System Identification Toolbox™

Reference

Lennart Ljung

MATLAB® & SIMULINK®

R2021b

MathWorks®

How to Contact MathWorks

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

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

System Identification Toolbox™ Reference

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)
March 2017	Online only	Revised for Version 9.6 (Release 2017a)
September 2017	Online only	Revised for Version 9.7 (Release 2017b)
March 2018	Online only	Revised for Version 9.8 (Release 2018a)
September 2018	Online only	Revised for Version 9.9 (Release 2018b)
March 2019	Online only	Revised for Version 9.10 (Release 2019a)
September 2019	Online only	Revised for Version 9.11 (Release 2019b)
March 2020	Online only	Revised for Version 9.12 (Release 2020a)
September 2020	Online only	Revised for Version 9.13 (Release 2020b)
March 2021	Online only	Revised for Version 9.14 (Release 2021a)
September 2021	Online only	Revised for Version 9.15 (Release 2021b)

Contents

Functions

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

(Not recommended) Add custom regressors to nonlinear ARX model

Note `addreg` is not recommended. Add linear, polynomial, and custom regressors directly to the `idnlarx` `Regressors` property instead. For more information, see “Compatibility Considerations”.

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],'idWaveletNetwork','nlr',[1:3]);
```

Create model with additional custom regressors, specified 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)
```

```
ans = 8x1 cell
    {'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],'idWaveletNetwork','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)
```

```
ans = 8x1 cell
    {'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)'}
```

Compatibility Considerations

`addreg` is not recommended

Not recommended starting in R2021a

Starting in R2021a, add regressor objects `linearRegressor`, `polynomialRegressor`, and `customRegressor` directly to the `idnlarx` model `Regressor` property by using the syntax `model.Regressors(end+1) = new_regressor_object`. There are no plans to remove `addreg` at this time.

See Also

`idnlarx` | `getreg` | `nlarx` | `customRegressor` | `polynomialRegressor` | `linearRegressor`

Topics

"Identifying Nonlinear ARX Models"

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 1-12 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 1-12 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:

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

  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:

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

- Bayesian Information Criteria, defined as:

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

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.

References

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

See Also

`fpe` | `goodnessOfFit`

Topics

"Loss Function and Model Quality Metrics"

Introduced before R2006a

append

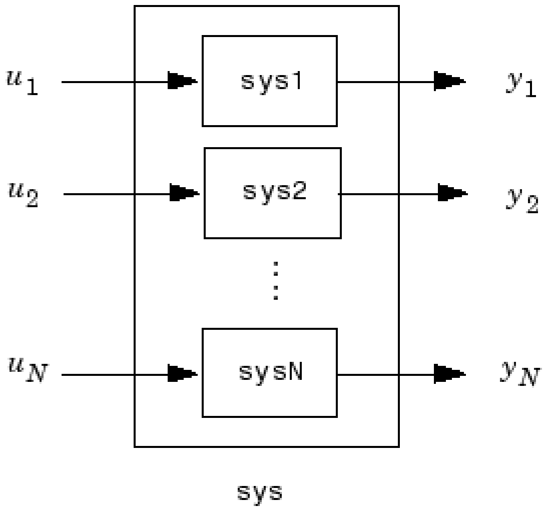
Group models by appending their inputs and outputs

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” (Control System Toolbox) 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)
```

```
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.
```

```
size(sys)
```

```
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 or ARI model for scalar time series

Syntax

```
sys = ar(y,n)
sys = ar(y,n,approach,window)
sys = ar(y,n, ___ ,Name,Value)
sys = ar(y,n, ___ ,opt)
[sys,refl] = ar(y,n,approach, ___ )
```

Description

`sys = ar(y,n)` estimates the parameters of an AR on page 1-25 `idpoly` model `sys` of order `n` using a least-squares method. The model properties include covariances (parameter uncertainties) and estimation goodness of fit.

`sys = ar(y,n,approach,window)` uses the algorithm specified by `approach` and the prewindowing and postwindowing specification in `window`. To specify `window` while accepting the default value for `approach`, use `[]` in the third position of the syntax.

`sys = ar(y,n, ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments. For instance, using the name-value pair argument `'IntegrateNoise',1` estimates an ARI on page 1-25 model, which is useful for systems with nonstationary disturbances. Specify `Name,Value` after any of the input argument combinations in the previous syntaxes.

`sys = ar(y,n, ___ ,opt)` specifies estimation options using the option set `opt`.

`[sys,refl] = ar(y,n,approach, ___ )` returns an AR model along with the reflection coefficients `refl` when `approach` is the lattice-based method `'burg'` or `'gl'`.

Examples

AR Model

Estimate an AR model and compare its response with the measured output.

Load the data, which contains the time series `z9` with noise.

```
load iddata9 z9
```

Estimate a fourth-order AR model.

```
sys = ar(z9,4)
```

```
sys =
Discrete-time AR model: A(z)y(t) = e(t)
  A(z) = 1 - 0.8369 z^-1 - 0.4744 z^-2 - 0.06621 z^-3 + 0.4857 z^-4

Sample time: 0.0039062 seconds
```

```
Parameterization:
   Polynomial orders:   na=4
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using AR ('fb/now') on time domain data "z9".
Fit to estimation data: 79.38%
FPE: 0.5189, MSE: 0.5108
```

The output displays the polynomial containing the estimated parameters alongside other estimation details. Under `Status`, `Fit to estimation data` shows that the estimated model has 1-step-ahead prediction accuracy above 75%.

You can find additional information about the estimation results by exploring the estimation report, `sys.Report`. For instance, you can retrieve the parameter covariance.

```
covar = sys.Report.Parameters.FreeParCovariance
```

```
covar = 4×4

    0.0015   -0.0015   -0.0005    0.0007
   -0.0015    0.0027   -0.0008   -0.0004
   -0.0005   -0.0008    0.0028   -0.0015
    0.0007   -0.0004   -0.0015    0.0014
```

For more information on viewing the estimation report, see “Estimation Report”.

Compare Burg's Method with Forward-Backward Approach

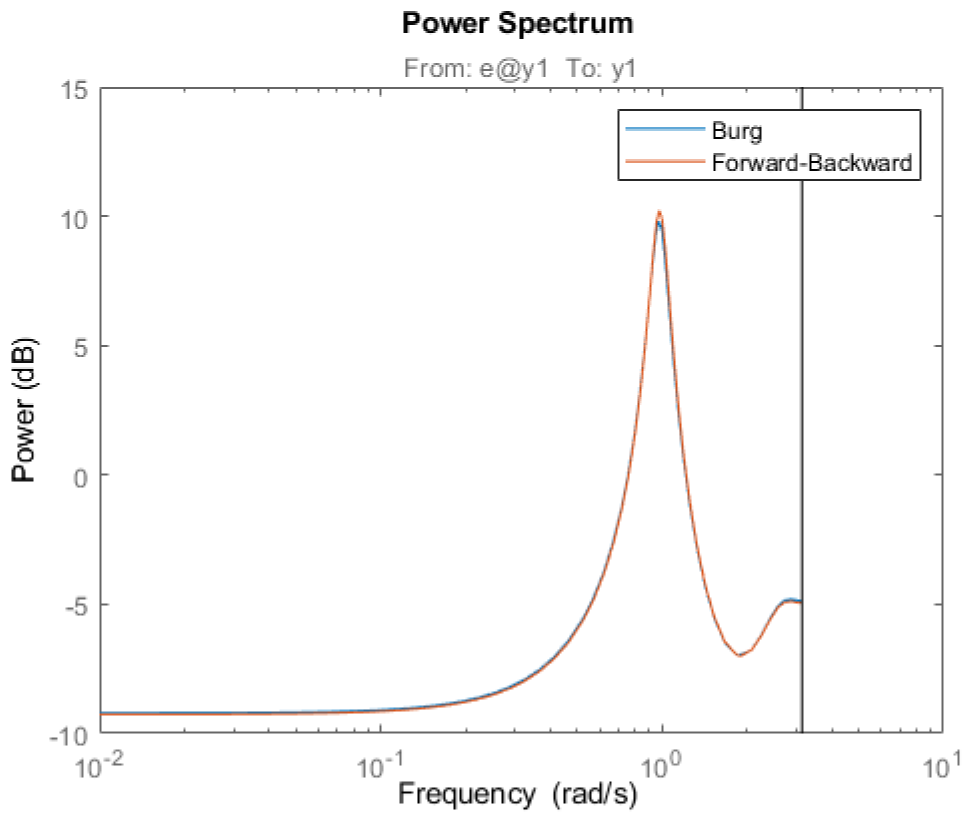
Given a sinusoidal signal with noise, compare the spectral estimates of Burg's method with those found using the forward-backward approach.

Generate an output signal and convert it into an `iddata` object.

```
y = sin([1:300]') + 0.5*randn(300,1);
y = iddata(y);
```

Estimate fourth-order AR models using Burg's method and using the default forward-backward approach. Plot the model spectra together.

```
sys_b = ar(y,4,'burg');
sys_fb = ar(y,4);
spectrum(sys_b,sys_fb)
legend('Burg','Forward-Backward')
```



The two responses match closely throughout most of the frequency range.

ARI Model

Estimate an ARI model, which includes an integrator in the noise source.

Load the data, which contains the time series `z9` with noise.

```
load iddata9 z9
```

Integrate the output signal.

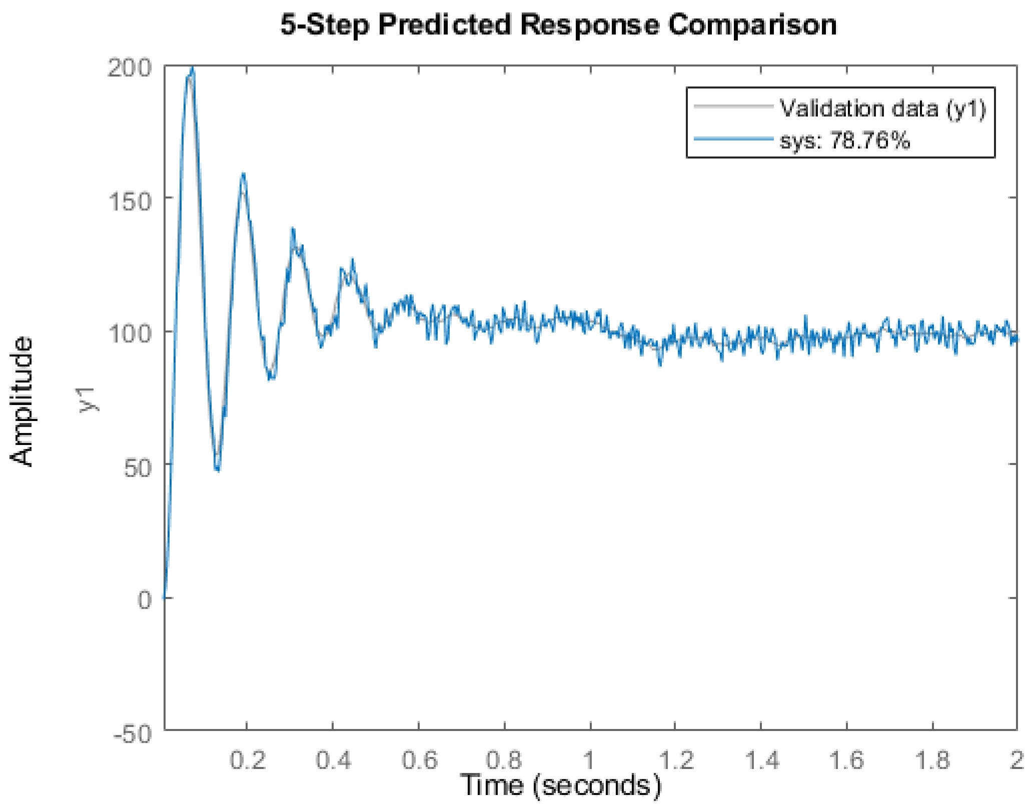
```
y = cumsum(z9.y);
```

Estimate an AR model with `'IntegrateNoise'` set to `true`. Use the least-squares method `'ls'`. Because `y` is a vector and not an `iddata` object, specify `Ts`.

```
Ts = z9.Ts;
sys = ar(y,4,'ls','Ts',Ts,'IntegrateNoise',true);
```

Predict the model output using 5-step prediction and compare the result with the integrated output signal `y`.

```
compare(y,sys,5)
```



Modify Default Options

Modify the default options for the AR function.

Load the data, which contains a time series `z9` with noise.

```
load iddata9 z9
```

Modify the default options so that the function uses the `'ls'` approach and does not estimate covariance.

```
opt = arOptions('Approach','ls','EstimateCovariance',false)
```

```
opt =
Option set for the ar command:

              Approach: 'ls'
                Window: 'now'
            DataOffset: 0
    EstimateCovariance: 0
               MaxSize: 250000

Description of options
```

Estimate a fourth-order AR model using the updated options.

```
sys = ar(z9,4,opt);
```

Retrieve Reflection Coefficients for Burg's Method

Retrieve reflection coefficients and loss functions when using Burg's method.

Lattice-based approaches such, as Burg's method `'burg'` and geometric lattice `'gl'`, compute reflection coefficients and corresponding loss function values as part of the estimation process. Use a second output argument to retrieve these values.

Generate an output signal and convert it into an `iddata` object.

```
y = sin([1:300]') + 0.5*randn(300,1);
y = iddata(y);
```

Estimate a fourth-order AR model using Burg's method and include an output argument for the reflection coefficients.

```
[sys,refl] = ar(y,4,'burg');
refl
```

```
refl = 2×5

         0   -0.3562    0.4430    0.5528    0.2385
    0.8494    0.7416    0.5960    0.4139    0.3904
```

Input Arguments

y — Time-series data
`iddata` object | double vector

Time-series data, specified as one of the following:

- An `iddata` object that contains a single output channel and an empty input channel.
- A double column vector containing output-channel data. When you specify `y` as a vector, you must also specify the sample time `Ts`.

n — Model order
positive integer

Model order, specified as a positive integer. The value of `n` determines the number of *A* parameters in the AR model.

Example: `ar(idy,2)` computes a second-order AR model from the single-channel `iddata` object `idy`

approach — Algorithm for computing AR model
`'fb'` (default) | `'burg'` | `'gl'` | `'ls'` | `'yw'`

Algorithm for computing the 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.

All of these algorithms are variants of the least-squares method. For more information, see "Algorithms" on page 1-26 .

Example: `ar(idy,2,'ls')` computes an AR model using the least-squares approach

`window` — Prewindowing and postwindowing
`'now'` | `'pow'` | `'ppw'` | `'prw`

Prewindowing and postwindowing outside the measured time interval (past and future values), specified as one of the following values:

- `'now'`: No windowing. This value is the default except when you set `approach` to `'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'`: Prewindowing and postwindowing. The software uses this value whenever you select the Yule-Walker approach `'yw'`, regardless of your `window` specification.
- `'prw'`: Prewindowing. Missing past values are replaced with zeros so that the summation in the criteria can start at time equal to zero.

Example: `ar(idy,2,'yw','ppw')` computes an AR model using the Yule-Walker approach with prewindowing and postwindowing.

`opt` — Estimation options
`arOptions` option set

Estimation options for AR model identification, specified as an `arOptions` option set. `opt` specifies the following options:

- Estimation approach
- Data windowing technique
- Data offset
- Maximum number of elements in a segment of data

For more information, see `arOptions`. For an example, see "Modify Default Options" on page 1-20.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'IntegrateNoise',true` adds an integrator in the noise source

Ts — Sample time
1 (default) | positive scalar

Sample time, specified as the comma-separated pair consisting of `'Ts'` and the sample time in seconds. If `y` is a double vector, then you must specify `'Ts'`.

Example: `ar(y_signal,2,'Ts',0.08)` computes a second-order AR model with sample time of 0.08 seconds

`IntegrateNoise` — Add integrator to noise channel
`false` (default) | logical vector

Noise-channel integration option for estimating ARI on page 1-25 models, specified as the comma-separated pair consisting of `'IntegrateNoise'` and a logical. Noise integration is useful in cases where the disturbance is nonstationary.

When using `'IntegrateNoise'`, you must also integrate the output-channel data. For an example, see “ARI Model” on page 1-19.

Output Arguments

sys — AR or ARI model
`idpoly` model object

AR on page 1-25 or ARI on page 1-25 model that fits the given estimation data, returned as a discrete-time `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.

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
`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:

Field	Description
`FitPercent`	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
`LossFcn`	Value of the loss function when the estimation completes.
`MSE`	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
`FPE`	Final prediction error for the model.
`AIC`	Raw Akaike Information Criteria (AIC) measure of model quality.
`AICc`	Small-sample-size corrected AIC.
`nAIC`	Normalized AIC.
`BIC`	Bayesian Information Criteria (BIC).

`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `arOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** — **Description** `Name` — Name of the data set. `Type` — Data type. `Length` — Number of data samples. `Ts` — Sample time. `InterSample` — Input intersample behavior, returned as one of the following values: • `'zoh'` — 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. `InputOffset` — Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`. `OutputOffset` — Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.

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

`refl` — Reflection coefficients and loss functions
array

Reflection coefficients and loss functions, returned as a 2-by-2 array. 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. For an example, see “Retrieve Reflection Coefficients for Burg's Method” on page 1-21.

More About

AR (Autoregressive) Model

The AR model structure has no input, and is given by the following equation:

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

This model structure accommodates estimation for scalar time-series data, which have no input channel. The structure is a special case of the ARX structure.

ARI (Autoregressive Integrated) Model

The ARI model is an AR model with an integrator in the noise channel. The ARI model structure is given by the following equation:

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

Algorithms

AR and ARI 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 with no windowing
Correlation method	Yule-Walker approach with prewindowing and postwindowing
Covariance method	Least squares approach with no windowing. `arx` uses this routine

References

[1] Marple, S. L., Jr. Chapter 8. *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice Hall, 1987.

See Also

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

Topics

“What Are Time Series Models?”
“Representing Data in MATLAB Workspace”
“Identify Time Series Models at the Command Line”

Introduced in R2006a

armax

Estimate parameters of ARMAX, ARIMAX, ARMA, or ARIMA 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)

[sys,ic] = armax(___)
```

Description

Estimate an ARMAX Model

`sys = armax(data,[na nb nc nk])` estimates the parameters of an ARMAX on page 1-40 or an ARMA on page 1-41 `idpoly` model `sys` using the prediction-error method and the polynomial orders specified in `[na nb nc nk]`. The model properties include estimation covariances (parameter uncertainties) and goodness of fit between the estimated and the measured data.

`sys = armax(data,[na nb nc nk],Name,Value)` specifies additional options using one or more name-value pair arguments. For instance, using the name-value pair argument `'IntegrateNoise',1` estimates an ARIMAX on page 1-41 or ARIMA on page 1-41 model, which is useful for systems with nonstationary disturbances.

Configure Initial Parameters

`sys = armax(data,init_sys)` uses the discrete-time linear model `init_sys` to configure the initial parameterization.

Specify Additional Options

`sys = armax(data,___,opt)` incorporates an option set `opt` that specifies options such as estimation objective, handling of initial conditions, regularization, and numerical search method to use for estimation. Specify `opt` after any of the previous input-argument combinations.

Return Estimated Initial Conditions

`[sys,ic] = armax(___)` returns the estimated initial conditions as an `initialCondition` object. Use this syntax if you plan to simulate or predict the model response using the same estimation input data and then compare the response with the same estimation output data. Incorporating the initial conditions yields a better match during the first part of the simulation.

Examples

Estimate ARMAX Model

Estimate an ARMAX model and view the fit of the model output to the estimation data.

Load the measurement data in `iddata` object `z2`.

```
load iddata2 z2
```

Estimate an ARMAX model with second-order *A*,*B*, and *C* polynomials and a transport delay of one sample period.

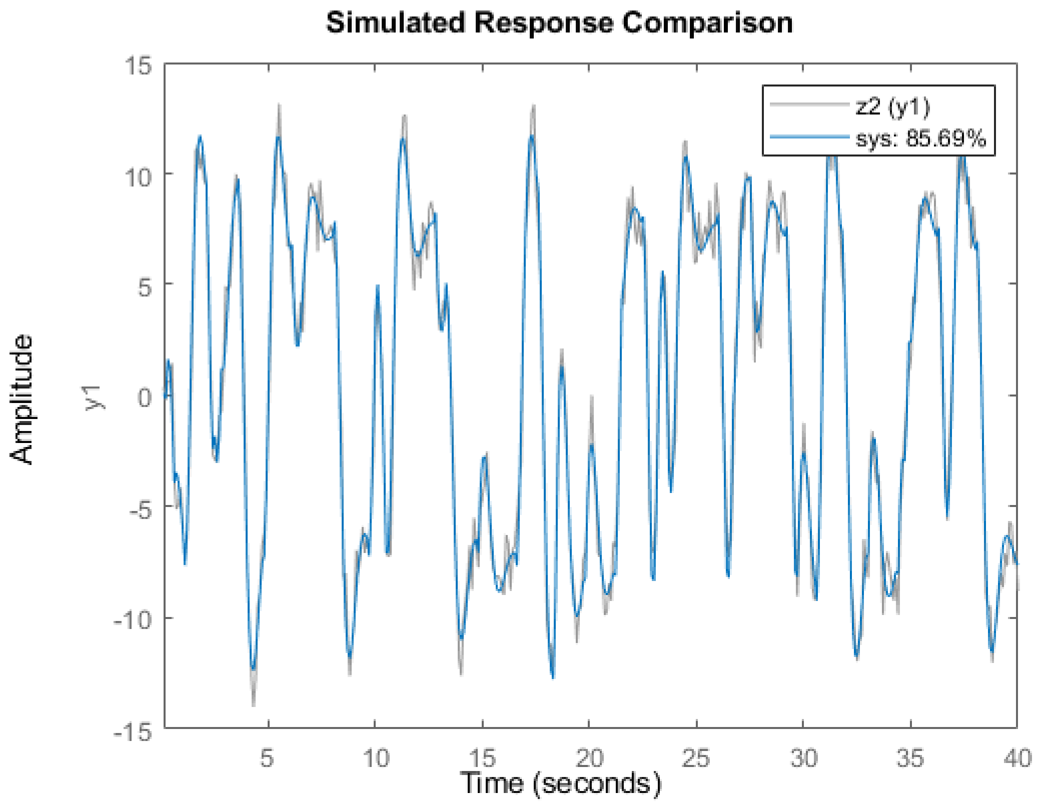
```
na = 2;
nb = 2;
nc = 2;
nk = 1;
sys = armax(z2,[na nb nc nk])
```

```
sys =
Discrete-time ARMAX model: A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 - 1.512 z^-1 + 0.7006 z^-2

  B(z) = -0.2606 z^-1 + 1.664 z^-2

  C(z) = 1 - 1.604 z^-1 + 0.7504 z^-2

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=2   nb=2   nc=2   nk=1
   Number of free coefficients: 6
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARMAX on time domain data "z2".
Fit to estimation data: 85.89% (prediction focus)
FPE: 1.086, MSE: 1.054
```

The output displays the polynomial containing the estimated parameters alongside the estimation details. Under `Status`, `Fit to estimation data` shows that the estimated model has 1-step-ahead prediction accuracy above 80%.

Compare the model simulated output to the measured data.

```
compare(z2,sys)
```



The fit of the simulated model to the measured data is nearly the same as the estimation fit.

ARMA Model

Estimate an ARMA model and compare its response with both the measured output and an AR model.

Load the data, which contains the time series `z9` with noise.

```
load iddata9 z9
```

Estimate a fourth-order ARMA model with a first-order *C* polynomial.

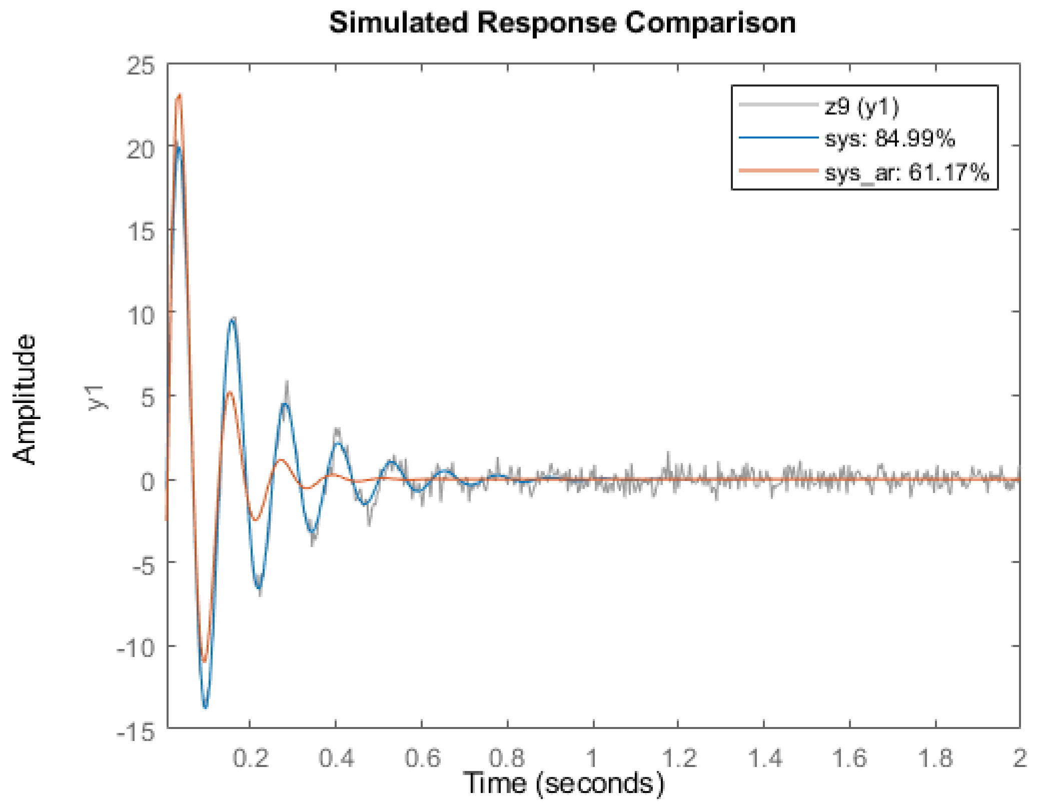
```
na = 4;
nc = 1;
sys = armax(z9,[na nc]);
```

Estimate a fourth-order AR model.

```
sys_ar = ar(z9,na);
```

Compare the model outputs with the measured data.

```
compare(z9,sys,sys_ar)
```



The ARMA model has the better fit to the data.

Specify Estimation Options

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

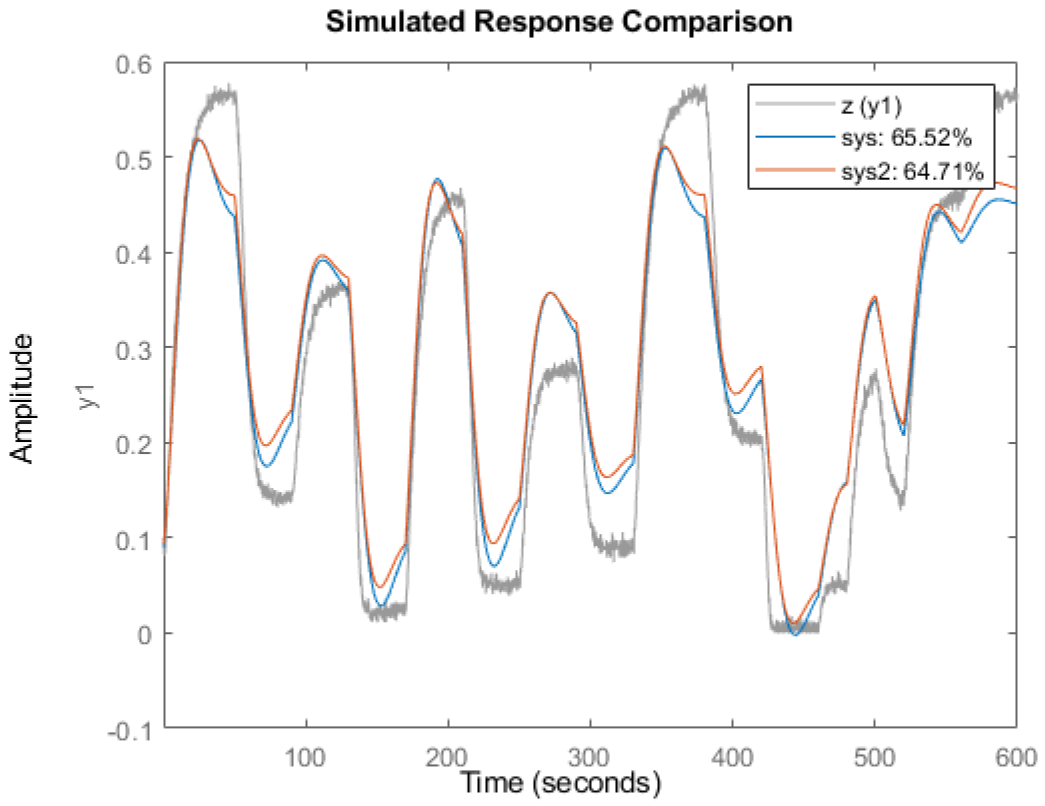
Load the data and create an `iddata` object. Initialize option set `opt`, and set options for `Focus`, `SearchMethod`, `MaxIterations`, and `Display`. Then estimate the ARMAX model using the updated option set.

```
load twotankdata;
z = iddata(y,u,0.2);
opt = armaxOptions;
opt.Focus = 'simulation';
opt.SearchMethod = 'lm';
opt.SearchOptions.MaxIterations = 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 `MaxIterations`, 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.

ARMAX Model with 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 30.

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

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

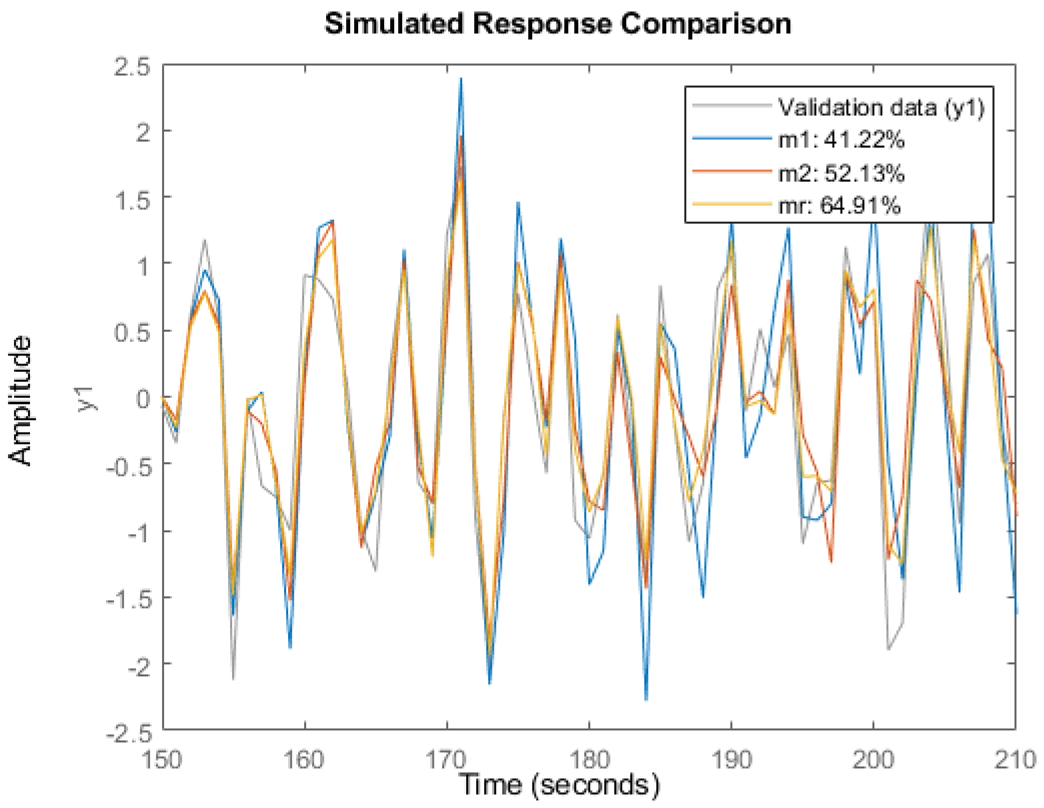
```
opt = armaxOptions;
opt.Regularization.Lambda = 1;
m2 = armax(m0simdata(1:150),[30 30 30 1],opt);
```

Obtain a lower order ARMAX model by converting a regularized ARX model and then performing 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 the data.

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



ARIMA Model

Estimate a fourth-order ARIMA model for univariate time-series data.

Load data that contains a time series with noise.

```
load iddata9 z9;
```

Integrate the output signal and use the result to replace the original output signal in `z9`.

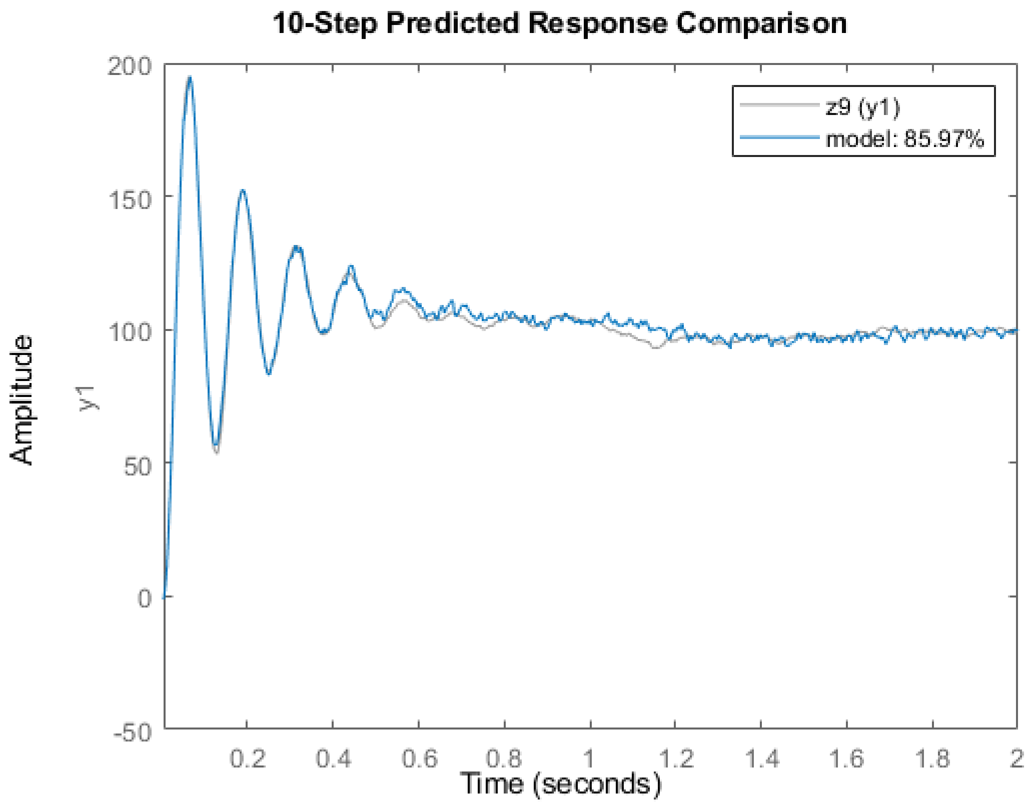
```
z9.y = cumsum(z9.y);
```

Estimate a fourth-order ARIMA model with a first-order*C*polynomial by setting `'IntegrateNoise'` to `true`.

```
model = armax(z9,[4 1],'IntegrateNoise',true);
```

Predict the model output using 10-step ahead prediction, and compare the predicted output with the estimation data.

```
compare(z9,model,10)
```



Estimate ARMAX Models Iteratively

Estimate ARMAX models of varying orders iteratively from measured data.

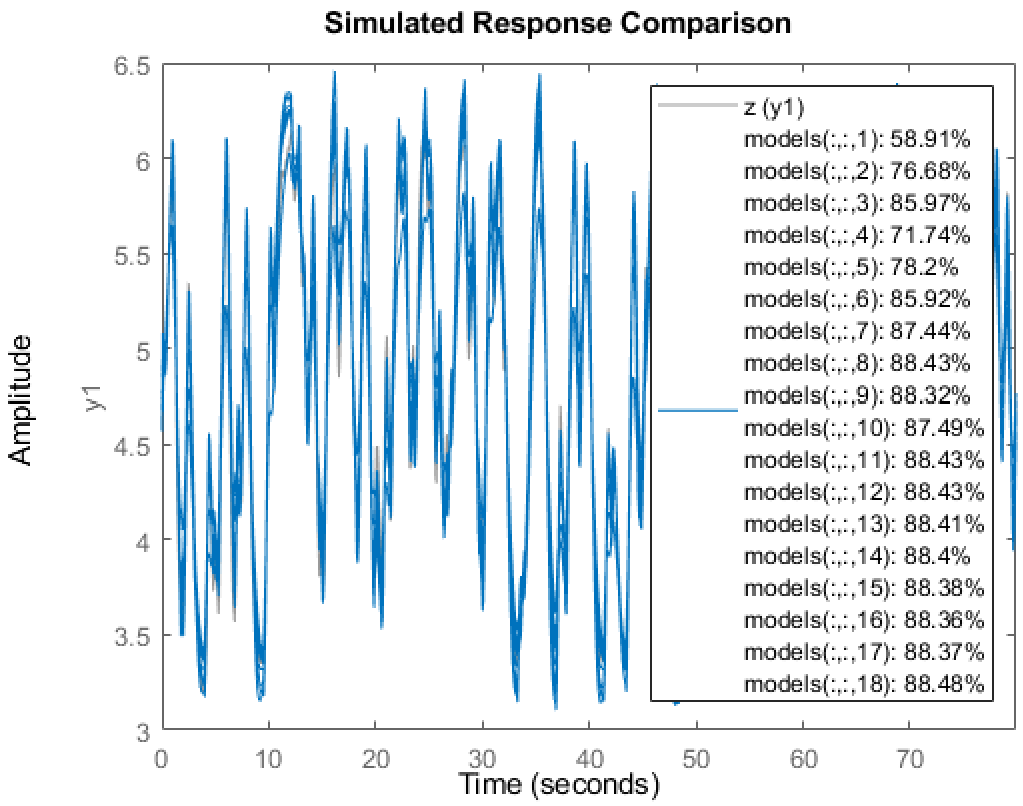
Load `dryer2` data and perform estimation for combinations of polynomial orders `na`, `nb`, `nc`, and input delay `nk`.

```
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 the estimation data z.

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



Initialize ARMAX Model Parameters Using State-Space Model

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 to initialize the parameters.

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

Obtain Initial Conditions

Load the data.

```
load iddata1ic z1i
```

Estimate a second-order ARMAX model `sys` and return the initial conditions in `ic`.

```
na = 2;
nb = 2;
nc = 2;
nk = 1;
[sys,ic] = armax(z1i,[na nb nc nk]);
ic
```

```
ic = 
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [0 1]
    Ts: 0.1000
```

`ic` is an `initialCondition` object that encapsulates the free response of `sys`, in state-space form, to the initial state vector in `X0`. You can incorporate `ic` when you simulate `sys` with the `z1i` input signal and compare the response with the `z1i` output signal.

Input Arguments

`data` — Time-domain estimation data
`iddata` object

Time-domain estimation data, specified as an `iddata` object. For ARMA and ARIMA time-series models, the input channel in `data` must be empty. For examples, see “ARMA Model” on page 1-29 and “ARIMA Model” on page 1-32.

`[na nb nc nk]` — Polynomial orders
integer row vector | row vector of integer matrices | scalar

Polynomial orders and delays for the model, specified as a 1-by-4 vector or vector of matrices `[na nb nc nk]`. The polynomial order is equal to the number of coefficients to estimate in that polynomial.

For an ARMA or ARIMA time-series model, which has no input, set `[na nb nc nk]` to `[na nc]`. For an example, see “ARMA Model” on page 1-29.

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, also known at the transport delay, specified as an *Ny*-by-*Nu* matrix of nonnegative integers. `nk` is represented in ARMAX models by fixed leading zeros of the *B* polynomial.

For an example, see “Estimate ARMAX Model” on page 1-27.

`init_sys` — System for configuring initial parameterization
discrete-time linear model

System for configuring the initial parameterization 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 with the 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.

For an example, see “Initialize ARMAX Model Parameters Using State-Space Model” on page 1-34.

`opt` — Estimation options
`armaxOptions` option set

Estimation options for ARMAX model identification, specified as an `armaxOptions` option set. Options specified by `opt` include the following:

- Initial condition handling — Use this option to determine how the initial conditions are set or estimated.
- Input and output data offsets — Use these options to remove offsets from data during estimation.
- Regularization — Use this option to control the tradeoff between bias and variance errors during the estimation process.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InputDelay',2` applies an input delay of two sample periods to all input channels

`InputDelay` — Input delays
`0` (default) | integer scalar | positive integer vector

Input delays expressed as integer multiples of the sample time, specified as the comma-separated pair consisting of `'InputDelay'` and one of the following:

- N_u-by-1 vector, where N_u is the number of inputs — Each entry is a numerical value representing the input delay for the corresponding input channel.
- Scalar value — Apply the same delay to all input channels.
- `0` — No input delays.

Example: `armax(data,[2 1 1 0],'InputDelay',1)` estimates a second-order ARX model with first-order B and C polynomials that has an input delay of two samples.

`IODelay` — Transport delays
`0` (default) | scalar | matrix

Transport delays for each input-output pair, expressed as integer multiples of the sample time, and specified as the comma-separated pair consisting of `'IODelay'` and one of the following:

- N_y-by-N_u matrix, where N_y is the number of outputs and N_u is the number of inputs — Each entry is an integer value representing the transport delay for the corresponding input-output pair.
- Scalar value — Apply the same delay 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` (default) | logical vector

Addition of integrators in the noise channel, specified as the comma-separated pair consisting of `'IntegrateNoise'` and a logical vector of length N_y, where N_y is the number of outputs.

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

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

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

Use `'IntegrateNoise'` to create ARIMA or ARIMAX models.

For an example, see “ARIMA Model” on page 1-32.

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.
`Method`	Estimation command used.
`InitialCondition`	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:

Field	Description
`FitPercent`	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
`LossFcn`	Value of the loss function when the estimation completes.
`MSE`	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
`FPE`	Final prediction error for the model.
`AIC`	Raw Akaike Information Criteria (AIC) measure of model quality.
`AICc`	Small-sample-size corrected AIC.
`nAIC`	Normalized AIC.
`BIC`	Bayesian Information Criteria (BIC).

Report Field	Description
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `armaxOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
DataUsed	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** / **Description**: `Name` — Name of the data set. `Type` — Data type. `Length` — Number of data samples. `Ts` — Sample time. `InterSample` — Input intersample behavior, returned as one of the following values: • `'zoh'` — 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. `InputOffset` — Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`. `OutputOffset` — Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.
Termination	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** / **Description**: `WhyStop` — Reason for terminating the numerical search. `Iterations` — Number of search iterations performed by the estimation algorithm. `FirstOrderOptimality` — ∞-norm of the gradient search vector when the search algorithm terminates. `FcnCount` — Number of times the objective function was called. `UpdateNorm` — Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `LastImprovement` — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `Algorithm` — Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used. For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.

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

ic — Initial conditions
`initialCondition` object | object array of `initialCondition` values

Estimated initial conditions, returned as an `initialCondition` object or an object array of `initialCondition` values.

- For a single-experiment data set, `ic` represents, in state-space form, the free response of the transfer function model (*A* and *C* matrices) to the estimated initial states (x_0).
- For a multiple-experiment data set with N_e experiments, `ic` is an object array of length N_e that contains one set of `initialCondition` values for each experiment.

If `armax` returns `ic` values of `0` and the you know that you have non-zero initial conditions, set the `'InitialCondition'` option in `armaxOptions` to `'estimate'` and pass the updated option set to `armax`. For example:

```
opt = armaxOptions('InitialCondition','estimate')
[sys,ic] = armax(data,np,nz,opt)
```

The default `'auto'` setting of `'InitialCondition'` uses the `'zero'` method when the initial conditions have a negligible effect on the overall estimation-error minimization process. Specifying `'estimate'` ensures that the software estimates values for `ic`.

For more information, see `initialCondition`. For an example of using this argument, see “Obtain Initial Conditions” on page 1-35.

More About

ARMAX Model

The ARMAX (Autoregressive Moving Average with Extra Input) 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,

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

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

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

ARMA Time-Series Model

The ARMA (Autoregressive Moving Average) model is a special case of an “ARMAX Model” on page 1-40 with no input channels. The ARMA single-output model structure is given by the following equation:

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

ARIMAX Model

The ARIMAX (Autoregressive Integrated Moving Average with Extra Input) model structure is similar to the ARMAX model, except that it contains an integrator in the noise source *e(t)*:

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

ARIMA Model

The ARIMA (Autoregressive Integrated Moving Average) model structure is a reduction of the ARIMAX model with no inputs:

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

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 `MaxIterations` and `Tolerance`, using the `'SearchOptions'` 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. The cutoff value 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'`. When `'Display'` is `'Full'`, both the current and the previous parameter estimates are displayed in column-vector form, and the parameters are listed in alphabetical order. Also, the values of the criterion function (cost) are given and the Gauss-Newton vector and its norm are displayed. When `'Display'` is `'On'`, only the criterion values are displayed.

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.

References

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

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox™). To enable parallel computing, use `armaxOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = armaxOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

`armaxOptions` | `arx` | `bj` | `oe` | `polyest` | `ssest` | `tfest` | `idpoly` | `iddata` | `compare` | `aic` | `fpe`

Topics

“What Are Polynomial Models?”
“What Are Time Series Models?”
“Estimate Models Using armax”
“Estimation Report”
“Loss Function and Model Quality Metrics”

"Regularized Estimates of Model Parameters"
"Apply Initial Conditions when Simulating Identified Linear Models"

Introduced in 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 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`

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

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

If `EstimateCovariance` 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` — Numerical search method used for iterative parameter estimation
`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

SearchOptions Structure When SearchMethod is Specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description	Default
Tolerance	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
MaxIterations	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as Tolerance. Setting MaxIterations = 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

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields: (see nested table below)	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	`1e-6`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`.	`100`

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

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

  - 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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`

Topics

“Loss Function and Model Quality Metrics”

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 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, ARIX, AR, or ARI model

Syntax

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

Description

`sys = arx(data,[na nb nk])` estimates the parameters of an ARX on page 1-67 or an AR on page 1-68 `idpoly` model `sys` using a least-squares method and the polynomial orders specified in `[na nb nk]`. The model properties include covariances (parameter uncertainties) and goodness of fit between the estimated and measured data.

`sys = arx(data,[na nb nk],Name,Value)` specifies additional options using one or more name-value pair arguments. For instance, using the name-value pair argument `'IntegrateNoise',1` estimates an ARIX on page 1-68 or ARI structure model, which is useful for systems with nonstationary disturbances.

`sys = arx(data,[na nb nk],___,opt)` specifies estimation options using the option set `opt`. Specify `opt` after all other input arguments.

`[sys,ic] = arx(___)` returns the estimated initial conditions as an `initialCondition` object. Use this syntax if you plan to simulate or predict the model response using the same estimation input data and then compare the response with the same estimation output data. Incorporating the initial conditions yields a better match during the first part of the simulation.

Examples

ARX Model

Generate output data based on a specified ARX model and use the output data to estimate the model.

Specify a polynomial model `sys0` with the ARX structure. The model includes an input delay of one sample, expressed as a leading zero in the `B` polynomial.

```
A = [1  -1.5  0.7];
B = [0 1 0.5];
sys0 = idpoly(A,B);
```

Generate a measured input signal `u` that contains random binary noise and an error signal `e` that contains normally distributed noise. With these signals, simulate the measured output signal `y` of `sys0`.

```
u = iddata([],idinput(300,'rbs'));
e = iddata([],randn(300,1));
y = sim(sys0,[u e]);
```

Combine y and u into a single iddata object z. Estimate a new ARX model using z and the same polynomial orders and input delay as the original model.

```
z = [y,u];
sys = arx(z,[2 2 1])
```

```
sys =
Discrete-time ARX model: A(z)y(t) = B(z)u(t) + e(t)
  A(z) = 1 - 1.524 z^-1 + 0.7134 z^-2

  B(z) = z^-1 + 0.4748 z^-2

Sample time: 1 seconds

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

Status:
Estimated using ARX on time domain data "z".
Fit to estimation data: 81.36% (prediction focus)
FPE: 1.025, MSE: 0.9846
```

The output displays the polynomial containing the estimated parameters alongside other estimation details. Under Status, Fit to estimation data shows that the estimated model has 1-step-ahead prediction accuracy above 80%.

AR Model

Estimate a time-series AR model using the arx function. An AR model has no measured input.

Load the data, which contains the time series z9 with noise.

```
load iddata9 z9
```

Estimate a fourth-order AR model by specifying only the na order in [na nb nk].

```
sys = arx(z9,4);
```

Examine the estimated A polynomial parameters and the fit of the estimate to the data.

```
param = sys.Report.Parameters.ParVector
```

```
param = 4×1

   -0.7923
   -0.4780
   -0.0921
    0.4698
```

```
fit = sys.Report.Fit.FitPercent
```

```
fit = 79.4835
```

ARIX Model

Estimate the parameters of an ARIX model. An ARIX model is an ARX model with integrated noise.

Specify a polynomial model `sys0` with an ARX structure. The model includes an input delay of one sample, expressed as a leading zero in `B`.

```
A = [1  -1.5  0.7];
B = [0 1 0.5];
sys0 = idpoly(A,B);
```

Simulate the output signal of `sys0` using the random binary input signal `u` and the normally distributed error signal `e`.

```
u = iddata([],idinput(300,'rbs'));
e = iddata([],randn(300,1));
y = sim(sys0,[u e]);
```

Integrate the output signal and store the result `yi` in the `iddata` object `zi`.

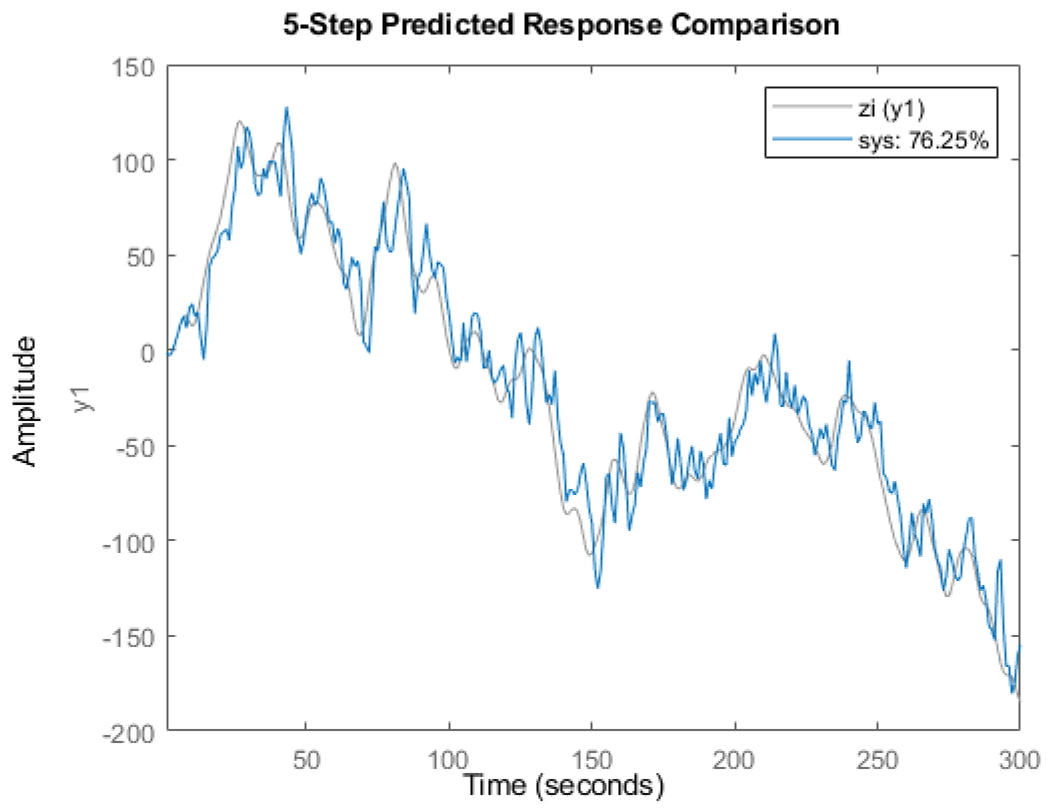
```
yi = iddata(cumsum(y.y),[]);
zi = [yi,u];
```

Estimate an ARIX model from `zi`. Set the name-value pair argument `'IntegrateNoise'` to `true`.

```
sys = arx(zi,[2 2 1],'IntegrateNoise',true);
```

Predict the model output using 5-step prediction and compare the result with `yi`.

```
compare(zi,sys,5)
```



ARX Model with Regularization

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

Obtain the `lambda` and `R` values.

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

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

```
opt = arxOptions;
opt.Regularization.Lambda = lambda;
opt.Regularization.R = R;
sys = arx(eData,orders,opt);
```

Obtain Initial Conditions

Load the data.

```
load iddata1ic z1i
```

Estimate a second-order ARX model sys and return the initial conditions in ic.

```
na = 2;
nb = 2;
nk = 1;
[sys,ic] = arx(z1i,[na nb nk]);
ic
```

```
ic = 
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [0 2]
    Ts: 0.1000
```

ic is an initialCondition object that encapsulates the free response of sys, in state-space form, to the initial state vector in X0. You can incorporate ic when you simulate sys with the z1i input signal and compare the response with the z1i output signal.

Input Arguments

data — Estimation data
iddata object | frd object | idfrd object

Estimation data, specified as an iddata object, an frd object, or an idfrd frequency-response object. For AR and ARI time-series models, the input channel in data must be empty.

[na nb nk] — Polynomial orders and delays
integer row vector | row vector of integer matrices | scalar

Polynomial orders and delays for the model, specified as a 1-by-3 vector or vector of matrices [na nb nk]. The polynomial order is equal to the number of coefficients to estimate in that polynomial.

For an AR or ARI time-series model, which has no input, set [na nb nk] to the scalar na. For an example, see “AR Model” on page 1-60.

For a model with N_y outputs and N_u inputs:

- na is the order of polynomial $A(q)$, specified as an N_y-by-N_y matrix of nonnegative integers.
- nb is the order of polynomial $B(q) + 1$, specified as an N_y-by-N_u matrix of nonnegative integers.
- nk is the input-output delay, also known as the transport delay, specified as an N_y-by-N_u matrix of nonnegative integers. nk is represented in ARX models by fixed leading zeros in the B polynomial.

  For instance, suppose that without transport delays, sys.b is [5 6].

  - Because sys.b + 1 is a second-order polynomial, nb = 2.
  - Specify a transport delay of nk = 3. Specifying this delay adds three leading zeros to sys.b so that sys.b is now [0 0 0 5 6], while nb remains equal to 2.
  - These coefficients represent the polynomial $B(q) = 5\ q^{-3} + 6q^{-4}$.

You can also implement transport delays using the name-value pair argument `'IODelay'`.

.

Example: `arx(data,[2 1 1])` computes, from an `iddata` object, a second-order ARX model with one input channel that has an input delay of one sample.

`opt` — Estimation options
`arxOptions` option set

Estimation options for ARX model identification, specified as an `arOptions` option set. Options specified by `opt` include the following:

- Initial condition handling — Use this option only for frequency-domain data. For time-domain data, the signals are shifted such that unmeasured signals are never required in the predictors.
- Input and output data offsets — Use these options to remove offsets from time-domain data during estimation.
- Regularization — Use this option to control the tradeoff between bias and variance errors during the estimation process.

For more information, see `arxOptions`. For an example, see “ARX Model with Regularization” on page 1-62.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'IntegrateNoise',true` adds an integrator in the noise source s

`InputDelay` — Input delays
0 (default) | integer scalar | positive integer vector

Input delays expressed as integer multiples of the sample time, specified as the comma-separated pair consisting of `'InputDelay'` and one of the following:

- N_u-by-1 vector, where N_u is the number of inputs — Each entry is a numerical value representing the input delay for the corresponding input channel.
- Scalar value — Apply the same delay to all input channels.

Example: `arx(data,[2 1 3],'InputDelay',1)` estimates a second-order ARX model with one input channel that has an input delay of three samples.

`IODelay` — Transport delays
0 (default) | integer scalar | integer array

Transport delays for each input-output pair, expressed as integer multiples of the sample time, and specified as the comma-separated pair consisting of `'IODelay'` and one of the following:

- N_y-by-N_u matrix, where N_y is the number of outputs and N_u is the number of inputs — Each entry is an integer value representing the transport delay for the corresponding input-output pair.
- Scalar value — Apply the same delay is applied to all input-output pairs. This approach is useful when the input-output delay parameter `nk` results in a large number of fixed leading zeros in the *B*

polynomial. You can factor out `max(nk-1,0)` lags by moving those lags from `nk` into the `'IODelay'` value.

For instance, suppose that you have a system with two inputs, where the first input has a delay of three samples and the second input has a delay of six samples. Also suppose that the *B* polynomials for these inputs are order `n`. You can express these delays using the following:

- `nk` = `[3 6]` — This results in B polynomials of `[0 0 0 b11 ... b1n]` and `[0 0 0 0 0 0 b21 ... b2n]`.
- `nk` = `[3 6]` and `'IODelay',3` — This results in B polynomials of `[b11 ... b1n]` and `[0 0 0 b21 ... b2n]`.

IntegrateNoise — Addition of integrators in noise channel
`false` (default) | logical vector

Addition of integrators in the 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 creates an ARIX on page 1-68 or ARI model for that channel. Noise integration is useful in cases where the disturbance is nonstationary.

When using `'IntegrateNoise'`, you must also integrate the output channel data. For an example, see “ARIX Model” on page 1-61.

Output Arguments

sys — ARX model
`idpoly` object

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.

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.
`InitialCondition`	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. This field is especially useful to view how the initial conditions were handled when the `InitialCondition` option in the estimation option set is `'auto'`.

Report Field	Description
`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:

Field	Description
`FitPercent`	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
`LossFcn`	Value of the loss function when the estimation completes.
`MSE`	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
`FPE`	Final prediction error for the model.
`AIC`	Raw Akaike Information Criteria (AIC) measure of model quality.
`AICc`	Small-sample-size corrected AIC.
`nAIC`	Normalized AIC.
`BIC`	Bayesian Information Criteria (BIC).

Report Field	Description
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `arxOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields.

Field	Description
`Name`	Name of the data set.
`Type`	Data type.
`Length`	Number of data samples.
`Ts`	Sample time.
`InterSample`	Input intersample behavior, returned as one of the following values: • `'zoh'` — 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.
`InputOffset`	Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`.
`OutputOffset`	Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.

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

`ic` — Initial conditions
`initialCondition` object | object array of `initialCondition` values

Estimated initial conditions, returned as an `initialCondition` object or an object array of `initialCondition` values.

- For a single-experiment data set, `ic` represents, in state-space form, the free response of the transfer function model (*A* and *C* matrices) to the estimated initial states (x_0).
- For a multiple-experiment data set with N_e experiments, `ic` is an object array of length N_e that contains one set of `initialCondition` values for each experiment.

For more information, see `initialCondition`. For an example of using this argument, see “Obtain Initial Conditions” on page 1-62.

More About

ARX Structure

The ARX model name stands for Autoregressive with Extra Input, because, unlike the AR model, the ARX model includes an input term. ARX is also known as Autoregressive with Exogenous Variables, where the exogenous variable is the input term. The ARX model structure is given by the following equation:

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

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 zeros
- 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,

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

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

ARIX Model

The ARIX (Autoregressive Integrated with Extra Input) model is an ARX model with an integrator in the noise channel. The ARIX model structure is given by the following equation:

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

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

AR Time-Series Models

For time-series data that contains no inputs, one output, and the *A* polynomial order *na*, the model has an AR structure of order *na*.

The AR (Autoregressive) model structure is given by the following equation:

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

ARI Model

The ARI (Autoregressive Integrated) model is an AR model with an integrator in the noise channel. The ARI model structure is given by the following equation:

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

Multiple-Input, Single-Output Models

For multiple-input, single-output systems (MISO) with *nu* inputs, *nb* and *nk* are row vectors where the *i*th element corresponds to the order and delay associated with the *i*th input in column vector $u(t)$. Similarly, the coefficients of the *B* polynomial are row vectors. The ARX MISO structure is then given by the following equation:

$$A(q)y(t) = B_1(q)u_1(t - nk_1) + B_2(q)u_2(t - nk_2) + \cdots + B_{nu}(q)u_{nu}(t - nk_{nu})$$

Multiple-Input, Multiple-Output Models

For multiple-input, multiple-output systems, `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 norm to an arbitrary quadratic norm using a weighting matrix `Lambda`

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

use the following syntax:

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

Initial Conditions

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

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

Set the `'InitialCondition'` estimation option (see `arxOptions`) to one of the following values:

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

Algorithms

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

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 the regularization term

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

where λ and R are the regularization constants. For more information on the regularization constants, see `arxOptions`.

When the regression matrix is larger than the `MaxSize` specified in `arxOptions`, the data is segmented and QR factorization is performed iteratively on the data segments.

See Also

`arxOptions` | `arxRegul` | `arxstruc` | `ar` | `armax` | `iv4` | `idinput` | `iddata` | `idfrd`

Topics
“What Are Polynomial Models?”
“What Are Time Series Models?”
“Estimate Polynomial Models at the Command Line”
“Regularized Estimates of Model Parameters”
“Estimating Models Using Frequency-Domain Data”
“Apply Initial Conditions when Simulating Identified Linear Models”

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 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`

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

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

If `EstimateCovariance` 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

See Also

`arx` | `arxRegul` | `idfilt`

Topics

“Loss Function and Model Quality Metrics”

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('RegularizationKernel','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('RegularizationKernel','DC');
[Lambda,R] = arxRegul(z1,orders,opt,'IntegrateNoise',true);
```

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 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.

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 the regularization term

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

where λ and R are the regularization constants. For more information on the regularization constants, see `arxOptions`.

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`

Topics
“Estimate Regularized ARX Model Using System Identification App”
“Regularized Estimates of Model Parameters”

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('RegularizationKernel','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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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

`RegularizationKernel` — 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.

Default: `'fmincon'`

Output Arguments

`opt` — Regularization options
`arxRegulOptions` options set

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

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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`

Topics
“Regularized Estimates of Model Parameters”

Introduced in R2014a

arxstruc

Compute loss functions for single-output ARX models

Syntax

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

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]
```

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`.

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);
```

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,info] = balred(sys,order)
[~,info] = balred(sys)
[ ___ ] = balred( ___ ,opts)
```

```
balred(sys)
```

Description

`[rsys,info] = balred(sys,order)` computes a reduced-order approximation `rsys` of the LTI model `sys`. The desired order (number of states) is specified by `order`. You can try multiple orders at once by setting `order` to a vector of integers, in which case `rsys` is an array of reduced models. `balred` also returns a structure `info` with additional information like the Hankel singular values (HSV), error bound, regularization level and the Cholesky factors of the gramians.

`[~,info] = balred(sys)` returns the structure `info` without computing the reduced-order model. You can use this information to select the reduced order `order` based on your desired fidelity.

Note When performance is a concern, avoid computing the Hankel singular values twice by using the information obtained from the above syntax to select the desired model order and then use `rsys = balred(sys,order,info)` to compute the reduced-order model.

`[ ___ ] = balred( ___ ,opts)` computes the reduced model using the options set `opts` that you specify using `balredOptions`. You can specify additional options for eliminating states, using absolute vs. relative error control, emphasizing certain time or frequency bands, and separating the stable and unstable modes. See `balredOptions` to create and configure the option set `opts`.

`balred(sys)` displays the Hankel singular values and approximation error on a plot. Use `hsvplot` to customize this plot.

Examples

Reduced-Order Model using Hankel Singular Values

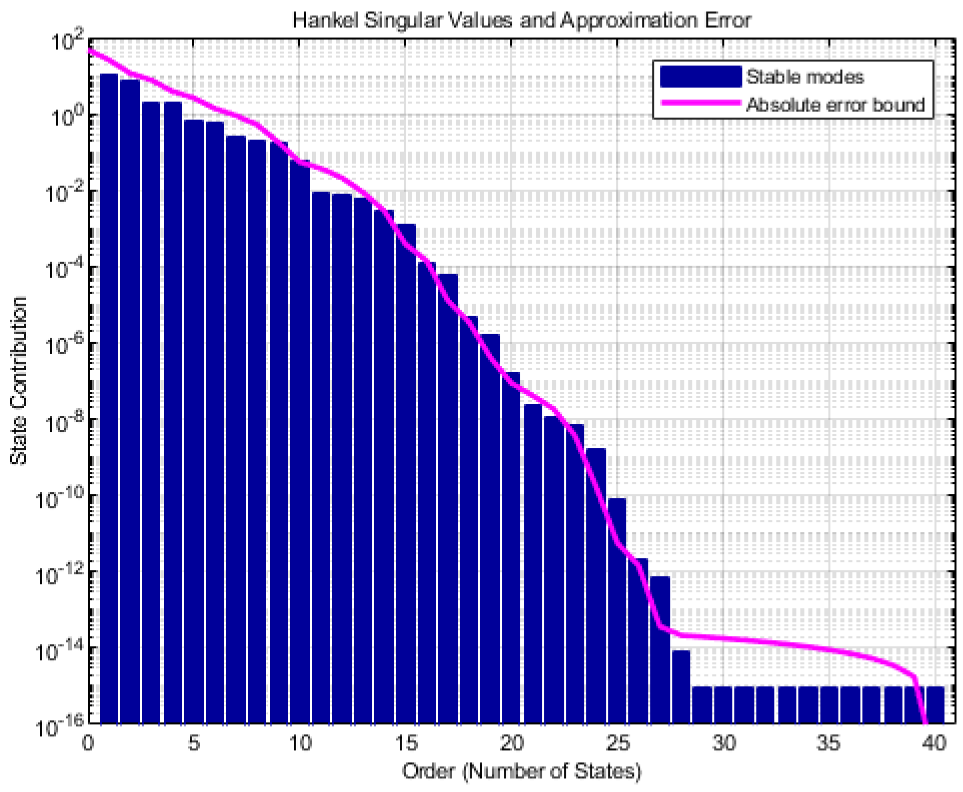
For this example, use the Hankel singular value plot to select suitable order and compute the reduced-order model.

For this instance, generate a random discrete-time state-space model with 40 states.

```
rng(0)
sys = drss(40);
```

Plot the Hankel singular values using `balred`.

```
balred(sys)
```

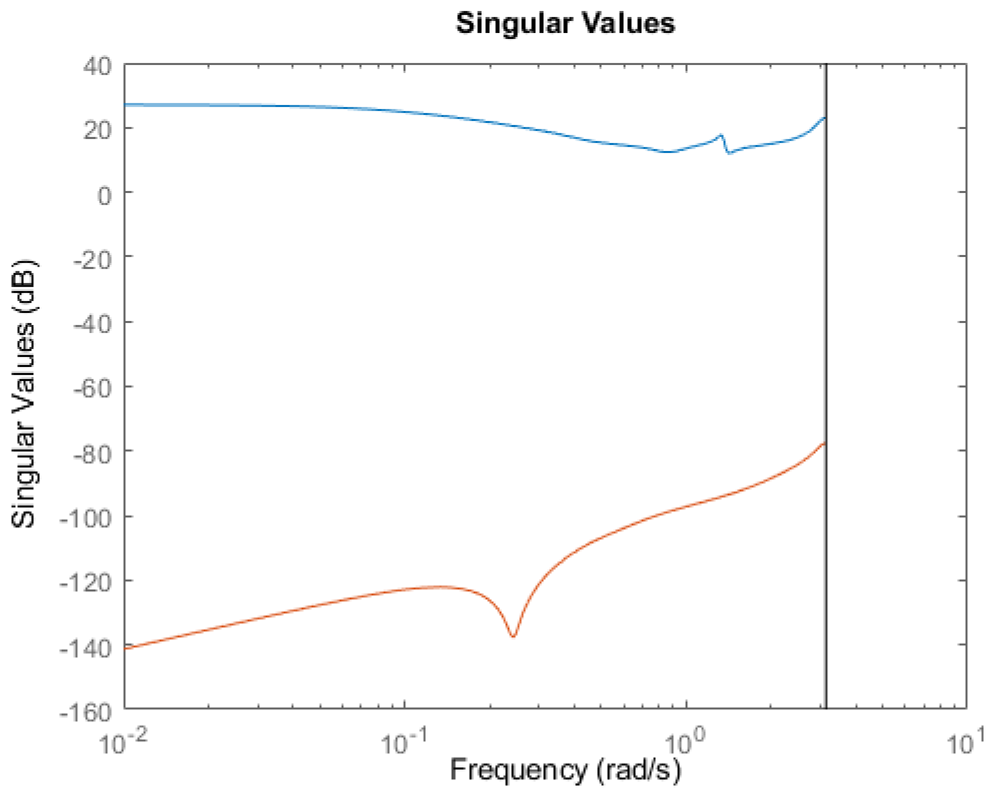


For this example, select order of `16` since it is the first order with an absolute error less than `1e-4`. In general, you select the order based on the desired absolute or relative fidelity. Then, compute the reduced-order model.

```
rsys = balred(sys,16);
```

Verify the absolute error by plotting the singular value response using `sigma`.

```
sigma(sys,sys-rsys)
```



Observe from the plot that the error, represented by the red curve, is below `-80 dB` (`1e-4`).

Array of Reduced-Order Models

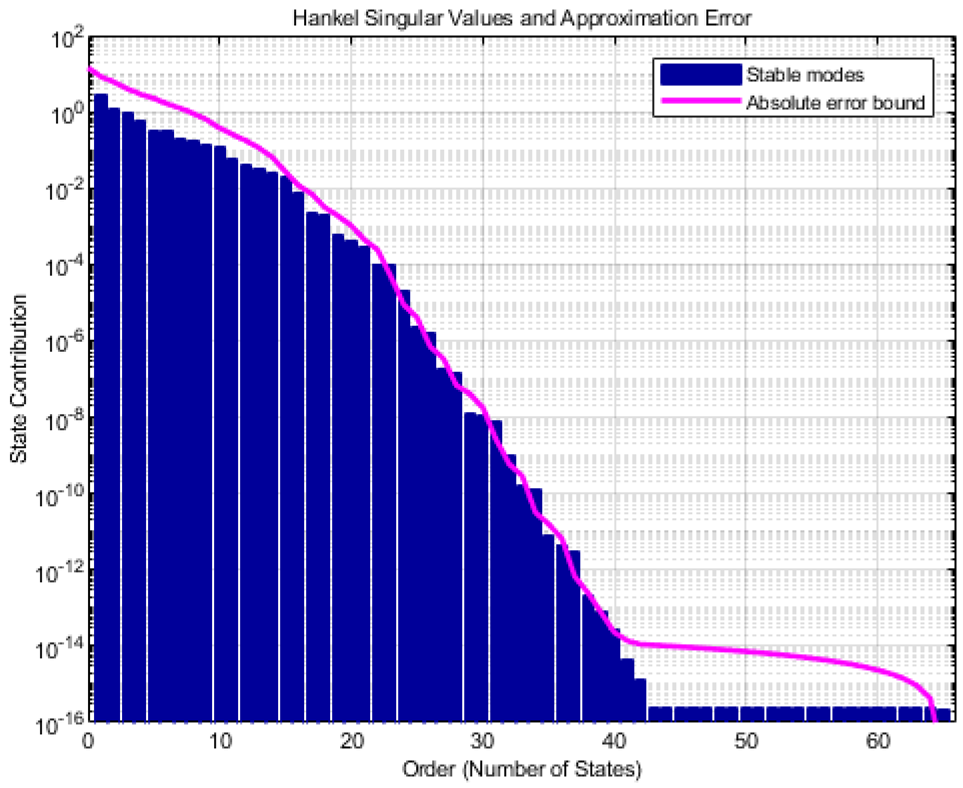
For this example, consider a random continuous-time state-space model with 65 states.

```
rng(0)
sys = rss(65);
size(sys)
```

```
State-space model with 1 outputs, 1 inputs, and 65 states.
```

Visualize the Hankel singular values on a plot.

```
balred(sys)
```



For this instance, compute reduced-order models with 25, 30 and 35 states.

```
order = [25,30,35];
rsys = balred(sys,order);
size(rsys)
```

```
3x1 array of state-space models.
Each model has 1 outputs, 1 inputs, and between 25 and 35 states.
```

Reduced-Order Approximation with Offset Option

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

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

Create the model.

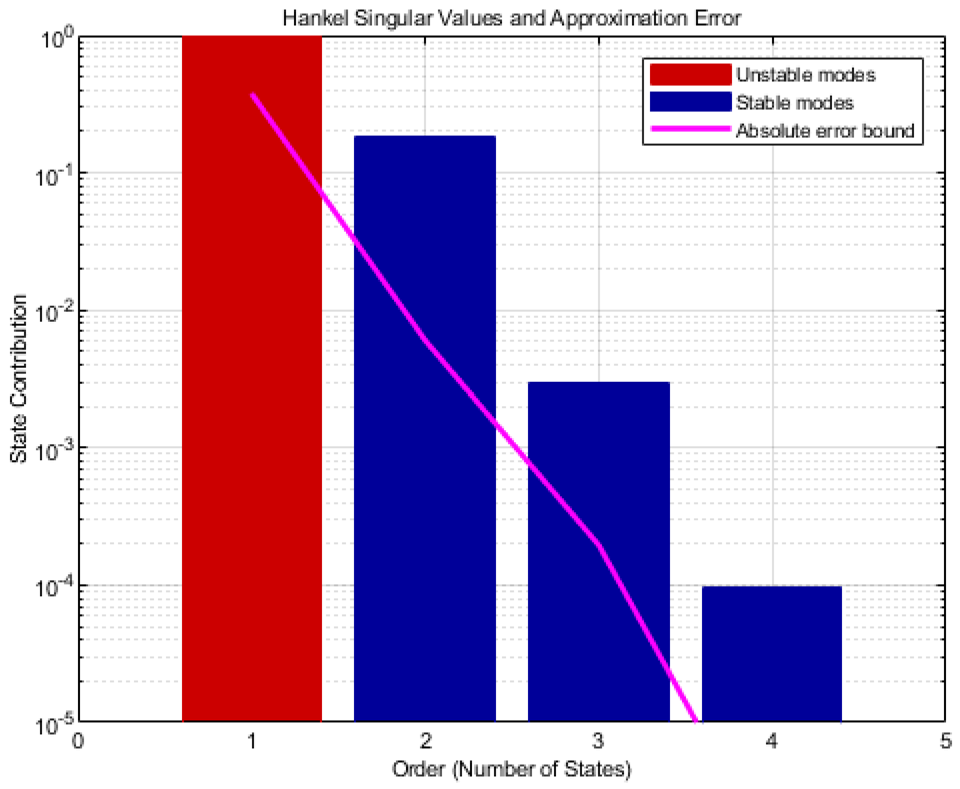
```
sys = zpk([-0.5 -1.1 -2.9],[-1e-6 -2 -1 -3],1);
```

Exclude the pole at $s = 10^{-6}$ from the stable term of the stable/unstable decomposition. To do so, set the `Offset` option of `balredOptions` to a value larger than the pole you want to exclude.

```
opts = balredOptions('Offset',0.001,'StateProjection','Truncate');
```

Visualize the Hankel singular values (HSV) and the approximation error.

```
balred(sys,opts)
```



Observe that the first HSV is red which indicates that it is associated with an unstable mode.

Now, compute a second-order approximation with the specified options.

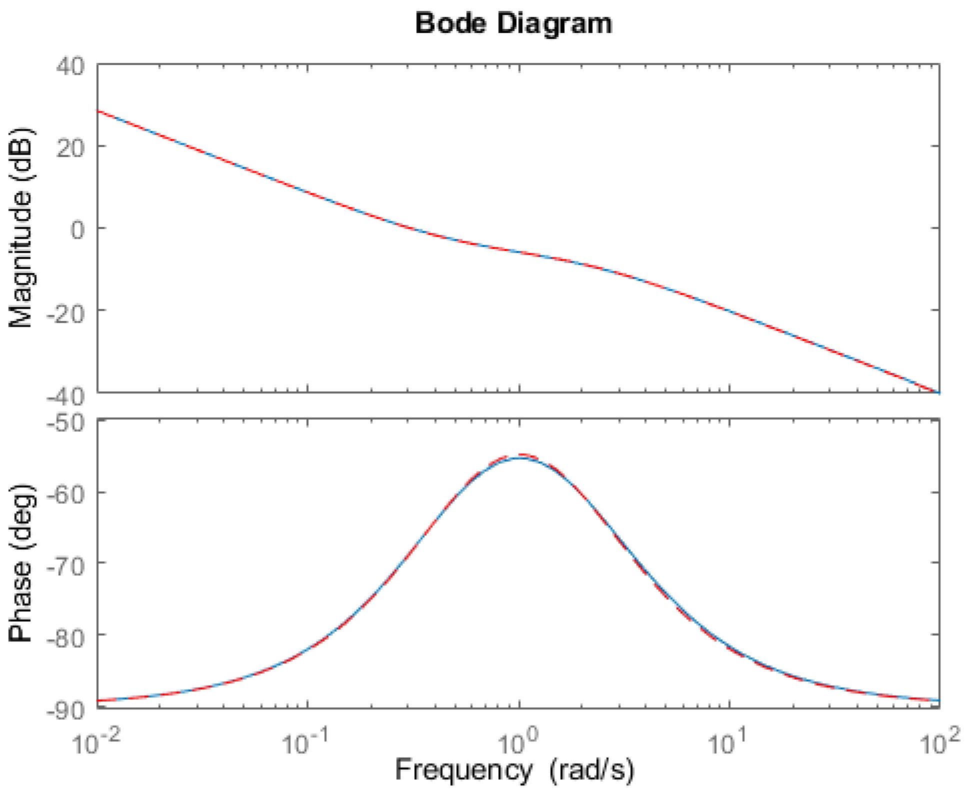
```
[rsys,info] = balred(sys,2,opts);
rsys
```

```
rsys =

  0.99113 (s+0.5235)
  -------------------
  (s+1e-06) (s+1.952)

Continuous-time zero/pole/gain model.
```

Notice that the pole at `-1e-6` appears unchanged in the reduced model `rsys`.

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

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



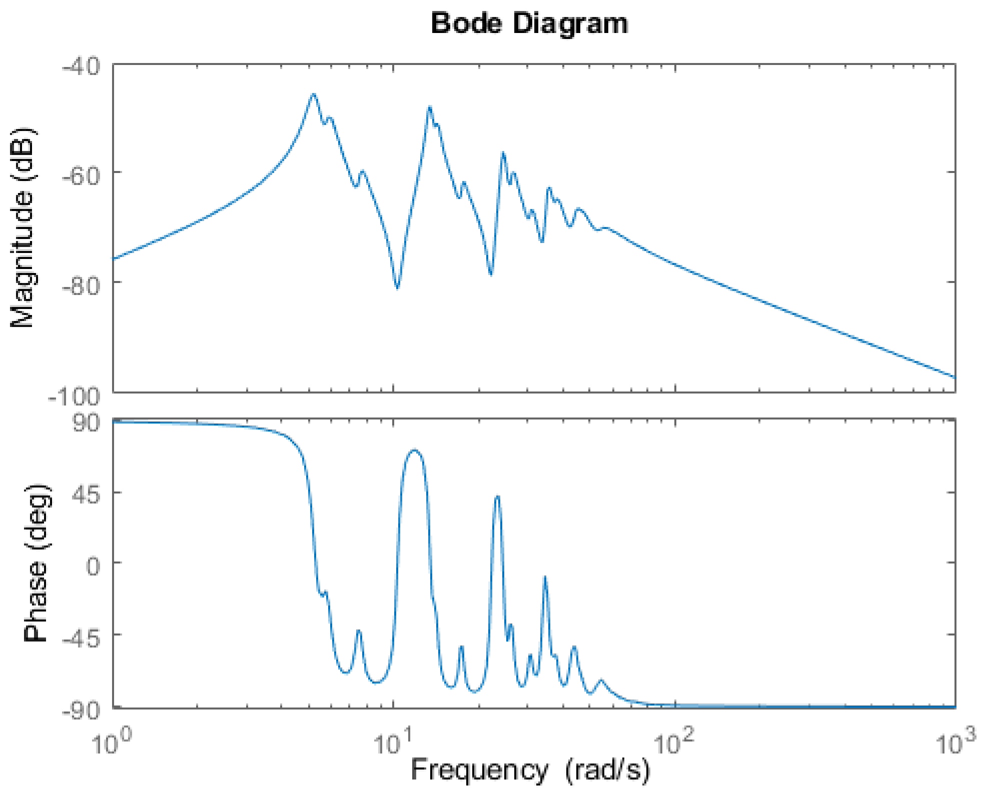
Observe that the bode response of the original model and the reduced-order model nearly match.

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('highOrderModel.mat','G')
bodeplot(G)
```

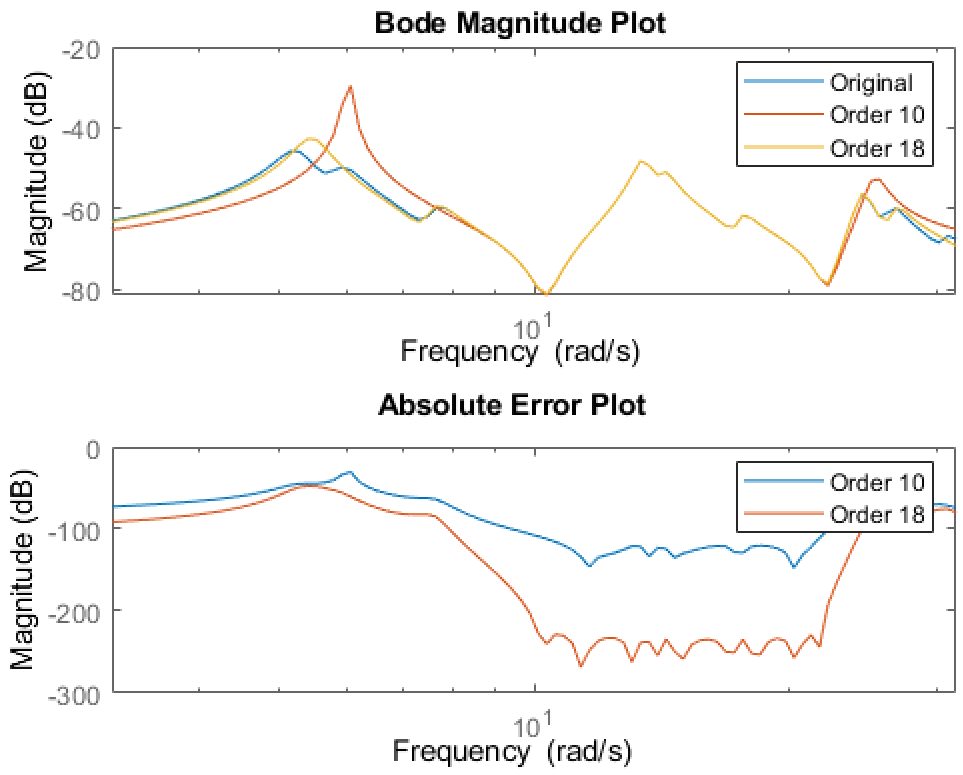


G is a 48th-order model with several large peak regions around 5.2 rad/s, 13.5 rad/s, and 24.5 rad/s, and smaller peaks scattered across many frequencies. Suppose that for your application you are only interested in the dynamics near the second large peak, between 10 rad/s and 22 rad/s. Focus the model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

```
bopt = balredOptions('StateProjection','Truncate','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.

Model-Order Reduction with Relative Error Approximation

For this example, consider the SISO state-space model `cdrom` with 120 states. You can use absolute or relative error control when approximating models with `balred`. This example compares the two approaches when applied to a 120-state model of a portable CD player device `crdom` [1,2] on page 1-0 .

Load the CD player model `cdrom`.

```
load cdromData.mat cdrom
size(cdrom)
```

```
State-space model with 1 outputs, 1 inputs, and 120 states.
```

To compare results with absolute vs. relative error control, create one option set for each approach.

```
opt_abs = balredOptions('ErrorBound','absolute','StateProjection','truncate');
opt_rel = balredOptions('ErrorBound','relative','StateProjection','truncate');
```

Compute reduced-order models of order 15 with both approaches.

```
rsys_abs = balred(cdrom,15,opt_abs);
rsys_rel = balred(cdrom,15,opt_rel);
size(rsys_abs)
```

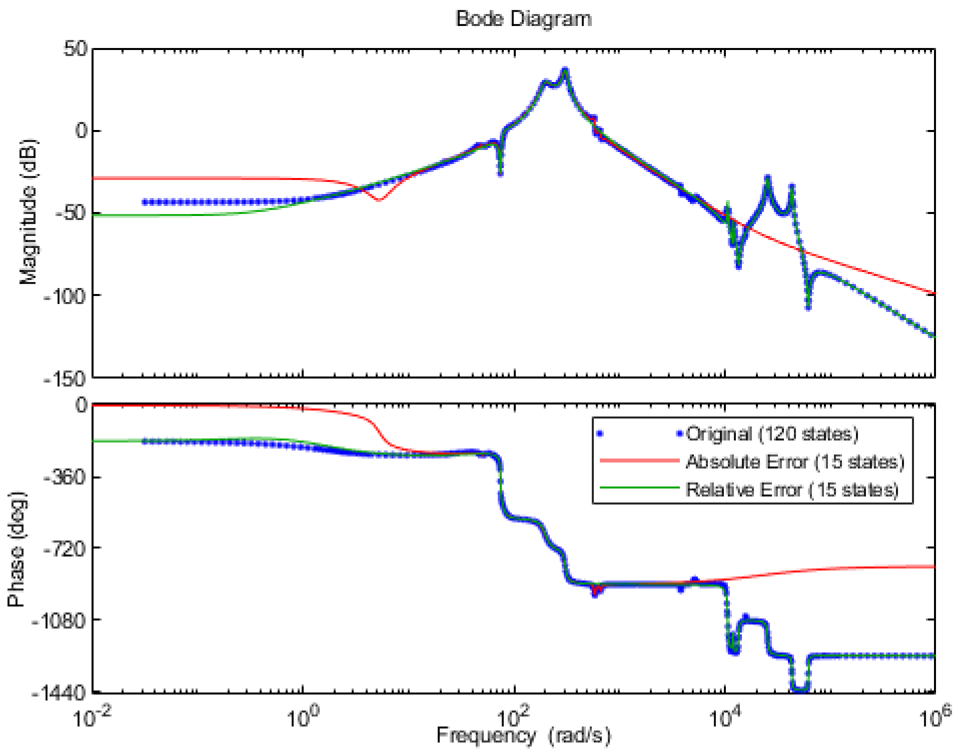
```
State-space model with 1 outputs, 1 inputs, and 15 states.
```

```
size(rsys_rel)
```

```
State-space model with 1 outputs, 1 inputs, and 15 states.
```

Plot the Bode response of the original model along with the absolute-error and relative-error reduced models.

```
bo = bodeoptions;
bo.PhaseMatching = 'on';
bodeplot(cdrom,'b.',rsys_abs,'r',rsys_rel,'g',bo)
legend('Original (120 states)','Absolute Error (15 states)','Relative Error (15 states)')
```



Observe that the Bode response of:

- The relative-error reduced model `rsys_rel` nearly matches the response of the original model `sys` across the complete frequency range.
- The absolute-error reduced model `rsys_abs` matches the response of the original model `sys` only in areas with the most gain.

References

1. Benchmark Examples for Model Reduction, Subroutine Library in Systems and Control Theory (SLICOT). The CDROM data set is reproduced with permission, see BSD3-license for details.
2. A.Varga, “On stochastic balancing related model reduction”, *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187)*, Sydney, NSW, 2000, pp. 2385-2390 vol.3, doi: 10.1109/CDC.2000.914156.

Input Arguments

`sys` — Dynamic system
dynamic system model

Dynamic system, specified as a SISO or MIMO dynamic system model (Control System Toolbox). Dynamic systems that you can use can be continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss`models.

When `sys` has unstable poles, `balred` decomposes `sys` to its stable and unstable parts and only the stable part is approximated. Use `balredOptions` to specify additional options for the stable/unstable decomposition.

`balred` does not support frequency response data models, uncertain and generalized state-space models, PID models or sparse model objects.

`order` — Desired number of states
integer | vector of integers

Desired number of states, specified as an integer or a vector of integers. You can try multiple orders at once by setting `order` to a vector of integers, in which case `rys` is returned as an array of reduced models.

You can also use the Hankel singular values and error bound information to select the reduced-model order based on the desired model fidelity.

`opts` — Additional options for model reduction
options set

Additional options for model reduction, specified as an options set. You can specify additional options for eliminating states, using absolute vs. relative error control, emphasizing certain time or frequency bands, and separating the stable and unstable modes.

See `balredOptions` to create and configure the option set `opts`.

Output Arguments

`rsys` — Reduced-order model
dynamic system model | array of dynamic system models

Reduced-order model, returned as a dynamic system model or an array of dynamic system models.

`info` — Additional information about the LTI model
structure

Additional information about the LTI model, returned as a structure with the following fields:

- `HSV` — Hankel singular values (state contributions to the input/output behavior). In state coordinates that equalize the input-to-state and state-to-output energy transfers, the Hankel singular values measure the contribution of each state to the input/output behavior. Hankel singular values are to model order what singular values are to matrix rank. In particular, small Hankel singular values signal states that can be discarded to simplify the model.
- `ErrorBound` — Bound on absolute or relative approximation error. `info.ErrorBound(J+1)` bounds the error for order J.
- `Regularization` — Regularization level □ (for relative error only). Here, `sys` is replaced by `[sys,ρ*I]` or `[sys;ρ*I]` that ensures a well-defined relative error at all frequencies.
- `Rr`, `Ro` — Cholesky factors of gramians.

Algorithms

1. `balred` first decomposes G into its stable and unstable parts:

   $$G = G_s + G_u$$
2. When you specify `ErrorBound` as `absolute`, `balred` uses the balanced truncation method of [1] to reduce G_s. This computes the Hankel singular values (HSV) σ_j based on the controllability and observability gramians. For order r, the absolute error $\|G_s - G_r\|_\infty$ is bounded by $2\sum_{j=r+1}^{n}\sigma_j$. Here, n is the number of states in G_s.
3. When you specify `ErrorBound` as `relative`, `balred` uses the balanced stochastic truncation method of [2] to reduce G_s. For square G_s, this computes the HSV σ_j of the phase matrix $F = (W')^{-1}G$ where $W(s)$ is a stable, minimum-phase spectral factor of GG':

   $$W'(s)W(s) = G(s)G'(s)$$

   For order r, the relative error $\|G_s{-1}(G_s - G_r)\|_\infty$ is bounded by:

   $$\prod_{j=r+1}^{H}\left(\frac{1+\sigma_j}{1-\sigma_j}\right) - 1 \approx 2\sum_{j=r+1}^{n}\sigma_j$$

   when, $2\sum_{j=r+1}^{n}\sigma_j \ll 1$.

Alternative Functionality

App

Model Reducer

Live Editor Task

Reduce Model Order (Control System Toolbox)

Compatibility Considerations

`MatchDC` option honored when specified frequency or time intervals exclude DC

Behavior changed in R2017b

When you use balred for model reduction, you can use balredOptions to restrict the computation to specified frequency or time intervals. If the StateProjection option of balredOptions is set to 'MatchDC' (the default value), then balred attempts to match the DC gain of the original and reduced models, even if the specified intervals exclude DC (frequency = 0 or time = Inf).

Prior to R2017b, if you specified time or frequency intervals that excluded DC, balred did not attempt to match the DC gain of the original and reduced models, even if StateProjection = 'MatchDC'.

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.

[2] Green, M., "A Relative Error Bound for Balanced Stochastic Truncation", *IEEE Transactions on Automatic Control*, Vol. 33, No. 10, 1988

See Also

Functions
balredOptions

Apps
Model Reducer

Live Editor Tasks
Reduce Model Order

Topics
“Model Reduction Basics” (Control System Toolbox)
“Balanced Truncation Model Reduction” (Control System Toolbox)

Introduced before R2006a

bandwidth

Frequency response bandwidth

Syntax

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

Description

`fb = bandwidth(sys)` returns the bandwidth of the SISO dynamic system model `sys`. The bandwidth is the first frequency where the gain drops below 70.79% (-3 dB) of its DC value. The bandwidth is expressed in `rad/TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `sys`.

This command requires a Control System Toolbox™ license.

`fb = bandwidth(sys,dbdrop)` returns the bandwidth for a specified gain drop.

Examples

Compute System Bandwidth

Compute the bandwidth of the transfer function `sys = 1/(s+1)`.

```
sys = tf(1,[1 1]);
fb = bandwidth(sys)
```

```
fb = 0.9976
```

This result shows that the gain of `sys` drops to 3 dB below its DC value at around 1 rad/s.

Find Bandwidth of System with Custom Gain Drop

Compute the frequency at which the gain of a system drops to 3.5 dB below its DC value. Create a state-space model.

```
A = [-2,-1;1,0];
B = [1;0];
C = [1,2];
D = 1;
sys = ss(A,B,C,D);
```

Find the 3.5 dB bandwidth of `sys`.

```
dbdrop = -3.5;
fb = bandwidth(sys,dbdrop)
```

```
fb = 0.8348
```

Find Bandwidth of Model Array

Find the bandwidth of each entry in a 5-by-1 array of transfer function models. Use a `for` loop to create the array, and confirm its dimensions.

```
sys = tf(zeros(1,1,5));
s = tf('s');
for m = 1:5
    sys(:,:,m) = m/(s^2+s+m);
end
size(sys)
```

```
5x1 array of transfer functions.
Each model has 1 outputs and 1 inputs.
```

Find the bandwidths.

```
fb = bandwidth(sys)
```

```
fb = 5×1

    1.2712
    1.9991
    2.5298
    2.9678
    3.3493
```

`bandwidth` returns an array in which each entry is the bandwidth of the corresponding entry in `sys`. For instance, the bandwidth of `sys(:,:,2)` is `fb(2)`.

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO dynamic system model or an array of SISO dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models such as `tf`, `zpk`, or `ss` models.
- Frequency-response data models such as `frd` models. For such models, `bandwidth` uses the first frequency point to approximate the DC gain.

If `sys` is an array of models, `bandwidth` returns an array of the same size, where each entry is the bandwidth of the corresponding model in `sys`. For more information on model arrays, see “Model Arrays” (Control System Toolbox).

`dbdrop` — Gain drop

-3 (default) | negative scalar

Gain drop in dB, specified as a real negative scalar.

Output Arguments

`fb` — Frequency response bandwidth
scalar | array

Frequency response bandwidth, returned as a scalar or an array. If `sys` is:

- A single model, then `fb` is the bandwidth of `sys`.
- A model array, then `fb` is an array of the same size as the model array `sys`. Each entry is the bandwidth of the corresponding entry in `sys`.

`fb` is expressed in `rad/TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `sys`.

See Also
`dcgain` | `issiso` | `bodeplot`

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)
[sys,ic] = bj(___)
```

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.

`[sys,ic] = bj(___)` returns the estimated initial conditions as an `initialCondition` object. Use this syntax if you plan to simulate or predict the model response using the same estimation input data and then compare the response with the same estimation output data. Incorporating the initial conditions yields a better match during the first part of the simulation.

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 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.

Report Field	Description
`InitialCondition`	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:
	Field — **Description**
	`FitPercent` — Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
	`LossFcn` — Value of the loss function when the estimation completes.
	`MSE` — Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
	`FPE` — Final prediction error for the model.
	`AIC` — Raw Akaike Information Criteria (AIC) measure of model quality.
	`AICc` — Small-sample-size corrected AIC.
	`nAIC` — Normalized AIC.
	`BIC` — Bayesian Information Criteria (BIC).
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `bjOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
DataUsed	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** — **Description** Name — Name of the data set. Type — Data type. Length — Number of data samples. Ts — Sample time. InterSample — Input intersample behavior, returned as one of the following values: • 'zoh' — 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. InputOffset — Offset removed from time-domain input data during estimation. For nonlinear models, it is []. OutputOffset — Offset removed from time-domain output data during estimation. For nonlinear models, it is [].
Termination	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** — **Description** WhyStop — Reason for terminating the numerical search. Iterations — Number of search iterations performed by the estimation algorithm. FirstOrderOptimality — ∞-norm of the gradient search vector when the search algorithm terminates. FcnCount — Number of times the objective function was called. UpdateNorm — Norm of the gradient search vector in the last iteration. Omitted when the search method is 'lsqnonlin' or 'fmincon'. LastImprovement — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is 'lsqnonlin' or 'fmincon'. Algorithm — Algorithm used by 'lsqnonlin' or 'fmincon' search method. Omitted when other search methods are used. For estimation methods that do not require numerical search optimization, the Termination field is omitted.

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

`[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.

`ic`

Estimated initial conditions, returned as an `initialCondition` object or an object array of `initialCondition` values.

- For a single-experiment data set, `ic` represents, in state-space form, the free response of the transfer function model (*A* and *C* matrices) to the estimated initial states (x_0).
- For a multiple-experiment data set with N_e experiments, `ic` is an object array of length N_e that contains one set of `initialCondition` values for each experiment.

If `bj` returns `ic` values of `0` and the you know that you have non-zero initial conditions, set the `'InitialCondition'` option in `bjOptions` to `'estimate'` and pass the updated option set to `bj`. For example:

```
opt = bjOptions('InitialCondition','estimate')
[sys,ic] = bj(data,[nb nc nd nf nk],opt)
```

The default `'auto'` setting of `'InitialCondition'` uses the `'zero'` method when the initial conditions have a negligible effect on the overall estimation-error minimization process. Specifying `'estimate'` ensures that the software estimates values for `ic`.

For more information, see `initialCondition`. For an example of using this argument, see “Obtain Initial Conditions” on page 1-109.

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.

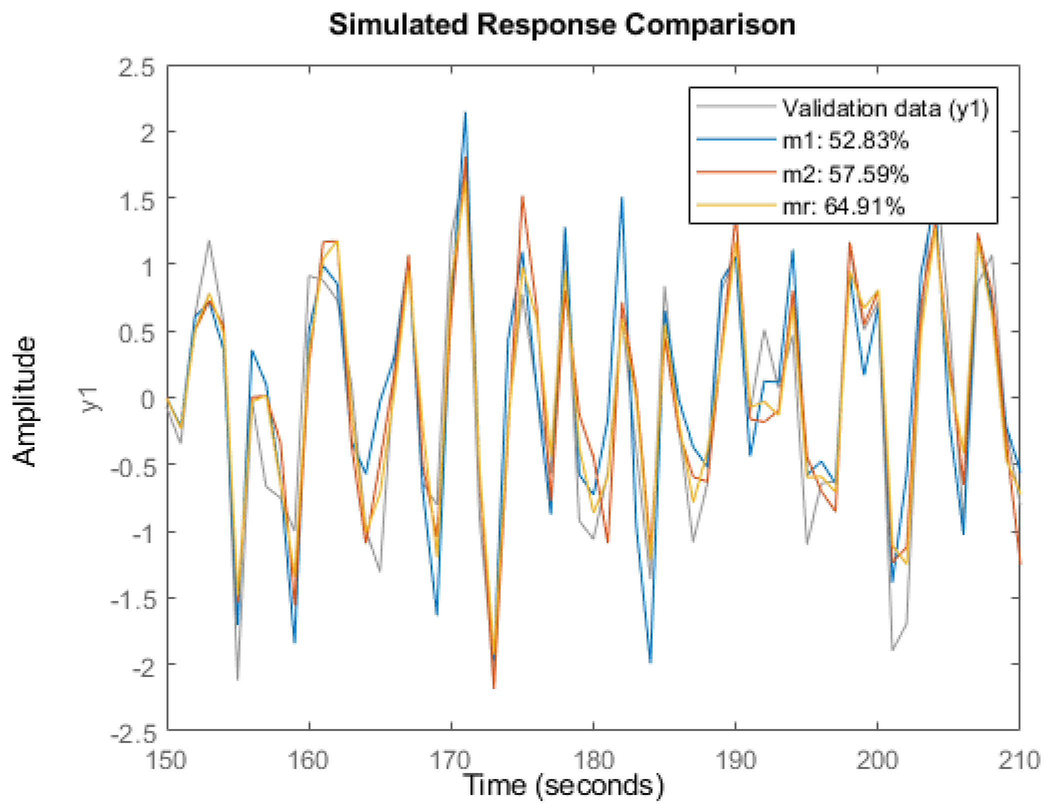
```
opt = bjOptions;
opt.Regularization.Lambda = 1;
m2 = bj(m0simdata(1:150),[15 15 15 15 1],opt);
```

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.SearchOptions.MaxIterations = 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.

Obtain Initial Conditions

Load the data.

```
load iddata1ic z1i
```

Estimate a second-order Box-Jenkins model `sys` and return the initial conditions in `ic`.

```
nb = 2;
nc = 2;
nd = 2;
nf = 2;
nk = 1;
[sys,ic] = bj(z1i,[nb nc nd nf nk]);
ic
```

```
ic = 
  initialCondition with properties:

     A: [4x4 double]
    X0: [4x1 double]
     C: [0.8744 0.5426 0.4647 -0.5285]
    Ts: 0.1000
```

`ic` is an `initialCondition` object that encapsulates the free response of `sys`, in state-space form, to the initial state vector in `X0`. You can incorporate `ic` when you simulate `sys` with the `z1i` input signal and compare the response with the `z1i` output signal.

More About

Box-Jenkins Model Structure

The general Box-Jenkins model structure is:

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

where *nu* is the number of input channels.

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

$$\begin{aligned} nb&: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1} \\ nc&: \quad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc} \\ nd&: \quad D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd} \\ nf&: \quad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf} \end{aligned}$$

Alternatives

To estimate a continuous-time model, use:

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

References

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

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `bjOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = bjOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

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

Topics
“Regularized Estimates of Model Parameters”
“Apply Initial Conditions when Simulating Identified Linear Models”

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 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`

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

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

If `EstimateCovariance` 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` — Numerical search method used for iterative parameter estimation
`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOptions` Structure When `SearchMethod` is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Tolerance`	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`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

SearchOptions Structure When SearchMethod is Specified as 'lsqnonlin'

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of FunctionTolerance is the same as that of opt.SearchOptions.Advanced.TolFun.	1e-5
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of StepTolerance is the same as that of opt.SearchOptions.Advanced.TolX.	1e-6
MaxIterations	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance. The value of MaxIterations is the same as that of opt.SearchOptions.Advanced.MaxIter.	20
Advanced	Advanced search settings, specified as an option set for lsqnonlin. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use optimset('lsqnonlin') to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	`1e-6`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`.	`100`

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

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

  - 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 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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`

Topics

“Loss Function and Model Quality Metrics”

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

Perform Block-Diagonal Concatenation

Perform block-diagonal concatenation of a transfer function model and a state-space model.

Create the SISO continuous-time transfer function model, `1/s`.

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

Create a SISO continuous-time state-space model with state-space matrices 1,2,3, and 4.

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

Concatenate `sys1`, a SISO static gain system, and `sys2`. The resulting model is a 3-input, 3-output state-space model.

```
sys = blkdiag(sys1,10,sys2)
```

```
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.
```

Alternatively, use the `append` command.

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

See Also
`append` | `series` | `parallel` | `feedback`

Introduced in R2009a

bode

Bode plot of frequency response, or magnitude and phase data

Syntax

```
bode(sys)
bode(sys1,sys2,...,sysN)
bode(sys1,LineSpec1,...,sysN,LineSpecN)
bode( ___ ,w)

[mag,phase,wout] = bode(sys)
[mag,phase,wout] = bode(sys,w)
[mag,phase,wout,sdmag,sdphase] = bode(sys,w)
```

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. `bode` automatically determines frequencies to plot based on system dynamics.

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

If `sys` is a model with complex coefficients, then in:

- Log frequency scale, the plot shows two branches, one for positive frequencies and one for negative frequencies. The plot also shows arrows to indicate the direction of increasing frequency values for each branch. See “Bode Plot of Model with Complex Coefficients” on page 1-135.
- Linear frequency scale, the plot shows a single branch with a symmetric frequency range centered at a frequency value of zero.

`bode(sys1,sys2,...,sysN)` plots the frequency response of multiple dynamic systems on the same plot. All systems must have the same number of inputs and outputs.

`bode(sys1,LineSpec1,...,sysN,LineSpecN)` specifies a color, line style, and marker for each system in the plot.

`bode( ___ ,w)` plots system responses for frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `bode` plots the response at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `bode` plots the response at each specified frequency. The vector `w` can contain both negative and positive frequencies.

You can use `w` with any of the input-argument combinations in previous syntaxes.

`[mag,phase,wout] = bode(sys)` returns the magnitude and phase of the response at each frequency in the vector `wout`. The function automatically determines frequencies in `wout` based on system dynamics. This syntax does not draw a plot.

`[mag,phase,wout] = bode(sys,w)` returns the response data at the frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `wout` contains frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `wout` = `w`.

`[mag,phase,wout,sdmag,sdphase] = bode(sys,w)` also returns the estimated standard deviation of the magnitude and phase values for the identified model `sys`. If you omit `w`, then the function automatically determines frequencies in `wout` based on system dynamics.

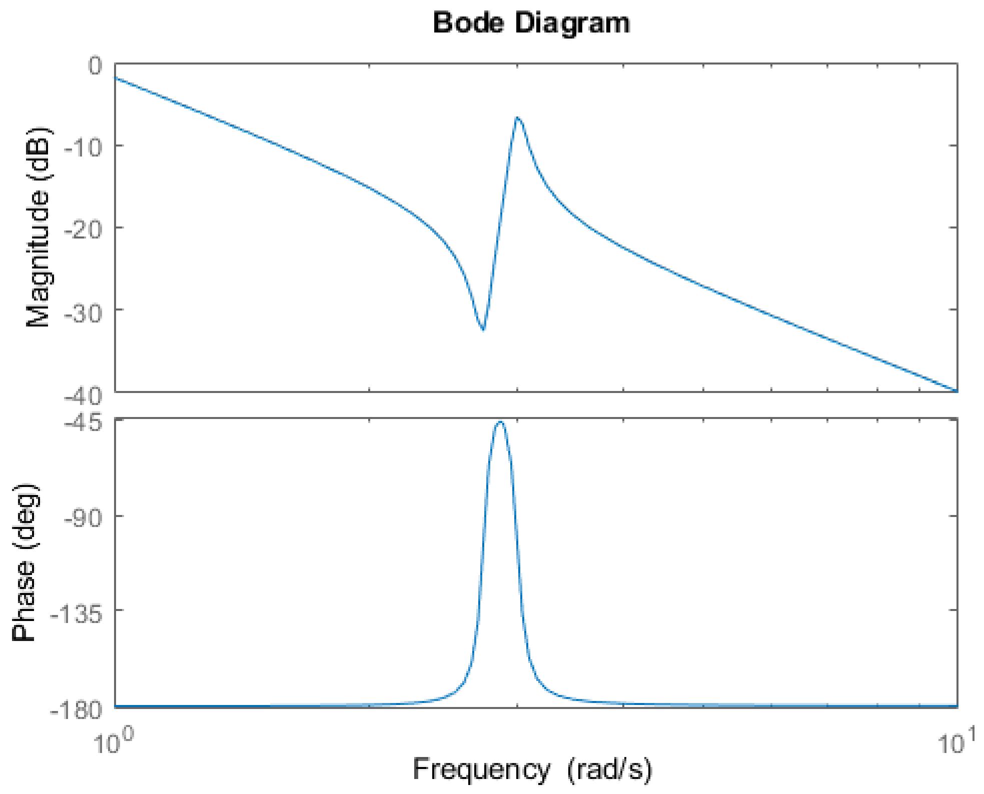
Examples

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)
```

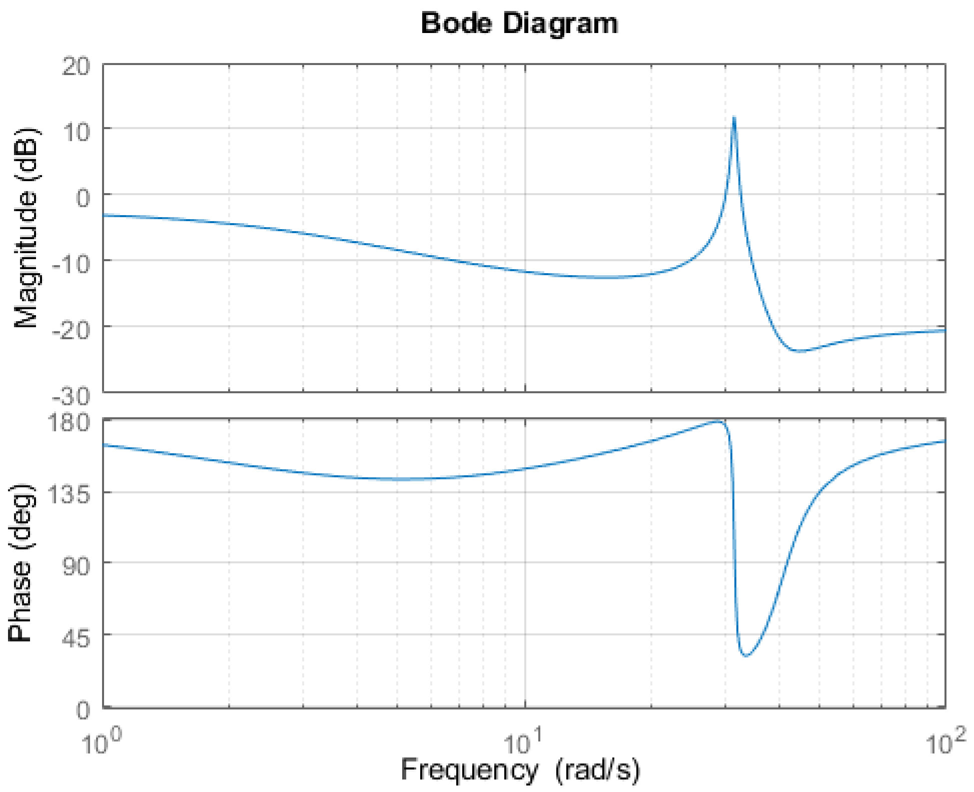


`bode` automatically selects the plot range based on the system dynamics.

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.

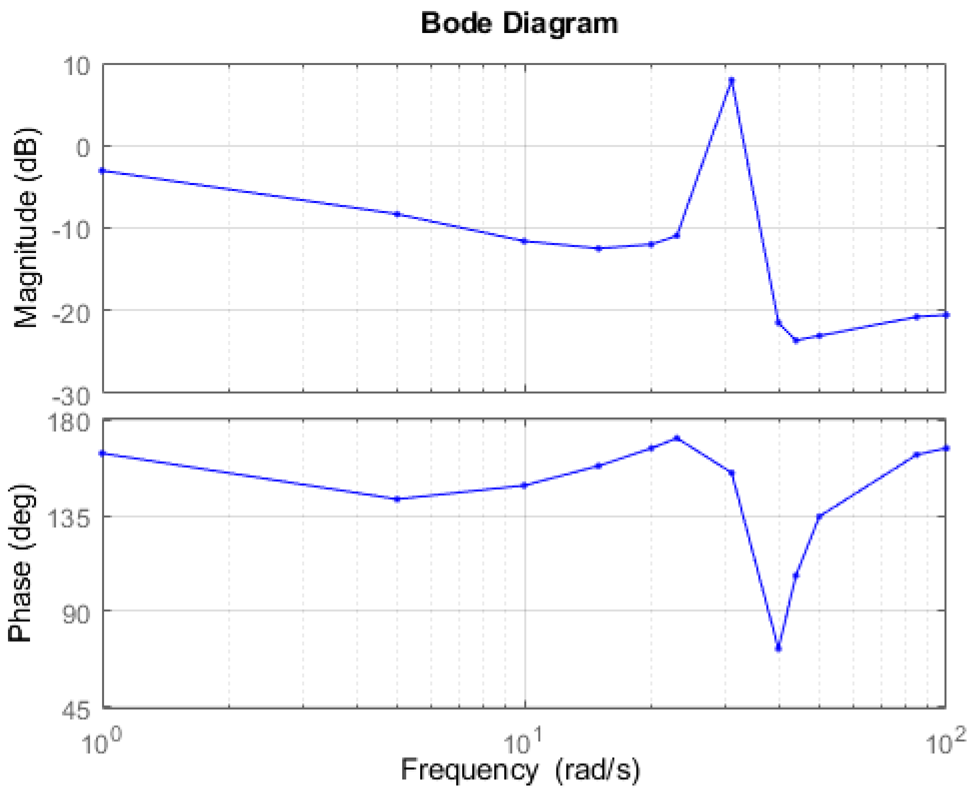
```
H = tf([-0.1,-2.4,-181,-1950],[1,3.3,990,2600]);
bode(H,{1,100})
grid on
```



The cell array `{1,100}` specifies the minimum and maximum frequency values in the Bode plot. When you provide frequency bounds in this way, the function selects intermediate points for frequency response data.

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

```
w = [1 5 10 15 20 23 31 40 44 50 85 100];
bode(H,w,'.-')
grid on
```



bode plots the frequency response at the specified frequencies only.

Compare Bode Plots of Several Dynamic Systems

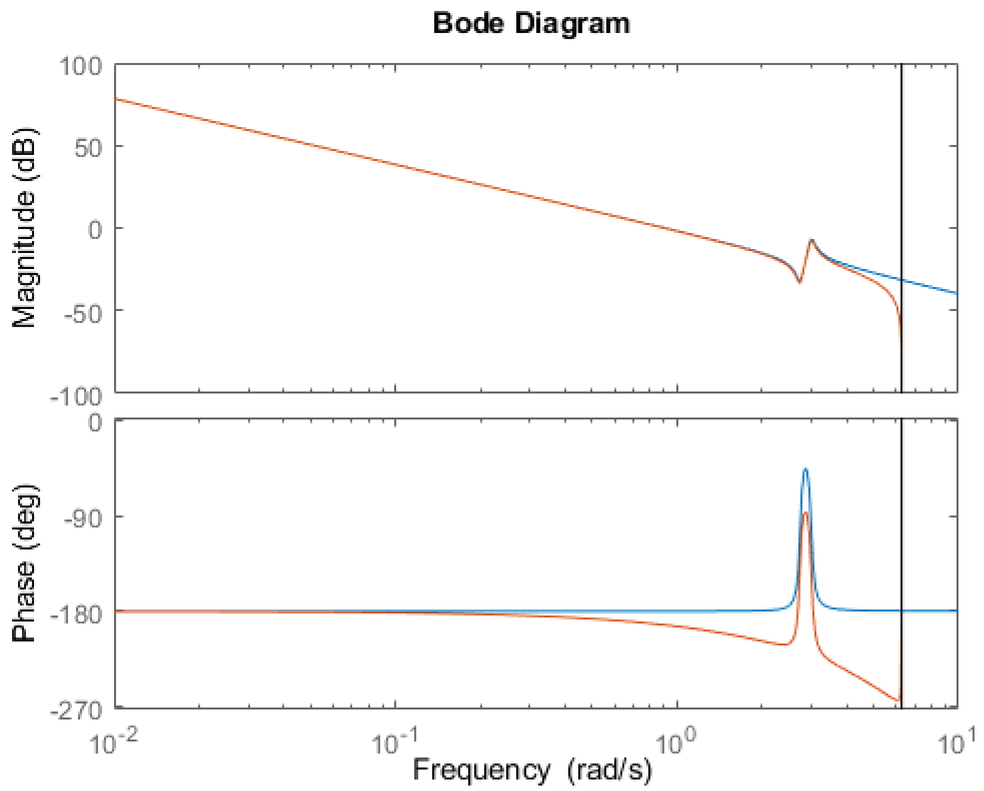
Compare the frequency response of a continuous-time system to an equivalent discretized system on the same Bode plot.

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)
```

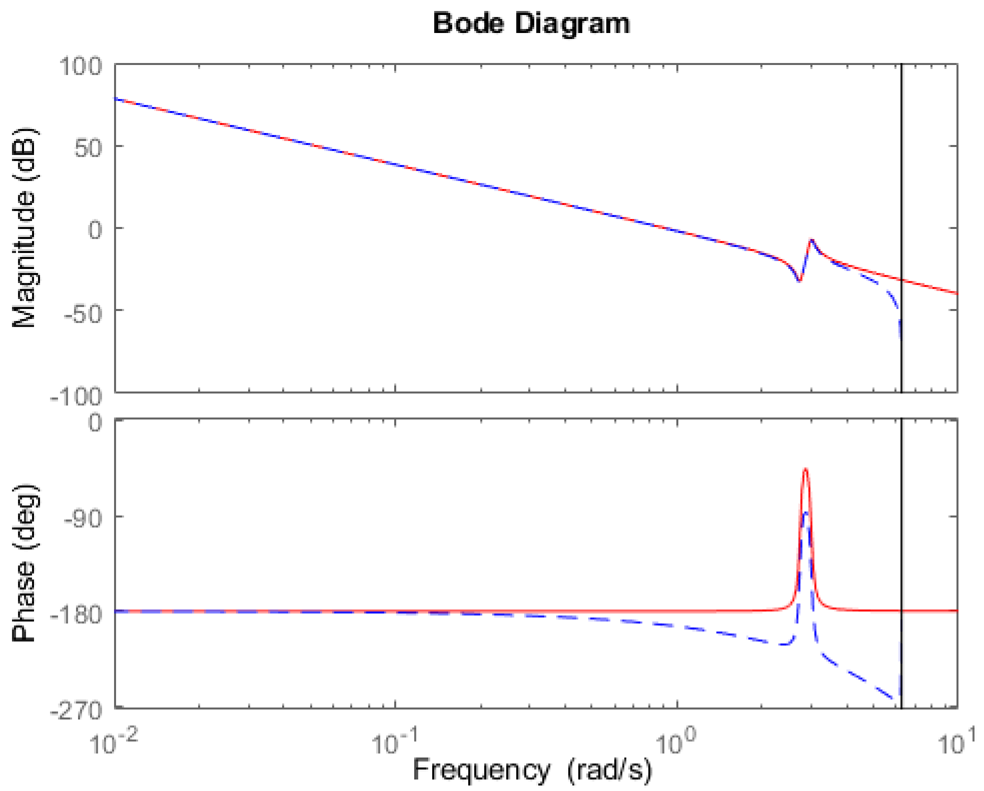


The Bode plot of a discrete-time system includes a vertical line marking the Nyquist frequency of the system.

Bode Plot with Specified Line Attributes

Specify the line style, color, or marker for each system in a Bode plot using the `LineSpec` input argument.

```
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 `LineSpec`, `'r'`, specifies a solid red line for the response of `H`. The second `LineSpec`, `'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 SISO system.

If you do not specify frequencies, `bode` chooses frequencies based on the system dynamics and returns them in the third output argument.

```
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`.

```
size(mag)
```

```
ans = 1×3

     1     1    41
```

```
length(wout)
```

```
ans = 41
```

Thus, each entry along the third dimension of mag gives the magnitude of the response at the corresponding frequency in wout.

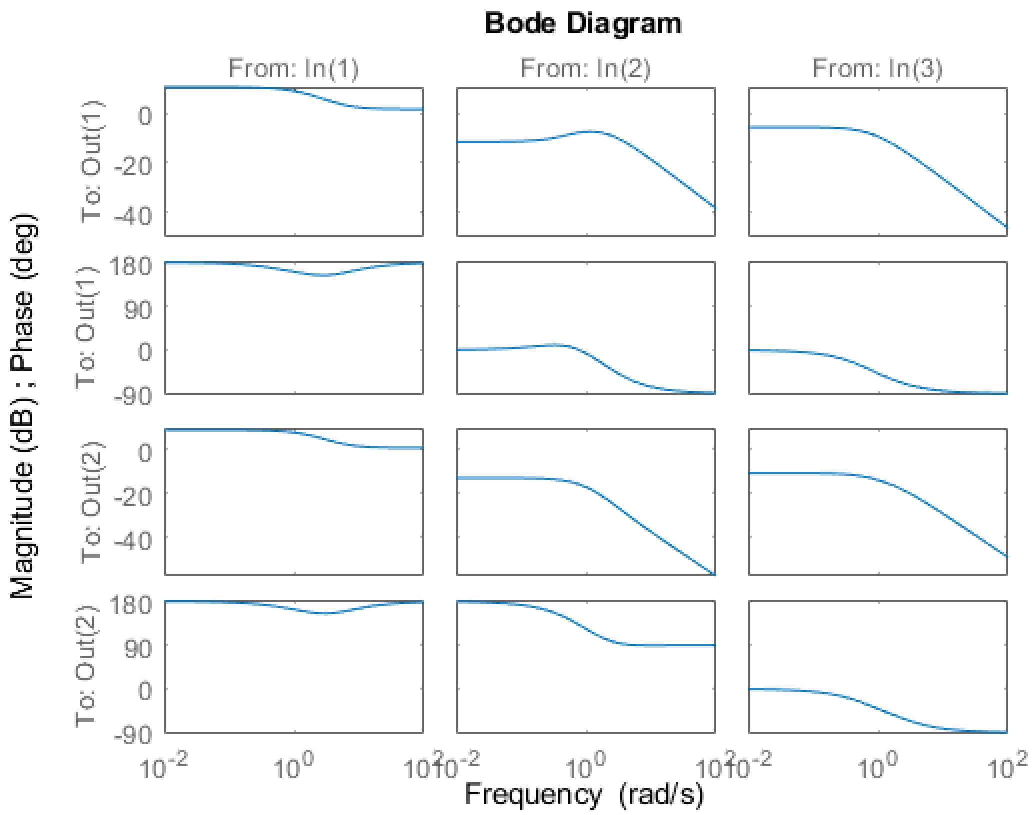
Magnitude and Phase of MIMO System

For this example, create a 2-output, 3-input system.

```
rng(0,'twister'); % For reproducibility
H = rss(4,2,3);
```

For this system, bode plots the frequency responses of each I/O channel in a separate plot in a single figure.

```
bode(H)
```



Compute the magnitude and phase of these responses at 20 frequencies between 1 and 10 radians.

```
w = logspace(0,1,20);
[mag,phase] = bode(H,w);
```

mag and phase are three-dimensional arrays, in which the first two dimensions correspond to the output and input dimensions of H, and the third dimension is the number of frequencies. For instance, examine the dimensions of mag.

```
size(mag)
```

```
ans = 1×3

     2     3    20
```

Thus, for example, `mag(1,3,10)` is the magnitude of the response from the third input to the first output, computed at the 10th frequency in `w`. Similarly, `phase(1,3,10)` contains the phase of the same response.

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 nonparametric models based on data.

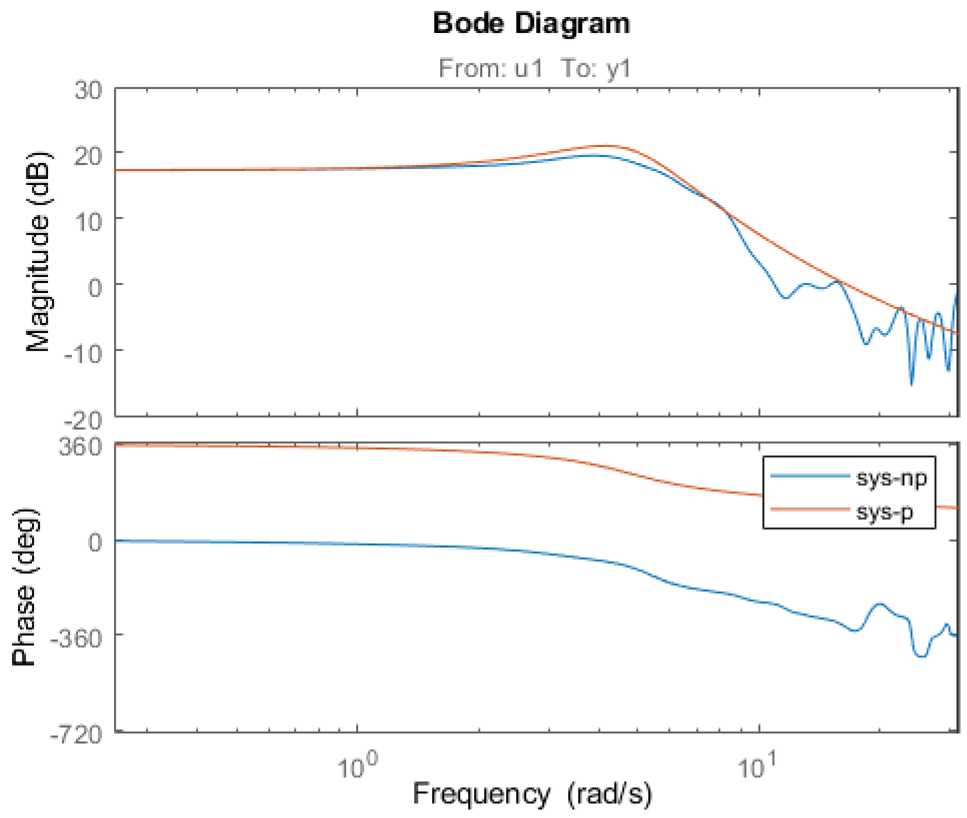
```
load iddata2 z2;
w = linspace(0,10*pi,128);
sys_np = spa(z2,[],w);
sys_p = tfest(z2,2);
```

Using the `spa` and `tfest` commands requires 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.

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



You can display the confidence region on the Bode plot by right-clicking the plot and selecting **Characteristics > Confidence Region**.

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.

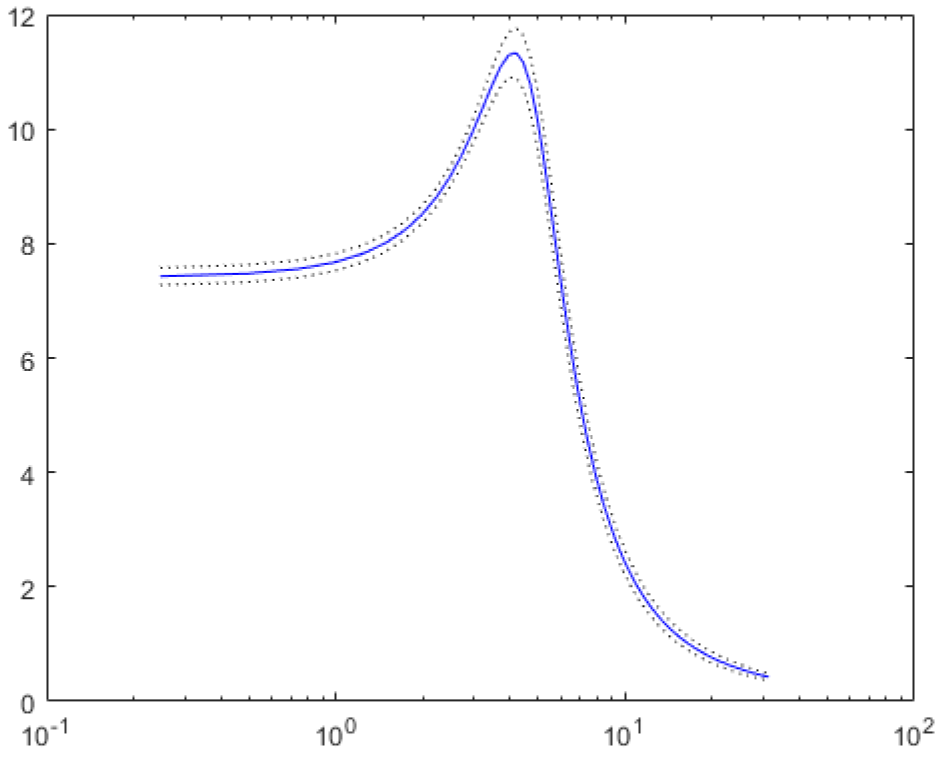
```
load iddata2 z2;
sys_p = tfest(z2,2);
w = linspace(0,10*pi,128);
[mag,ph,w,sdmag,sdphase] = bode(sys_p,w);
```

Using the `tfest` command requires System Identification Toolbox™ software.

`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 3σ plot corresponding to the confidence region.

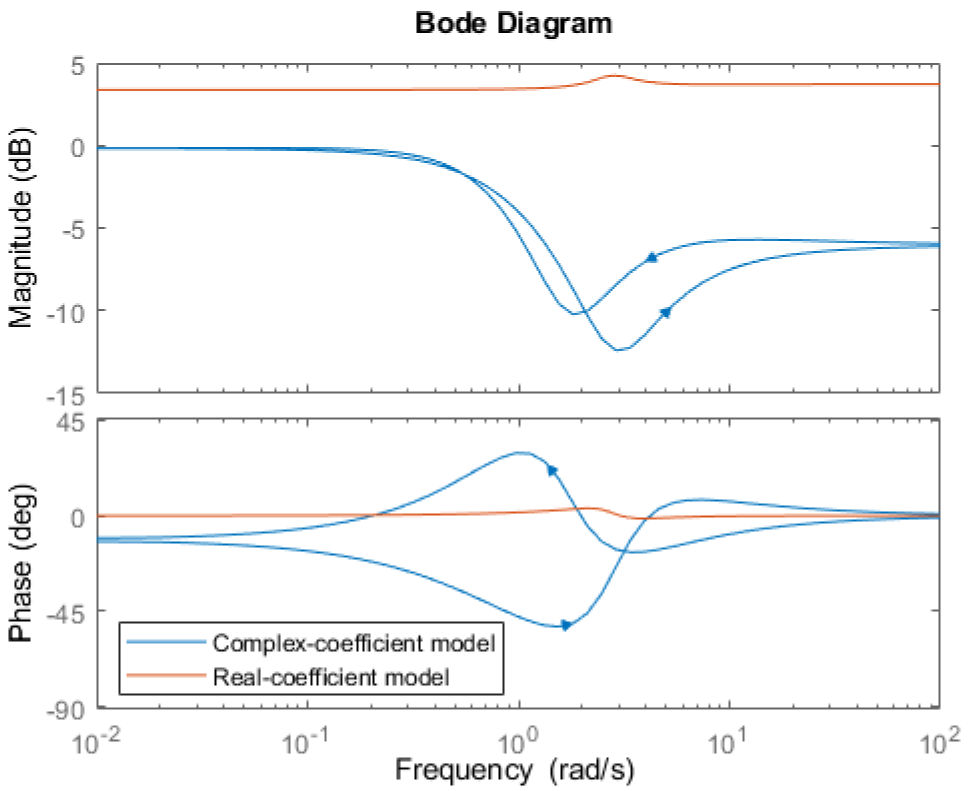
```
mag = squeeze(mag);
sdmag = squeeze(sdmag);
semilogx(w,mag,'b',w,mag+3*sdmag,'k:',w,mag-3*sdmag,'k:');
```



Bode Plot of Model with Complex Coefficients

Create a Bode plot of a model with complex coefficients and a model with real coefficients on the same plot.

```
rng(0)
A = [-3.50,-1.25-0.25i;2,0];
B = [1;0];
C = [-0.75-0.5i,0.625-0.125i];
D = 0.5;
Gc = ss(A,B,C,D);
Gr = rss(5);
bode(Gc,Gr)
legend('Complex-coefficient model','Real-coefficient model','Location','southwest')
```



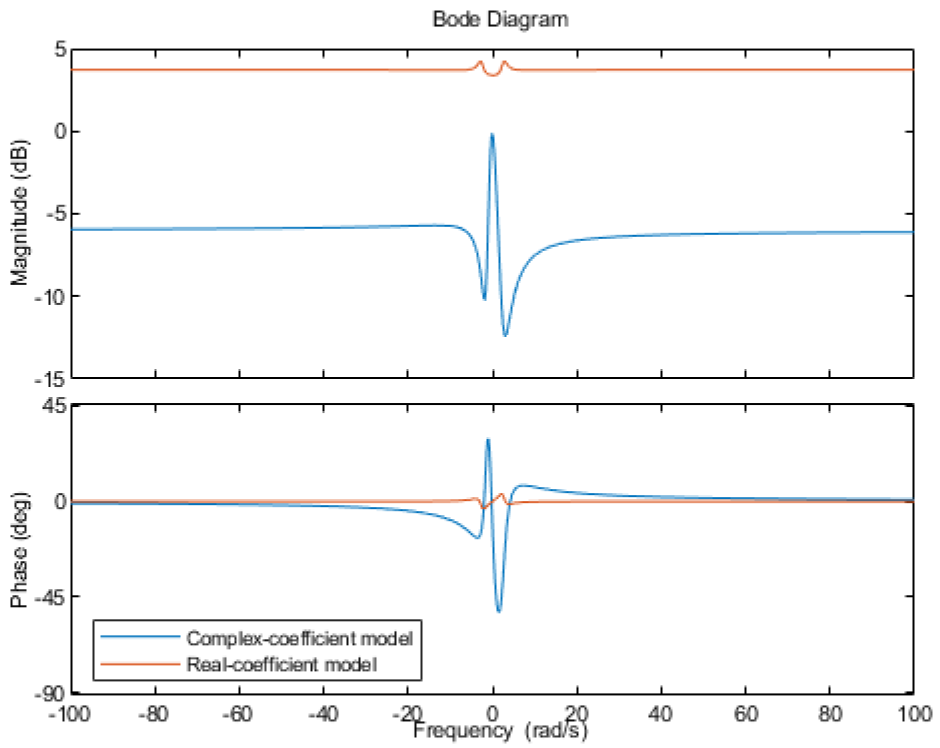
In log frequency scale, the plot shows two branches for complex-coefficient models, one for positive frequencies, with a right-pointing arrow, and one for negative frequencies, with a left-pointing arrow. In both branches, the arrows indicate the direction of increasing frequencies. The plots for real-coefficient models always contain a single branch with no arrows.

You can change the frequency scale of the Bode plot by right-clicking the plot and selecting **Properties**. In the Property Editor dialog, on the **Units** tab, set the frequency scale to `linear scale`. Alternatively, you can use the `bodeplot` function with a `bodeoptions` object to create a customized plot.

```
opt = bodeoptions;
opt.FreqScale = 'Linear';
```

Create the plot with customized options.

```
bodeplot(Gc,Gr,opt)
legend('Complex-coefficient model','Real-coefficient model','Location','southwest')
```



In linear frequency scale, the plot shows a single branch with a symmetric frequency range centered at a frequency value of zero. The plot also shows the negative-frequency response of a real-coefficient model when you plot the response along with a complex-coefficient model.

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox™ software.)
  - For tunable control design blocks, the function evaluates the model at its current value for both plotting and returning frequency response data.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model. When you use output arguments, the function returns frequency response data for the nominal model only.
- Frequency-response data models such as `frd` models. For such models, the function plots the response at frequencies defined in the model.

- Identified LTI models, such as `idtf`, `idss`, or `idproc` models. For such models, the function can also plot confidence intervals and return standard deviations of the frequency response. See "Bode Plot of Identified Model" on page 1-133.

If `sys` is an array of models, the function plots the frequency responses of all models in the array on the same axes.

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a string or vector of one, two, or three characters. The characters can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line. For more information about configuring this argument, see the `LineSpec` input argument of the `plot` function.

Example: `'r--'` specifies a red dashed line

Example: `'*b'` specifies blue asterisk markers

Example: `'y'` specifies a yellow line

`w` — Frequencies
`{wmin,wmax}` | vector

Frequencies at which to compute and plot frequency response, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function computes the response at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function computes the response at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values. The vector `w` can contain both positive and negative frequencies.

For models with complex coefficients, if you specify a frequency range of $[w_{\mathrm{min}},w_{\mathrm{max}}]$ for your plot, then in:

- Log frequency scale, the plot frequency limits are set to $[w_{\mathrm{min}},w_{\mathrm{max}}]$ and the plot shows two branches, one for positive frequencies $[w_{\mathrm{min}},w_{\mathrm{max}}]$ and one for negative frequencies $[-w_{\mathrm{max}},-w_{\mathrm{min}}]$.
- Linear frequency scale, the plot frequency limits are set to $[-w_{\mathrm{max}},w_{\mathrm{max}}]$ and the plot shows a single branch with a symmetric frequency range centered at a frequency value of zero.

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

Output Arguments

`mag` — Magnitude of system response
3-D array

Magnitude of the system response in absolute units, returned as a 3-D array. The dimensions of this array are (number of system outputs) × (number of system inputs) × (number of frequency points).

- For SISO systems, `mag(1,1,k)` gives the magnitude of the response at the `k`th frequency in `w` or `wout`. For an example, see "Obtain Magnitude and Phase Data" on page 1-131.

- For MIMO systems, `mag(i,j,k)` gives the magnitude of the response at the `k`th frequency from the `j`th input to the `i`th output. For an example, see “Magnitude and Phase of MIMO System” on page 1-132.

To convert the magnitude from absolute units to decibels, use:

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

phase — Phase of system response
3-D array

Phase of the system response in degrees, returned as a 3-D array. The dimensions of this array are (number of outputs) × (number of inputs) × (number of frequency points).

- For SISO systems, `phase(1,1,k)` gives the phase of the response at the `k`th frequency in `w` or `wout`. For an example, see “Obtain Magnitude and Phase Data” on page 1-131.
- For MIMO systems, `phase(i,j,k)` gives the phase of the response at the `k`th frequency from the `j`th input to the `i`th output. For an example, see “Magnitude and Phase of MIMO System” on page 1-132.

`wout` — Frequencies
vector

Frequencies at which the function returns the system response, returned as a column vector. The function chooses the frequency values based on the model dynamics, unless you specify frequencies using the input argument `w`.

`wout` also contains negative frequency values for models with complex coefficients.

Frequency values are in radians/`TimeUnit`, where `TimeUnit` is the value of the `TimeUnit` property of `sys`.

sdmag — Standard deviation of magnitude
3-D array | [ ]

Estimated standard deviation of the magnitude of the response at each frequency point, returned as a 3-D array. `sdmag` has the same dimensions as `mag`.

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

sdphase — Standard deviation of phase
3-D array | [ ]

Estimated standard deviation of the phase of the response at each frequency point, returned as a 3-D array. `sdphase` has the same dimensions as `phase`.

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

Tips

- When you need additional plot customization options, use `bodeplot` instead.

Algorithms

`bode` computes the frequency response as follows:

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

   - For continuous-time systems, `bode` evaluates the frequency response on the imaginary axis $s = j\omega$ and considers only positive frequencies.
   - 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:

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

where T_s is the sample time and ω_N is the Nyquist frequency. The equivalent continuous-time frequency ω is then used as the x-axis variable. Because $H\left(e^{j\omega T_s}\right)$ is periodic with period $2\omega_N$, `bode` plots the response only up to the Nyquist frequency ω_N. If `sys` is a discrete-time model with unspecified sample time, `bode` uses $T_s = 1$.

See Also

`bodeplot` | `freqresp` | `nyquist` | `spectrum` | `step`

Topics

"Plot Bode and Nyquist Plots at the Command Line"
"Dynamic System Models"

Introduced before R2006a

bodemag

Magnitude-only Bode plot of frequency response

Syntax

```
bodemag(sys)
bodemag(sys1,sys2,...,sysN)
bodemag(sys1,LineSpec1,...,sysN,LineSpecN)
bodemag( ___ ,w)
```

Description

`bodemag` enables you to generate magnitude-only plots to visualize the magnitude frequency response of a dynamic system.

For a more comprehensive function, see `bode`. `bode` provides magnitude and phase information. If you have System Identification toolbox, `bode` also returns the computed values, including statistical estimates.

For more customizable plotting options, see `bodeplot`.

`bodemag(sys)` creates a Bode magnitude plot of the frequency response of the dynamic system model `sys`. The plot displays the magnitude (in dB) of the system response as a function of frequency. `bodemag` automatically determines frequencies to plot based on system dynamics.

If `sys` is a multi-input, multi-output (MIMO) model, then `bodemag` produces an array of Bode magnitude plots in which each plot shows the frequency response of one I/O pair.

`bodemag(sys1,sys2,...,sysN)` plots the frequency response of multiple dynamic systems on the same plot. All systems must have the same number of inputs and outputs.

`bodemag(sys1,LineSpec1,...,sysN,LineSpecN)` specifies a color, line style, and marker for each system in the plot.

`bodemag( ___ ,w)` plots system responses for frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `bodemag` plots the response at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `bodemag` plots the response at each specified frequency.

You can use this syntax with any of the input-argument combinations in previous syntaxes.

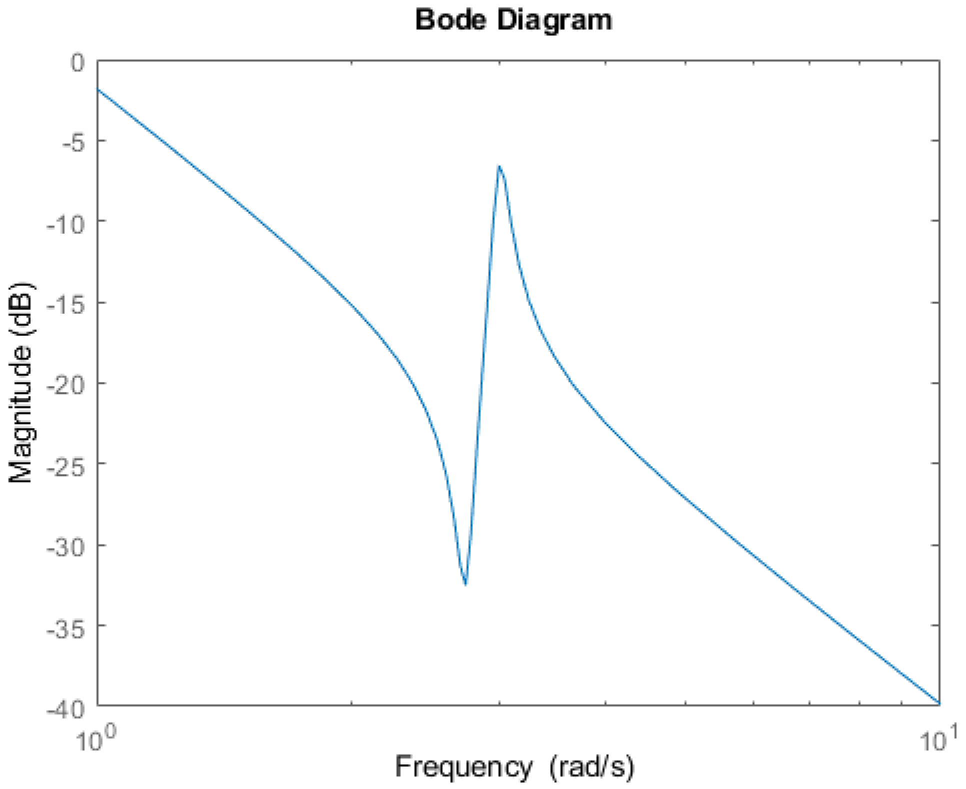
Examples

Bode Magnitude Plot of Dynamic System

Create a Bode magnitude 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]);
bodemag(H)
```

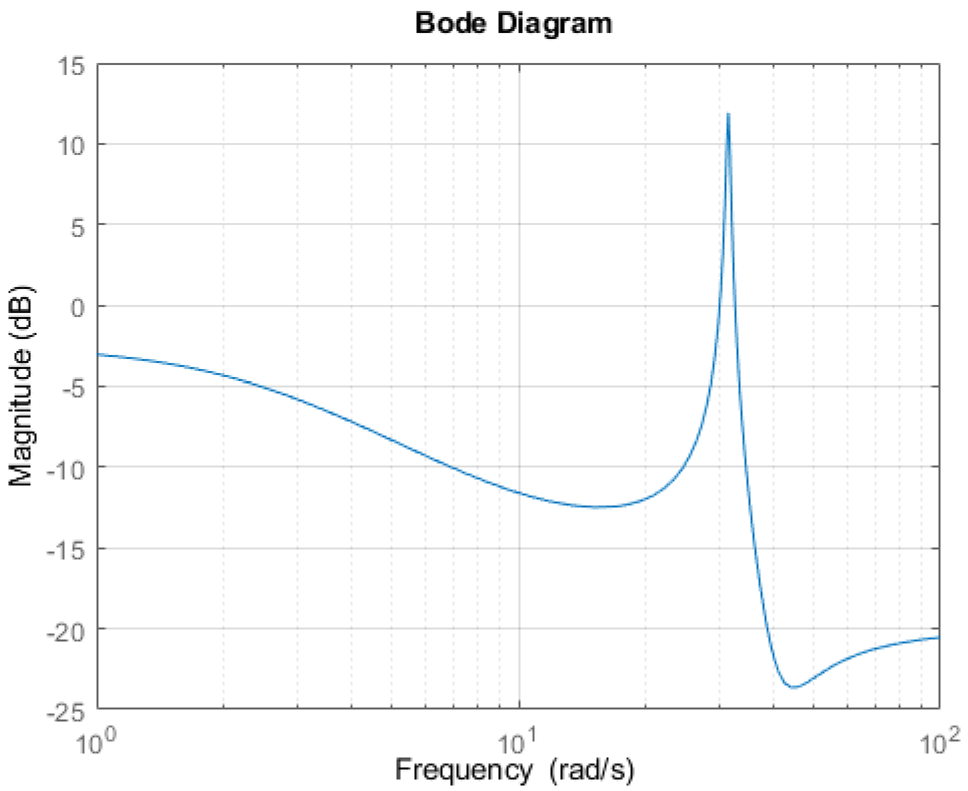


bodemag automatically selects the plot range based on the system dynamics.

Bode Magnitude Plot at Specified Frequencies

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

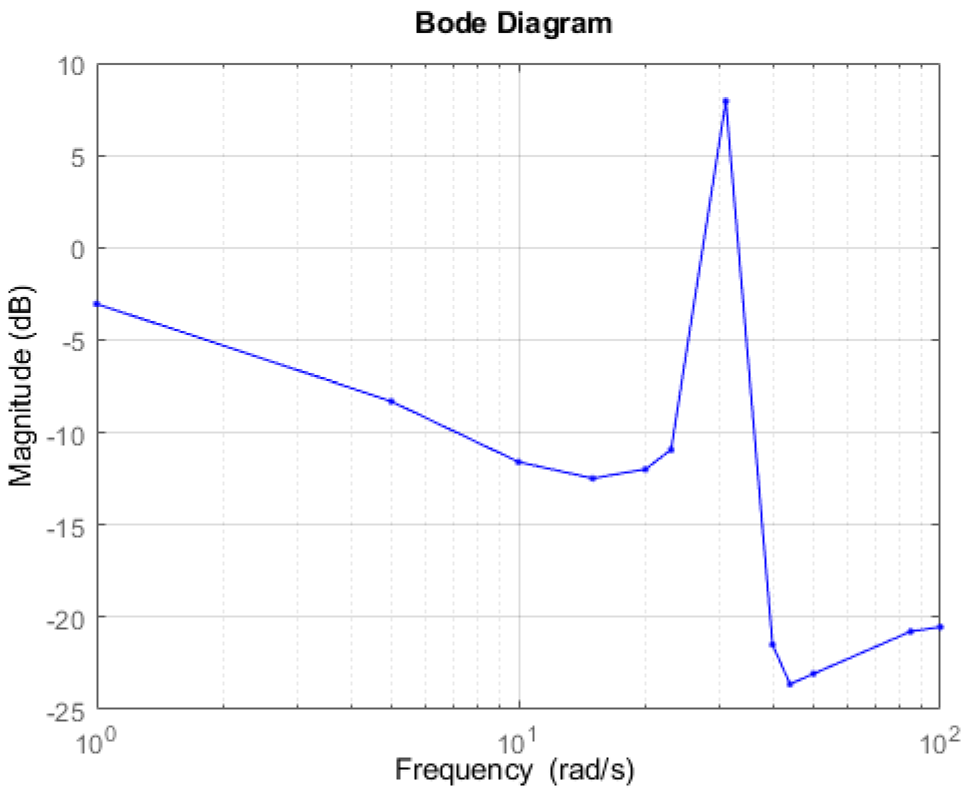
```
H = tf([-0.1,-2.4,-181,-1950],[1,3.3,990,2600]);
bodemag(H,{1,100})
grid on
```



The cell array `{1,100}` specifies the minimum and maximum frequency values in the Bode magnitude plot. When you provide frequency bounds in this way, the function selects intermediate points for frequency response data.

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

```
w = [1 5 10 15 20 23 31 40 44 50 85 100];
bodemag(H,w,'.-')
grid on
```



`bodemag` plots the frequency response at the specified frequencies only.

Compare Bode Magnitude Plots of Several Dynamic Systems

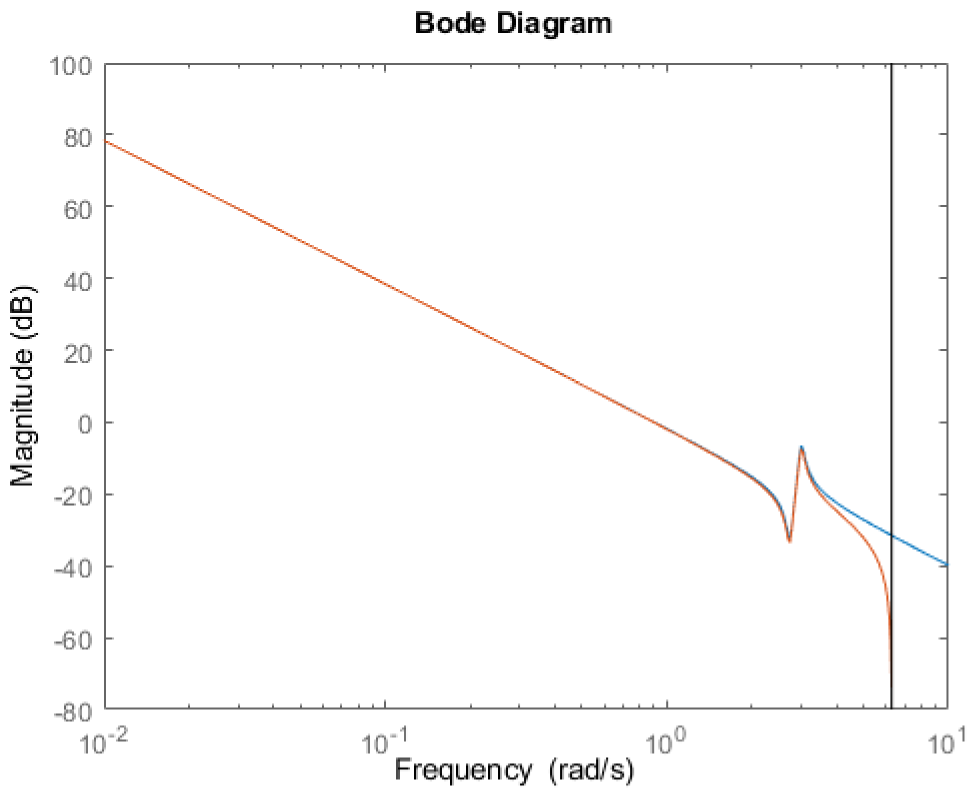
Compare the magnitude of the frequency response of a continuous-time system to an equivalent discretized system on the same Bode plot.

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 magnitude plot that displays the responses of both systems.

```
bodemag(H,Hd)
```

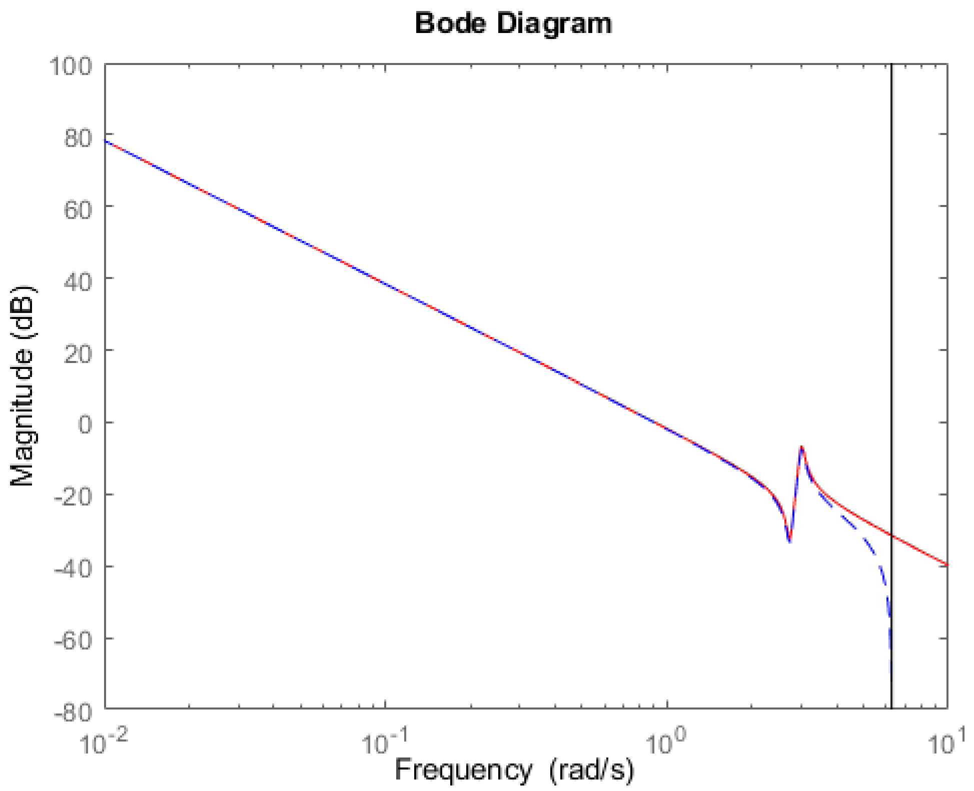


The Bode magnitude plot of a discrete-time system includes a vertical line marking the Nyquist frequency of the system.

Bode Magnitude Plot with Specified Line and Marker Attributes

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

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



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

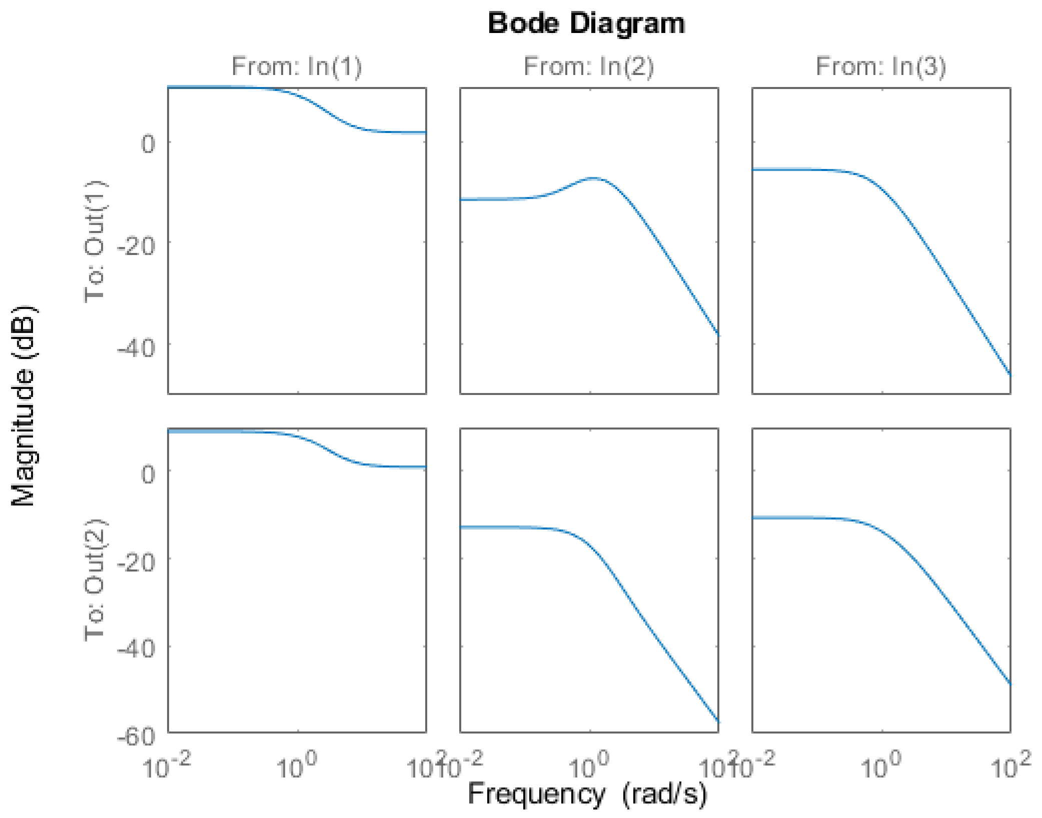
Magnitude of MIMO System

For this example, create a 2-output, 3-input system.

```
rng(0,'twister'); % For reproducibility
H = rss(4,2,3);
```

For this system, `bodemag` plots the magnitude-only frequency responses of each I/O channel in a separate plot in a single figure.

```
bodemag(H)
```



Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value for both plotting and returning frequency response data.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model. When you use output arguments, the function returns frequency response data for the nominal model only.
- Frequency-response data models such as `frd` models. For such models, the function plots the response at frequencies defined in the model.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models.

If `sys` is an array of models, the function plots the frequency responses of all models in the array on the same axes.

LineSpec — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a string or vector of one, two, or three characters. The characters can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line. For more information about configuring this argument, see the `LineSpec` input argument of the `plot` function.

Example: `'r--'` specifies a red dashed line

Example: `'*b'` specifies blue asterisk markers

Example: `'y'` specifies a yellow line

w — Frequencies
`{wmin,wmax}` | vector

Frequencies at which to compute and plot frequency response, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function computes the index at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function computes the index at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values.

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

Algorithms

`bodemag` computes the frequency response as follows:

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

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

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

where T_s is the sample time and ω_N is the Nyquist frequency. The equivalent continuous-time frequency ω is then used as the x-axis variable. Because $H\left(e^{j\omega T_s}\right)$ is periodic with period $2\omega_N$, `bodemag` plots the response only up to the Nyquist frequency ω_N. If `sys` is a discrete-time model with unspecified sample time, `bodemag` uses $T_s = 1$.

See Also
`bode` | `bodeplot` | `freqresp` | `nyquist` | `spectrum` | `step`

Topics
"Plot Bode and Nyquist Plots at the Command Line"
"Dynamic System Models"

Introduced in R2012a

bodeoptions

Create list of Bode plot options

Description

Use the `bodeoptions` command to create a `BodePlotOptions` object to customize Bode plot appearance. You can also use the command to override the plot preference settings in the MATLAB session in which you create the Bode plots.

Creation

Syntax

```
plotoptions = bodeoptions
plotoptions = bodeoptions('cstprefs')
```

Description

`plotoptions = bodeoptions` returns a default set of plot options for use with the `bodeplot` command. 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.

`plotoptions = bodeoptions('cstprefs')` initializes the plot options with the options you selected in the Control System Toolbox and System Identification Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor”. 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.

Properties

`FreqUnits` — Frequency units
`'rad/s'` (default)

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
'`log`' (default) | '`linear`'

Frequency scale, specified as either '`log`' or '`linear`' .

`MagUnits` — Magnitude units
'`dB`' (default) | '`abs`'

Magnitude units, specified as either '`dB`' or absolute value '`abs`'.

`MagScale` — Magnitude scale
'`linear`' (default) | '`log`'

Magnitude scale, specified as either '`log`' or '`linear`'.

`MagVisible` — Toggle magnitude plot visibility
'`on`' (default) | '`off`'

Toggle magnitude plot visibility, specified as either '`on`' or '`off`'.

`MagLowerLimMode` — Lower magnitude limit mode
'`auto`' (default) | '`manual`'

Lower magnitude limit mode, specified as either '`auto`' or '`manual`'.

`MagLowerLim` — Lower magnitude limit value
'`-inf`' (default) | scalar

Lower magnitude limit value, specified as a scalar.

`PhaseUnits` — Phase units
'`deg`' (default) | '`rad`'

Phase units, specified as either '`deg`' or '`rad`' to change to degrees or radians, respectively.

PhaseVisible — Toggle phase plot visibility
'on' (default) | 'off'

Toggle phase plot visibility, specified as either 'on' or 'off'.

PhaseWrapping — Enable phase wrapping
'off' (default) | 'on'

Enable phase wrapping, specified as either 'on' or 'off'. When you set PhaseWrapping to 'on', the plot wraps accumulated phase at the value specified by the PhaseWrappingBranch property.

PhaseWrappingBranch — Phase wrapping value
-180 (default) | integer

Phase wrapping value at which the plot wraps accumulated phase when PhaseWrapping is set to 'on'. By default, phase wraps into the interval [-180°,180°].

PhaseMatching — Enable phase matching
'off' (default) | 'on'

Enable phase matching, specified as either 'on' or 'off'. Turning PhaseMatching 'on' matches the phase to the value specified in PhaseMatchingValue at the frequency specified in PhaseMatchingFreq

PhaseMatchingFreq — Phase matching frequency
0 (default) | scalar

Phase matching frequency, specified as a scalar.

PhaseMatchingValue — Phase matching response value
0 (default) | scalar

Phase matching response value, specified as a scalar.

ConfidenceRegionNumberSD — Number of standard deviations to use to plot the confidence region
1 (default) | scalar

Number of standard deviations to use to plot the confidence region, specified as a scalar. This is applicable to identified models only.

IOGrouping — Grouping of input-output pairs
'none' (default) | 'inputs' | 'outputs' | 'all'

Grouping of input-output (I/O) pairs, specified as one of the following:

- 'none' — No input-output grouping.
- 'inputs' — Group only the inputs.
- 'outputs' — Group only the outputs.
- 'all' — Group all the I/O pairs.

InputLabels — Input label style
structure (default)

Input label style, specified as a structure with the following fields:

- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is dark grey with the RGB triplet `[0.4,0.4,0.4]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`OutputLabels` — Output label style
structure (default)

Output label style, specified as a structure with the following fields:

- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is dark grey with the RGB triplet `[0.4,0.4,0.4]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`InputVisible` — Toggle display of inputs
`{'on'}` (default) | `{'off'}` | cell array

Toggle display of inputs, specified as either `{'on'}`, `{'off'}` or a cell array with multiple elements .

`OutputVisible` — Toggle display of outputs
`{'on'}` (default) | `{'off'}` | cell array

Toggle display of outputs, specified as either `{'on'}`, `{'off'}` or a cell array with multiple elements.

`Title` — Title text and style
structure (default)

Title text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a character vector. By default, the plot is titled `'Bode Diagram'`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`XLabel` — X-axis label text and style
structure (default)

X-axis label text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a character vector. By default, the axis is titled based on the frequency units `FreqUnits`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`YLabel` — Y-axis label text and style
structure (default)

Y-axis label text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a cell array of character vectors. By default, the axis label is a 1x2 cell array with `'Magnitude'` and `'Phase'`.

- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`TickLabel` — Tick label style
structure (default)

Tick label style, specified as a structure with the following fields:

- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.

`Grid` — Toggle grid display
'`off`' (default) | '`on`'

Toggle grid display on the plot, specified as either '`off`' or '`on`'.

`GridColor` — Color of the grid lines
`[0.15,0.15,0.15]` (default) | RGB triplet

Color of the grid lines, specified as an RGB triplet. The default color is light grey specified by the RGB triplet `[0.15,0.15,0.15]`.

`XLimMode` — X-axis limit selection mode
'`auto`' (default) | '`manual`' | cell array

Selection mode for the x-axis limits, specified as one of these values:

- '`auto`' — Enable automatic limit selection, which is based on the total span of the plotted data.
- '`manual`' — Manually specify the axis limits. To specify the axis limits, set the `XLim` property.

`YLimMode` — Y-axis limit selection mode
'`auto`' (default) | '`manual`' | cell array

Selection mode for the y-axis limits, specified as one of these values:

- `'auto'` — Enable automatic limit selection, which is based on the total span of the plotted data.
- `'manual'` — Manually specify the axis limits. To specify the axis limits, set the `YLim` property.

`XLim` — X-axis limits
`'{[1,10]}'` (default) | cell array of two-element vector of the form `[min,max]` | cell array

X-axis limits, specified as a cell array of two-element vector of the form `[min,max]`.

`YLim` — Y-axis limits
`'{[1,10]}'` (default) | cell array of two-element vector of the form `[min,max]` | cell array

Y-axis limits, specified as a cell array of two-element vector of the form `[min,max]`.

Object Functions

bode	Bode plot of frequency response, or magnitude and phase data
bodeplot	Plot Bode frequency response with additional plot customization options
getoptions	Return plot options handle or plot options property
setoptions	Set plot options handle or plot options property

Examples

Custom Bode Plot Settings Independent of Preferences

For this example, create a Bode plot that uses 15-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 using `bodeoptions`.

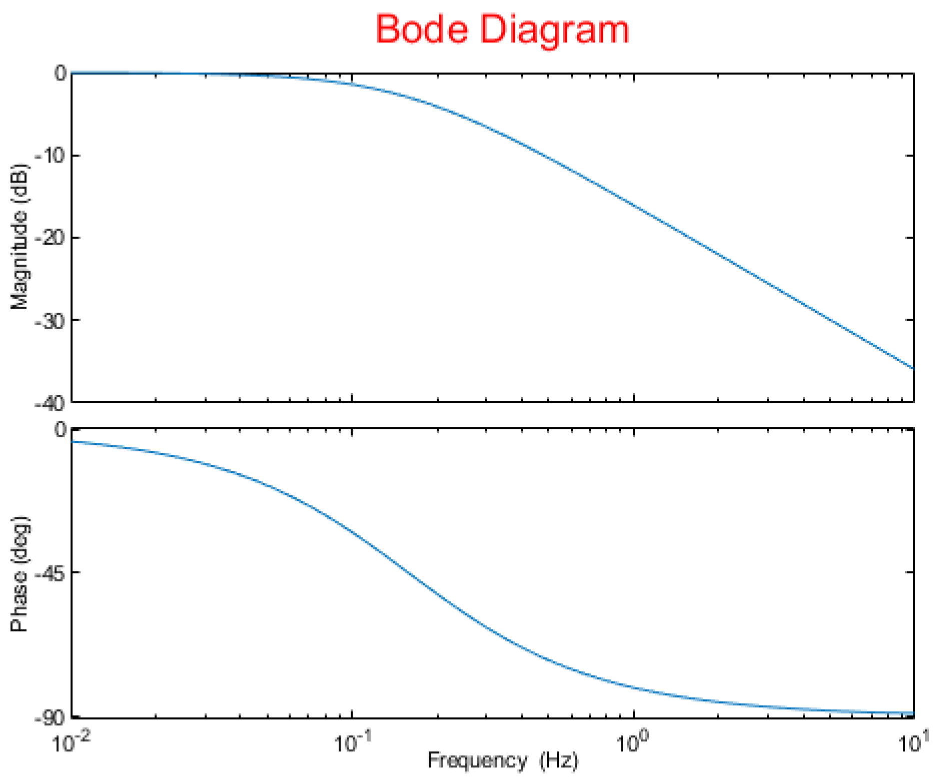
```
opts = bodeoptions;
```

Next, change the required properties of the options set `opts`.

```
opts.Title.FontSize = 15;
opts.Title.Color = [1 0 0];
opts.FreqUnits = 'Hz';
```

Now, create a Bode plot using the options set `opts`.

```
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.

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.

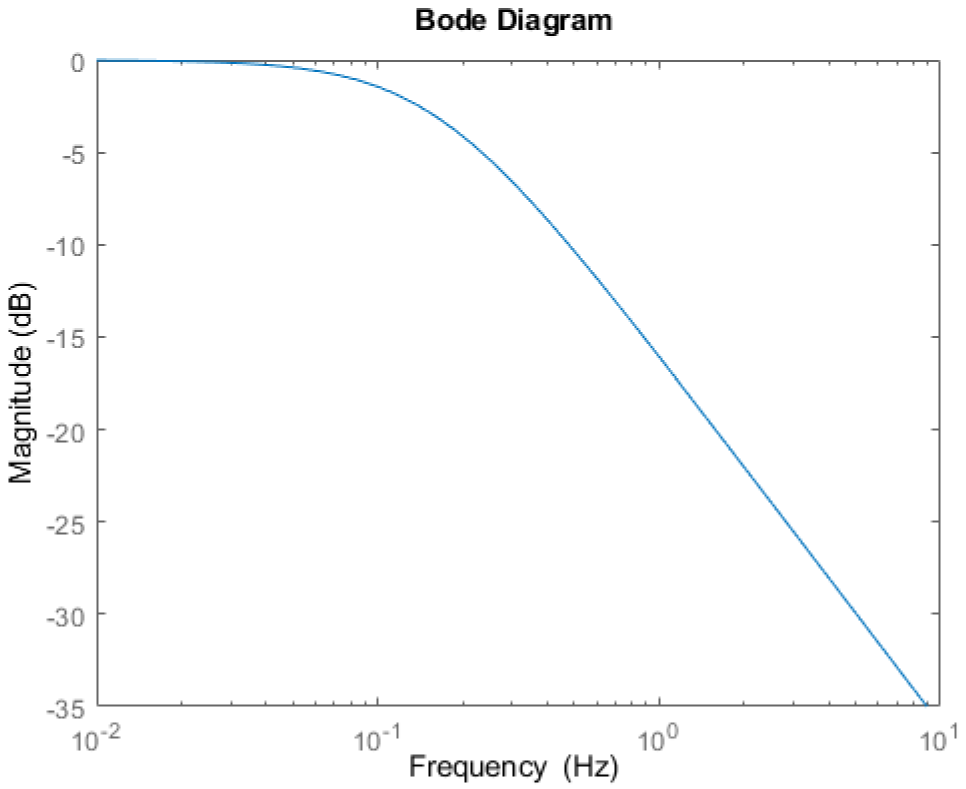
```
opts = bodeoptions('cstprefs');
```

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.

See Also

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

Topics

"Toolbox Preferences Editor"

Introduced in R2012a

bodeplot

Plot Bode frequency response with additional plot customization options

Syntax

```
h = bodeplot(sys)
h = bodeplot(sys1,sys2,...,sysN)
h = bodeplot(sys1,LineSpec1,...,sysN,LineSpecN)
h = bodeplot(AX, ___ )
h = bodeplot( ___ ,plotoptions)
h = bodeplot( ___ ,w)
```

Description

`bodeplot` lets you plot the Bode magnitude and phase of a dynamic system model with a broader range of plot customization options than `bode`. You can use `bodeplot` to obtain the plot handle and use it to customize the plot, such as modify the axes labels, limits and units. You can also use `bodeplot` to draw a Bode response plot on an existing set of axes represented by an axes handle. To customize an existing Bode plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox). To create Bode plots with default options or to extract the frequency response data, use `bode`.

`h = bodeplot(sys)` plots 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 `h` to customize the plot with the `getoptions` and `setoptions` commands. If `sys` is a multi-input, multi-output (MIMO) model, then `bodeplot` produces a grid of Bode plots, each plot displaying the frequency response of one I/O pair.

`h = bodeplot(sys1,sys2,...,sysN)` plots the frequency response of multiple dynamic systems `sys1,sys2,…,sysN` on the same plot. All systems must have the same number of inputs and outputs to use this syntax.

`h = bodeplot(sys1,LineSpec1,...,sysN,LineSpecN)` sets the line style, marker type, and color for the Bode response of each system. All systems must have the same number of inputs and outputs to use this syntax.

`h = bodeplot(AX, ___ )` plots the Bode response on the `Axes` or `UIAxes` object in the current figure with the handle `AX`. Use this syntax when creating apps using `bodeplot` in the App Designer.

`h = bodeplot( ___ ,plotoptions)` plots the Bode frequency response with the options set specified in `plotoptions`. You can use these options to customize the Bode plot appearance using the command line. Settings you specify in `plotoptions` overrides the preference settings in the MATLAB session in which you run `bodeplot`. Therefore, this syntax is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

`h = bodeplot( ___ ,w)` plots system responses for frequencies specified by the frequencies in `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `bodeplot` plots the response at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `bodeplot` plots the response at each specified frequency.

You can use `w` with any of the input-argument combinations in previous syntaxes.

See `logspace` to generate logarithmically spaced frequency vectors.

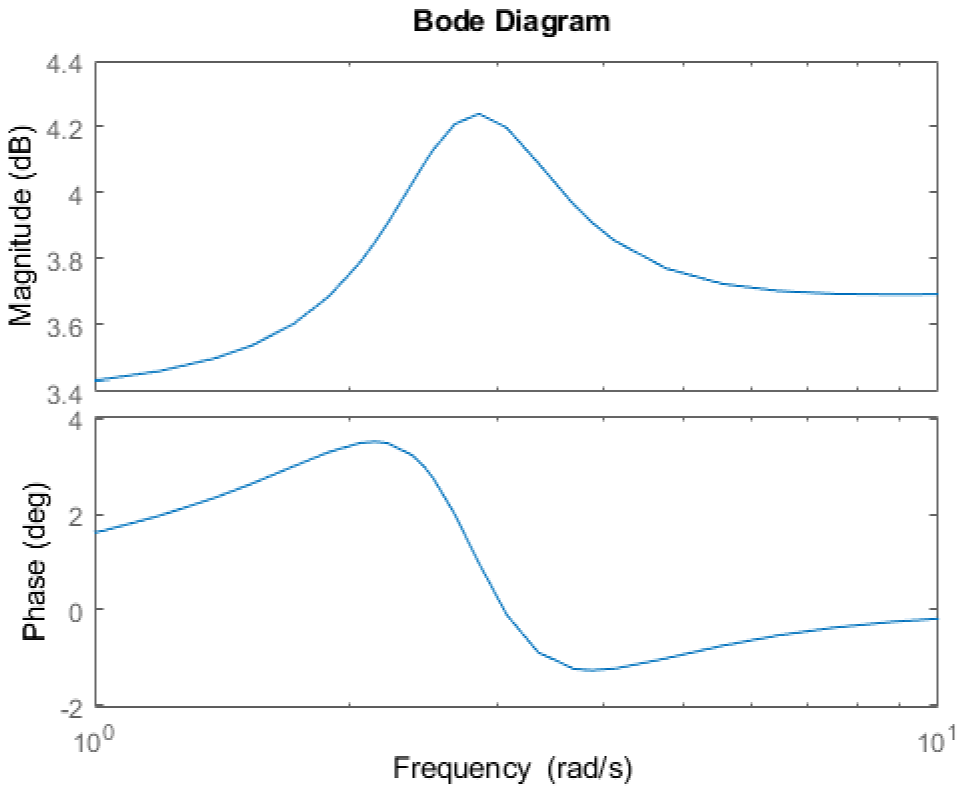
Examples

Customize Bode Plot using Plot Handle

For this example, use the plot handle to change the frequency units to Hz and turn off the phase plot.

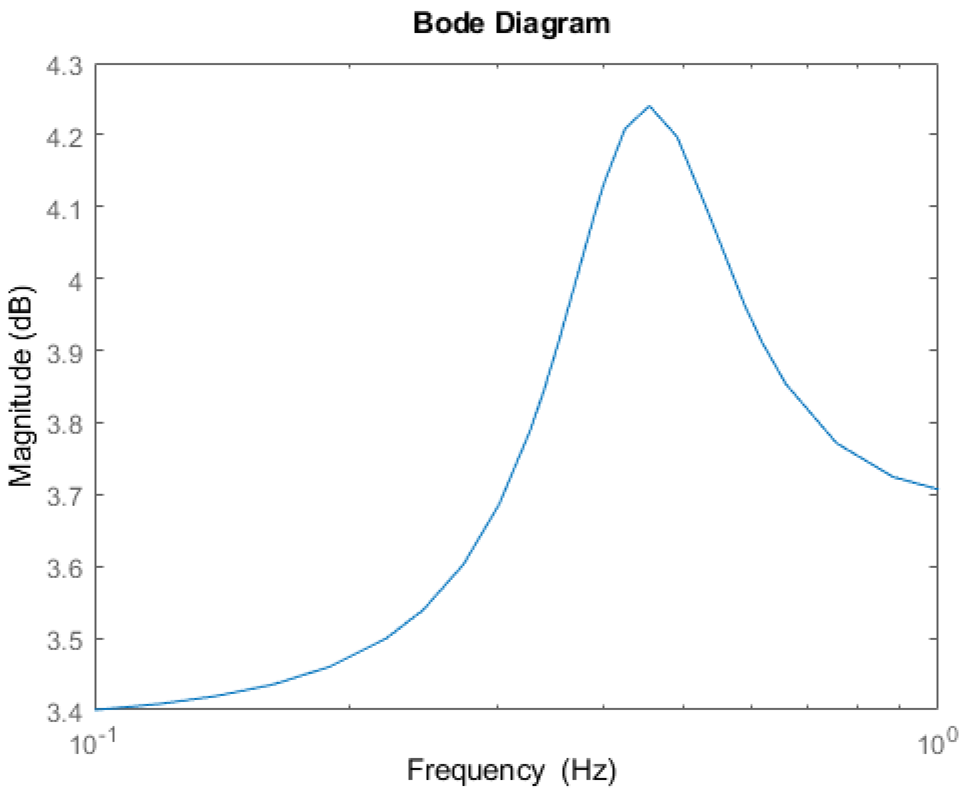
Generate a random state-space model with 5 states and create the Bode plot with plot handle `h`.

```
rng("default")
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` using `setoptions`.

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



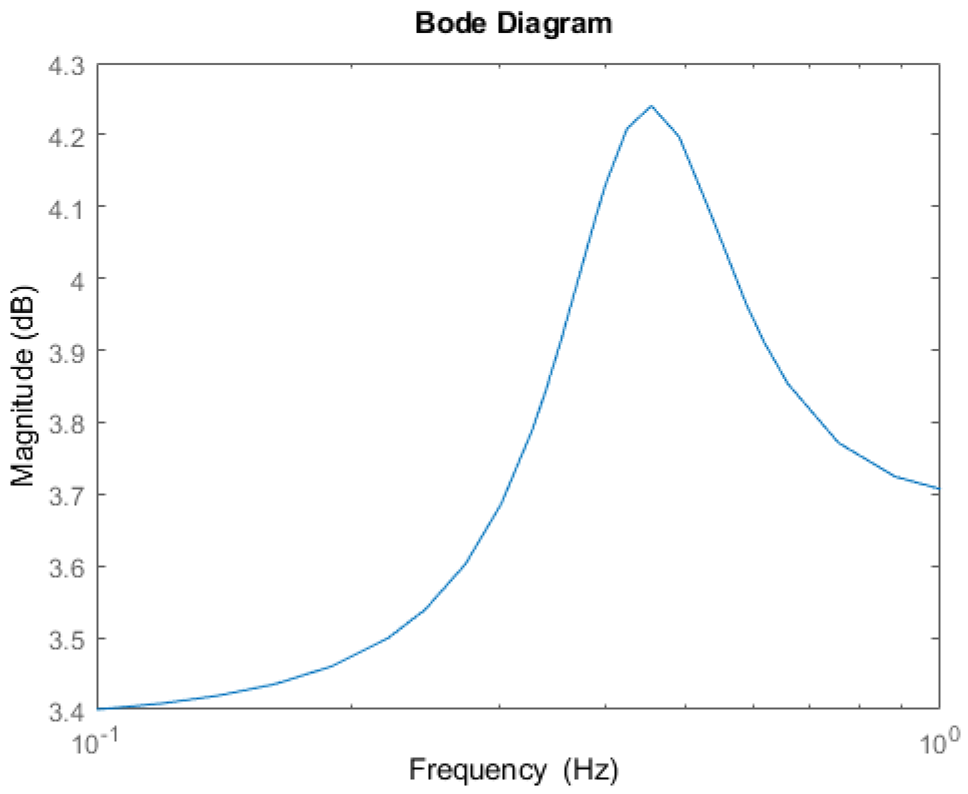
The Bode plot automatically updates when you call `setoptions`.

Alternatively, you can also use the `bodeoptions` command to specify the required plot options. First, create an options set based on the toolbox preferences.

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

Change properties of the options set by setting the frequency units to Hz and hide the phase plot.

```
p.FreqUnits = 'Hz';
p.PhaseVisible = 'off';
bodeplot(sys,p);
```



You can use the same option set to create multiple Bode plots with the same customization. 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 Bode Plot Settings Independent of Preferences

For this example, create a Bode plot that uses 15-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 using `bodeoptions`.

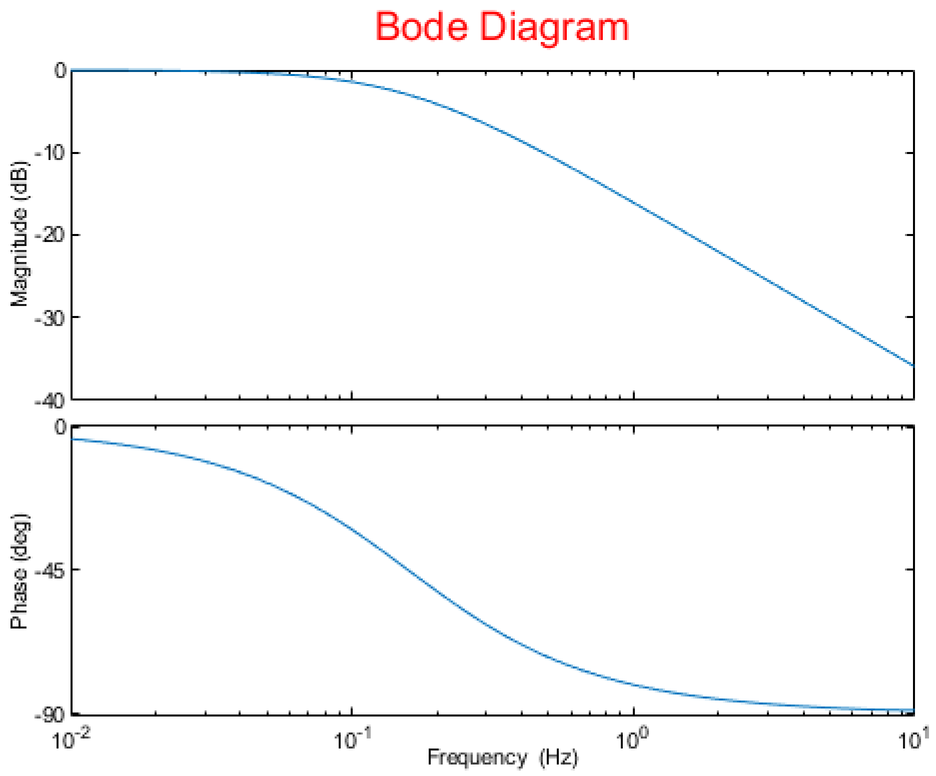
```
opts = bodeoptions;
```

Next, change the required properties of the options set `opts`.

```
opts.Title.FontSize = 15;
opts.Title.Color = [1 0 0];
opts.FreqUnits = 'Hz';
```

Now, create a Bode plot using the options set `opts`.

```
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.

Customized Bode Plot of Transfer Function

For this example, create a Bode plot of the following continuous-time SISO dynamic system. Then, turn the grid on, rename the plot and change the frequency scale.

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

Create the transfer function `sys`.

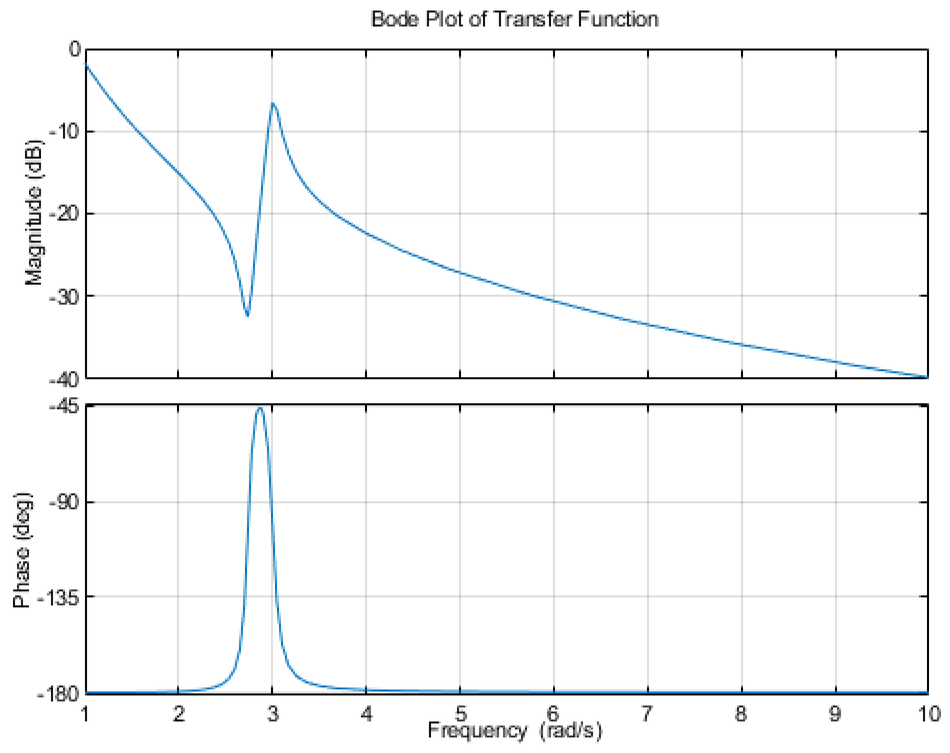
```
sys = tf([1 0.1 7.5],[1 0.12 9 0 0]);
```

Next, create the options set using `bodeoptions` and change the required plot properties.

```
plotoptions = bodeoptions;
plotoptions.Grid = 'on';
plotoptions.FreqScale = 'linear';
plotoptions.Title.String = 'Bode Plot of Transfer Function';
```

Now, create the Bode plot with the custom option set `plotoptions`.

```
bodeplot(sys,plotoptions)
```



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

Bode Plot with Specified Frequency Scale and Units

For this example, consider a MIMO state-space model with 3 inputs, 3 outputs and 3 states. Create a Bode plot with linear frequency scale, specify frequency units in Hz and turn the grid on.

Create the MIMO state-space model `sys_mimo`.

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
size(sys_mimo)
```

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

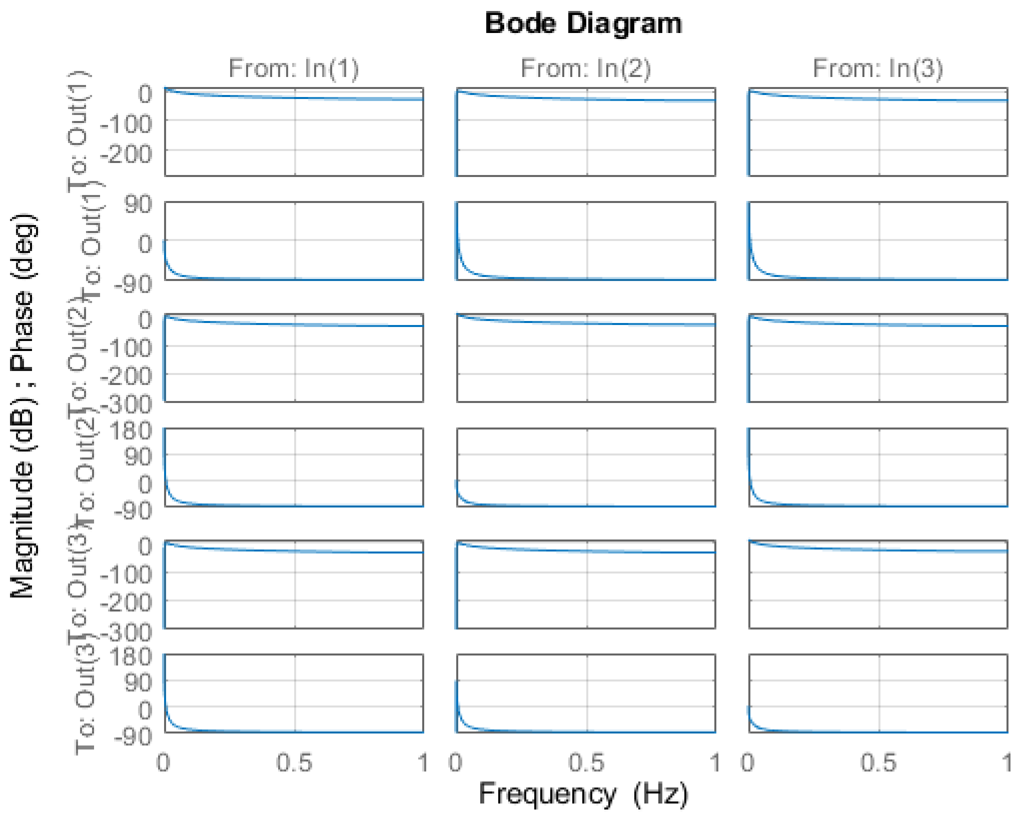
Create a Bode plot with plot handle `h` and use `getoptions` for a list of the options available.

```
h = bodeplot(sys_mimo);
p = getoptions(h)
```

```
p =

                  FreqUnits: 'rad/s'
                  FreqScale: 'log'
                   MagUnits: 'dB'
                   MagScale: 'linear'
                 MagVisible: 'on'
            MagLowerLimMode: 'auto'
                 PhaseUnits: 'deg'
               PhaseVisible: 'on'
              PhaseWrapping: 'off'
              PhaseMatching: 'off'
          PhaseMatchingFreq: 0
    ConfidenceRegionNumberSD: 1
                MagLowerLim: 0
         PhaseMatchingValue: 0
        PhaseWrappingBranch: -180
                 IOGrouping: 'none'
                InputLabels: [1x1 struct]
               OutputLabels: [1x1 struct]
               InputVisible: {3x1 cell}
              OutputVisible: {3x1 cell}
                      Title: [1x1 struct]
                     XLabel: [1x1 struct]
                     YLabel: [1x1 struct]
                  TickLabel: [1x1 struct]
                       Grid: 'off'
                  GridColor: [0.1500 0.1500 0.1500]
                       XLim: {3x1 cell}
                       YLim: {6x1 cell}
                   XLimMode: {3x1 cell}
                   YLimMode: {6x1 cell}
```

Use `setoptions` to update the plot with the requires customization.

```
setoptions(h,'FreqScale','linear','FreqUnits','Hz','Grid','on');
```



The Bode plot automatically updates when you call `setoptions`. For MIMO models, `bodeplot` produces an array of Bode plots, each plot displaying the frequency response of one I/O pair.

Match Phase at Specified Frequency

For this example, match the phase of your system response such that the phase at 1 rad/sec is 150 degrees.

First, create a Bode plot of transfer function system with plot handle `h`.

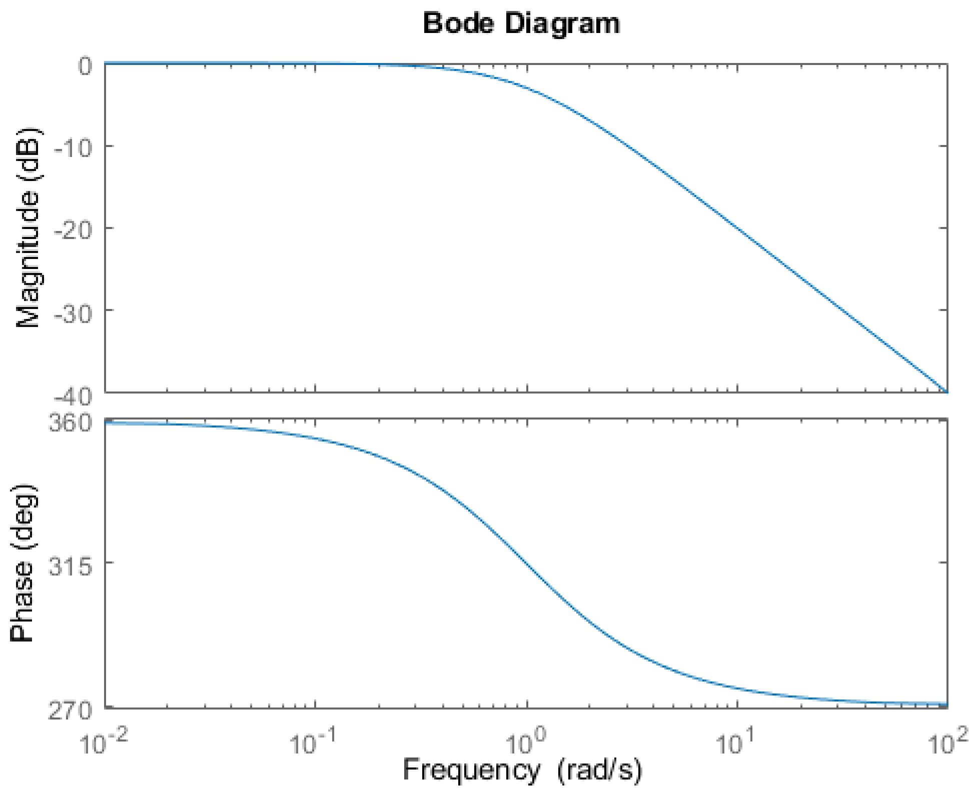
```
sys = tf(1,[1 1]);
h = bodeplot(sys);
```

Use `getoptions` to obtain the plot properties. Change the properties `PhaseMatchingFreq` and `PhaseMatchingValue` to match a phase to a specified frequency.

```
p = getoptions(h);
p.PhaseMatching = 'on';
p.PhaseMatchingFreq = 1;
p.PhaseMatchingValue = 150;
```

Update the plot using `setoptions`.

```
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 150 degrees yields the second Bode plot. Note that, however, the phase can only be -45 + N*360, where N is an integer. So the plot is set to the nearest allowable phase, namely 315 degrees (or $1 * 360 - 45 = 315^o$).

Display Confidence Regions of Identified Models

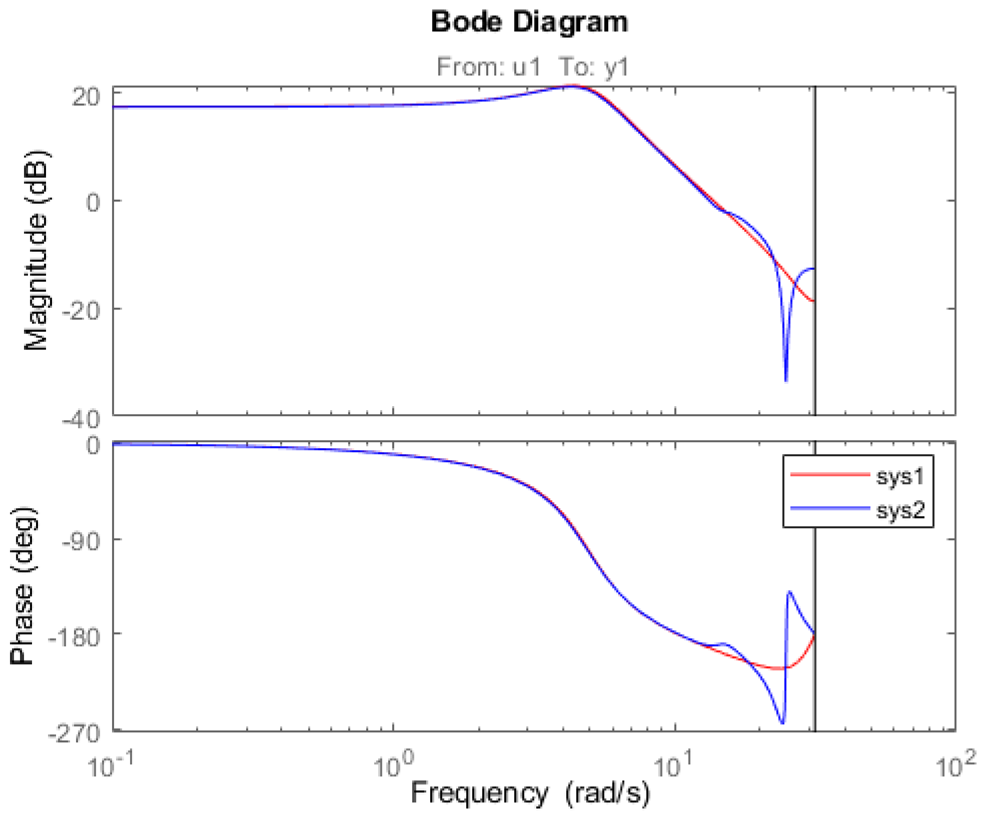
For this example, compare the frequency responses of two identified state-space models with 2 and 6 states along with their 2 σ confidence regions.

Load the identified state-space model data and estimate the two models using `n4sid`. Using `n4sid` requires a System Identification Toolbox license.

```
load iddata1
sys1 = n4sid(z1,2);
sys2 = n4sid(z1,6);
```

Create a Bode plot of the two systems.

```
bodeplot(sys1,'r',sys2,'b');
legend('sys1','sys2');
```

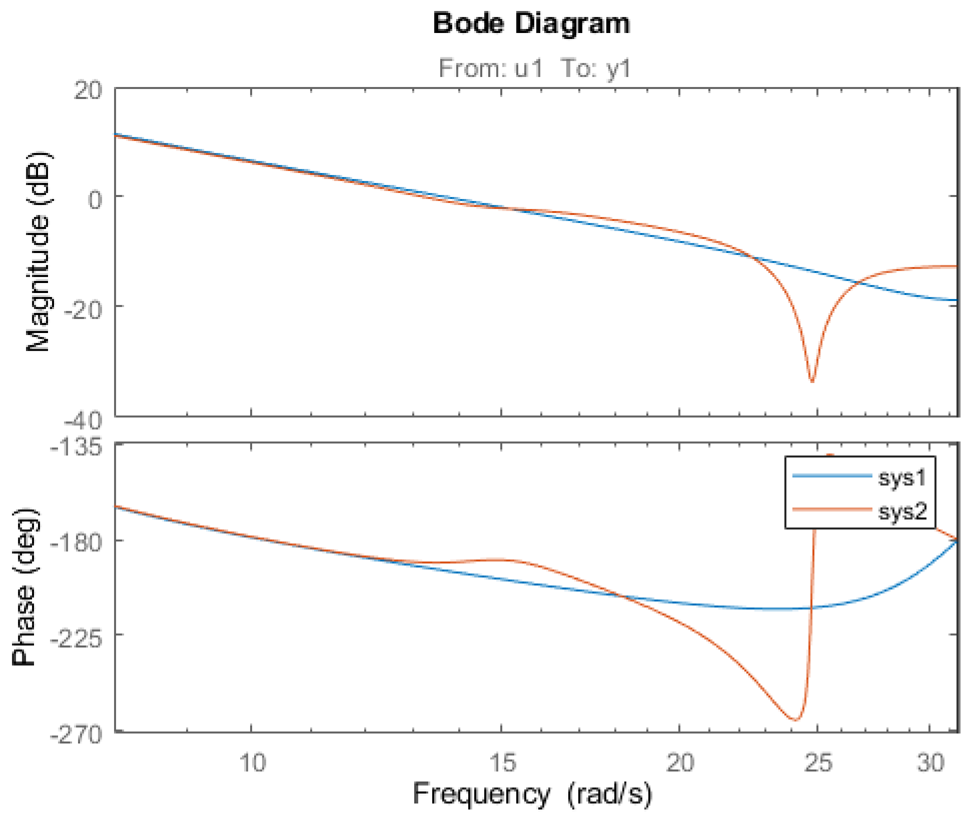


From the plot, observe that both models produce about 70% fit to data. However, `sys2` shows higher uncertainty in its frequency response, especially close to the Nyquist frequency. Now, use `linspace` to create a vector of frequencies and plot the Bode response using the frequency vector `w`.

```
w = linspace(8,10*pi,256);
h = bodeplot(sys1,sys2,w);
legend('sys1','sys2');
```

Use `setoptions` to turn on phase matching and to specify the standard deviation of the confidence region.

```
setoptions(h,'PhaseMatching','on','ConfidenceRegionNumberSD',2);
```



You can use the showconfidence command to display the confidence regions on the Bode plot.

```
showConfidence(h)
```

Frequency Response of Identified Parametric and Nonparametric Models

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

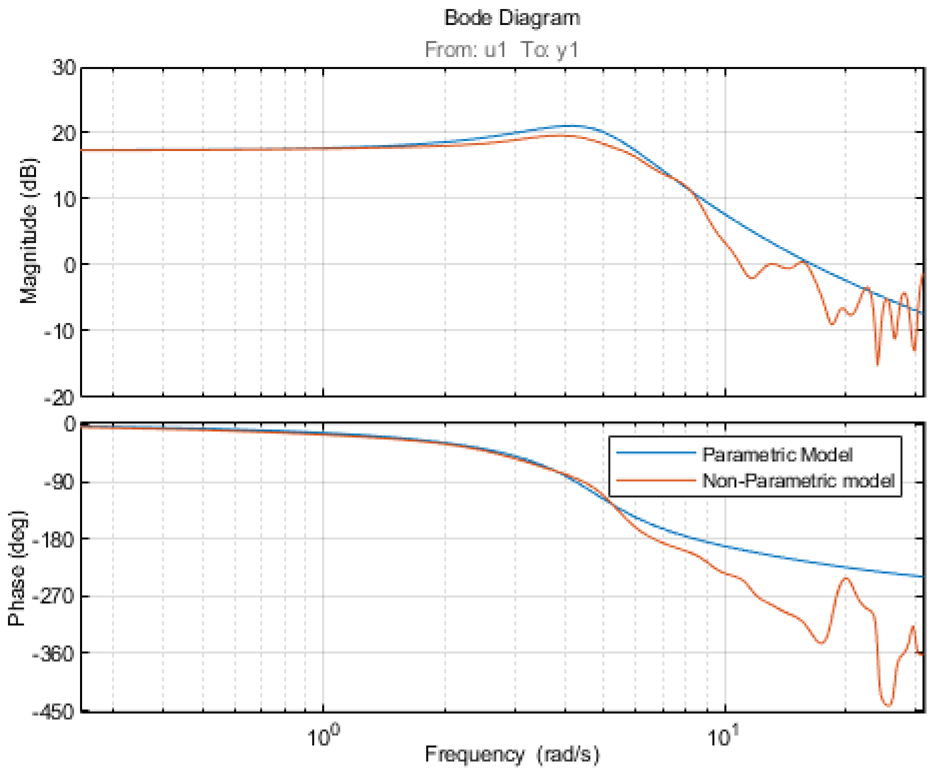
Load the data and create the parametric and non-parametric models using `tfest` and `spa`, respectively.

```
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. The model `sys_np` is a non-parametric identified model while, `sys_p` is a parametric identified model.

Create an options set to turn phase matching and the grid on. Then, create a Bode plot that includes both systems using this options set.

```
plotoptions = bodeoptions;
plotoptions.PhaseMatching = 'on';
plotoptions.Grid = 'on';
bodeplot(sys_p,sys_np,w,plotoptions);
legend('Parametric Model','Non-Parametric model');
```



Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Sparse state-space models, such as `sparss` or `mechss` models. Frequency grid `w` must be specified for sparse models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value to plot the frequency response data.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model.

- Frequency-response data models such as `frd` models. For such models, the function plots the response at frequencies defined in the model.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models. For such models, the function can also plot confidence intervals and return standard deviations of the frequency response. See “Bode Plot of Identified Model” on page 1-133.

If `sys` is an array of models, the function plots the frequency responses of all models in the array on the same axes.

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'_'`	Horizontal line
`'|'`	Vertical line
`'s'`	Square
`'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'p'`	Pentagram
`'h'`	Hexagram

Color	Description
`y`	yellow

Color	Description
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

AX — Target axes
Axes object | UIAxes object

Target axes, specified as an Axes or UIAxes object. If you do not specify the axes and if the current axes are Cartesian axes, then bodeplot plots on the current axes. Use AX to plot into specific axes when creating apps in the App Designer.

plotoptions — Bode plot options set
BodePlotOptions object

Bode plot options set, specified as a BodePlotOptions object. You can use this option set to customize the Bode plot appearance. Use bodeoptions to create the option set. Settings you specify in plotoptions overrides the preference settings in the MATLAB session in which you run bodeplot. Therefore, plotoptions is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

For the list of available options, see bodeoptions.

w — Frequencies
{wmin,wmax} | vector

Frequencies at which to compute and plot frequency response, specified as the cell array {wmin,wmax} or as a vector of frequency values.

- If w is a cell array of the form {wmin,wmax}, then the function computes the response at frequencies ranging between wmin and wmax.
- If w is a vector of frequencies, then the function computes the response at each specified frequency. For example, use logspace to generate a row vector with logarithmically spaced frequency values.

Specify frequencies in units of rad/TimeUnit, where TimeUnit is the TimeUnit property of the model.

Output Arguments

h — Plot handle
handle object

Plot handle, returned as a handle object. Use the handle h to get and set the properties of the Bode plot using getoptions and setoptions. For the list of available options, see the *Properties and Values Reference* section in “Customizing Response Plots from the Command Line” (Control System Toolbox).

See Also

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

Topics

"Customizing Response Plots from the Command Line" (Control System Toolbox)

Introduced before R2006a

c2d

Convert model from continuous to discrete time

Syntax

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

Description

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

`sysd = c2d(sysc,Ts,method)` specifies the discretization method.

`sysd = c2d(sysc,Ts,opts)` specifies additional options for the discretization.

`[sysd,G] = c2d( ___ )`, where `sysc` is a state-space model, returns a matrix, `G` that maps the continuous initial conditions x_0 and u_0 of the state-space model to the discrete-time initial state vector $x[0]$.

Examples

Discretize a Transfer Function

Discretize the following continuous-time transfer function:

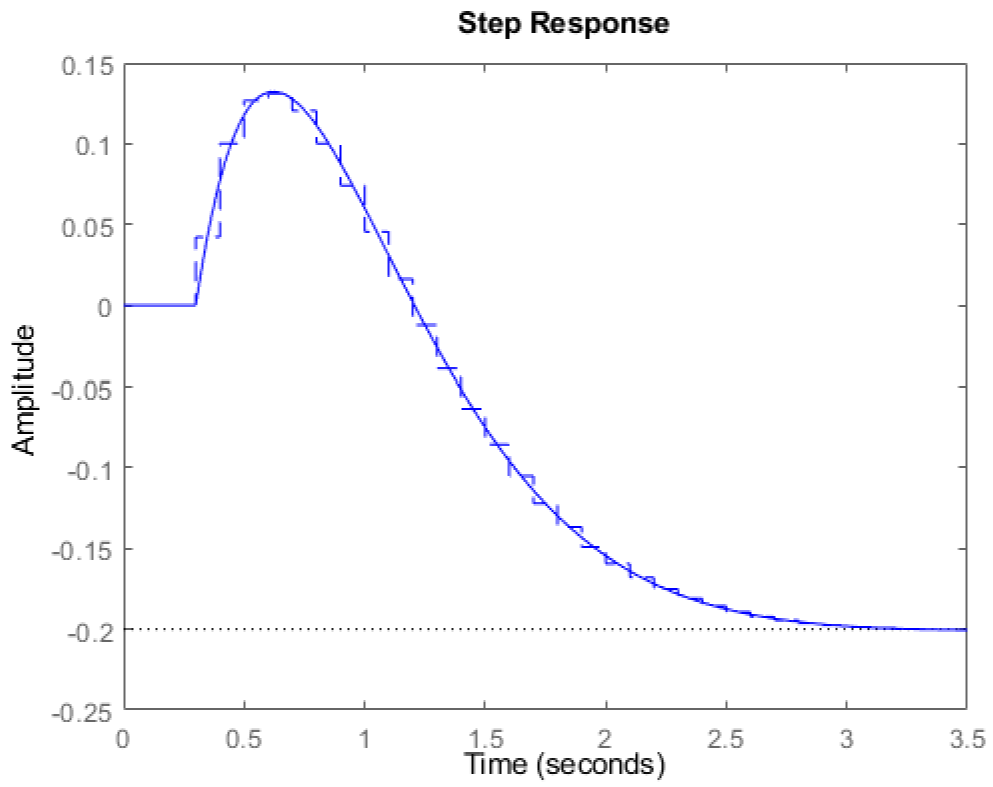
$$H(s) = e^{-0.3s}\frac{s-1}{s^2+4s+5}.$$

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.

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

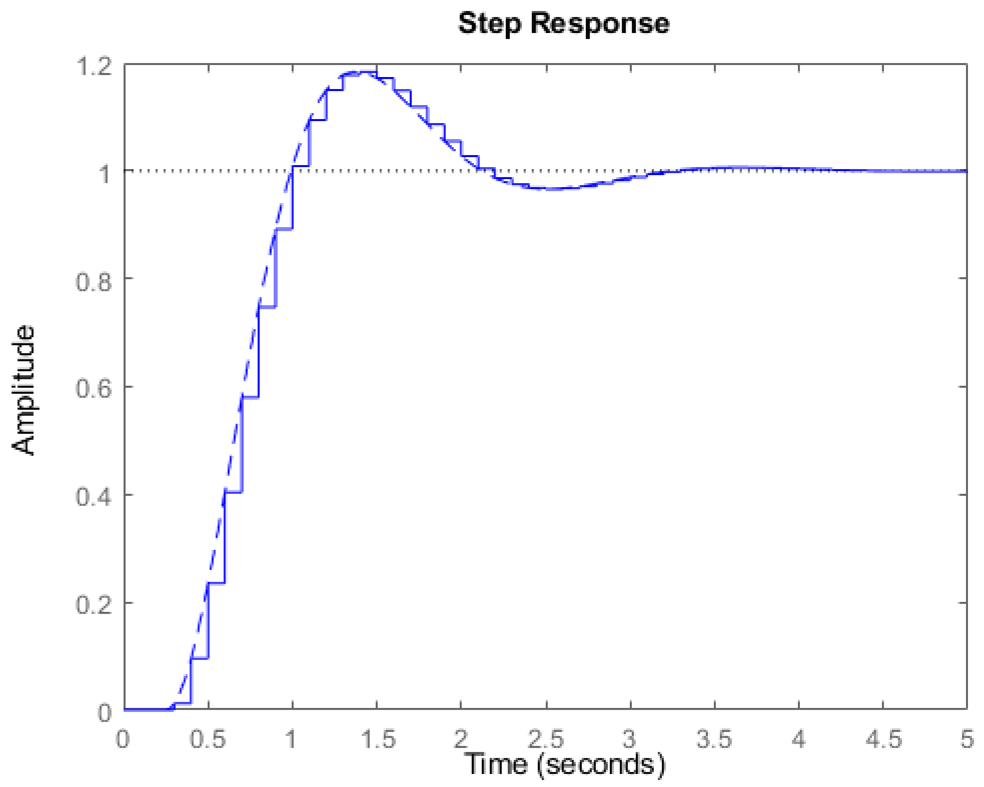
```
h = tf(10,[1 3 10],'IODelay',0.25);
hd = c2d(h,0.1)
```

```
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.

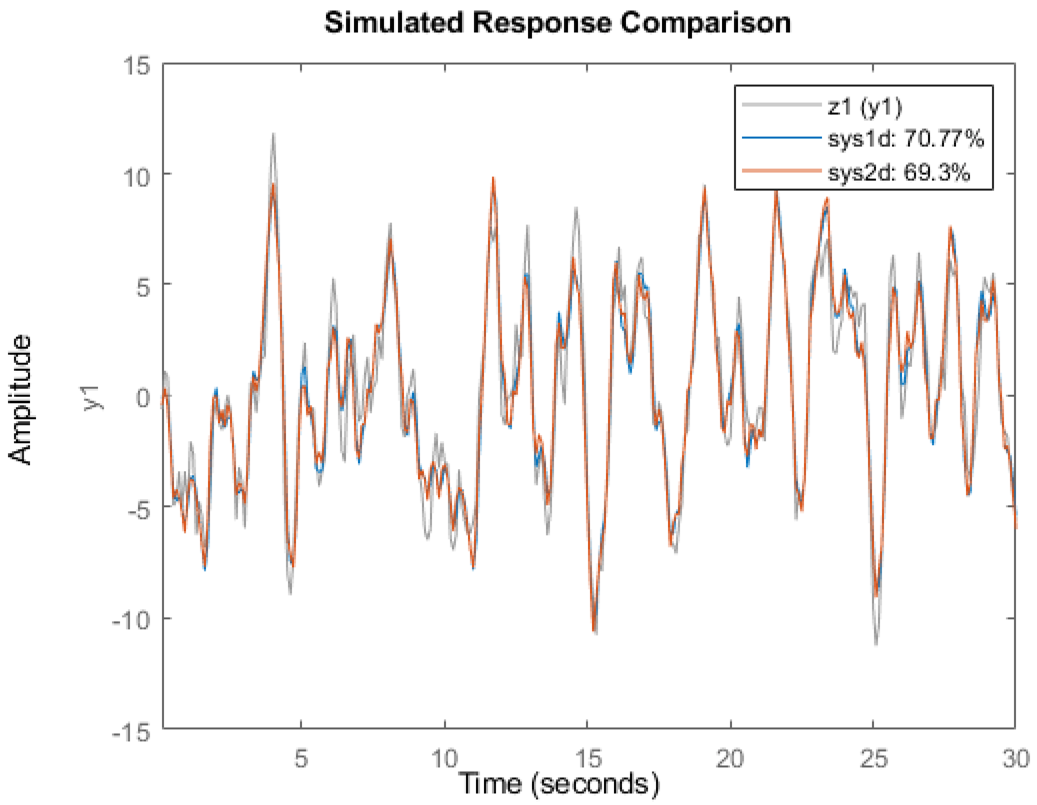
```
load iddata1
sys1c = tfest(z1,2);
sys1d = c2d(sys1c,0.1,'zoh');
```

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

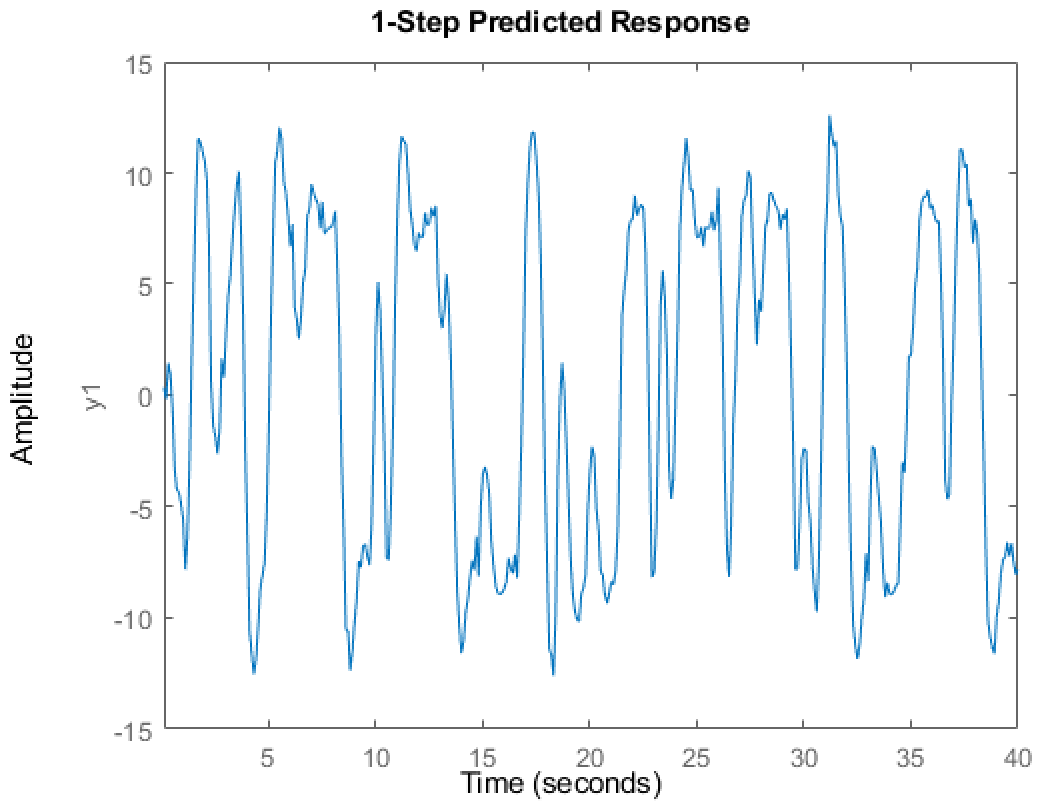
Discretize an identified state-space model to build a one-step ahead predictor of its response.

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');
```

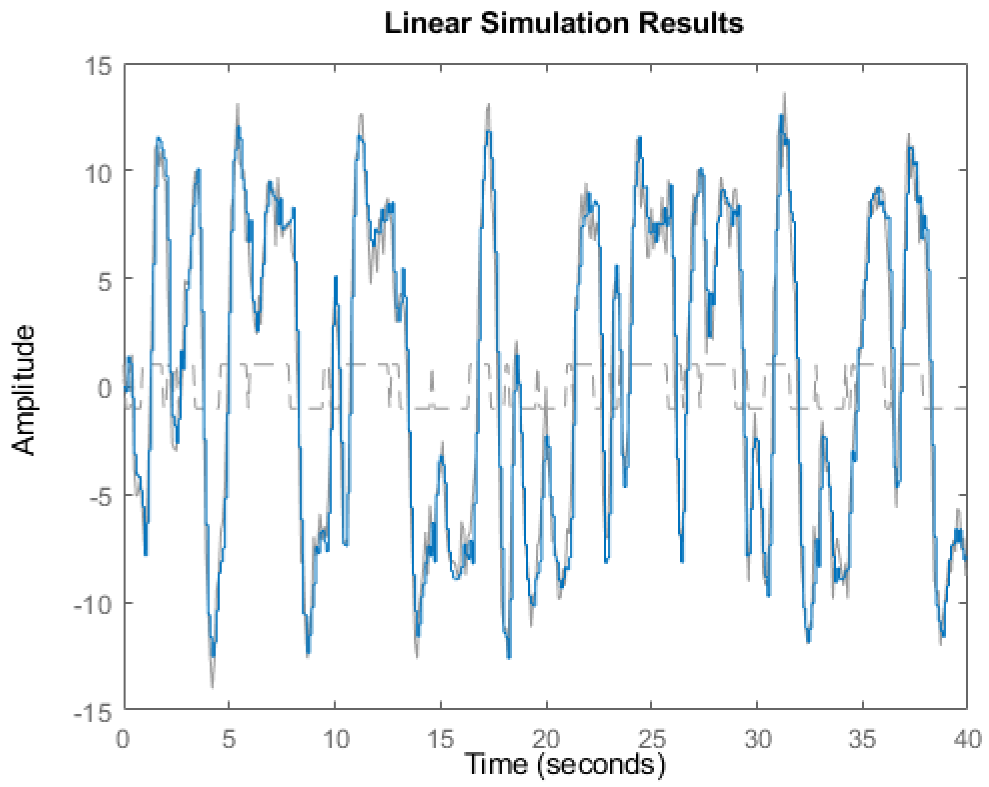
Build a predictor model from the discretized model, `sysd`.

```
[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)`.

Input Arguments

`sysc` — Continuous-time dynamic system

dynamic system model

Continuous-time model, specified as a dynamic system model such as `idtf`, `idss`, or `idpoly`. `sysc` cannot be a frequency response data model. `sysc` can be a SISO or MIMO system, except that the `'matched'` discretization method supports SISO systems only.

`sysc` can have input/output or internal time delays; however, the `'matched'`, `'impulse'`, and `'least-squares'` 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.

`Ts` — Sample time

positive scalar

Sample time, specified as a positive scalar that represents the sampling period of the resulting discrete-time system. `Ts` is in `TimeUnit`, which is the `sysc.TimeUnit` property.

`method` — Discretization method
`'zoh'` (default) | `'foh'` | `'impulse'` | `'tustin'` | `'matched'` | `'least-squares'`

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. To specify this method with frequency prewarping (formerly known as the `'prewarp'` method), use the `PrewarpFrequency` option of `c2dOptions`.
- `'matched'` — Zero-pole matching method
- `'least-squares'` — Least-squares method
- `'damped'` — Damped Tustin approximation based on the `TRBDF2` formula for sparse models only.

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

`opts` — Discretization options
`c2dOptions` object

Discretization options, specified as a `c2dOptions` object. For example, specify the prewarp frequency, order of the Thiran filter or discretization method as an option.

Output Arguments

`sysd` — Discrete-time model
dynamic system model

Discrete-time model, returned as a dynamic system model of the same type as the input system `sysc`.

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

- Includes both measured and noise components of `sysc`. The innovations variance λ of the continuous-time identified model `sysc`, 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 `sysc`. If you want to translate the covariance while discretizing the model, use `translatecov`.

`G` — Mapping of continuous initial conditions of state-space model to discrete-time initial state vector
matrix

Mapping of continuous-time initial conditions x_0 and u_0 of the state-space model `sysc` to the discrete-time initial state vector $x[0]$, returned as a matrix. The mapping of initial conditions to the initial state vector is 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.

See Also

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

Topics

"Dynamic System Models"
"Transforming Between Discrete-Time and Continuous-Time Representations"
"Continuous-Discrete Conversion Methods"

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] on page 1-184, 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] on page 1-184, 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.
`'least-squares'`	Least-squares method. Minimize the error between the frequency responses of the continuous-time and discrete-time systems up to the Nyquist frequency.

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

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

`FitOrder`

Fit order for `'least-squares'` method, specified as an integer. Specifies the order of the discrete-time model to be fitted to the continuous-time frequency response. Leave the default option '`auto`' to use the order of the continuous-time model, and change it to an integer `N` to use an N^{th}-order fit. Reducing the order helps with unstable poles or pole/zero cancellations at `z = -1`.

Default: '`auto`'

`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

Canonical state-space realization

Syntax

```
csys = canon(sys,type)
csys = canon(sys,'modal',condt)

[csys,T]= canon(___)
```

Description

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

For information on controllable and observable canonical forms, see “Canonical State-Space Realizations”.

`csys = canon(sys,'modal',condt)` specifies an upper bound `condt` on the condition number of the block-diagonalizing transformation. Use `condt` if you have close lying eigenvalues in `csys`.

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

Examples

Convert State-Space Model to Companion Canonical Form

`aircraftPitchSSModel.mat` contains the state-space matrices of an aircraft where the input is elevator deflection angle δ and the output is the aircraft pitch angle θ.

$$\begin{bmatrix}\dot{\alpha}\\ \dot{q}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}-0.313 & 56.7 & 0\\ -0.0139 & -0.426 & 0\\ 0 & 56.7 & 0\end{bmatrix}\begin{bmatrix}\alpha\\ q\\ \theta\end{bmatrix} + \begin{bmatrix}0.232\\ 0.0203\\ 0\end{bmatrix}[\delta]$$

$$y = [0\ 0\ 1]\begin{bmatrix}\alpha\\ q\\ \theta\end{bmatrix} + [0][\delta]$$

Load the model data to the workspace and create the state-space model `sys`.

```
load('aircraftPitchSSModel.mat');
sys = ss(A,B,C,D)
```

```
sys =

  A = 
            x1       x2       x3
   x1   -0.313     56.7        0
   x2  -0.0139   -0.426        0
   x3        0     56.7        0

  B =
            u1
   x1   0.232
   x2  0.0203
   x3       0

  C =
       x1  x2  x3
   y1   0   0   1

  D =
       u1
   y1   0

Continuous-time state-space model.
```

Convert the resultant state-space model `sys` to companion canonical form.

```
csys = canon(sys,'companion')
```

```
csys =

  A =
               x1          x2          x3
   x1           0           0  -1.709e-16
   x2           1           0     -0.9215
   x3           0           1      -0.739

  B =
       u1
   x1   1
   x2   0
   x3   0

  C =
            x1       x2       x3
   y1        0    1.151  -0.6732

  D =
       u1
   y1   0

Continuous-time state-space model.
```

`csys` is the companion canonical form of `sys`.

Convert State-Space Model to Modal Canonical Form

`pendulumCartSSModel.mat` contains the state-space model of an inverted pendulum on a cart where the outputs are the cart displacement `x` and the pendulum angle θ. The control input `u` is the horizontal force on the cart.

$$\begin{bmatrix}\dot{x}\\ \ddot{x}\\ \dot{\theta}\\ \ddot{\theta}\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 0\\ 0 & -0.1 & 3 & 0\\ 0 & 0 & 0 & 1\\ 0 & -0.5 & 30 & 0\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\\ \theta\\ \dot{\theta}\end{bmatrix} + \begin{bmatrix}0\\ 2\\ 0\\ 5\end{bmatrix}u$$

$$y = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\\ \theta\\ \dot{\theta}\end{bmatrix} + \begin{bmatrix}0\\ 0\end{bmatrix}u$$

First, load the state-space model `sys` to the workspace.

```
load('pendulumCartSSModel.mat','sys');
```

Convert `sys` to modal canonical form and extract the transformation matrix.

```
[csys,T] = canon(sys,'modal')
```

```
csys =
 
  A = 
             x1       x2       x3       x4
   x1         0        0        0        0
   x2         0    -0.05        0        0
   x3         0        0   -5.503        0
   x4         0        0        0    5.453
 
  B = 
           u1
   x1   1.875
   x2   6.298
   x3    12.8
   x4   12.05
 
  C = 
                x1          x2          x3          x4
   y1           16      -4.763   -0.003696    0.003652
   y2            0    0.003969    -0.03663     0.03685
 
  D = 
       u1
   y1   0
   y2   0
 
Continuous-time state-space model.

T = 4×4

    0.0625    1.2500   -0.0000   -0.1250
         0    4.1986    0.0210   -0.4199
         0    0.2285  -13.5873    2.4693
         0   -0.2251   13.6287    2.4995
```

`csys` is the modal canonical form of `sys`, while `T` represents the transformation between the state vectors of `sys` and `csys`.

Convert Zero-Pole-Gain Model to Modal Canonical Form

For this example, consider the following system with doubled poles and clusters of close poles:

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

Create a `zpk` model of this system and convert it to modal canonical form using the string `'modal'`.

```
sys = zpk([1 -1],[0 -10 -10.0001 1+1i 1-1i 1+1i 1-1i],100);
csys1 = canon(sys,'modal');
csys1.A
```

```
ans = 7×7

         0         0         0         0         0         0         0
         0    1.0000    1.0000         0         0         0         0
         0   -1.0000    1.0000    3.2459         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    4.0571
         0         0         0         0         0         0  -10.0001
```

```
csys1.B
```

```
ans = 7×1

    0.1600
   -0.0052
    0.0427
   -0.0975
    0.5319
         0
    4.0095
```

`sys` has a pair of poles at `s = -10` and `s = -10.0001`, and two complex poles of multiplicity 2 at `s = 1+i` and `s = 1-i`. As a result, the modal form `csys1` is a state-space model with a block of size 2 for the two poles near `s = -10`, and a block of size 4 for the complex eigenvalues.

Now, separate the two poles near `s = -10` by increasing the value of the condition number of the block-diagonalizing transformation. Use a value of `1e10` for this example.

```
csys2 = canon(sys,'modal',1e10);
csys2.A
```

```
ans = 7×7

         0         0         0         0         0         0         0
         0    1.0000    1.0000         0         0         0         0
         0   -1.0000    1.0000    3.2459         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
```

```
format shortE
csys2.B
```

```
ans = 7×1

    0.0000
   -0.0000
    0.0000
   -0.0000
    0.0000
    1.6267
    1.6267
```

The `A` matrix of `csys2` includes separate diagonal elements for the poles near `s = -10`. Increasing the condition number results in some very large values in the `B` matrix.

Convert System to Companion Canonical Form

The file `icEngine.mat` contains one data set with 1500 input-output samples collected at the a sampling rate of 0.04 seconds. The input `u(t)` is the voltage (V) controlling the By-Pass Idle Air Valve (BPAV), and the output `y(t)` is the engine speed (RPM/100).

Use the data in `icEngine.mat` to create a state-space model with identifiable parameters.

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

Convert the identified state-space model `sys` to companion canonical form.

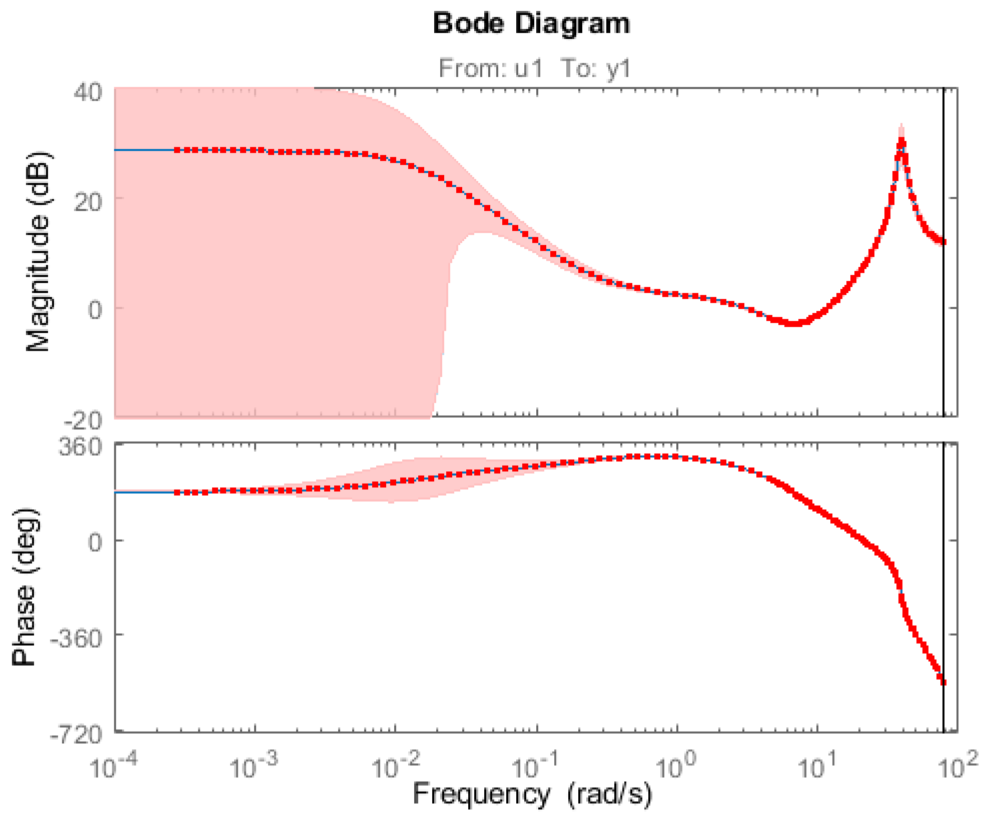
```
csys = canon(sys,'companion');
```

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

```
opt = ssestOptions;
opt.SearchOptions.MaxIterations = 0;
csys = ssest(z,csys,opt);
```

Compare frequency response confidence bounds of `sys` to `csys`.

```
h = bodeplot(sys,csys,'r.');
showConfidence(h)
```



The frequency response confidence bounds are identical.

Input Arguments

sys — Dynamic system
dynamic system model

Dynamic system, specified as a SISO, or MIMO dynamic system model. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, `ss`, or `pid` models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)

  The resulting canonical state-space model assumes

  - current values of the tunable components for tunable control design blocks.
  - nominal model values for uncertain control design blocks.
- Identified LTI models, such as `idtf`, `idss`, `idproc`, `idpoly`, and `idgrey` models.

You cannot use frequency-response data models such as `frd` models.

type — Transformation type
`'modal'` (default) | `'companion'`

Transformation type, specified as either `'modal'` or `'companion'`. If `type` is unspecified, then `canon` converts the specified dynamic system model to modal canonical form by default.

The companion canonical form is the same as the observable canonical form. For information on controllable and observable canonical forms, see “Canonical State-Space Realizations”.

- *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

$$A = \begin{bmatrix} 0 & 0 & 0 & \ldots & 0 & -\alpha_n \\ 1 & 0 & 0 & \ldots & 0 & -\alpha_{n-1} \\ 0 & 1 & 0 & \ldots & 0 & -\alpha_{n-2} \\ 0 & 0 & 1 & \ldots & 0 & -\alpha_{n-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & -\alpha_1 \end{bmatrix}$$

  The companion transformation requires that the system is controllable from the first input. The transformation to companion form is based on the controllability matrix which is almost always numerically singular for mid-range orders. Hence, avoid using it when possible.

  The companion canonical form is the same as the observable canonical form. For more information on observable and controllable canonical forms, see “Canonical State-Space Realizations”.

`condt` — Upper bound on the condition number of the block-diagonalizing transformation
`1e8` (default) | positive scalar

Upper bound on the condition number of the block-diagonalizing transformation, specified as a positive scalar. This argument is available only when `type` is set to `'modal'`.

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

Output Arguments

`csys` — Canonical state-space form of the dynamic model
`ss` model object

Canonical state-space form of the dynamic model, returned as an `ss` model object. `csys` is a state-space realization of `sys` in the canonical form specified by `type`.

`T` — Transformation matrix
matrix

Transformation matrix, returned as an n-by-n matrix, where n is the number of states. `T` is 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 an `ss` model object.

Limitations

- You cannot use frequency-response data models to convert to canonical state-space form.
- The companion form is poorly conditioned for most state-space computations, that is, the transformation to companion form is based on the controllability matrix which is almost always numerically singular for mid-range orders. Hence, avoid using it 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, `canon` 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` | `tf` | `zpk` | `ss` | `pid` | `genss` | `uss` | `idtf` | `idss` | `idproc` | `idpoly` | `idgrey`

Topics
"Canonical State-Space Realizations"

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('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.

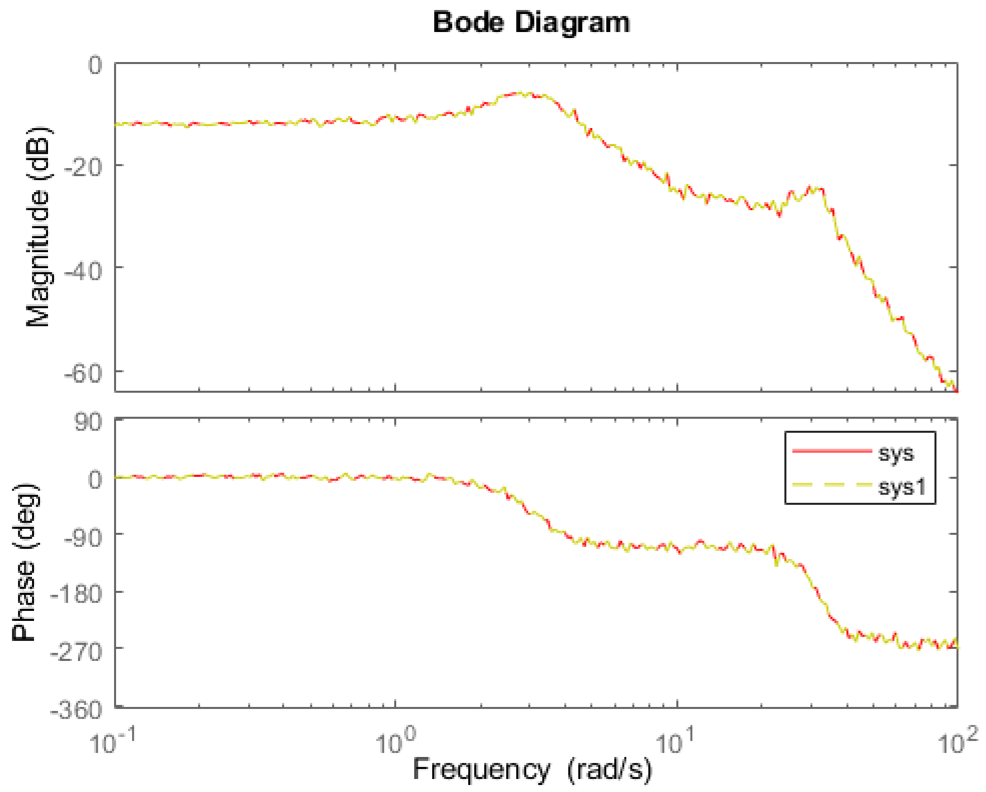
Change the frequency units.

```
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')
```

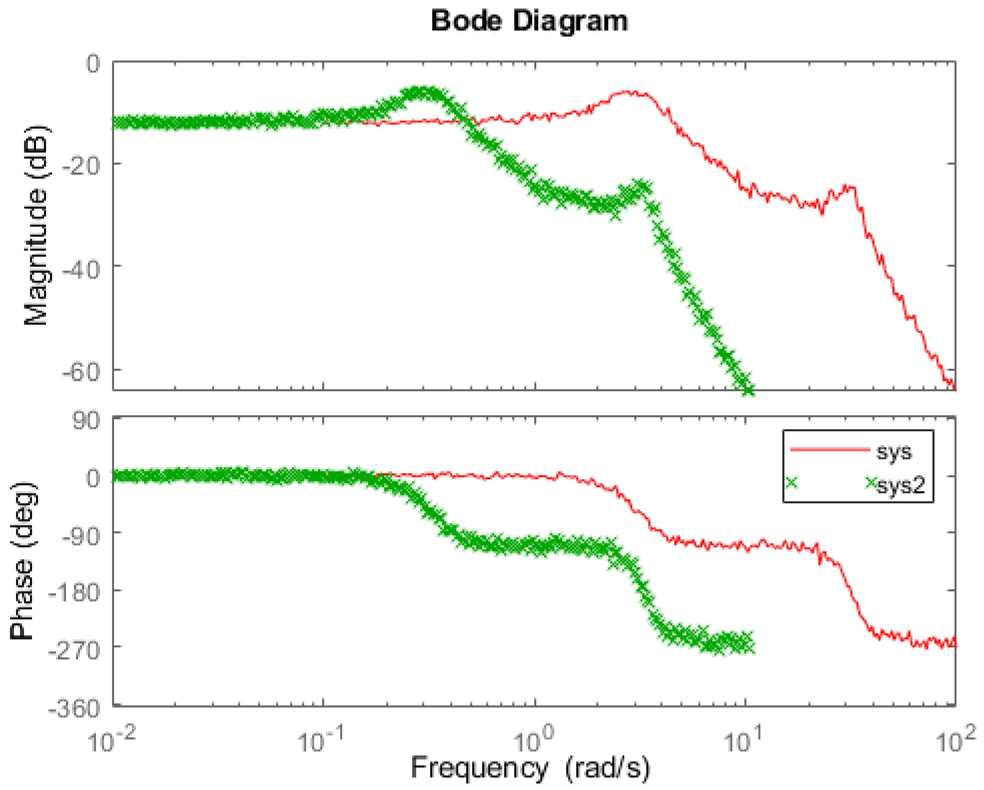


The magnitude and phase of `sys` and `sys1` match because `chgFreqUnit` command changes the units of frequency points in `sys` without modifying system behavior.

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).

Tips

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

See Also

`chgTimeUnit` | `frd` | `idfrd`

Topics

“Specify Frequency Units of Frequency-Response Data Model” (Control System Toolbox)

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.

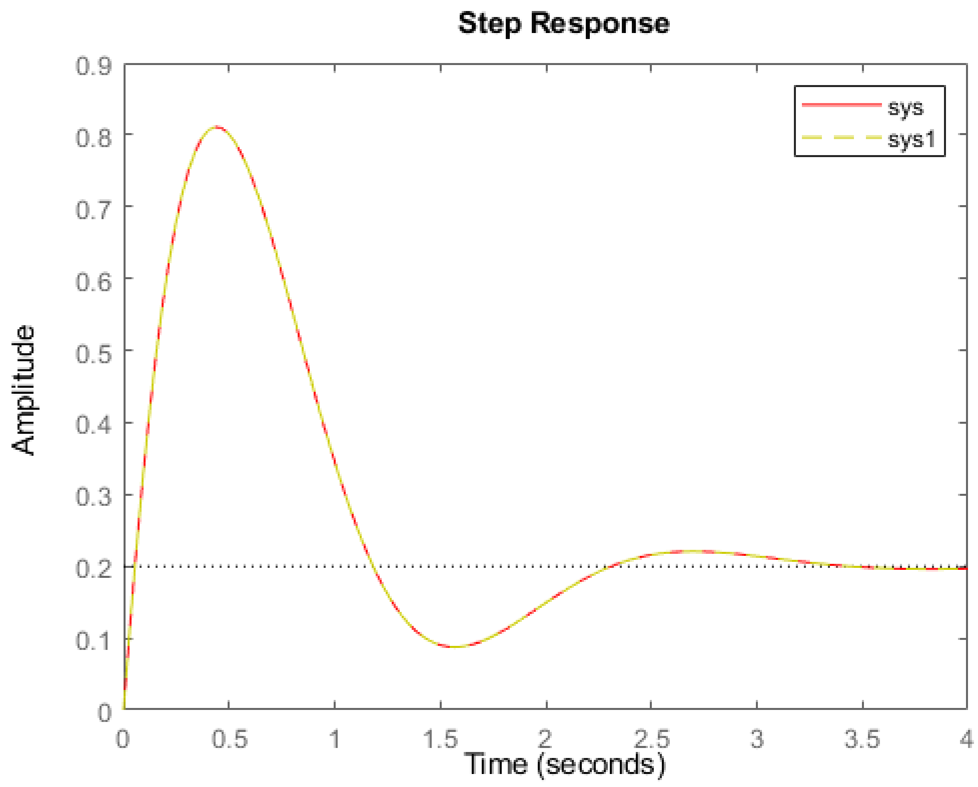
```
num = [4 2];
den = [1 3 10];
sys = tf(num,den);
```

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`.

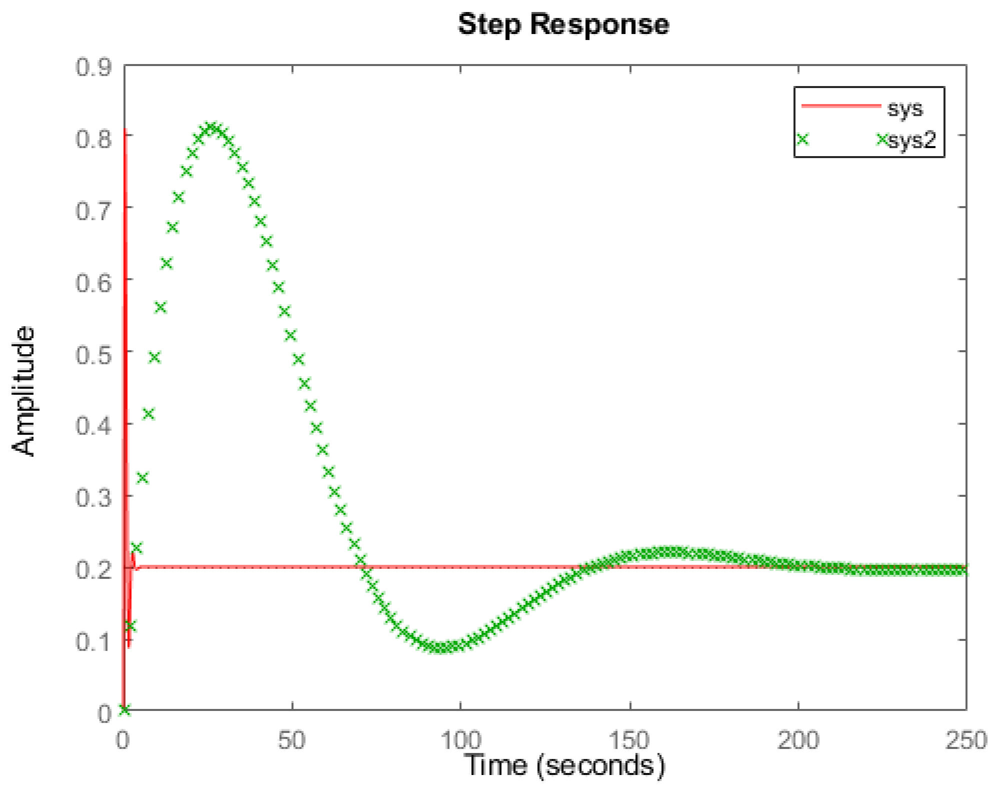
```
stepplot(sys,'r',sys1,'y--');
legend('sys','sys1');
```



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.

Tips

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

See Also

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

Topics

"Specify Model Time Units" (Control System Toolbox)

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: [2x2 double]
             EstimationMethod: 'ForgettingFactor'
             ForgettingFactor: 1
             EnableAdaptation: true
                      History: 'Infinite'
              InputProcessing: 'Sample-based'
                     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: [2x2 double]
           EstimationMethod: 'ForgettingFactor'
           ForgettingFactor: 1
           EnableAdaptation: true
                    History: 'Infinite'
            InputProcessing: 'Sample-based'
                   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.

See Also

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

Topics
"What Is Online Estimation?"

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: [2x1 double]
                StateCovariance: [2x2 double]
                   ProcessNoise: [2x2 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: [2x1 double]
                StateCovariance: [2x2 double]
                   ProcessNoise: [2x2 double]
               MeasurementNoise: 1
```

Modify the `MeasurementNoise` property of `obj2`.

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

Verify that the `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 | `particleFilter` object

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

- `extendedKalmanFilter`
- `unscentedKalmanFilter`
- `particleFilter`

Output Arguments

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

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

See Also
`predict` | `correct` | `extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter` | `initialize`

Topics
"What Is Online Estimation?"

Introduced in R2016b

compare

Compare identified model output and measured output

Syntax

```
compare(data,sys)
compare(data,sys,kstep)
compare(data,sys,LineSpec,kstep)
compare(data,sys1,...,sysN,kstep)
compare(data,sys1,LineSpec1,...,sysN,LineSpecN,kstep)
compare(___,opt)

[y,fit,ic] = compare(___)
```

Description

Plot Results

`compare(data,sys)` simulates the response of a dynamic system model, and superimposes that response over plotted measurement data. The plot also displays the normalized root mean square (NRMSE) measure of the goodness of the fit between simulated response and measurement data. Use this function when you want to evaluate a set of candidate models identified from the same measurement data, or when you want to validate a model you have selected. You can use `compare` with time-domain or frequency-domain models and data.

`compare(data,sys,kstep)` also predicts the response of `sys`, using a prediction horizon specified by `kstep`. Prediction uses output measurements as well as input measurements to project a future response. `kstep` represents the number of time samples between the timepoint of each output measurement and the timepoint of the resulting predicted response. For more information on prediction, see “Simulate and Predict Identified Model Output”.

`compare(data,sys,LineSpec,kstep)` also specifies the line type, marker symbol, and color for the model response.

`compare(data,sys1,...,sysN,kstep)` compares the responses of multiple dynamic systems on the same axes. `compare` automatically chooses the line specifications.

`compare(data,sys1,LineSpec1,...,sysN,LineSpecN,kstep)` also compares the responses of multiple systems 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. Options include initial condition handling, data offsets, and data selection. You can use this syntax with any of the previous input-argument combinations.

Return Results

`[y,fit,ic] = compare(___)` returns the model response `y`, goodness of fit value `fit`, and the initial conditions `ic`. This syntax does not generate a plot, so any line specifications are ignored.

Examples

Compare Response of Estimated Model to Measured Data

Identify a linear model and visualize the simulated model response with the data from which it was generated.

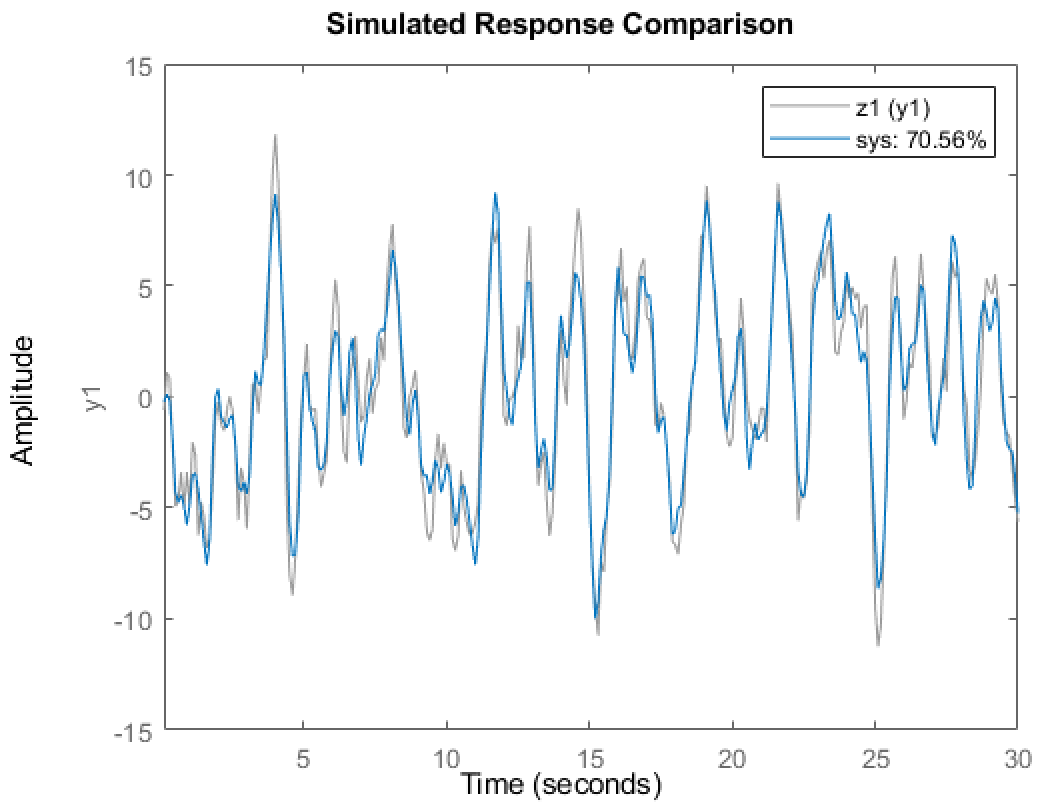
Load input/output measurements `z1`, and identify a third-order state-space model `sys`.

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

`sys` is a continuous-time identified state-space (`idss`) model.

Use `compare` to simulate the `sys` response and plot it alongside the data `z1`.

```
figure
compare(z1,sys)
```



The plot illustrates the differences between the model response and the original data. The percentage shown in the legend is the NRMSE fitness value. It represents how close the predicted model output is to the data.

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**.
- To view the confidence region for the simulated response, select **Characteristics** -> **ConfidenceRegion**.
- To specify number of standard deviations to plot, double-click the plot and open the Property Editor 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.

Compare Predicted Response of Identified Time-Domain Model to Measured Data

Identify a linear model and visualize the predicted model response with the data from which it was computed.

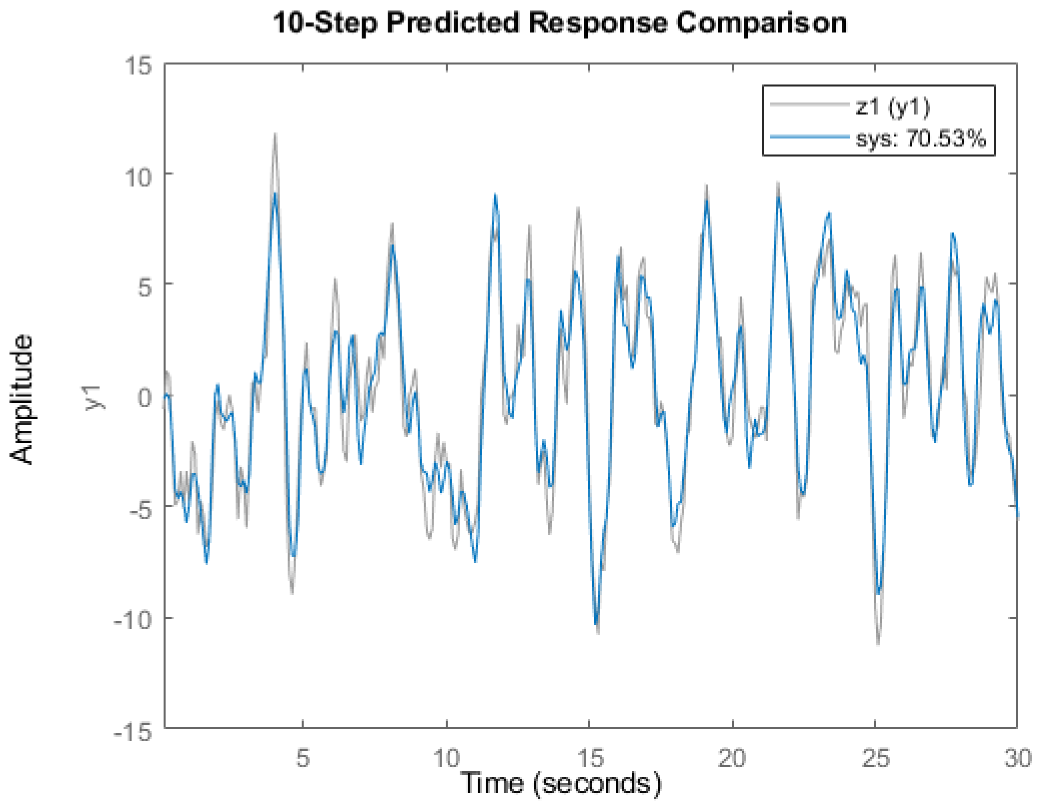
Identify a third-order state-space model using the input/output measurements in `z1`.

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

`sys` is a continuous-time identified state-space (`idss`) model.

Now use `compare` to plot the predicted response. Prediction differs from simulation in that it uses both measured input and measured output when computing the system response. The prediction horizon defines how far in the future to predict, relative to your current measured output point. For this example, set the prediction horizon `kstep` to 10 steps, and use `compare` to plot the predicted response against the original measurement data.

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



In this plot, each `sys` data point represents the predicted output associated with output measurement data that was taken at least 10 steps earlier. For instance, the point at t = 15s is based on output measurements taken at or prior to t = 5s. The calculation of this t = 15s `sys` data point also uses input measurements up to t = 15s, just as a simulation would.

The plot illustrates the differences between the model response and the original data. The percentage shown in the legend is the NRMSE fitness value. It represents how closely the predicted model output matches the data.

To change display and simulation 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. To change the prediction horizon value, or to toggle between simulation and prediction, select **Prediction Horizon** from the context menu.

Compare Multiple Identified Models to Measured Time-Domain Data

Identify several model types for the same data, and compare the results to see which best fits the data.

Load the data, which contains `iddata` object `z1` with single input and output.

```
load iddata1;
```

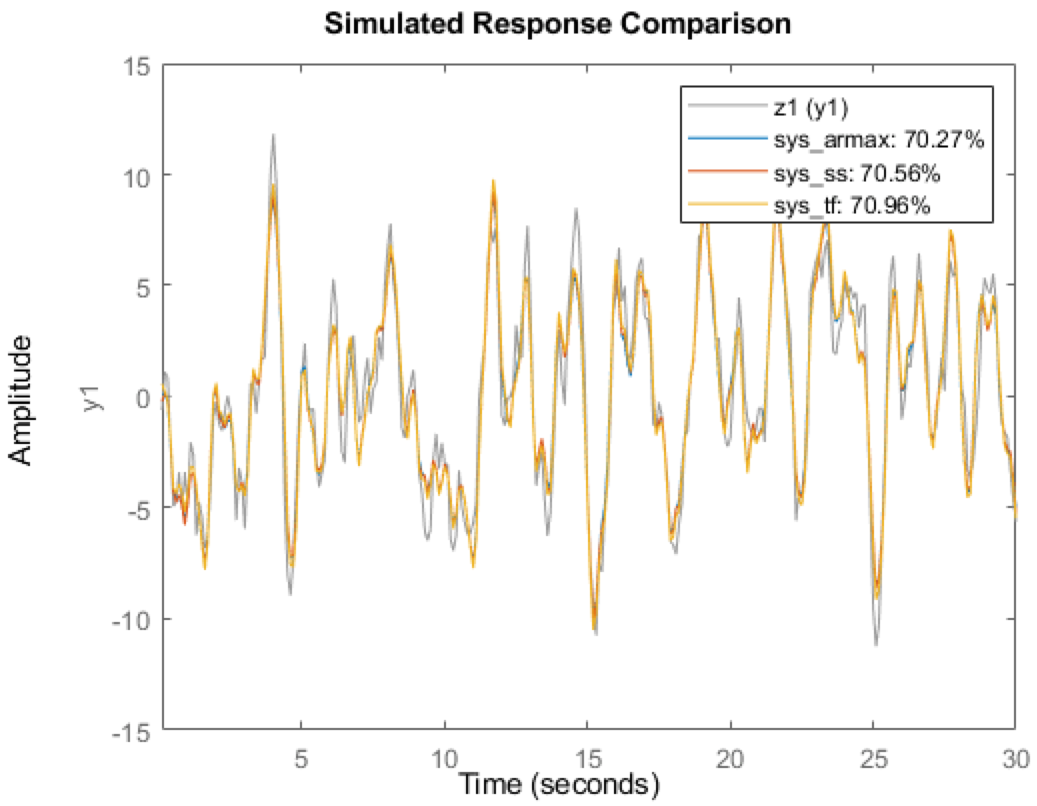
From `z1`, identify a model for each of the following linear forms:

- ARMAX (`idpoly`) of orders 2, 3, and 1, with dead time of 0
- State space (`idss`) with three states
- Transfer function (`idtf`) with three poles

```
sys_armax = armax(z1,[2 3 1 0]);
sys_ss = ssest(z1,3);
sys_tf = tfest(z1,3);
```

Using `compare`, plot the simulated responses for the three models with `z1`.

```
compare(z1,sys_armax,sys_ss,sys_tf)
```



For this set of data, along with the default settings for all the models, the transfer-function form has the best NRMSE fit. However, the fits for all models are within about 1% of each other.

You can interactively control which model responses are displayed in the plot by right-clicking on the plot and hovering over **Systems**.

Compare Multiple Estimated Models to Measured Frequency-Domain Data

Compare the outputs of multiple estimated models of differing types to measured frequency-domain data.

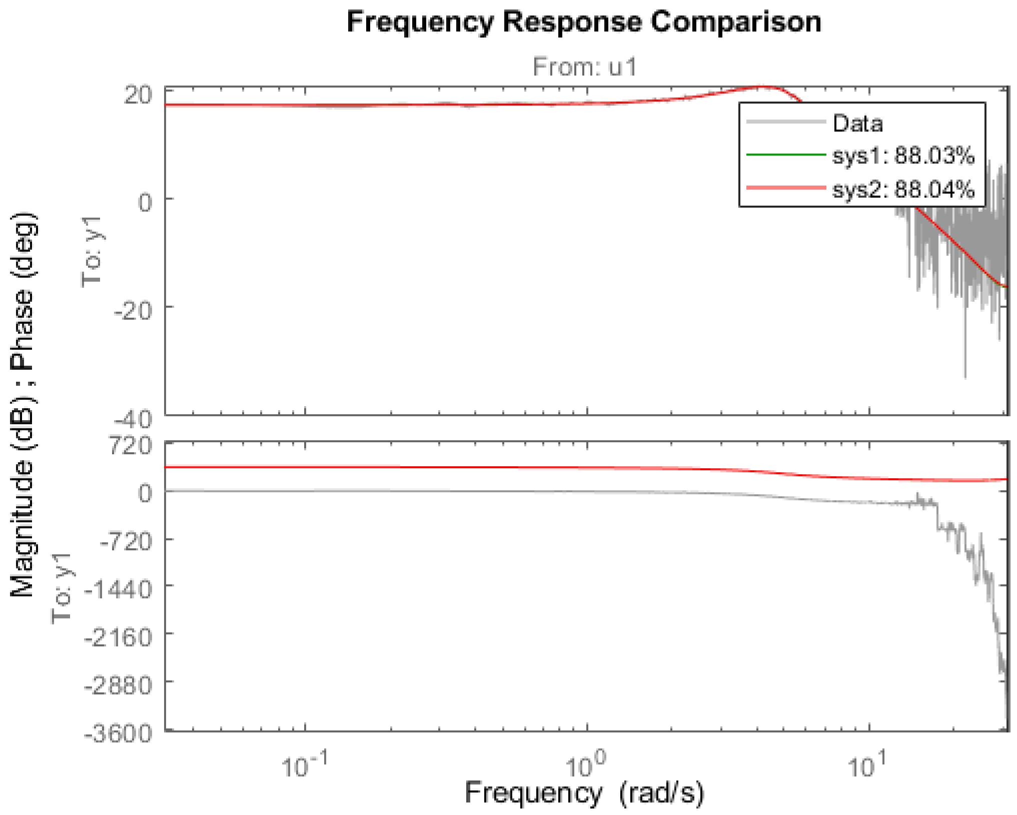
For this example, estimate a process model and an output-error polynomial from frequency response data.

```
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');
```



The two models have NRMSE fit values that are nearly equal with respect to the data from which they were calculated.

Compare Estimated Model to Data and Specify Comparison Options

Modify default behavior when you compare an estimated model to measured data.

Estimate a transfer function for measured data.

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

`sys` is a continuous-time identified transfer function (`idtf`) model.

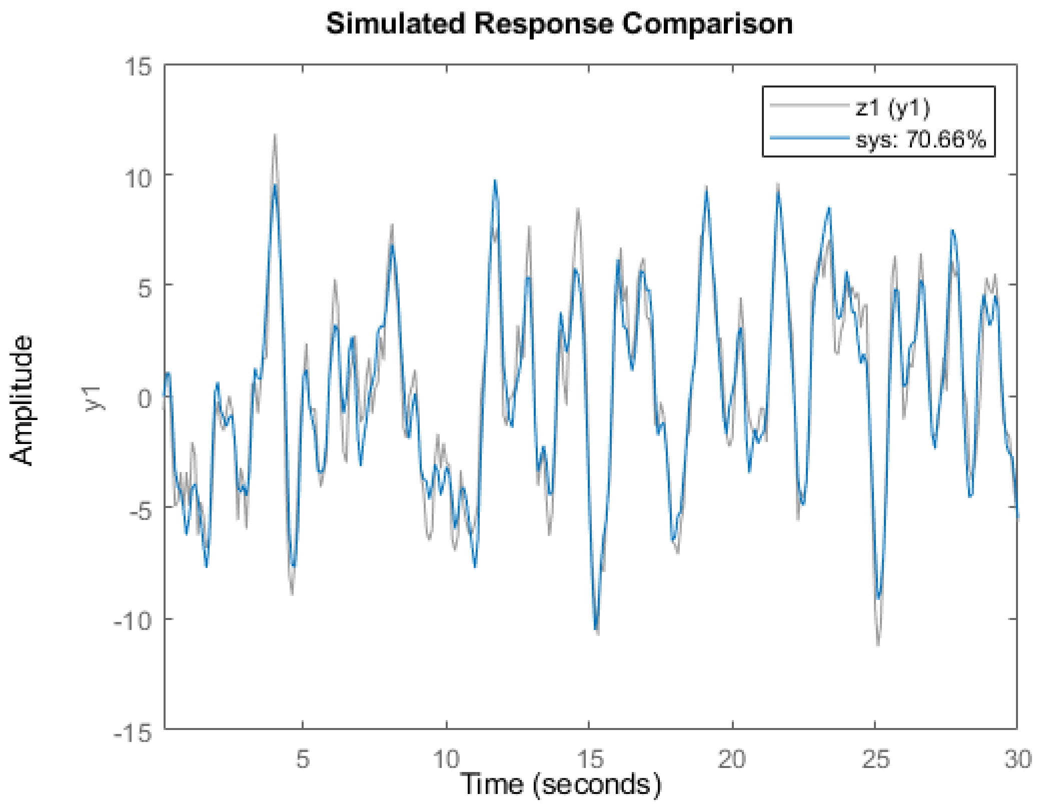
Suppose you want your initial conditions to be zero. The default for `compare` is to estimate initial conditions from the data.

Create an option set to specify the initial condition handling. To use zero for initial conditions, specify `'z'` for the `'InitialCondition'` option.

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

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

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



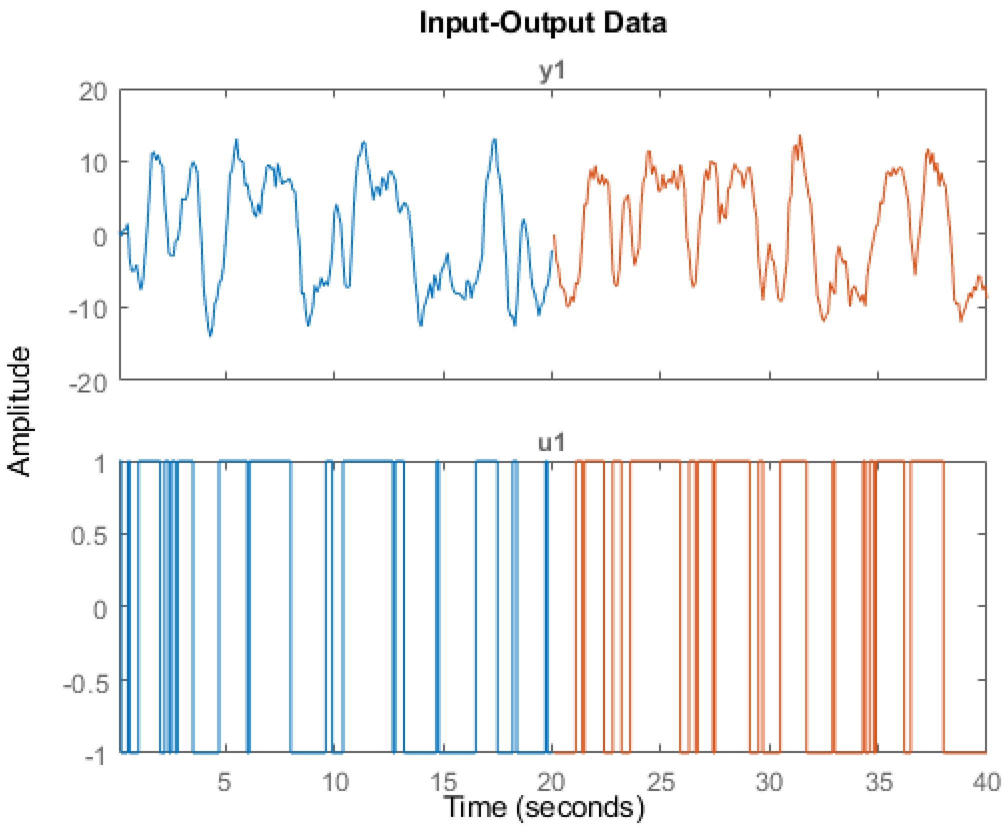
Obtain Initial Conditions

Load the data.

```
load iddata2 z2
```

Split the data into estimation and validation sets.

```
z2e = z2(1:200);
z2v = z2(201:400);
plot(z2e,z2v)
```



Estimate a state-space model and a transfer function model using the estimation data.

```
sys_ss = ssest(z2e,2);
sys_tf = tfest(z2e,2,1);
```

Use `compare` to obtain initial conditions for `sys_ss`.

```
[y_ss,fit_ss,ic_ss] = compare(z2e,sys_ss);
ic_ss
```

```
ic_ss = 2×1

   -0.0018
    0.0016
```

`ic_ss` is a numeric vector of initial states.

```
[y_tf,fit_tf,ic_tf] = compare(z2e,sys_tf);
ic_tf
```

```
ic_tf = 
  initialCondition with properties:
     A: [2x2 double]
    X0: [2x1 double]
     C: [-1.6093 5.1442]
    Ts: 0
```

`ic_tf` is an `initialCondition` object that contains, in state-space form, a model of the free response of `sys_tf` to the initial conditions. `A` and `C` contain the free-response information and `X0` contains the initial states.

Now obtain initial conditions for both models at once using the validation data.

```
[y_sstf,fit_sstf,ic_sstf] = compare(z2v,sys_ss,sys_tf);
ic_sstf
```

```
ic_sstf=2×1 cell array
    {2x1 double          }
    {1x1 initialCondition}
```

`ic_sstf` is a cell array that contains an initial state vector for `sys_ss` and an `initialCondition` object for `sys_tf`.

`compare` can provide initial conditions for an existing model with any measurement data set.

Input Arguments

`data` — Validation data

`iddata object` | `idfrd` object | `frd` object

Validation data, specified as an `iddata`, `idfrd`, or `frd` 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
- A frequency-response data (FRD) model (defined as either `idfrd` or `frd`), then `data` must also be FRD
- A parametric model (such as `idss`), then `data` can be `iddata` or FRD

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

For examples, see:

- “Compare Predicted Response of Identified Time-Domain Model to Measured Data” on page 1-206
- “Compare Multiple Estimated Models to Measured Frequency-Domain Data” on page 1-208

.

`sys` — Identified model

dynamic system model | `iddata` object | model array

Identified model, specified as a dynamic system model, an `iddata` object, or a model array.

When the time or frequency units of `data` do not match the units of `sys`, `compare` rescales `sys` to match the units of `data`.

`kstep` — Prediction Horizon — steps ahead to predict
`Inf` (default) | integer

Prediction horizon, specified as one of the following:

- `Inf` — Compare simulated response of `sys` to `data`.
- Positive finite integer — Compare predicted response of `sys` to `data`, where each predicted response point is based not only on measured input data up to that timepoint, but also on measured output data up to `kstep` timepoints earlier.

`compare` ignores `kstep` when `sys` is an `iddata` object, an FRD model, or a dynamic system with no noise component. `compare` also ignores `kstep` when using frequency response validation data.

If you specify `kstep` that is greater than the number of data samples, `compare` sets `kstep` to `Inf` and provides a warning message.

For more information on simulation and prediction, see "Simulate and Predict Identified Model Output".

For an example, see "Compare Predicted Response of Identified Time-Domain Model to Measured Data" on page 1-206.

`LineSpec` — Line style, marker, and color
character vector

Line style, marker, and color of both the line and marker, specified as a character vector, such as `'b'` or `'b+:'`.

For more information about configuring `LineSpec`, see the `Linespec` input argument of `plot`. For an example, see "Compare Multiple Estimated Models to Measured Frequency-Domain Data" on page 1-208.

`opt` — Comparison options
compareOptions object

Comparison options, specified as an option set you create using `compareOptions`.

Available options include:

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

For examples, see:

- "Compare Estimated Model to Data and Specify Comparison Options" on page 1-209
- "Resolve Fit Value Differences Between Model Identification and compare Command"

Output Arguments

y — Model response

iddata object | idfrd object | cell array | array

Model response, returned as an iddata object, an idfrd object, a cell array, or an array. The output depends on the models and data you provide, as follows:

- For a single model and single-experiment data set, y is an iddata object or idfrd object
- For multimodel comparisons, y is a cell array with one iddata or idfrd object 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 N_{sys}-by-N_{exp} cell array, where N_{sys} is the number of models, and N_{exp} is the number of experiments
- If sys is a model array, y is an array with an element corresponding to each model in sys and experiment in data. For more information on model arrays, see stack

If kstep is not specified or is Inf, then compare returns the simulated response in y.

Otherwise, compare returns the predicted response. Measured output values in data up to time $t_{n\text{-kstep}}$ are used to predict the output of sys at time t_n. For more information on simulation and prediction, see “Simulate and Predict Identified Model Output”.

The compare response computation requires specification of initial condition handling. By default, compare estimates the initial conditions to maximize the fit to data. See compareOptions for more information on how compare determines the initial conditions to use.

fit — NRMSE fitness value

vector | matrix | cell array

NRMSE fitness value indicator of how well the simulated or predicted model response matches the measurement data, returned as a vector, a matrix, or a cell array. The output depends on the models and data you provide, as follows:

- If data is an iddata object, fit is a vector of length N_y, where N_y is the number of outputs
- If data is an FRD model, fit is an N_y-by-N_u matrix, where N_u is the number of inputs in data
- For a single model and single-experiment data set, fit is a vector or matrix
- 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 N_{sys}-by-N_{exp} cell array, where N_{sys} is the number of models, and N_{exp} is the number of experiments
- If sys is a model array, fit is an array with an element corresponding to each model in sys and experiment in data

compare calculates fit (in percentage) using:

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

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

For FRD models — `compare` calculates `fit` by comparing the complex frequency response. The fits of the magnitude and phase curves shown in the `compare` plot are not computed by `compare` separately.

`ic` — Initial conditions used to compute system response
`[]` | vector | `initialCondition` object | cell array

Initial conditions used to compute system response, returned as an empty array, a vector, an `initialCondition` object, or a cell array.

For a single model and single-experiment data, the form of the output depends on the model type.

- For state-space models, `ic` is a numeric vector that contains the initial states.
- For transfer function and polynomial models, `ic` is an `initialCondition` object. An `initialCondition` represents, in state-space form, the free response of the model (*A* and *C* matrices) to the estimated initial states (x_0).
- When `sys` is an `frd` or `iddata` object, `ic` is the empty array `[]`, because initial conditions cannot be used with these objects.

For multiple model and/or experiments, the form of the output is as follows:

- For multimodel comparisons, `ic` is a cell or object array, with one vector, matrix, or `initialCondition` entry for each input model.
- For multiexperiment data, `ic` is a cell or object array, with one entry for each experiment.
- For multimodel comparisons using multiexperiment data, `ic` is an N_{sys}-by-N_{exp} cell or object array, where N_{sys} is the number of models and N_{exp} is the number of experiments.
- If `sys` is a model array, `ic` is an array with an element corresponding to each model in `sys` and experiment in `data`.

By default, `compare` uses `findstates` to estimate the initial states in `ic`. To change this behavior, set the `'InitialCondition'` option in `opt` (see `compareOptions`). If you have input/output history that immediately precedes your start point, you can set `'InitialCondition'` to that history data. `compare` then uses `data2state` to compute the end state of the history data, and thus the start state for the simulation. Other choices include setting initial conditions to zero, or to specific values that you determine previously. For more information about finding initial conditions, see “Estimate Initial Conditions for Simulating Identified Models”.

If you are using an estimation model that does not explicitly use states, `compare` first converts the model to its state-space representation and then maps the data to the initial states. `compare` then packages the initial state vector and the `A` and `C` state-space matrices into an `initialCondition` object. For an example of using `compare` with such a model, see “Obtain Initial Conditions” on page 1-210.

Tips

- The NRMSE fit result you obtain with `compare` may not precisely match the fit value reported in model identification. These differences typically arise from mismatches in initial conditions, and in the differences in the prediction horizon defaults for identification and for validation. The differences are generally small, and should not impact your model selection and validation workflow. For more information, see “Resolve Fit Value Differences Between Model Identification and compare Command”.

- `compare` matches the input/output channels in `data` and `sys` 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`. This flexibility allows you to compare multiple models which were each identified independently from different sets of input/output channels.
- The `compare` plot allows you to vary key parameters. For example, you can interactively control:

  - Whether you generate a simulated or predicted response
  - Prediction horizon value
  - Initial condition handling
  - Which experiment data you view
  - Which system models you view

  To access the controls, right-click the plot to bring up the options menu.

See Also

`compareOptions` | `sim` | `predict` | `forecast` | `goodnessOfFit` | `chgTimeUnit` | `chgFreqUnit` | `plot`

Topics

"Compare Simulated Output with Measured Validation Data"
"Validating Models After Estimation"
"Model Validation"
"Estimate Initial Conditions for Simulating Identified Models"
"Apply Initial Conditions when Simulating Identified Linear Models"

Introduced in 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 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`

Handling of initial conditions.

Specify `InitialCondition` as one of the following:

- `'z'` — Zero initial conditions.
- `'e'` — Estimate initial conditions such that the prediction error for observed output is minimized.

  For nonlinear grey-box models, only those initial states `i` that are designated as free in the model (`sys.InitialStates(i).Fixed = false`) are estimated. To estimate all the states of the model, first specify all the `Nx` states of the `idnlgrey` model `sys` as free.

  ```
  for i = 1:Nx
  sys.InitialStates(i).Fixed = false;
  end
  ```

  Similarly, to fix all the initial states to values specified in `sys.InitialStates`, first specify all the states as fixed in the `sys.InitialStates` property of the nonlinear grey-box model.

- `'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 linear models only.
- Vector or Matrix — Initial guess for state values, specified as a numerical column vector of length equal to the number of states. For multiexperiment data, specify a matrix with *Ne* columns, where *Ne* is the number of experiments. Otherwise, use a column vector to specify the same initial conditions for all experiments. Use this option for state-space (`idss` and `idgrey`) and nonlinear models (`idnlarx`, `idnlhw`, and `idnlgrey`) only.
- `initialCondition` object — `initialCondition` object that represents a model of the free response of the system to initial conditions. For multiexperiment data, specify a 1-by-N_e array of objects, where N_e is the number of experiments.

  Use this option for individual linear models only. If you are analyzing more than one model and want to specify an `initialCondition` object for each model, you must specify the object and use `compare` for each model separately.
- 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 by `compare`:

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 `[]`. The number of rows must be greater than or equal to the model order.
`Output`	Output history, specified as a matrix with *Ny* columns, where *Ny* is the number of output channels. The number of rows must be greater than or equal to the model order.

  For multiexperiment data, configure the initial conditions separately for each experiment by specifying `InitialCondition` as a structure array with *Ne* elements. To specify the same initial conditions for all experiments, use a single structure.

  The software uses `data2state` to map the historical data to states. If your model is not `idss`, `idgrey`, `idnlgrey`, or `idnlarx`, the software first converts the model to its state-space representation and then maps the data to states. If conversion of your model to `idss` is not possible, the estimated states are returned empty.
- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space (`idss` and `idgrey`) and nonlinear grey-box (`idnlgrey`) models only. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum or maximum bounds.

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 before computing the model response. After computing the model response, the software adds the offset to the response to give the final model response.

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, or particle filter and measurements

Syntax

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

Description

The `correct` command updates the state and state estimation error covariance of an `extendedKalmanFilter`, `unscentedKalmanFilter` or `particleFilter` object using measured system outputs. To implement extended or unscented Kalman filter, or particle filter, 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 1-228.

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

You create `obj` using the `extendedKalmanFilter`, `unscentedKalmanFilter` or `particleFilter` 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 the 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`, based on 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;
```

Initialize arrays to capture results of the estimation.

```
residBuf = [];
xcorBuf = [];
xpredBuf = [];
```

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 the 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. Compute the residual between the predicted and actual measurement to assess how well the filter is performing and converging. Computing the residual is an optional step. When you use `residual`, place the command immediately before the `correct` command. If the prediction matches the measurement, the residual is zero.

After you perform the real-time commands for the time step, buffer the results so that you can plot them after the run is complete.

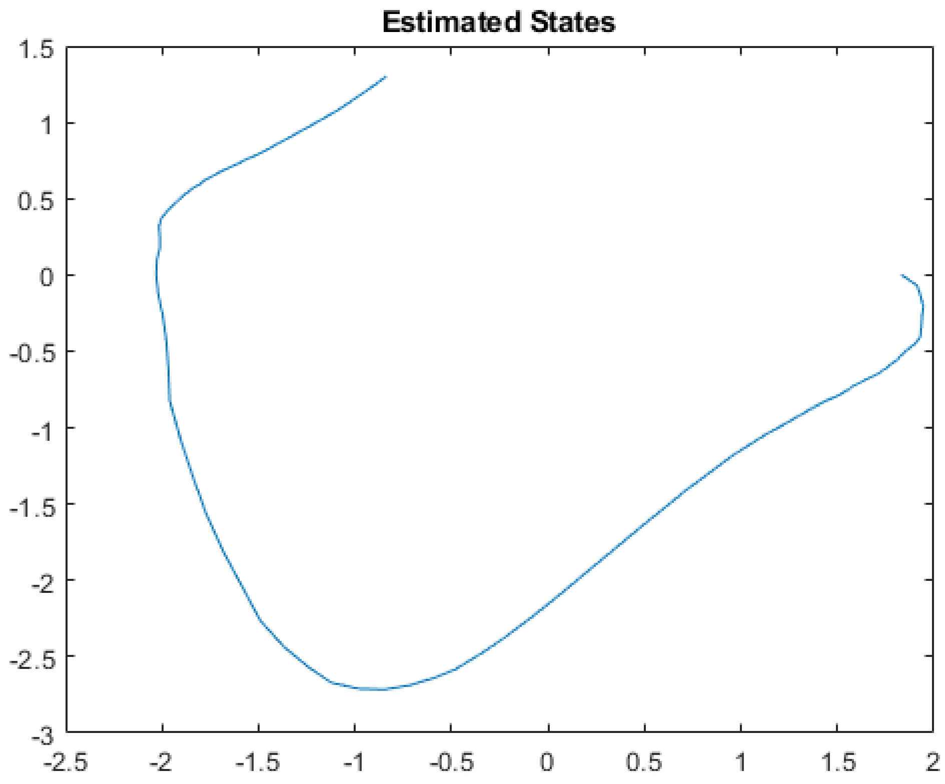
```
for k = 1:size(y)
    [Residual,ResidualCovariance] = residual(obj,y(k));
    [CorrectedState,CorrectedStateCovariance] = correct(obj,y(k));
    [PredictedState,PredictedStateCovariance] = predict(obj);

     residBuf(k,:) = Residual;
     xcorBuf(k,:) = CorrectedState';
     xpredBuf(k,:) = PredictedState';

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`. When you use the `residual` command, you do not modify any `obj` properties.

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` based on 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` based on measurements 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” on page 1-228.

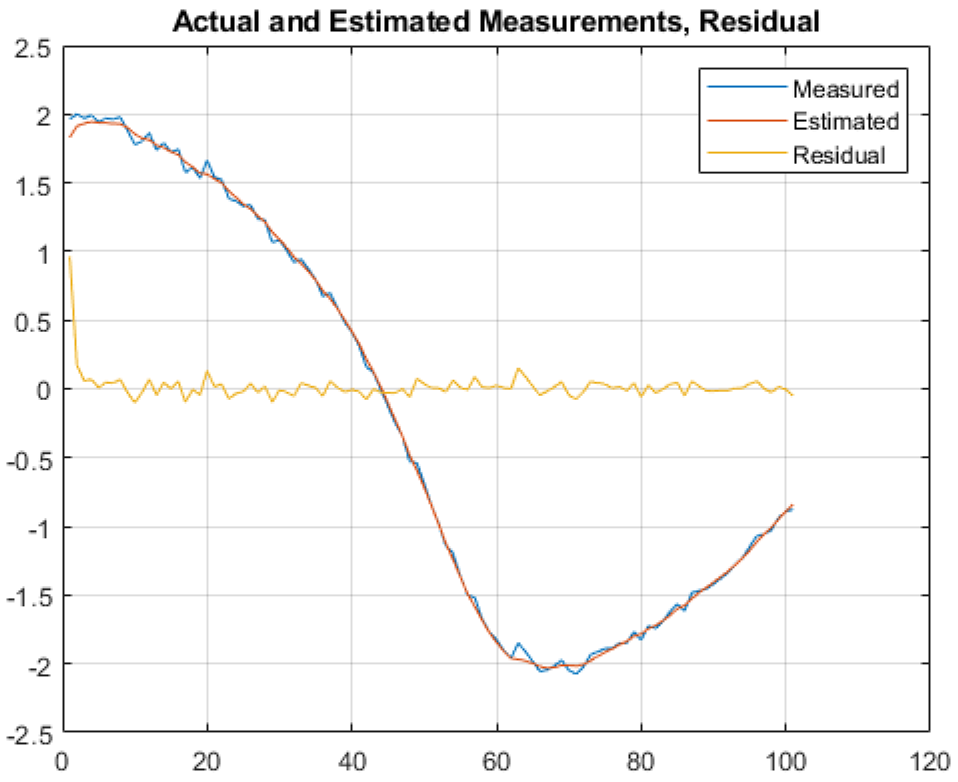
Plot the estimated states, using the postcorrection values.

```
plot(xcorBuf(:,1), xcorBuf(:,2))
title('Estimated States')
```



Plot the actual measurement, the corrected estimated measurement, and the residual. For the measurement function in `vdpMeasurementFcn`, the measurement is the first state.

```
M = [y,xcorBuf(:,1),residBuf];
plot(M)
grid on
title('Actual and Estimated Measurements, Residual')
legend('Measured','Estimated','Residual')
```



The estimate tracks the measurement closely. After the initial transient, the residual remains relatively small throughout the run.

Estimate States Online using Particle Filter

Load the van der Pol ODE data, and specify the sample time.

`vdpODEdata.mat` contains a simulation of the van der Pol ODE with nonlinearity parameter mu=1, using ode45, with initial conditions `[2;0]`. The true state was extracted with sample time `dt = 0.05`.

```
load ('vdpODEdata.mat','xTrue','dt')
tSpan = 0:dt:5;
```

Get the measurements. For this example, a sensor measures the first state with a Gaussian noise with standard deviation `0.04`.

```
sqrtR = 0.04;
yMeas = xTrue(:,1) + sqrtR*randn(numel(tSpan),1);
```

Create a particle filter, and set the state transition and measurement likelihood functions.

```
myPF = particleFilter(@vdpParticleFilterStateFcn,@vdpMeasurementLikelihoodFcn);
```

Initialize the particle filter at state `[2; 0]` with unit covariance, and use `1000` particles.

```
initialize(myPF,1000,[2;0],eye(2));
```

Pick the `mean` state estimation and `systematic` resampling methods.

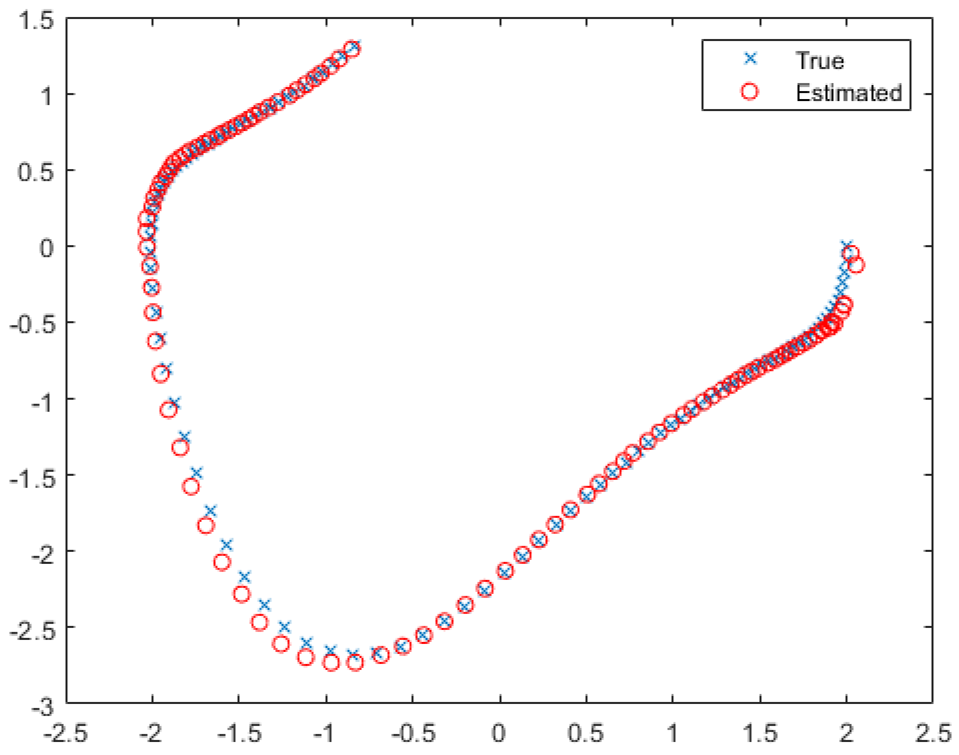
```
myPF.StateEstimationMethod = 'mean';
myPF.ResamplingMethod = 'systematic';
```

Estimate the states using the `correct` and `predict` commands, and store the estimated states.

```
xEst = zeros(size(xTrue));
for k=1:size(xTrue,1)
    xEst(k,:) = correct(myPF,yMeas(k));
    predict(myPF);
end
```

Plot the results, and compare the estimated and true states.

```
figure(1)
plot(xTrue(:,1),xTrue(:,2),'x',xEst(:,1),xEst(:,2),'ro')
legend('True','Estimated')
```



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 following 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 the next time step, given `u[k]=0.2`.

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

Retrieve the error, or *residual*, between the prediction and the measurement.

```
[Residual, ResidualCovariance] = residual(obj,0.8,0.2);
```

Input Arguments

obj — Extended or unscented Kalman filter, or particle filter object
`extendedKalmanFilter` object | `unscentedKalmanFilter` object | `particleFilter` object

Extended or unscented Kalman filter, or particle 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.
- `particleFilter` — Uses the particle 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` or `MeasurementLikelihoodFcn` 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 or the measurement likelihood function to calculate estimated outputs. You can specify multiple arguments.

For example, suppose that your measurement or measurement likelihood 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`, `unscentedKalmanFilter` and `particleFilter` 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, or particle filter object, `obj`, to implement the estimation 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 1-222 or "Estimate States Online using Particle Filter" on page 1-1265.

See Also

`predict` | `clone` | `extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter` | `initialize` | `residual`

Topics

"Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter"
"Generate Code for Online State Estimation in MATLAB"
"What Is Online Estimation?"
"Extended and Unscented Kalman Filter Algorithms for Online State Estimation"

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`.

The `cra` command first computes an autoregressive model for the input u as $A(q)u(t) = e(t)$, where e is uncorrelated (white) noise, q is the time-shift operator, and $A(q)$ is a polynomial of order `na`. The command then filters u and output data y with $A(q)$ to obtain the prewhitened data. The command then computes and plots the covariance functions of the prewhitened y and u and the cross-correlation function between them. 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 impulse response. This is also plotted 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.

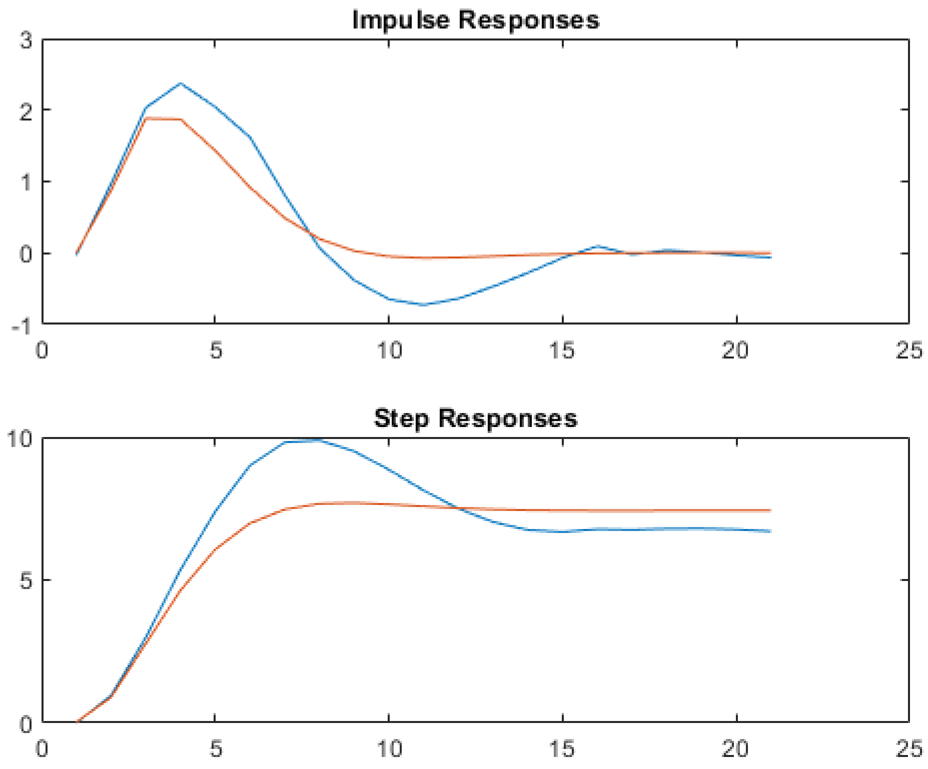
Examples

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')
```



Alternatives

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

See Also

`impulse` | `step` | `impulseest` | `spa`

Introduced before R2006a

customreg

(Not recommended) Custom regressor for nonlinear ARX models

Note The `customreg` command is not recommended. For polynomial regressors, use `polynomialRegressor` instead. For other custom regressors, use `customRegressor`. For more information, see “Compatibility Considerations”.

Syntax

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

Description

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

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

For simpler regressor expressions, specify custom regressors directly in the 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`

Property Name	Description
`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: • `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],'idLinear','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],'idLinear','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],idSigmoidNetwork,'CustomReg',C);
```

Compatibility Considerations

`customreg` is not recommended
Not recommended starting in R2021a

Starting in R2021a, create polynomial regressors using `polynomialRegressor`. To create other custom regressor forms, use `customRegressor`. Add the new regressor to the `idnlarx` `Regressors` property by using the syntax by using the syntax `model.Regressors(end+1) = new_regressor_object`.

There are no plans to remove `customreg` at this time.

See Also

`addreg` | `getreg` | `idnlarx` | `nlarx` | `polyreg`

Topics

"Identifying Nonlinear ARX Models"

Introduced in R2007a

customRegressor

Specify custom regressor for nonlinear ARX model

Description

A custom regressor represents a single user-provided formula that operates on delayed input and output variables. For example, $y(t-1)e^{u(t-1)}$ is a custom regressor that you can construct using the formula `@(x,y)x.*exp(y)`. A `customRegressor` object encapsulates a set of custom regressors. Use `customRegressor` objects when you create nonlinear ARX models using `idnlarx` or `nlarx`. You can specify `customRegressor` objects along with `linearRegressor` and `polynomialRegressor` objects and combine them into a single combined regressor set.

Creation

Syntax

```
cReg = customRegressor(Variables,Lags,Fcn)
cReg = customRegressor(Variables,Lags,Fcn,Vectorized)
```

Description

`cReg = customRegressor(Variables,Lags,Fcn)` creates a `customRegressor` object, with the output and input names in `Variables`, the corresponding lags in `Lags`, and the function handle in `Fcn`. `Fcn` sets the `VariablesToRegressorFcn` property. For example, if `Variables` contains `'y'`, `lags` contains the corresponding lag vector `[2 4]`, and the custom function is `@(x)sin(x)`, then the regressors that use `'y'` are $\sin(y(t-2))$ and $\sin(y(t-4))$.

`cReg = customRegressor(Variables,Lags,Fcn,Vectorized)` specifies whether `Fcn` can process a vector of inputs to return a vector of output values, based on the value of `Vectorized`.

Properties

`VariablesToRegressorFcn` — Custom Function
function handle

Custom function that transforms a set of delayed variables into a numeric scalar output, specified as a function handle.

Example: `@(x)sin(x)`

Example: `@(x,y)x.*exp(y)`

`Variables` — Output and input variable names
cell array of strings | `iddata` object properties

Output and input variable names, specified as a cell array of strings or a cell array that references the `OutputName` and `InputName` properties of an `iddata` object. Each entry must be a string with no special characters other than white space.

Example: `{'y1','u1'}`

Example: `[z.OutputName; z.InputName]`

Lags — Lags in each variable
cell array of non-negative integers

Lags in each variable, specified as a 1-by-n_v cell array of non-negative integer row vectors, where n_v is the total number of regressor variables. Each row vector contains n_r integers that specify the n_r regressor lags for the corresponding variable. When n_r>1 for at least one of the variables, then the software generates a regressor for every lag combination. For instance, suppose that you want to create the formula $r(t) = \sin(y_1(t–a))\cos(u_1(t–b))$, where lag a can be `1` or `2` and lag b can be `0` or `3`. Specify `Lags` as `{[1 2],[0 3]}`, which corresponds to the variables `{'y1','u1'}`. This specification creates the following set of regressors:

- `'sin(y1(t-1))*cos(u1(t))'`
- `'sin(y1(t-1))*cos(u1(t-3))'`
- `'sin(y1(t-2))*cos(u1(t))'`
- `'sin(y1(t-2))*cos(u1(t-3))'`

If a lag corresponds to an output variable of an `idnlarx` model, the minimum lag must be greater than or equal to 1.

Example: `{1 1}`

Example: `{[1 2],[1,3,4]}`

`Vectorized` — Vectorization indicator
`true` (default) | `false`

Vectorization indicator that determines whether `VariablesToRegressorFcn` is vectorized , specified as `true` or `false`.

For an example of setting this property, see “Use Absolute Value in Polynomial Regressor Set” on page 1-1240.

Example: `[true,false]`

`TimeVariable` — Name of time variable
`'t'` (default) | character array | string

Name of the time variable, specified as a valid MATLAB variable name that is distinct from values in `Variables`.

Example: `'ClockTime'`

Examples

Create Custom Regressor

Create a custom regressor that represents the formula xe^y.

Specify the input variables as `'u1'` and `'y1'` and corresponding lags of `1` and `3` delays.

```
vars = {'y1','u1'};
lags = {1 3};
```

Specify the custom function.

```
fcn = @(x,y)x.*exp(y);
```

Create the regressor.

```
cReg = customRegressor(vars,lags,fcn)
```

```
cReg =
Custom regressor: y1(t-1).*exp(u1(t-3))
    VariablesToRegressorFcn: @(x,y)x.*exp(y)
                  Variables: {'y1'  'u1'}
                       Lags: {[1]  [3]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

Create Custom Regressors in Multiple Variables

Create a set of custom regressors in three variables, all based on the formula $xy + \sin(z)$.

Specify the variable names and the lags.

```
vars = {'a','b','c'};
lags = {[1 5],[0 8],7};
```

Specify the custom function.

```
fcn = @(x,y,z)x.*y+sin(z);
```

Create the custom regressor set.

```
cReg = customRegressor(vars,lags,fcn)
```

```
cReg =
Custom regressor: @(x,y,z)x.*y+sin(z)
    VariablesToRegressorFcn: @(x,y,z)x.*y+sin(z)
                  Variables: {'a'  'b'  'c'}
                       Lags: {[1 5]  [0 8]  [7]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

`cReg` specifies regressors for all possible lag combinations.

Specify Linear, Polynomial, and Custom Regressors

Load the estimation data `z1`, which has one input and one output, and obtain the output and input names.

```
load iddata1 z1;
names = [z1.OutputName z1.InputName]
```

```
names = 1x2 cell
    {'y1'}    {'u1'}
```

Specify L as the set of linear regressors that represents $y_1(t-1)$, $u_1(t-2)$, and $u_1(t-5)$.

```
L = linearRegressor(names,{1,[2 5]});
```

Specify P as the polynomial regressor $y_1(t-1)^2$.

```
P = polynomialRegressor(names(1),1,2);
```

Specify C as the custom regressor $y_1(t-2)u_1(t-3)$. Use an anonymous function handle to define this function.

```
C = customRegressor(names,{2 3},@(x,y)x.*y)
```

```
C =
Custom regressor: y1(t-2).*u1(t-3)
    VariablesToRegressorFcn: @(x,y)x.*y
                  Variables: {'y1'  'u1'}
                       Lags: {[2]  [3]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

Combine the regressors in the column vector R.

```
R = [L;P;C]
```

```
R =
[3 1] array of linearRegressor, polynomialRegressor, customRegressor objects.
------------------------------------
1. Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1]  [2 5]}
     UseAbsolute: [0 0]
    TimeVariable: 't'

------------------------------------
2. Order 2 regressors in variables y1
               Order: 2
           Variables: {'y1'}
                Lags: {[1]}
         UseAbsolute: 0
    AllowVariableMix: 0
         AllowLagMix: 0
        TimeVariable: 't'

------------------------------------
3. Custom regressor: y1(t-2).*u1(t-3)
    VariablesToRegressorFcn: @(x,y)x.*y
                  Variables: {'y1'  'u1'}
                       Lags: {[2]  [3]}
```

```
                 Vectorized: 1
               TimeVariable: 't'


Regressors described by this set
```

Estimate a nonlinear ARX model with R.

```
sys = nlarx(z1,R)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables y1
  3. Custom regressor: y1(t-2).*u1(t-3)
  List of all regressors

Output function: Wavelet network with 1 units
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 59.73% (prediction focus)
FPE: 3.356, MSE: 3.147
```

View the full regressor set.

```
getreg(sys)
```

```
ans = 5x1 cell
    {'y1(t-1)'         }
    {'u1(t-2)'         }
    {'u1(t-5)'         }
    {'y1(t-1)^2'       }
    {'y1(t-2).*u1(t-3)'}
```

See Also

`idnlarx` | `nlarx` | `getreg` | `linearRegressor` | `polynomialRegressor`

Introduced in R2021a

d2c

Convert model from discrete to continuous time

Syntax

```
sysc = d2c(sysd)
sysc = d2c(sysd,method)
sysc = d2c(sysd,opts)
[sysc,G] = d2c( ___ )
```

Description

`sysc = d2c(sysd)` converts a the discrete-time dynamic system model `sysd` to a continuous-time model using zero-order hold on the inputs.

`sysc = d2c(sysd,method)` specifies the conversion method.

`sysc = d2c(sysd,opts)` specifies conversion options for the discretization.

`[sysc,G] = d2c( ___ )`, where `sysd` is a state-space model, returns a matrix `G` that maps the states `xd[k]` of the discrete-time state-space model to the states `xc(t)` of `sysc`.

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 - 8.407e-13
  ---------------------
  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.

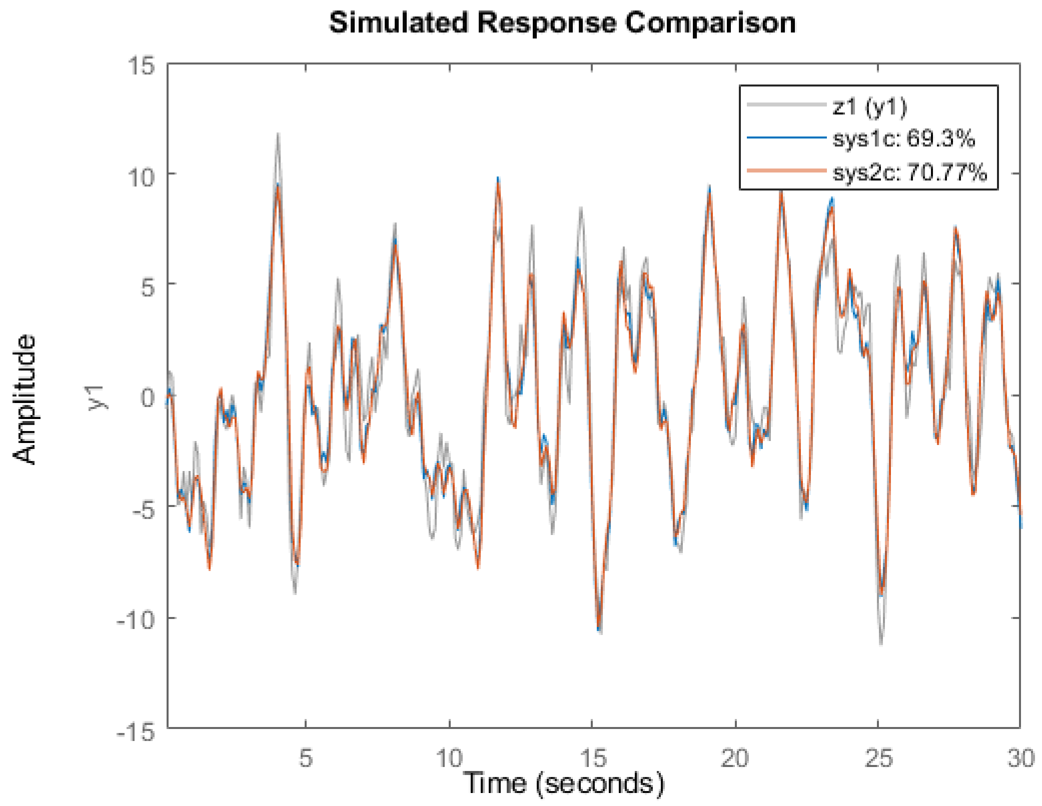
```
load iddata1
sys1d = tfest(z1,2,'Ts',0.1);
sys1c = d2c(sys1d,'zoh');
```

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');
```

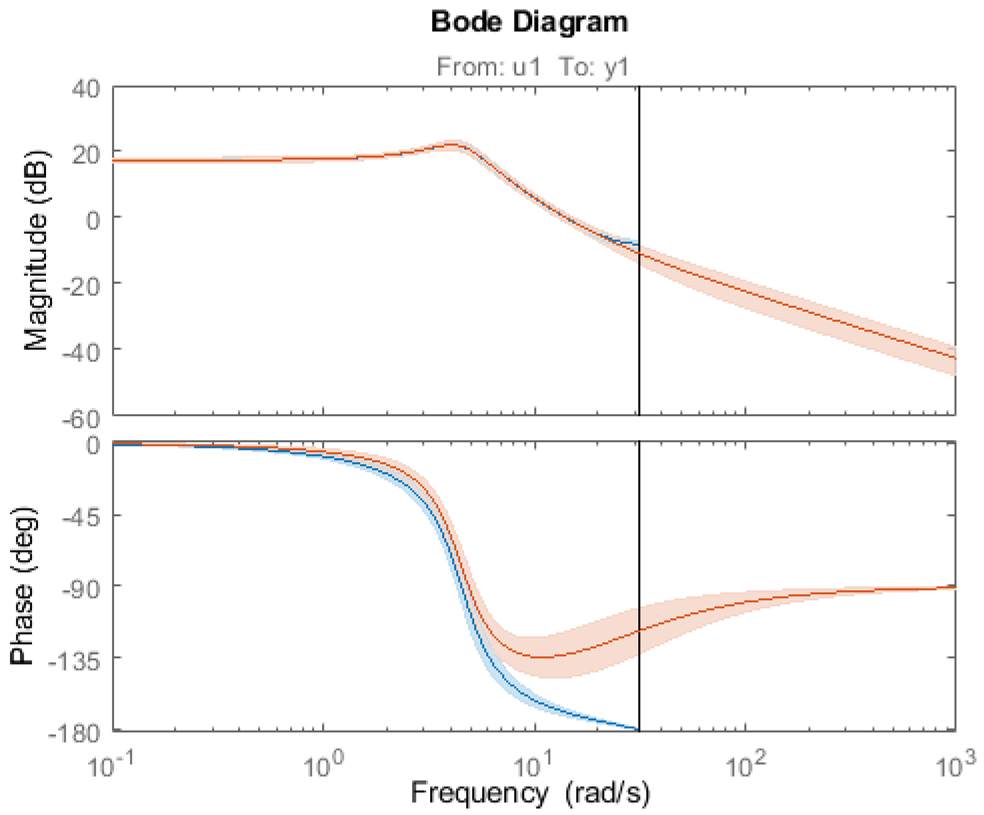
`sys1c` has no covariance information. The `d2c` operation leads to loss of covariance data of identified models.

Regenerate the covariance information using a zero iteration update with the same estimation command and estimation data.

```
opt = tfestOptions;
opt.SearchOptions.MaxIterations = 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.

Input Arguments

`sysd` — Discrete-time dynamic system

dynamic system model

Discrete-time model, specified as a dynamic system model such as `idtf`, `idss`, or `idpoly`.

You cannot directly use an `idgrey` model whose `FunctionType` is `'d'` with `d2c`. Convert the model into `idss` form first.

`method` — Discrete-to-continuous time conversion method

`'zoh'` (default) | `'foh'` | `'tustin'` | `'matched'`

Discrete-to-continuous time conversion method, specified as one of the following values:

- `'zoh'` — Zero-order hold on the inputs. Assumes that the control inputs are piecewise constant over the sampling period.

- `'foh'` — Linear interpolation of the inputs (modified first-order hold). Assumes that the control inputs are piecewise linear over the sampling period.
- `'tustin'` — Bilinear (Tustin) approximation to the derivative. To specify this method with frequency prewarping (formerly known as the `'prewarp'` method), use the `PrewarpFrequency` option of `d2cOptions`.
- `'matched'` — Zero-pole matching method (for SISO systems only). See [1] .

For information about the algorithms for each `d2c` conversion method, see “Continuous-Discrete Conversion Methods”.

`opts` — Discrete-to-continuous time conversion options
`d2cOptions` object

Discrete-to-continuous time conversion options, created using `d2cOptions`. For example, specify the prewarp frequency or the conversion method as an option.

Output Arguments

`sysc` — Continuous-time model
dynamic system model

Continuous-time model, returned as a dynamic system 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**λ.
- Does not include the estimated parameter covariance of `sysd`. If you want to translate the covariance while converting the model, use `translatecov`.

`G` — Mapping of discrete-time states of state-space model to continuous-time states
matrix

Mapping of the states `xd[k]` of the state-space model `sysd` to the states `xc(t)` of `sysc`, returned as a matrix. The mapping of the states is as follows:

$$x_c(kT_s) = G\begin{bmatrix} x_d[k] \\ u[k] \end{bmatrix}.$$

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:

$$x_c(0) = G\begin{bmatrix} x_0 \\ u_0 \end{bmatrix}.$$

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`

Topics

"Dynamic System Models"
"Transforming Between Discrete-Time and Continuous-Time Representations"
"Continuous-Discrete Conversion Methods"

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 that the control inputs are piecewise constant over the sample time `Ts`.
`'foh'`	Linear interpolation of the inputs (modified first-order hold). Assumes that 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] on page 1-250, p. 224.)

For information about the algorithms for each `d2c` conversion method, see “Continuous-Discrete Conversion Methods”.

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

Output Arguments

`opts` — Option set for `d2c`
`d2cOptions` option set

Option set for `d2c`, returned as an `d2cOptions` option set.

Examples

Specify Model Discretization Method

Consider the following discrete-time transfer function.

$$H(z) = \frac{z+1}{z^2+z+1}$$

Create the discrete-time transfer function with a sample time of 0.1 seconds.

```
Hd = tf([1 1],[1 1 1],0.1);
```

Specify the discretization method as bilinear (Tustin) approximation and the prewarp frequency as 20 rad/seconds.

```
opts = d2cOptions('Method','tustin','PrewarpFrequency',20);
```

Convert the discrete-time model to continuous-time using the specified discretization method.

```
Hc = d2c(Hd,opts);
```

You can use the discretization option set `opts` to discretize 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

For information about the algorithms for each `d2d` conversion method, see “Continuous-Discrete Conversion Methods”.

sys1 = d2d(*sys*, *Ts*, *opts*) resamples `sys` using the option set with `d2dOptions`.

Examples

Resample a Discrete-Time Model

Create the following zero-pole-gain-model with sample time 0.1 seconds.

$$H(z) = \frac{z - 0.7}{z - 0.5}$$

```
H = zpk(0.7,0.5,1,0.1);
```

Resample the model at 0.05 s.

```
H2 = d2d(H,0.05)
```

```
H2 =

  (z-0.8243)
  ----------
  (z-0.7071)

Sample time: 0.05 seconds
Discrete-time zero/pole/gain model.
```

Resample `H2` at 0.1 seconds to obtain the original model `H`.

```
H3 = d2d(H2,0.1)

H3 =

  (z-0.7)
  -------
  (z-0.5)

Sample time: 0.1 seconds
Discrete-time zero/pole/gain model.
```

Resample an Identified Discrete-Time Model

Suppose that you estimate a discrete-time output-error polynomial model with sample time commensurate with the estimation data (0.1 seconds). However, your deployment application requires a faster sampling frequency (0.01 seconds). You can use d2d to resample your estimated model.

Load the estimation data.

```
load iddata1 z1
z1.Ts

ans = 0.1000
```

z1 is an iddata object containing the estimation input-output data with sample time 0.1 seconds.

Estimate an output-error polynomial model of order [2 2 1].

```
sys = oe(z1,[2 2 1]);
sys.Ts

ans = 0.1000
```

Resample the model at sample time 0.01 seconds.

```
sys2 = d2d(sys,0.01);
sys2.Ts

ans = 0.0100
```

d2d resamples the model using the zero-order hold method.

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, use the syntax sys1 = d2d(sys,Ts,opts). For more information, see d2dOptions.
- 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 that 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.

For information about the algorithms for each d2d conversion method, see "Continuous-Discrete Conversion Methods".

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

Specify Method for Resampling a Discrete-Time Model

Create the following discrete-time transfer function with sample time 0.1 seconds.

$$H(z) = \frac{z+1}{z^2+z+1}$$

```
h1 = tf([1 1],[1 1 1],0.1);
```

Specify the discretization method as bilinear Tustin method with a prewarping frequency of 20 rad/seconds.

```
opts = d2dOptions('Method','tustin','PrewarpFrequency',20);
```

Resample the discrete-time model using the specified options.

```
h2 = d2d(h1,0.05,opts);
```

You can use the option set `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 the damping ratio, 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. The poles are sorted in increasing order of frequency values.

`[wn,zeta] = damp(sys)` returns the natural frequencies `wn`, and damping ratios `zeta` of the poles of `sys`.

`[wn,zeta,p] = damp(sys)` also returns the poles `p` of `sys`.

Examples

Display Natural Frequency, Damping Ratio, and Poles of Continuous-Time System

For this example, consider the following continuous-time transfer function:

$$sys(s) = \frac{2s^2 + 5s + 1}{s^3 + 2s - 3}.$$

Create the continuous-time transfer function.

```
sys = tf([2,5,1],[1,0,2,-3]);
```

Display the natural frequencies, damping ratios, time constants, and poles of `sys`.

```
damp(sys)
```

```
         Pole              Damping       Frequency      Time Constant
                                       (rad/seconds)      (seconds)

  1.00e+00                -1.00e+00       1.00e+00        -1.00e+00
 -5.00e-01 + 1.66e+00i     2.89e-01       1.73e+00         2.00e+00
 -5.00e-01 - 1.66e+00i     2.89e-01       1.73e+00         2.00e+00
```

The poles of `sys` contain an unstable pole and a pair of complex conjugates that lie int he left-half of the s-plane. The corresponding damping ratio for the unstable pole is -1, which is called a driving force instead of a damping force since it increases the oscillations of the system, driving the system to instability.

Display Natural Frequency, Damping Ratio, and Poles of Discrete-Time System

For this example, consider the following discrete-time transfer function with a sample time of 0.01 seconds:

$$sys(z) = \frac{5z^2 + 3z + 1}{z^3 + 6z^2 + 4z + 4}.$$

Create the discrete-time transfer function.

```
sys = tf([5 3 1],[1 6 4 4],0.01)
```

```
sys =

     5 z^2 + 3 z + 1
  ---------------------
  z^3 + 6 z^2 + 4 z + 4

Sample time: 0.01 seconds
Discrete-time transfer function.
```

Display information about the poles of `sys` using the `damp` command.

```
damp(sys)
```

```

         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.

Natural Frequency and Damping Ratio of Zero-Pole-Gain Model

For this example, create a discrete-time zero-pole-gain model with two outputs and one input. Use sample time of 0.1 seconds.

```
sys = zpk({0;-0.5},{0.3;[0.1+1i,0.1-1i]},[1;2],0.1)
```

```
sys =

  From input to output...
         z
   1:  -------
       (z-0.3)

            2 (z+0.5)
   2:  -------------------
```

```
       (z^2 - 0.2z + 1.01)

Sample time: 0.1 seconds
Discrete-time zero/pole/gain model.
```

Compute the natural frequency and damping ratio of the zero-pole-gain model `sys`.

```
[wn,zeta] = damp(sys)
```

```
wn = 3×1

   12.0397
   14.7114
   14.7114


zeta = 3×1

    1.0000
   -0.0034
   -0.0034
```

Each entry in `wn` and `zeta` corresponds to combined number of I/Os in `sys`. `zeta` is ordered in increasing order of natural frequency values in `wn`.

Compute Natural Frequency, Damping Ratio and Poles of a State-Space Model

For this example, compute the natural frequencies, damping ratio and poles of the following state-space model:

$$A = \begin{bmatrix} -2 & -1 \\ 1 & -2 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix}, \quad C = [1 \ \ 0], \quad D = [0 \ \ 1].$$

Create the state-space model using the state-space matrices.

```
A = [-2 -1;1 -2];
B = [1 1;2 -1];
C = [1 0];
D = [0 1];
sys = ss(A,B,C,D);
```

Use `damp` to compute the natural frequencies, damping ratio and poles of `sys`.

```
[wn,zeta,p] = damp(sys)
```

```
wn = 2×1

    2.2361
    2.2361


zeta = 2×1

    0.8944
    0.8944

p = 2×1 complex

  -2.0000 + 1.0000i
  -2.0000 - 1.0000i
```

The poles of `sys` are complex conjugates lying in the left half of the s-plane. The corresponding damping ratio is less than 1. Hence, `sys` is an underdamped system.

Input Arguments

`sys` — Linear dynamic system
dynamic system model

Linear dynamic system, specified as a SISO, or MIMO dynamic system model. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)

  `damp` assumes

  - current values of the tunable components for tunable control design blocks.
  - nominal model values for uncertain control design blocks.

Output Arguments

`wn` — Natural frequency of each pole
vector

Natural frequency of each pole of `sys`, returned as a vector sorted in ascending order of frequency values. 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. If the sample time is not specified, then `damp` assumes a sample time value of 1 and calculates `wn` accordingly. For more information, see “Algorithms” on page 1-260.

`zeta` — Damping ratio of each pole
vector

Damping ratios of each pole, returned as a vector sorted 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. If the sample time is not specified, then `damp` assumes a sample time value of 1 and calculates `zeta` accordingly. For more information, see “Algorithms” on page 1-260.

`p` — Poles of the dynamic system model
vector

Poles of the dynamic system model, returned as a vector sorted in the same order as `wn`. `p` is the same as the output of `pole(sys)`, except for the order. For more information on poles, see `pole`.

Algorithms

`damp` computes the natural frequency, time constant, and damping ratio of the system poles as 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	$\tau = \frac{1}{\omega_n \zeta}$	$\tau = \frac{1}{\omega_n \zeta}$

If the sample time is not specified, then `damp` assumes a sample time value of 1 and calculates `zeta` accordingly.

See Also

`eig` | `esort` | `dsort` | `pole` | `pzmap` | `zero`

Introduced before R2006a

data2state

Map past data to states of state-space and nonlinear ARX models

Syntax

```
X = data2state(sys,PastData)
[X,XCov] = data2state(sys,PastData)
```

Description

`X = data2state(sys,PastData)` maps the past data to the states of a state-space or a nonlinear ARX model `sys`. `X` contains the state values at the time instant immediately after the most recent data sample in `PastData`. The software computes the state estimates by minimizing the 1-step ahead prediction error between predicted response and output signal in `PastData`.

`data2state` is useful for continued model simulation. That is, suppose 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 Mapped States of Identified Model

Compute the mapped states of an identified model, and use the states as initial state values for model simulation.

Load estimation data.

```
load iddata3 z3
```

Estimate a second-order state-space model using the data.

```
sys = ssest(z3,2);
```

Simulate the model using the entire input signal in `z3`.

```
Input = z3(:,[],:); % |iddata| object containing only the input signal
y_all = sim(sys,Input);
```

`sim` uses zero initial conditions to compute `y_all`.

Now simulate the model using only the first-half of the input signal.

```
Input1 = Input(1:150);
y_1 = sim(sys,Input1);
```

Continue the simulation with the second-half of the input signal such that the results show no discontinuity owing to initial-condition-induced transients. To do so, first construct a past data set comprising of the input and simulated output from the first-half of the input signal. Then calculate the state values corresponding to the start of the second-half of the input signal (t = 151).

```
PastData = [y_1,Input1];
X = data2state(sys,PastData);
```

X contains 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 (second-half of the input signal).

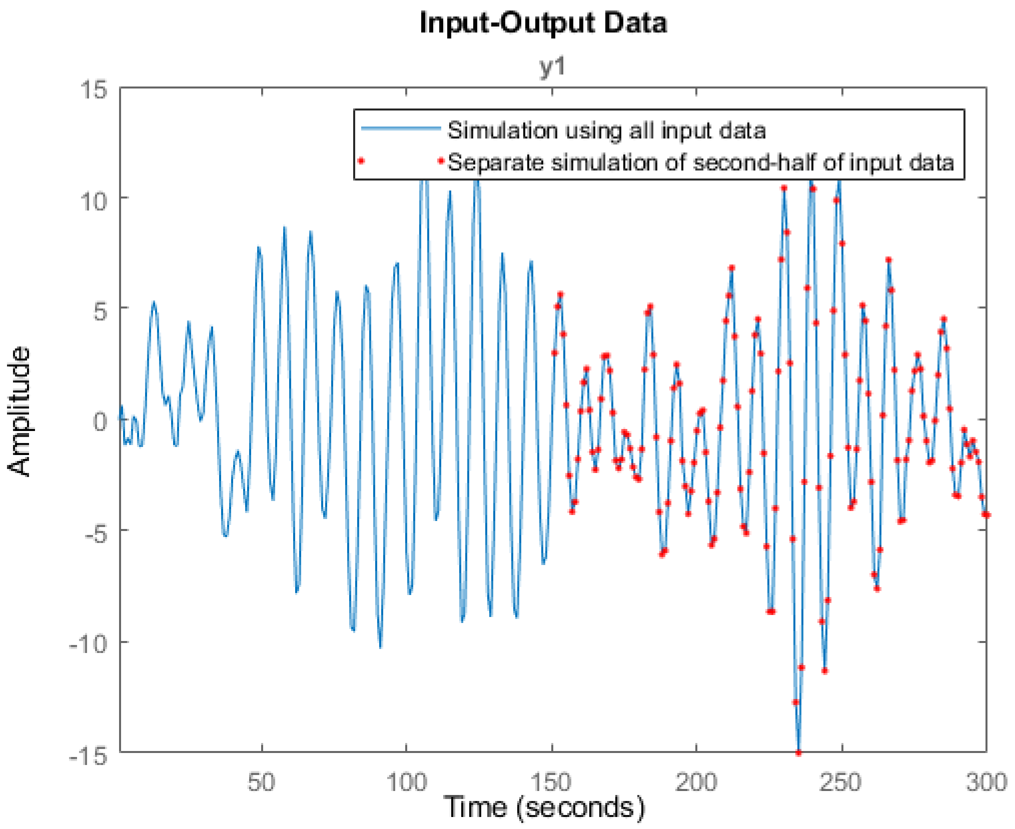
```
FutureData = Input(151:end);
```

Simulate the model using the second-half of the input signal and X as initial conditions.

```
opt = simOptions('InitialCondition',X);
y_2 = sim(sys,FutureData,opt);
```

Verify that `y_2` matches the second half of `y_all`.

```
plot(y_all,y_2,'r.')
legend('Simulation using all input data',...
    'Separate simulation of second-half of input data')
```



Calculate Mapped States and Covariance of States

Load the past data.

```
load iddata1 z1
PastData = z1;
```

Estimate an ARX model.

```
sys = arx(PastData,[1 1 0]);
```

Convert the model to a state-space model.

```
sys2 = idss(sys);
```

Calculate the mapped states and covariance of states using `PastData`.

```
[X,XCov] = data2state(sys2,PastData);
```

`X` is the state value at the time instant immediately after the most recent data sample in `PastData`.

Determine Mapped 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)],'idWaveletNetwork');
```

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,:));
```

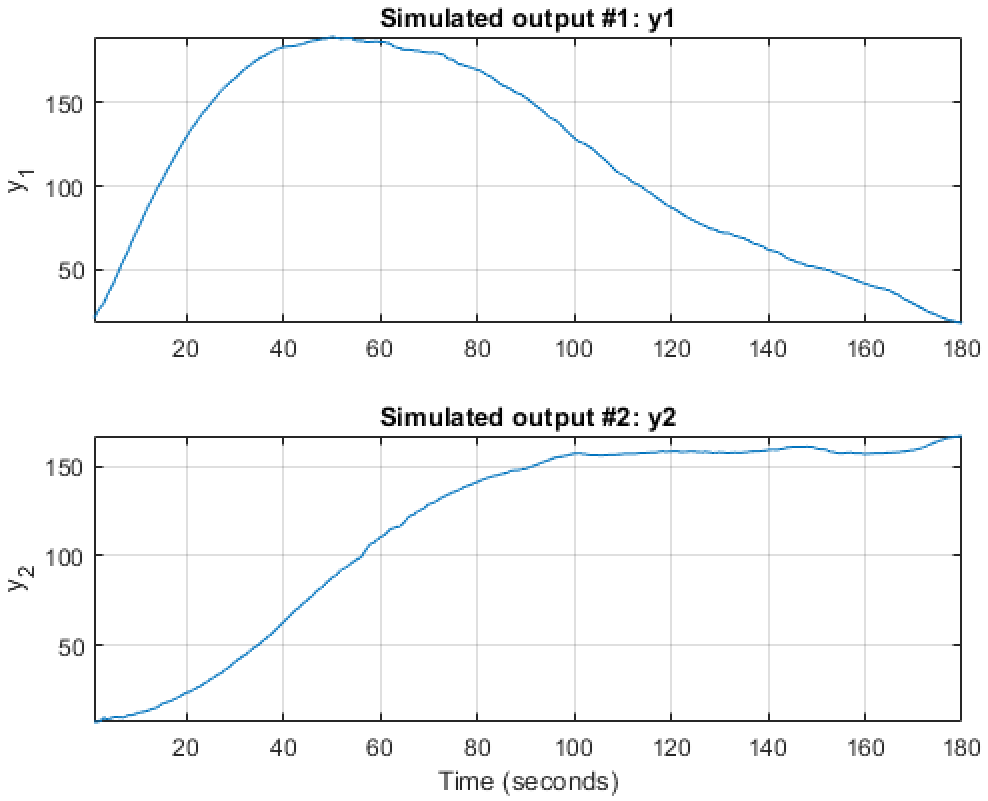
Compute the mapped states of the model.

```
X = data2state(mw1,PastData);
```

`X` is the state value at the time instant immediately after the most recent data sample in `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
idss | idgrey | idnlgrey | idnlarx

Identified model whose current states are estimated, specified as one of the following:

- State-space model (`idss`, `idgrey`, or `idnlgrey` object)
- Nonlinear ARX model (`idnlarx` object) — For a definition of the states of `idnlarx` models, see "Definition of idnlarx States" on page 1-623.

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 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. X then corresponds to the discretized (`c2d`) version of the model, where the discretization method is stored in the `InterSample`property of `PastData`.

- Structure — Specified as a structure with the following fields:
  - Input — Past input data, specified as an *N*-by-*Nu* matrix, where *N* is great than or equal to the model order.
  - Output — Past output data, specified as an *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 — Mapped states of model
row vector

Mapped states of model, returned as a row vector of size equal to the number of states. `X` contains the state value at the time instant immediately after the most recent data sample in `PastData`. That is, if `PastData` is an `iddata` object, `X` is the state value at time `t = PastData.SamplingInstants(end)+PastData.Ts`.

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 model.

See Also
`idnlarx/findop` | `findstates` | `getDelayInfo` | `sim` | `order`

Introduced in R2008a

db2mag

Convert decibels (dB) to magnitude

Syntax

```
y = db2mag(ydb)
```

Description

`y = db2mag(ydb)` returns the magnitude measurements, `y`, that correspond to the decibel (dB) values specified in `ydb`. The relationship between magnitude and decibels is $ydb = 20 * \log_{10}(y)$

Examples

Magnitude of Elements in an Array

For this example, generate a 2-by-3-by-4 array of Gaussian random numbers. Assume the numbers are expressed in decibels and compute the corresponding magnitudes.

```
rng('default');
ydb = randn(2,3,4);
y = db2mag(ydb)
```

```
y = 
y(:,:,1) =

    1.0639    0.7710    1.0374
    1.2351    1.1044    0.8602


y(:,:,2) =

    0.9513    1.5098    0.8561
    1.0402    1.3755    1.4182


y(:,:,3) =

    1.0871    1.0858    0.9858
    0.9928    0.9767    1.1871


y(:,:,4) =

    1.1761    1.0804    1.0861
    1.1772    0.8702    1.2065
```

Use the definition to check the calculation.

```
chck = 10.^(ydb/20)
chck =
chck(:,:,1) =

    1.0639    0.7710    1.0374
    1.2351    1.1044    0.8602


chck(:,:,2) =

    0.9513    1.5098    0.8561
    1.0402    1.3755    1.4182


chck(:,:,3) =

    1.0871    1.0858    0.9858
    0.9928    0.9767    1.1871


chck(:,:,4) =

    1.1761    1.0804    1.0861
    1.1772    0.8702    1.2065
```

Input Arguments

ydb — Input array in decibels
scalar | vector | matrix | array

Input array in decibels, specified as a scalar, vector, matrix, or an array. When `ydb` is nonscalar, `db2mag` is an element-wise operation.

Data Types: `single` | `double`

Output Arguments

y — Magnitude measurements
scalar | vector | matrix | array

Magnitude measurements, returned as a scalar, vector, matrix, or an array of the same size as `ydb`.

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

Compute the DC Gain of a MIMO Transfer Function

Create the following 2-input 2-output continuous-time transfer function.

$$H(s) = \begin{bmatrix} 1 & \dfrac{s-1}{s^2+s+3} \\ \dfrac{1}{s+1} & \dfrac{s+2}{s-3} \end{bmatrix}$$

```
H = [1 tf([1 -1],[1 1 3]) ; tf(1,[1 1]) tf([1 2],[1 -3])];
```

Compute the DC gain of the transfer function. For continuous-time models, the DC gain is the transfer function value at the frequency `s = 0`.

```
K = dcgain(H)
```

```
K = 2×2

    1.0000   -0.3333
    1.0000   -0.6667
```

The DC gain for each input-output pair is returned. `K(i,j)` is the DC gain from input j to output i.

Compute DC Gain of Identified Model

Load the estimation data.

```
load iddata1 z1
```

`z1` is an `iddata` object containing the input-output estimation data.

Estimate a process model from the data. Specify that the model has one pole and a time delay term.

```
sys = procest(z1,'P1D')
```

```
sys =
Process model with transfer function:
             Kp
  G(s) = ---------- * exp(-Td*s)
          1+Tp1*s

        Kp = 9.0754
       Tp1 = 0.25655
        Td = 0.068

Parameterization:
    {'P1D'}
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "z1".
Fit to estimation data: 44.85%
FPE: 6.02, MSE: 5.901
```

Compute the DC gain of the model.

```
K = dcgain(sys)
```

```
K = 9.0754
```

This DC gain value is stored in the `Kp` property of `sys`.

```
sys.Kp
```

```
ans = 9.0754
```

Tips

The DC gain is infinite for systems with integrators.

See Also

`evalfr` | `norm`

Introduced in R2012a

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:

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

`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 time-domain signals contained in `iddata` objects

Syntax

```
data_d = detrend(data)
data_d = detrend(data,Type)
[data_d,T_r] = detrend( ___ )

data_d = detrend(data,1,brkpt)
```

Description

`detrend` subtracts offsets or linear trends from time-domain input-output data represented in `iddata` objects. `detrend` either computes the trend data to subtract, or subtracts the trend that you specify.

For a more general detrending function that does not require `iddata` objects, see `detrend`.

`data_d = detrend(data)` computes and subtracts the mean value from each time-domain signal in `data`. The `iddata` objects `data_d` and `data` each contain input and output data originating from SISO, MIMO, or multiexperiment datasets.

`data_d = detrend(data,Type)` subtracts the trend you specify in `Type`. You can specify a mean-value, linear, or custom trend.

`[data_d,T_r] = detrend( ___ )` also returns the subtracted trend as a `TrendInfo` object `T_r`. You can obtain `T_r` with any of the input-argument combinations in previous syntaxes.

`data_d = detrend(data,1,brkpt)` computes and subtracts the piecewise-linear trends for data with segmented trends, using the breakpoints that you define with `brkpt`.

The second argument, which corresponds to `Type`, must be `1`.

With this syntax, you cannot retrieve the resulting piecewise-linear trend information as an output.

Examples

Remove Biases From Signals

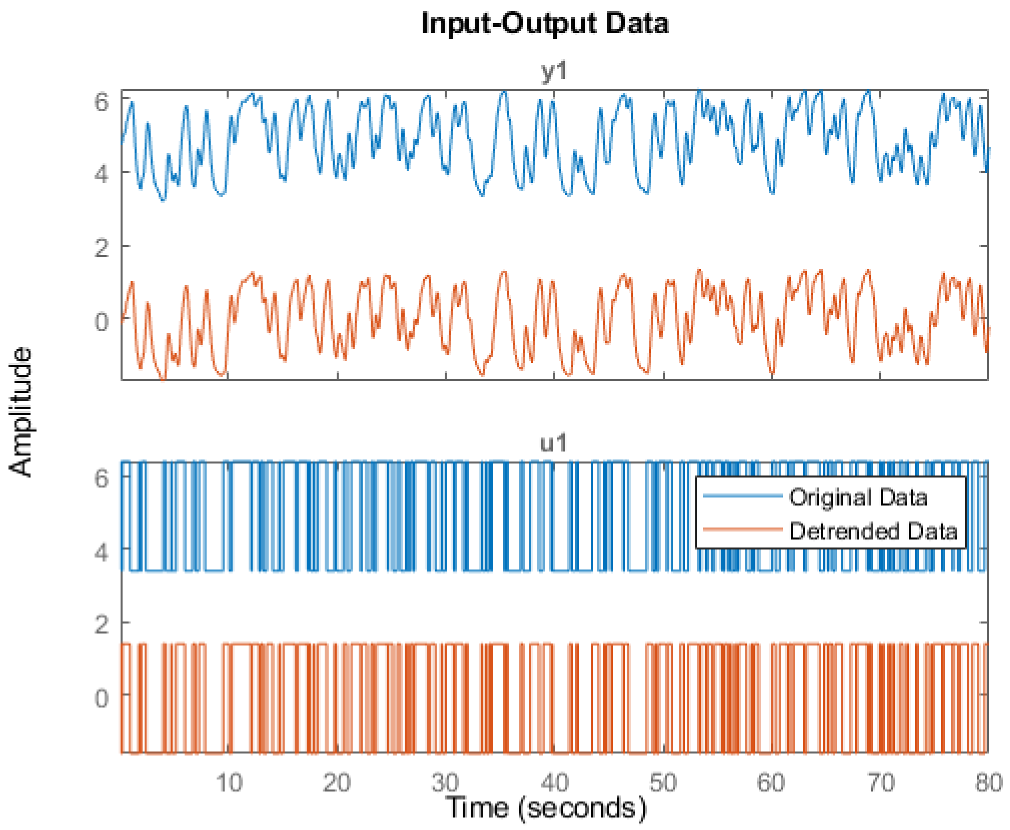
Remove biases from steady-state signals in an `iddata` object by using `detrend` to compute and subtract the mean values of the input and output.

Load the input and output time series data `y2` and `u2`. Construct the `iddata` object `data2`, using the data and a sample time of 0.08 seconds.

```
load dryer2 y2 u2
data2 = iddata(y2,u2,0.08);
```

Use `detrend` to both compute the mean values and subtract them from input and output signals. Use the input argument `Tr` to store the computed trend information. Plot the original data and detrended data together.

```
[data2_d,Tr] = detrend(data2);
plot(data2,data2_d)
legend('Original Data','Detrended Data')
```



The detrended data has shifted by about 5 units. Inspect `Tr` to obtain the precise mean values that `detrend` computed and subtracted. These values are returned in the `InputOffset` and `OutputOffset` properties.

```
Tr
```

```
Trend specifications for data "data2" with 1 input(s), 1 output(s), 1 experiment(s):
        DataName: 'data2'
     InputOffset: 5.0000
    OutputOffset: 4.8901
      InputSlope: 0
     OutputSlope: 0
```

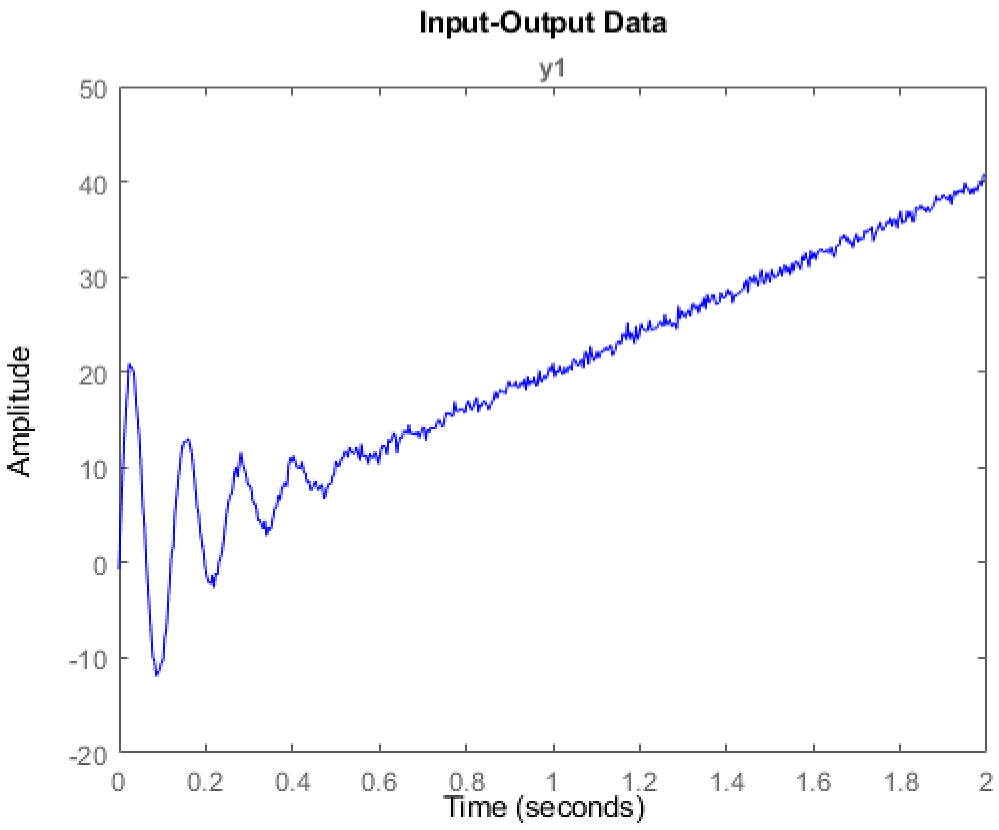
The mean of the original input is higher than the mean of the original output.

Remove Linear Trend from a Signal

Remove the linear trend from a signal in an `iddata` object, and overlay the trendline on a before-and-after data plot.

Load and plot signal data from the file `lintrend2`. For this example, only output data is provided in `iddata` object `dataL`.

```
load lintrend2 dataL
plot(dataL,'b')
```

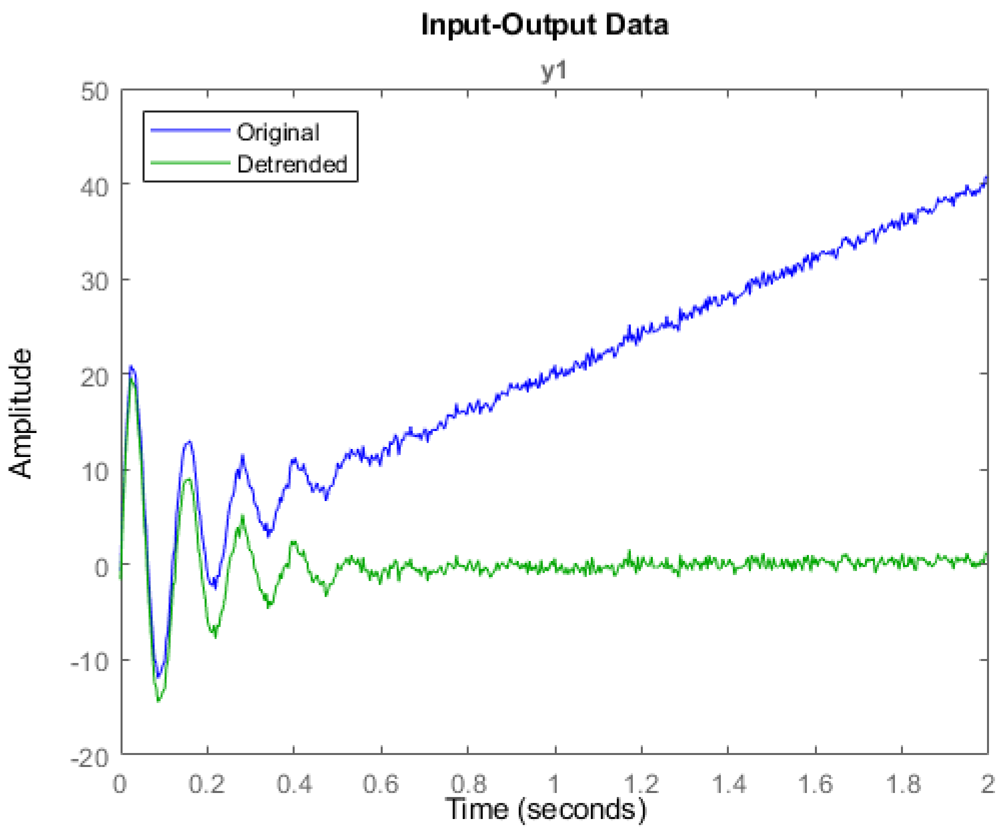


The plot shows a clear linear trend in the data. Use `detrend` linear option (`Type` = 1) to subtract the trend from the data. `detrend` fits the data and determines the linear trend to subtract. Include the `TrendInfo` object `Tr` as an output argument so you can see what `detrend` subtracts.

```
[dataL_d,Tr] = detrend(dataL,1);
```

Plot the detrended data against the original data.

```
hold on
plot(dataL_d,'g')
legend('Original','Detrended','Location','northwest')
```



The linear trend has been removed. Inspect `Tr` to get more information about the removed trend.

```
Tr
```

```
Trend specifications for data "dataL" with 0 input(s), 1 output(s), 1 experiment(s):
        DataName: 'dataL'
     InputOffset: [1x0 double]
    OutputOffset: 0.8888
      InputSlope: [1x0 double]
     OutputSlope: 19.3830
```

The `OutputOffset` and the `OutputSlope` properties provide the parameters of the line that `detrend` removed. You can reconstruct this line, and then overlay it on the before-and-after data plot. The `SamplingInstants` property of `DataL` provides the timepoints associated with the data.

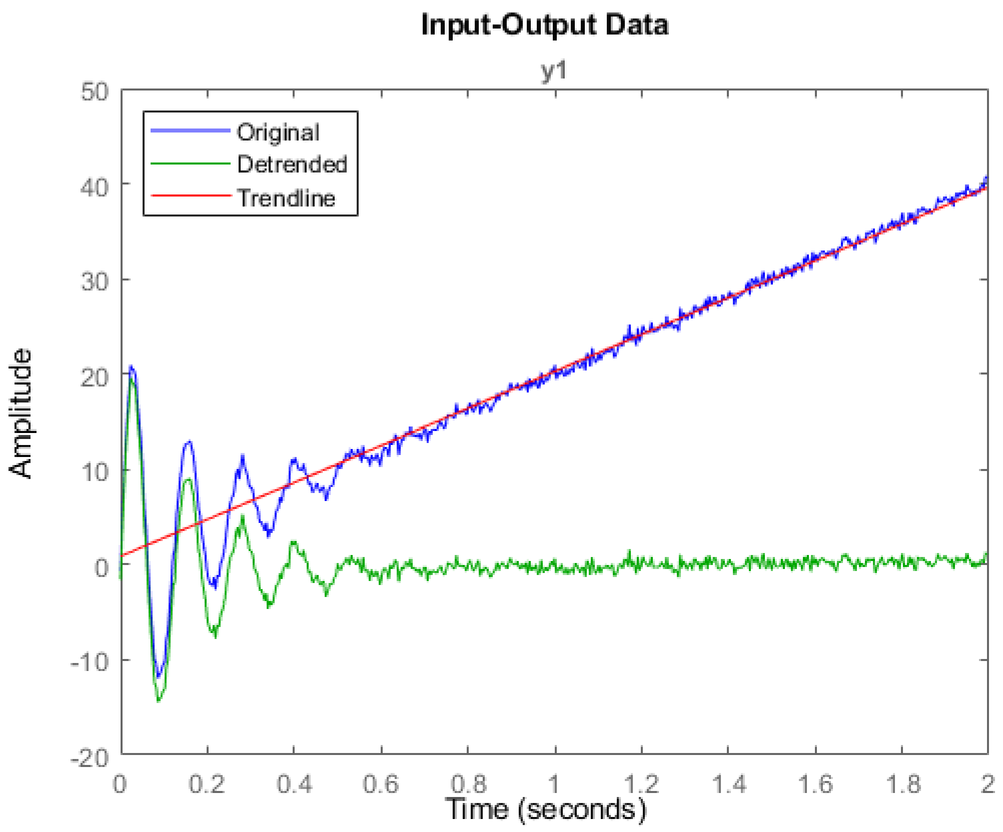
```
m = Tr.OutputSlope
```

```
m = 19.3830
```

```
b = Tr.OutputOffset
```

```
b = 0.8888
```

```
t = dataL.SamplingInstants;
TrLn = m*t+b;
plot(t,TrLn,'r')
legend('Original','Detrended','Trendline','Location','northwest')
```



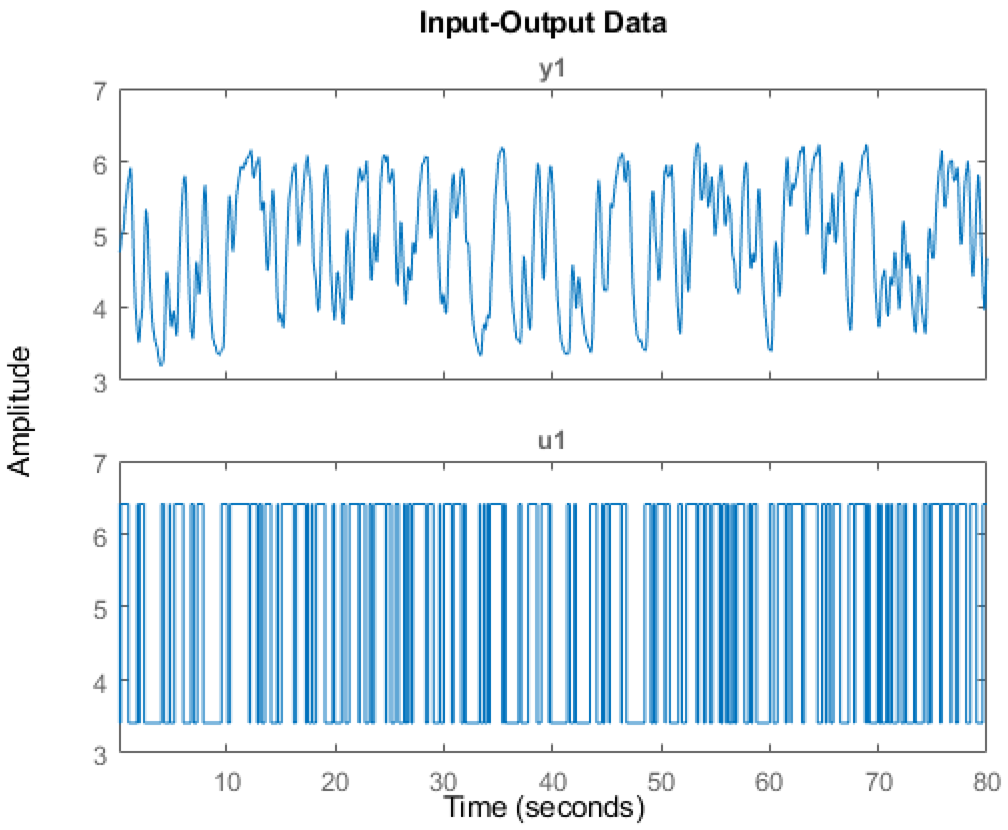
Remove Specified Offsets from Signals

Remove known offsets from an input-output signal pair contained in an `iddata object`.

`Detrend` can compute and subtract the mean values for input and output signals, resulting in zero-mean detrended signals. However, if you already know you have specific data offsets beforehand, you can have `detrend` subtract these from your signals instead. Specifying the offsets also allows you to retain a non-zero operating point in the `detrend` result.

Load SISO data containing vectors `u2` and `y2`. For this example, suppose that you know both signals have an offset of 4 from the expected operating point of 1. Combine these vectors into an `iddata object`, using a sample time of 0.08 seconds, and plot it.

```
load dryer2 u2 y2
data = iddata(y2,u2,0.08);
plot(data)
```



The known offset of 4 (from operating point 1) is visible in the plots. You can construct a `TrendInfo` object to capture this offset, using the function `getTrend`.

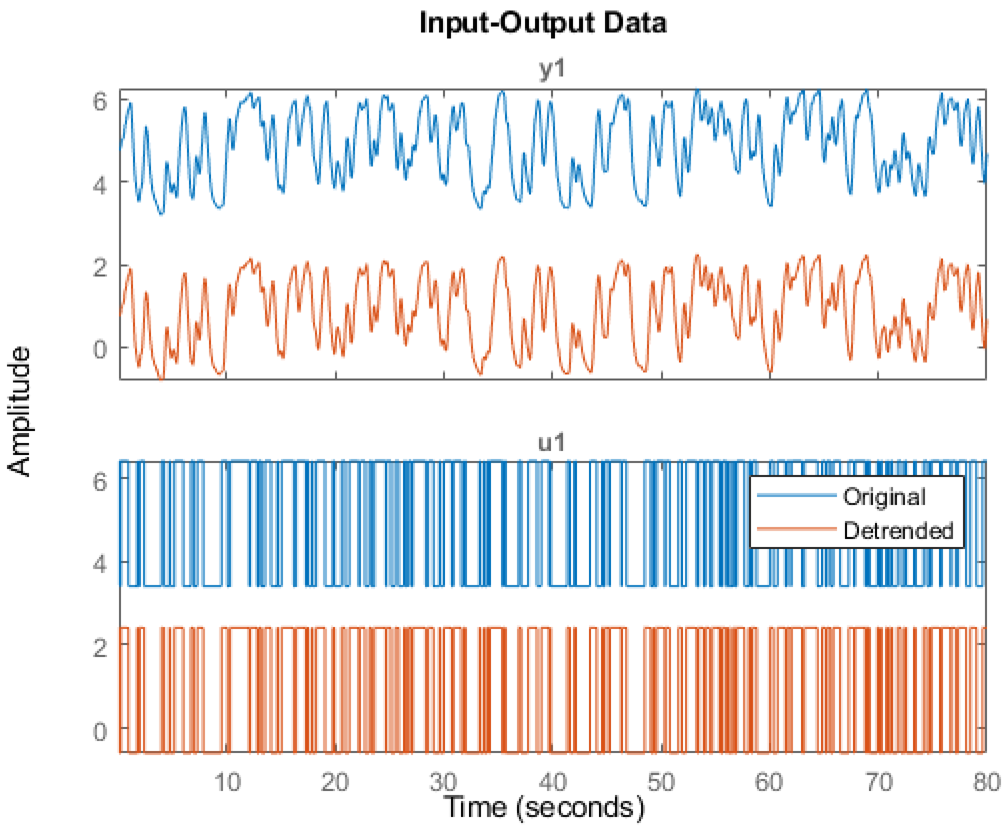
Create the `TrendInfo` object, and then set its offset properties.

```
T = getTrend(data);
T.InputOffset = 4;
T.OutputOffset = 4
```

```
Trend specifications for data "data" with 1 input(s), 1 output(s), 1 experiment(s):
        DataName: 'data'
     InputOffset: 4
    OutputOffset: 4
      InputSlope: 0
     OutputSlope: 0
```

Use `detrend` to subtract the offset from the data. Plot it alongside the original data.

```
data_d = detrend(data,T);
hold on
plot(data_d)
legend('Original','Detrended')
```



The offset of 4 has been removed.

Remove Segmented Linear Trends from Signals by using Breakpoints

Detrend data with segmented piecewise-linear trends by specifying breakpoints to delimit the segments.

Most of the `detrend` syntaxes assume and compute a single trend for each of the signals. However, in some cases there are discontinuities in the linear trends, caused by test configuration changes, environmental conditions, or other influences. When the signal displays such segmentation, you can have `detrend` operate on the test segments independently. To do so, specify breakpoints in the `brkpt` input argument. These are the indices to the timepoints in the signal at which linear trends change slope.

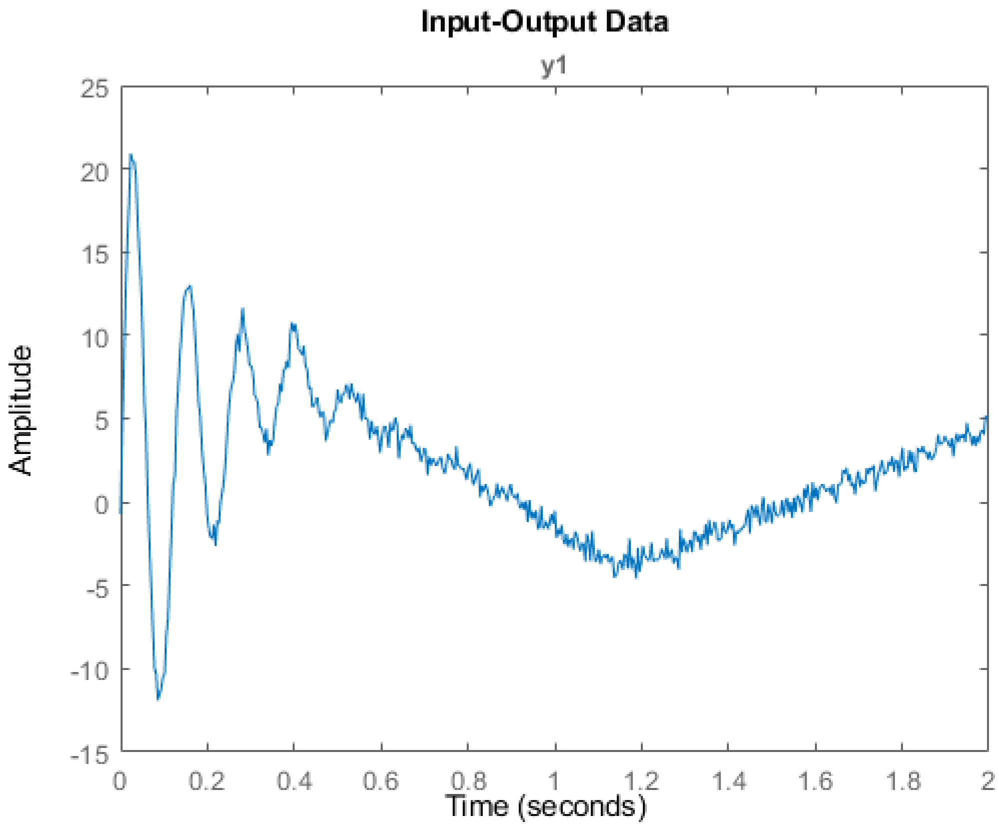
You may know these breakpoints up front, based on changes that you know occurred during data collection. Alternatively, you may need to approximate them by inspecting the data itself.

Load the data, inspect its structure and contents, and plot it. This data consists of output data only in the `iddata` object `dataLb2`.

```
load brkTrend dataLb2
dataLb2
```

```
dataLb2 =
Time domain data set with 512 samples.
Sample time: 0.00390625 seconds

Outputs      Unit (if specified)
   y1
```

```
plot(dataLb2)
```



For this example, the data has known breakpoints at indices [100 300]. Applying the sample time (property `Ts`), these breakpoints correspond to the actual timepoints as follows:

```
brkpt=[100 300];
brkpt_time = brkpt*dataLb2.Ts
```

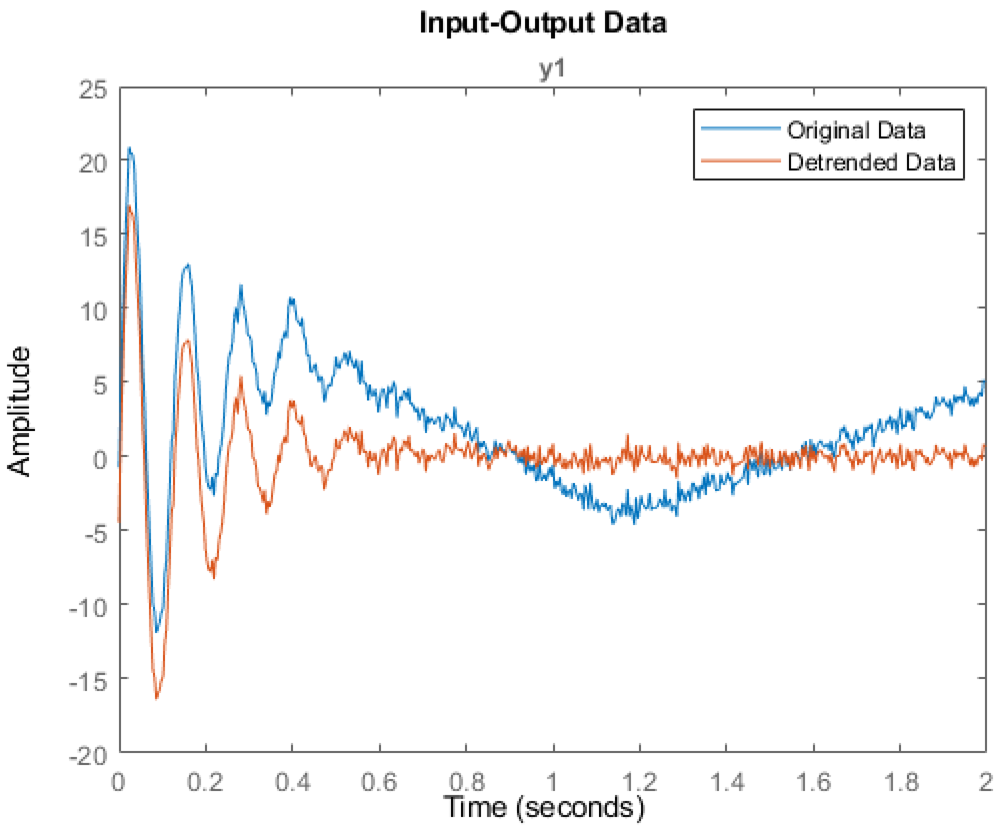
```
brkpt_time = 1×2

    0.3906    1.1719
```

Detrend the data using `brkpt`.

```
dataLb2_d = detrend(dataLb2,1,brkpt);
```

Plot the original and detrended data.

```
plot(dataLb2,dataLb2_d)
legend('Original Data','Detrended Data')
```



The linear trend segments have been removed.

Detrend Multiexperiment Signals using Multiple-Breakpoint Sets

Apply a unique set of breakpoints to each experiment when you detrend a Multiexperiment dataset.

Experiments within a multiexperiment dataset may contain unique linear trending discontinuities. You can apply a unique set of breakpoints to each experiment by expressing them in a cell array.

Load the data, which consists of:

- `datmult`, a multiexperiment `iddata` object containing three experiments (output only)
- `bp`*n* vectors, which provide known breakpoints for each experiment in the form of indices to timepoints

```
load multiexpdt datmult bp1 bp2 bp3
datmult
```

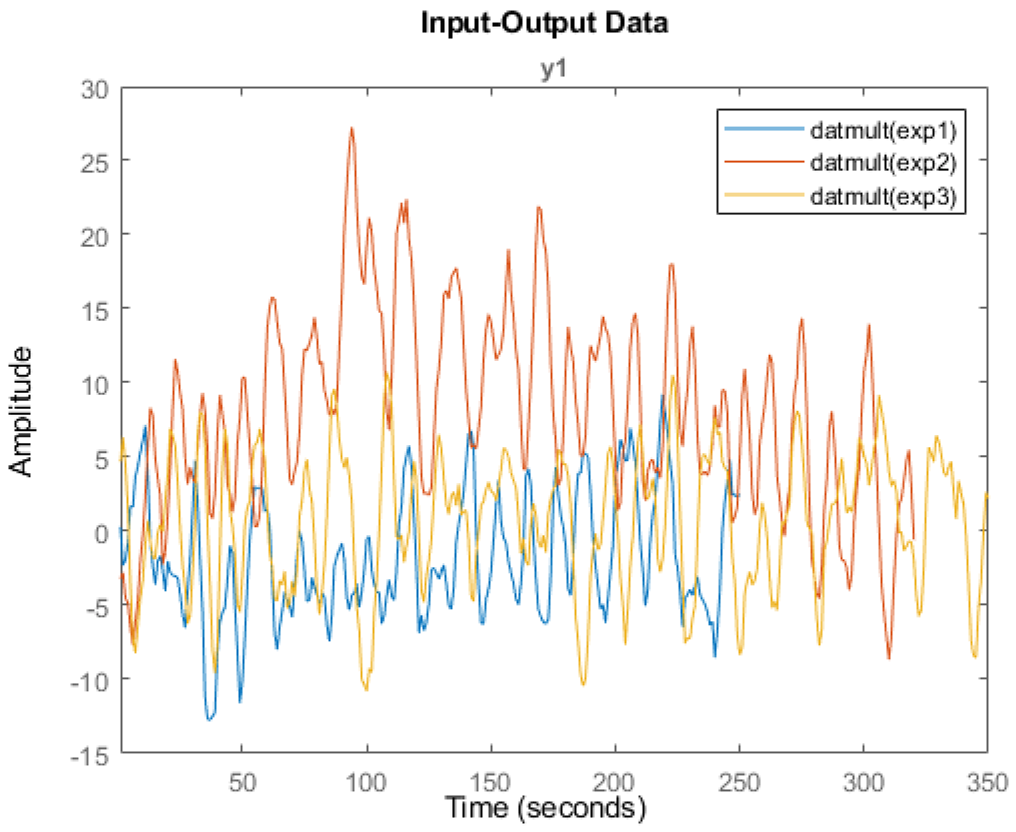
```
datmult =
Time domain data set containing 3 experiments.

Experiment    Samples      Sample Time
   exp1         250            1
   exp2         320            1
   exp3         350            1

Outputs      Unit (if specified)
   y1
```

```
bp1,bp2,bp3
```

```
bp1 = 1×2

    50   200


bp2 = 100

bp3 =

     []
```

Plot the data. There are significant differences among the streams, and they drift at different rates from zero mean.

```
plot(datmult)
legend
```



For this set of experiments, it is known that there is unique trending for each run and unique discontinuities indicated by the bp vectors.

`detrend` can incorporate these unique characteristics if the `bp` information is provided as a cell array.

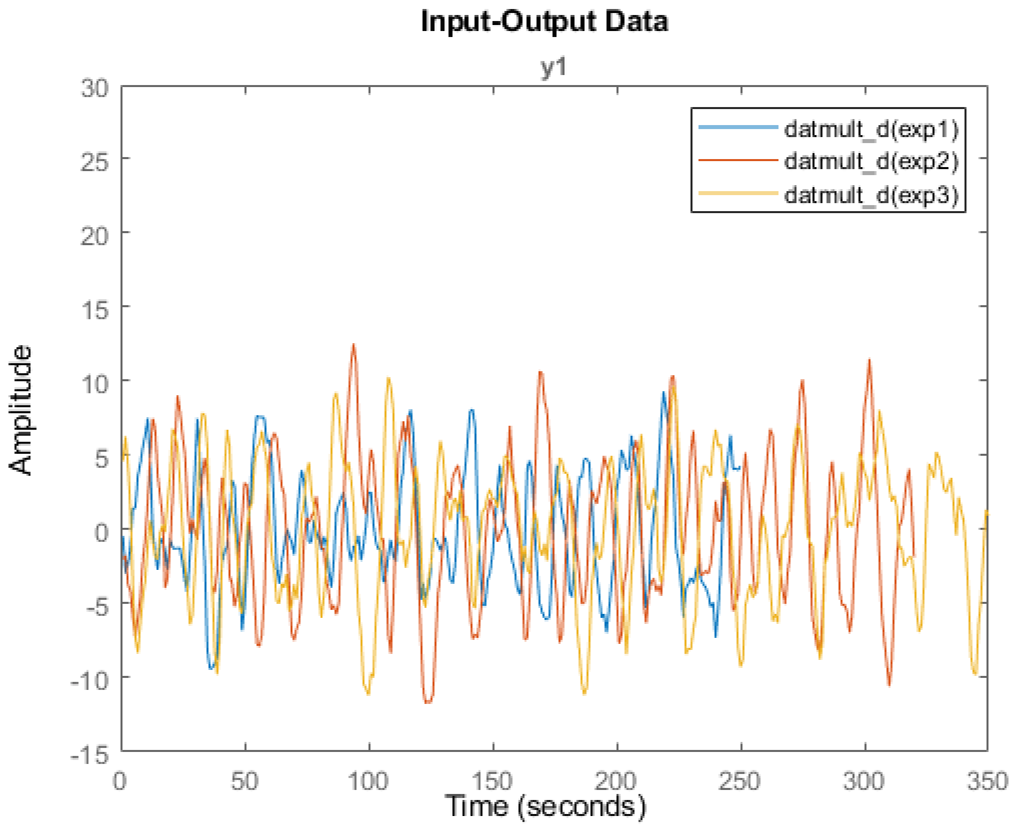
Construct the cell array.

```
bpcell = {bp1;bp2;bp3}
```

```
bpcell=3×1 cell array
    {[  50 200]}
    {[     100]}
    {0x0 double}
```

Apply `detrend` and plot the result, using the same scale as the original plot.

```
datmult_d = detrend(datmult,1,bpcell);
figure
plot(datmult_d)
axis([0,inf,-15,30])
legend
```



The experimental data are now better aligned, and do not drift significantly away from zero mean.

Detrend Input and Output Signals Separately

Apply different trend types to the input and output signals contained in an `iddata` object.

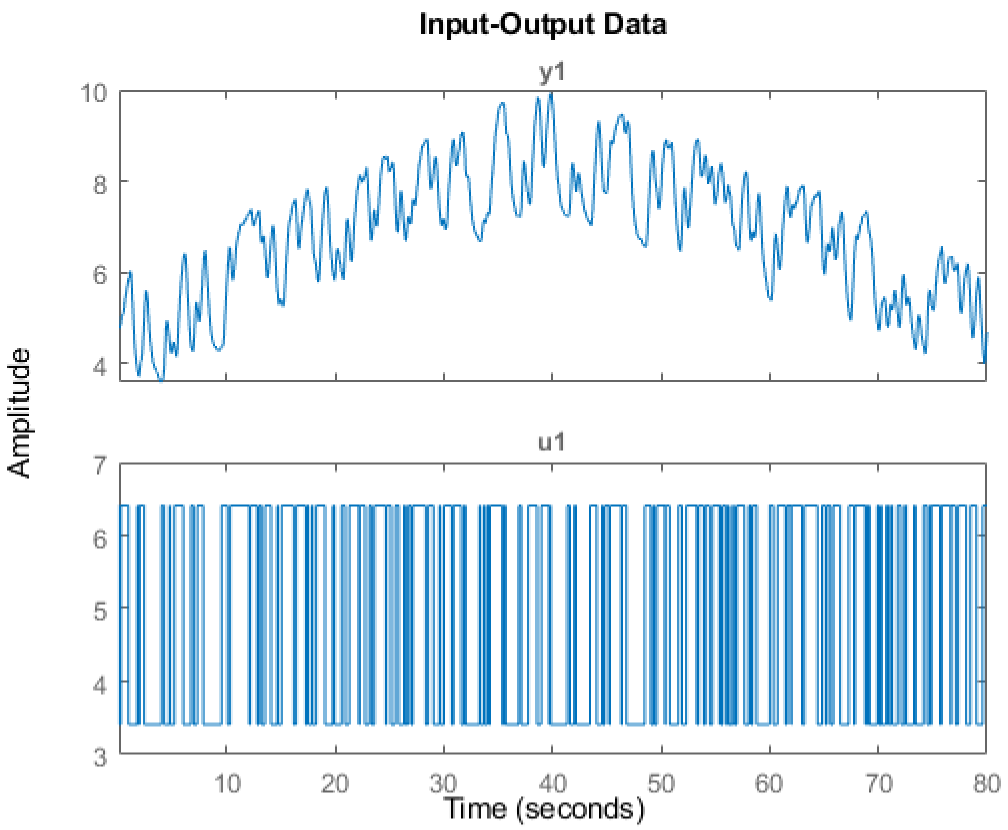
Detrend assumes that the same type of trend applies to both input and output signals. In some cases, there may be a trend type that is present in only one signal. You can perform detrend individually on each signal by extracting the signals into separate iddata objects. Apply detrend to each object using its individual signal trend type. Then reassemble the results back into a single detrended iddata object.

Load, examine, and plot the data in iodatab.

```
load septrend iodatab;
iodatab
```

```
iodatab =

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

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

```
plot(iodatab)
hold on
```



Both input and output plots show a bias. However, the output plot also shows an inverted V-shape trend that is not present in the input data.

Separate the input data and the output data into separate objects for detrending, using the `iddata` general data-selection form (see “Representing Time- and Frequency-Domain Data Using iddata Objects”):

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

```
idatab = iodatab(:,[],:);
odatab = iodatab(:,:,[]);
```

Remove the bias from the input data, using `detrend` to calculate and subtract the mean.

```
idatab_d = detrend(idatab,0);
```

Remove the bias and the inverted-V trend from the output data, using the midpoint index 500 as a breakpoint.

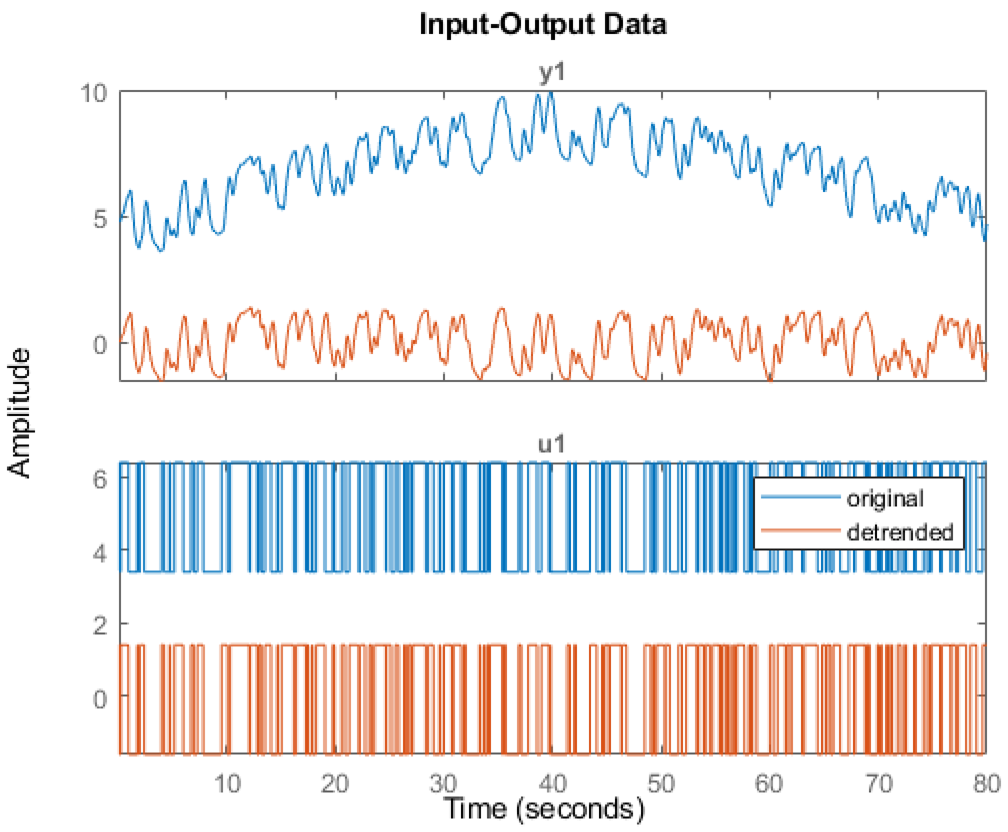
```
odatab_d = detrend(odatab,1,500);
```

Combine the detrended input and output data into a single `iddata` object.

```
iodatab_d = [odatab_d,idatab_d];
```

Overlay the detrended data on the original data.

```
plot(iodatab_d)
legend('original','detrended')
```



The input and output data now contain neither bias nor V-shape trend.

Input Arguments

data — Time-domain input-output data
iddata object

Time-domain input-output data, specified as an iddata object containing one or more sets of time-domain signals. The iddata object can contain SISO, MIMO, or multiexperiment data. The signal sets can contain either input and output data, or output data only.

Type — Trend type to be subtracted
0 (default) | 1 | TrendInfo object

Trend type to be subtracted, specified as one of:

- 0 — Compute and subtract the mean value
- 1 — Compute and subtract the linear trend (least-squares fit)
- TrendInfo object — subtract the trend you specify in the TrendInfo object. Use getTrend to create a TrendInfo object. For an example, see "Remove Specified Offsets from Signals" on page 1-275.

brkpt — Timepoint locations of trending discontinuities
integer row vector | cell array of integer vectors

Timepoint locations of trending discontinuities (breakpoints), specified as:

- An integer row vector — For single-experiment SISO and MIMO datasets. Doing so applies a single set of breakpoints to all input and output signals. For an example, see "Remove Segmented Linear Trends from Signals by using Breakpoints" on page 1-277.
- A cell array containing individually-sized integer row vectors — For multiple-experiment datasets. Doing so applies a unique set of breakpoints to the output and input signals for each experiment. For an example, see "Detrend Multiexperiment Signals using Multiple-Breakpoint Sets" on page 1-279.

Output Arguments

data_d — Detrended signals
iddata object

Detrended signals, returned as an iddata object. Dimensions of the contents are the same as dimensions of the contents of data.

T_r — Subtracted trend data
TrendInfo object

Trend data subtracted from data to produce data_d, returned as a TrendInfo object .

When you use brkpt to specify multiple trends, you cannot retrieve the computed trend data.

See Also

getTrend | iddata | idfilt | retrend

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

Estimate Process Model

Estimate continuous-time process model for single-input, single-output (SISO) system in either time or frequency domain in the Live Editor

Description

The **Estimate Process Model** task lets you interactively estimate and validate a process model for SISO systems. You can define and vary the model structure and specify optional parameters, such as initial condition handling and search methods. The task automatically generates MATLAB code for your live script. For more information about Live Editor tasks generally, see “Add Interactive Tasks to a Live Script”.

Process models are simple continuous-time transfer functions that describe the linear system dynamics. Process model elements include static gain, time constants, time delays, integrator, and process zero.

Process models are popular for describing system dynamics in numerous industries and are applicable to various production environments. The advantages of these models are that they are simple, they support transport delay estimation, and the model coefficients are easy to interpret as poles and zeros. For more information about process model estimation, see “What Is a Process Model?”

The **Estimate Process Model** task is independent of the more general **System Identification** app. Use the **System Identification** app when you want to compute and compare estimates for multiple model structures.

To get started, load experiment data that contains input and output data into your MATLAB workspace and then import that data into the task. Then select a model structure to estimate. The task gives you controls and plots that help you experiment with different model structures and compare how well the output of each model fits the measurements.

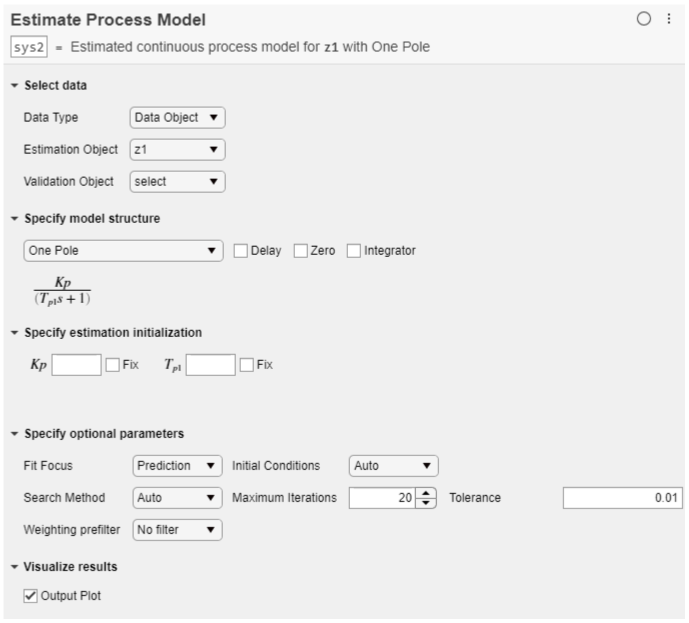
Related Functions

The code that **Estimate Process Model** generates uses the following functions.

- `iddata`
- `idfrd`

  `frd`
- `procest`
- `procestOptions`
- `compare`

The task estimates an `idproc` process model.



Estimate Process Model

sys2 = Estimated continuous process model for z1 with One Pole

Select data

Data Type: Data Object

Estimation Object: z1

Validation Object: select

Specify model structure

One Pole ☐ Delay ☐ Zero ☐ Integrator

$$\frac{Kp}{(T_{p1}s+1)}$$

Specify estimation initialization

Kp ☐ Fix T_{p1} ☐ Fix

Specify optional parameters

Fit Focus: Prediction | Initial Conditions: Auto

Search Method: Auto | Maximum Iterations: 20 | Tolerance: 0.01

Weighting prefilter: No filter

Visualize results

☑ Output Plot

Description (Collapsed Portion)

Related Functions

The code that **Estimate Process Model** generates uses the following functions.

- `iddata`
- `idfrd`

  `frd`
- `procest`
- `procestOptions`
- `compare`

The task estimates an `idproc` process model.

Open the Task

To add the **Estimate Process Model** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Estimate Process Model**.
- In a code block in your script, type a relevant keyword, such as `process` or `estimate`. Select `Estimate Process Model` from the suggested command completions.

Examples

Estimate Process Model with Live Editor Task

Use the **Estimate Process Model** Live Editor Task to estimate a state-space model and compare the model output to the measurement data.

Open this example to see a preconfigured script containing the task.

Set Up Data

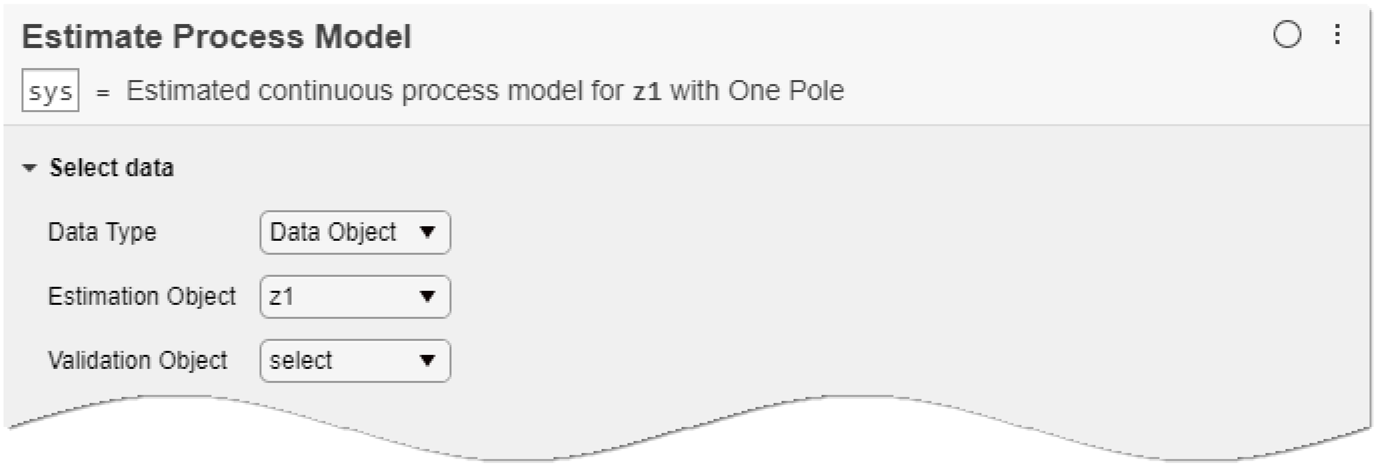
Load the measurement data `iddata1` into your MATLAB workspace.

```
load iddata1 z1
z1
```

```
z1 =

Time domain data set with 300 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Import Data into the Task

In the **Select data** section, set **Data Type** to `Data Object` and set **Estimation Object** to `z1`.



Estimate Process Model

sys = Estimated continuous process model for z1 with One Pole

Select data

Data Type: Data Object

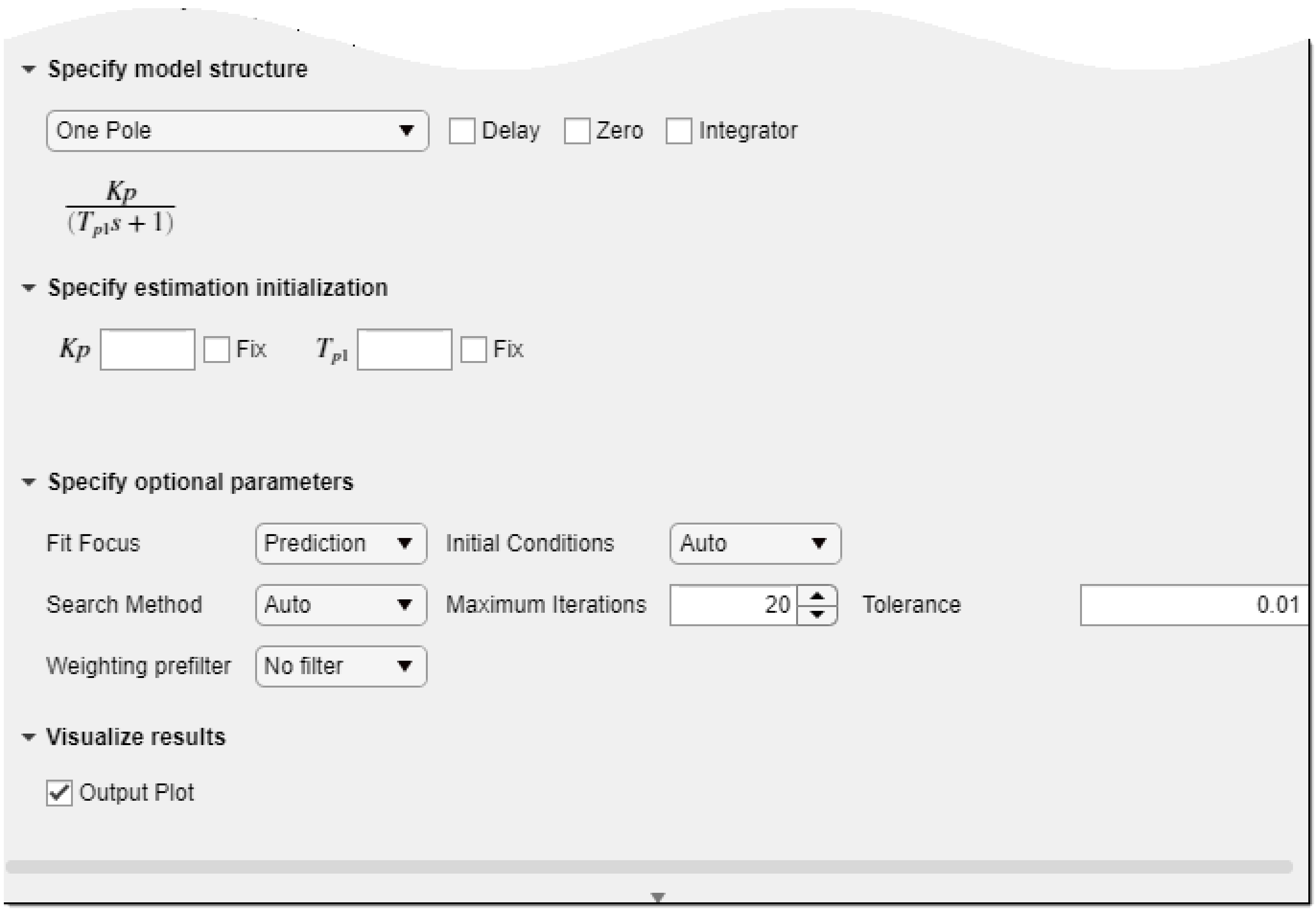
Estimation Object: z1

Validation Object: select

The data object contains the input and output variable names as well as the sample time, so you do not have to specify them.

Estimate the Model Using Default Settings

Examine the model structure and optional parameters.



Specify model structure

One Pole; Delay [ ]; Zero [ ]; Integrator [ ]

$$\frac{Kp}{(T_{p1}s+1)}$$

Specify estimation initialization

Kp: (blank), Fix [ ]; T_{p1}: (blank), Fix [ ]

Specify optional parameters

Fit Focus: Prediction; Initial Conditions: Auto

Search Method: Auto; Maximum Iterations: 20; Tolerance: 0.01

Weighting prefilter: No filter

Visualize results

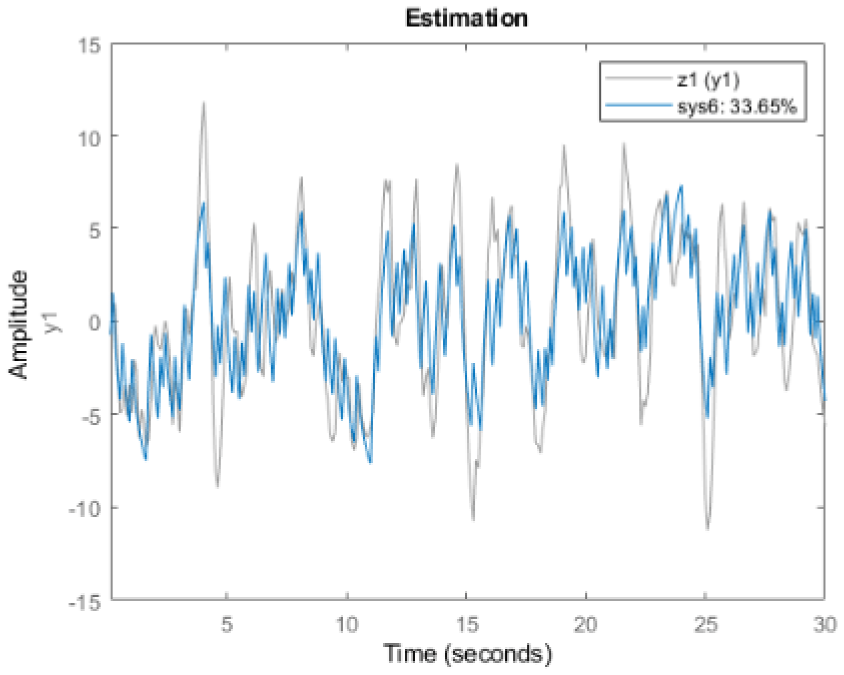
Output Plot [✓]

In the **Specify model structure** section, the default option is `One Pole` with no delay, zero, or integrator. Equations below the parameters in this section display the specified structure.

In the **Specify estimation initialization** section, initialization parameters matching the parameters in your model structure allow you to set starting points for estimation. If you select **Fix**, the parameter remains fixed to the value you specify. For this example, do not specify initialization. The task then uses default values for starting points.

In the **Specify optional parameters** section, the default options for process estimation are set.

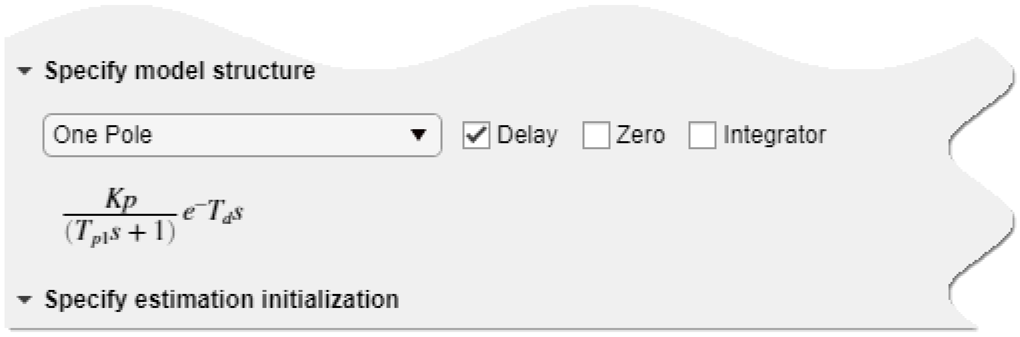
Execute the task from the **Live Editor** tab using **Run**. A plot displays the estimation data, the estimated model output, and the fit percentage.



Experiment with Parameter Settings

Experiment with the parameter settings and see how they influence the fit.

For instance, add delay to the `One Pole` structure and run the task.

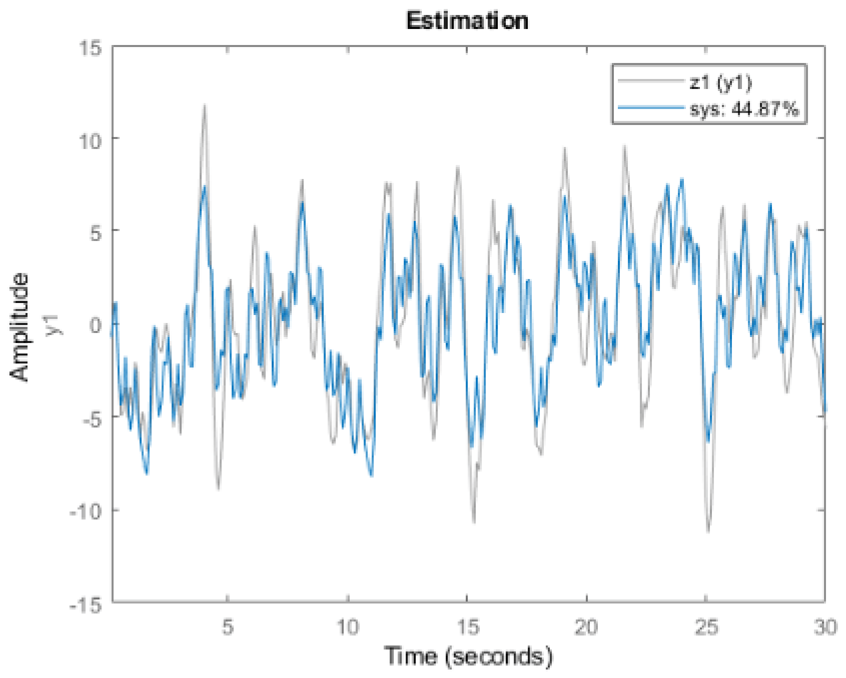


▾ Specify model structure

One Pole ▾ [✓ Delay] ☐ Zero ☐ Integrator

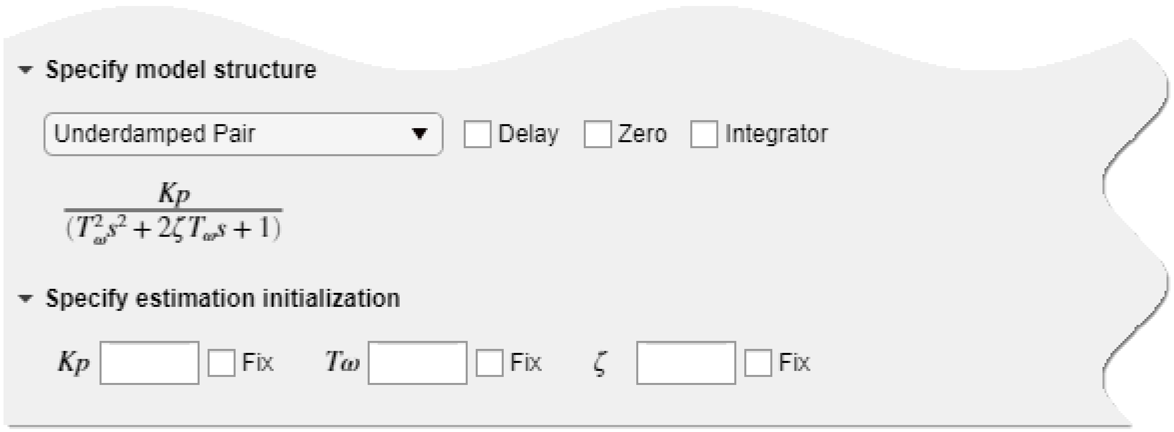
$$\frac{Kp}{(T_{p1}s+1)}e^{-T_d s}$$

▾ Specify estimation initialization



The estimation fit improves, although the fit percentage is still below 50%.

Try a different model structure. In **Specify model structure**, select `Underdamped Pair` with no delay and run the task.



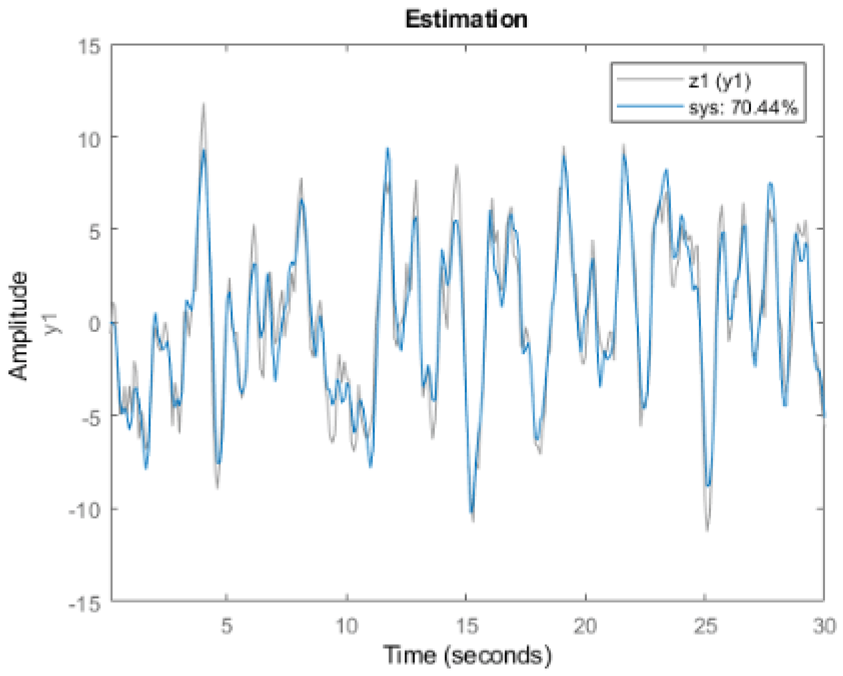
Specify model structure

Underdamped Pair ▼ ☐ Delay ☐ Zero ☐ Integrator

$$\frac{Kp}{(T_\omega^2 s^2 + 2\zeta T_\omega s + 1)}$$

Specify estimation initialization

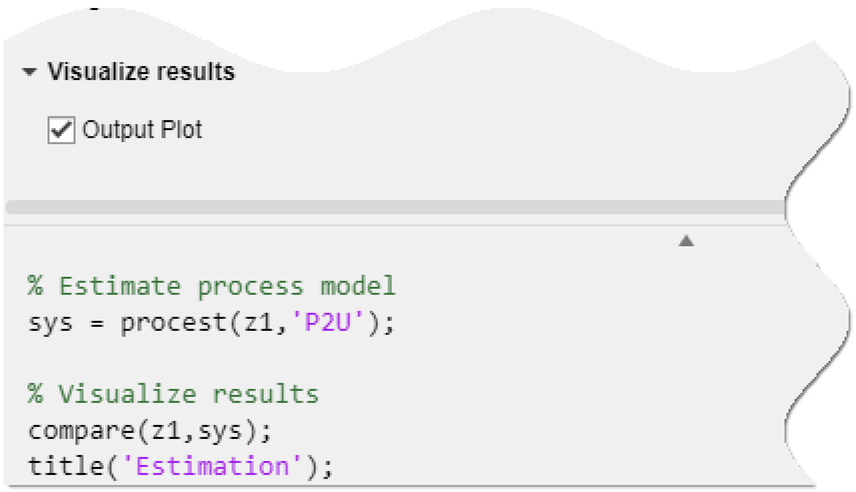
Kp ☐ Fix $T\omega$ ☐ Fix ζ ☐ Fix



The fit results improve significantly.

Generate Code

To display the code that the task generates, click ▼ at the bottom of the parameter section. The code that you see reflects the current parameter configuration of the task.



```
% Estimate process model
sys = procest(z1,'P2U');

% Visualize results
compare(z1,sys);
title('Estimation');
```

Use Validation Data Set to Validate Estimated Model

Use separate estimation and validation data so that you can validate the estimated process model.

Open this example to see a preconfigured script containing the task.

Set Up Data

Load the measurement data iddata1 into your MATLAB workspace and examine its contents.

```
load iddata1 z1
z1
```

```
z1 =

Time domain data set with 300 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Extract the input and output measurements.

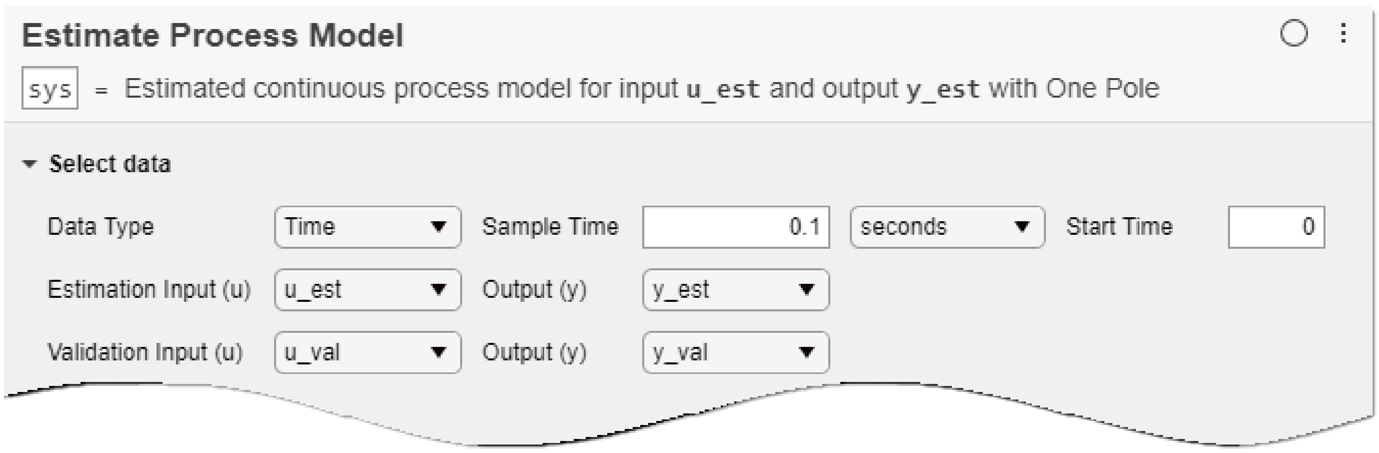
```
u = z1.u;
y = z1.y;
```

Split the data into two sets, with one half for estimation and one half for validation. The original data set has 300 samples, so each new data set has 150 samples.

```
u_est = u(1:150);
u_val = u(151:300);
y_est = y(1:150);
y_val = y(151:300);
```

Import Data into Task

In the **Select data** section, set **Data Type** to Time. Set the sample time to 0.1 seconds, which is the sample time in the original iddata object z1. Select the appropriate data sets for estimation and validation.



Estimate Process Model

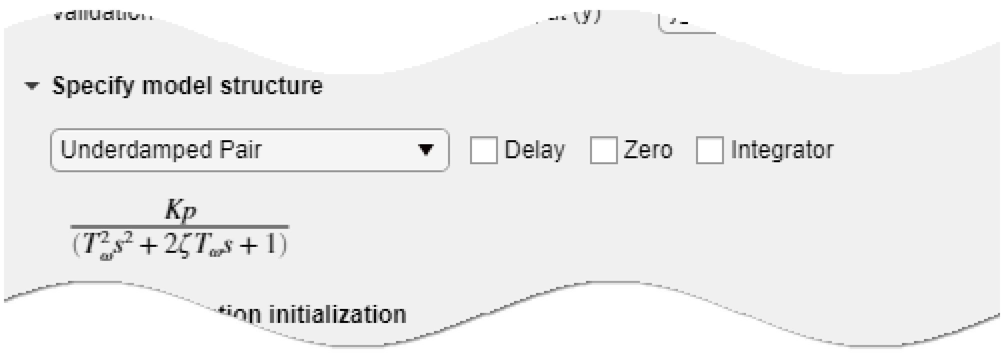
sys = Estimated continuous process model for input u_est and output y_est with One Pole

Select data

Data Type	Time	Sample Time	0.1	seconds	Start Time	0
Estimation Input (u)	u_est	Output (y)	y_est			
Validation Input (u)	u_val	Output (y)	y_val			

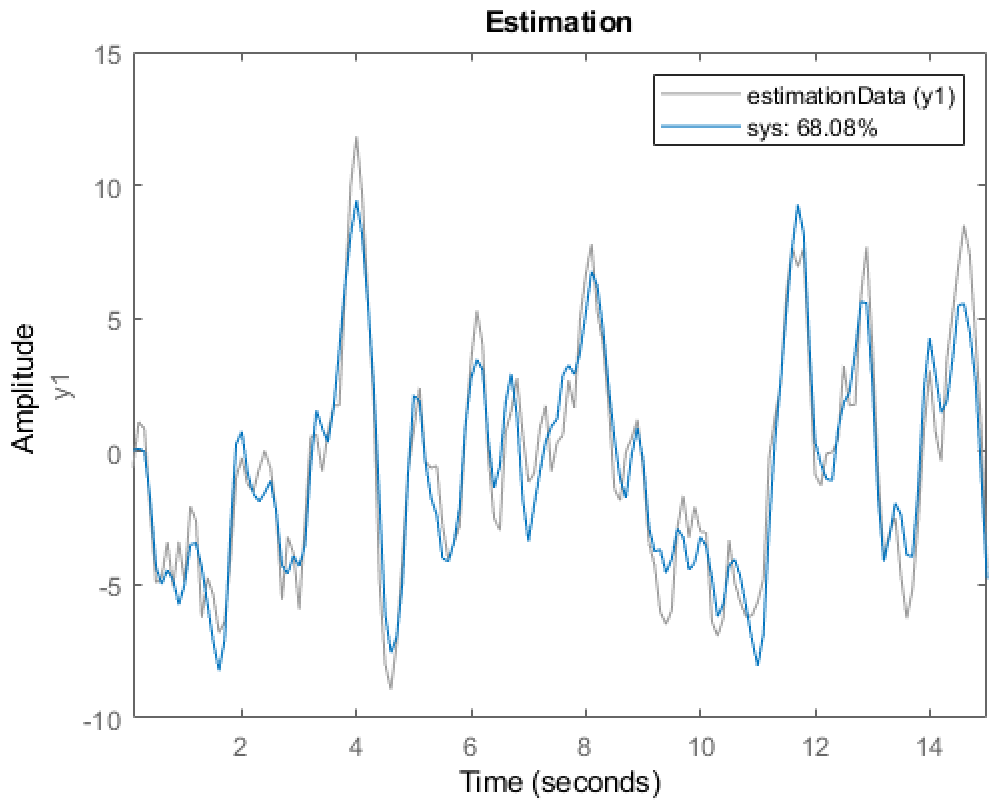
Estimate and Validate the Model

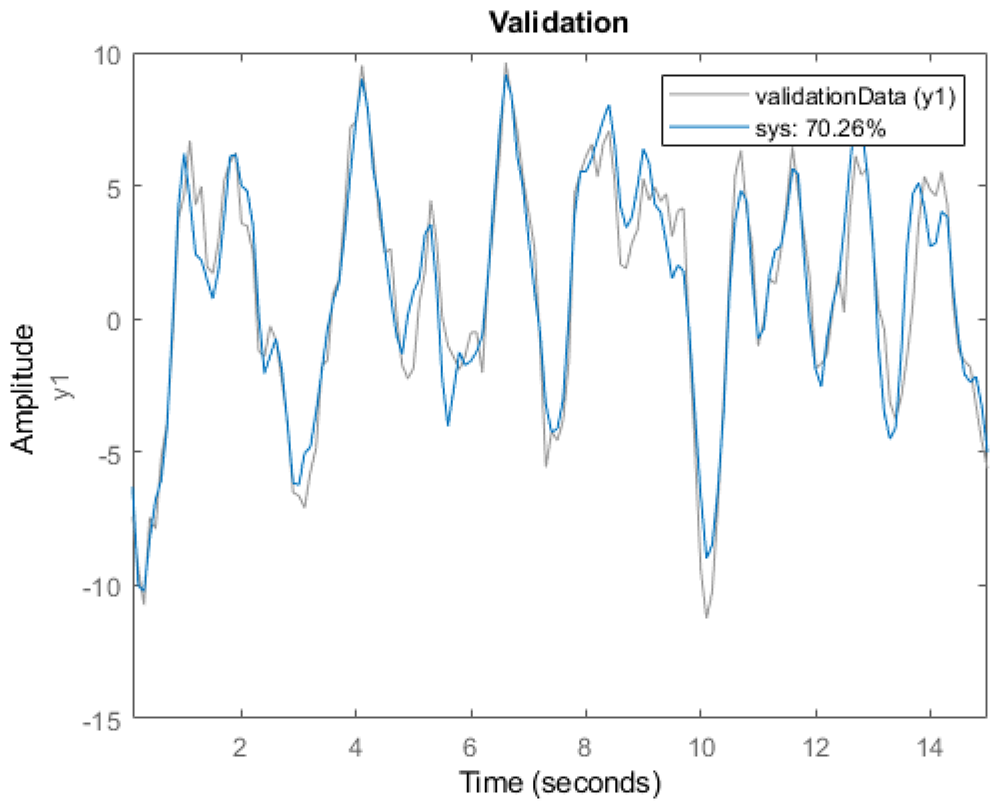
The example “Estimate Process Model with Live Editor Task” on page 1-288 achieves the best results using the model structure `Underdamped Pair`. Choose the same option for this example.



$$\frac{Kp}{(T_{\omega}^2 s^2 + 2\zeta T_{\omega} s + 1)}$$

Execute the task from the **Live Editor** tab using **Run**. Executing the task creates two plots.The first plot shows the estimation results and the second plot shows the validation results.


Estimation
estimationData (y1)
sys: 68.08%
Amplitude
y1
Time (seconds)


Validation
validationData (y1)
sys: 70.26%
Amplitude
y1
Time (seconds)

The fit to the estimation data is somewhat worse than in “Estimate Process Model with Live Editor Task” on page 1-288. Estimation in the current example has only half the data with which to estimate the model. The fit to the validation data, which represents the goodness of the model more generally, is better than the fit to the estimation data.

Parameters

Select Data

`Data Type` — Data type for input and output data

`Time` (default) | `Frequency` | `Data Object`

The task accepts single-channel numeric measurement values that are uniformly sampled in time. Data can be packaged as numeric arrays (`Time` or `Frequency` type) or in a data object, such as an `iddata` or `idfrd` object.

The data type you choose determines the additional parameters you must specify.

- `Time` — Specify **Sample Time** and **Start Time** in the time units that you select.
- `Frequency` — Specify **Frequency** by selecting the variable name of a frequency vector in your MATLAB workspace. Specify the units for this frequency vector. Specify **Sample Time** in seconds.
- `Data Object` — Specify no additional parameters, because the data object already contains information on time or frequency sampling.

`Estimation Input and Estimation Output` — Variable names of input and output data for estimation

valid variable names

Select the input and output variable names from the MATLAB workspace choices. Use these parameters when **Data Type** is `Time` or `Frequency`.

`Estimation Object` — Variable name of data object containing input and output data to be used for estimation

valid variable name

Select the data object variable name from the MATLAB workspace choices. Use this parameter when **Data Type** is `Data Object`.

`Validation Input (u) and Validation Output (y)` — Variable names of input and output data to be used for validation

valid variable names

Select the input and output variable names, or the data object name, from the workspace choices. Use these parameters when **Data Type** is `Time` or `Frequency`. Specifying validation data is optional but recommended.

`Validation Object` — Variable name of data object containing input and output data for validation

valid variable name

Select the data object variable name from the MATLAB workspace choices. Use this parameter when **Data Type** is `Data Object`. Specifying validation data is optional but recommended.

Specify Model Structure

`Structure` — Zeros and poles in model
`One Pole` (default) | `Two Real Poles` | `Underdamped Pair` | `Underdamped Pair + Real Pole`

The task allows you to specify one of four basic structures. These structures range from a simple first-order process to a more dynamic second-order or third-order process with complex conjugate (underdamped) poles.

- `One Pole`
- `Two Real Poles`
- `Underdamped Pair`
- `Underdamped Pair + Real Pole`

`Delay` — Include transport delay
off (default) | on

Include transport delay, or input-to-output delay, of one sample. The transport delay is also known as dead time.

`Zero` — Include process zero
off (default) | on

Include a process zero in the numerator.

`Integrator` — Include integrator
off (default) | on

Include an integrator, represented by an additional 1/ *s* term. Including an integrator creates a self-regulating process.

Specify Estimation Initialization

`Initial Values` — Initial values of structure parameters
0 | parameter values

Specify initial values for the estimation and whether these values are to be fixed or estimated. The values to specify depend on the model structure and your specifications for **Delay** and **Zero**. Below **Specify model structure**, the task displays the equation that represents the specified system. This equation contains all of the parameters that can be estimated, and that you can initialize or fix. The possible parameters are:

- Kp — Static gain
- T_{p1} — Time constant for first real pole
- T_{p2} — Time constant for second real pole
- $T\omega$ — Time constant for complex poles, equal to the inverse of the natural frequency
- ζ — Damping coefficient for complex poles
- T_d — Transport delay
- T_z — Time constant for the process zero

All time-based parameters are in the time units you select for **Sample Time**.

Specify Optional Parameters

`Fit Focus` — Minimize prediction error or simulation error

`Prediction` (default) | `Simulation`

Fit focus specifies what error to minimize in the loss function during estimation.

- `Prediction` — Minimize the one-step-ahead prediction error between measured and predicted outputs. This estimation approach focuses on producing a good predictor model for the estimation inputs and outputs. Prediction focus generally produces the best estimation results because it uses both input and output measurements, thus accounting for disturbances.
- `Simulation` — Minimize the error between measured and simulation outputs. This estimation approach focuses on producing a simulated model response that has a good fit with the estimation inputs and outputs. Simulation focus is generally best for validation, especially with data sets not used for the original estimation.

`Initial Conditions` — Handling of initial conditions

`Auto` (default) | `Zero` | `Estimate` | `Backcast`

Set this option when you want to choose a specific method for initializing the model. With the default setting of `Auto`, the software chooses the method based on the estimation data. Choices are:

- `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.

`Input Intersampling` — Intersampling behavior for input signal

`Zero-order hold` (default) | `Triangle approximation` | `Band-limited`

Input intersampling is a property of the input data. The task uses this property when estimating process models. Specify **Input Intersampling** when your data type is `Time` or `Frequency`. If you are using an `iddata` object, the object already contains the intersampling information. Choices for this property are:

- `Zero-order hold` — Piecewise-constant input signal between samples
- `Triangle approximation` — Piecewise-linear input signal between samples, also known as first-order hold
- `Band-limited` — Input signal has zero power above the Nyquist frequency

`Search Method` — Numerical search mode for iterative parameter estimation

`Auto` (default) | `Gauss-Newton` | `Adaptive Gauss-Newton` | `Levenberg-Marquardt` | `Gradient Search`

- `Auto` — For each iteration, the software cycles through the methods until it finds the first direction descent that leads to a reduction in estimation cost.
- `Gauss-Newton` — Subspace Gauss-Newton least-squares search.
- `Levenberg-Marquardt` — Levenberg-Marquardt least-squares search.
- `Adaptive Gauss-Newton` —Adaptive subspace Gauss-Newton search.
- `Gradient Search` — Steepest descent least-squares search.

`Max. Iterations` — Maximum number of iterations during error minimization

20 (default) | positive integer

Set the maximum number of iterations during error minimization. The iterations stop when **Max. Iterations** is reached or another stopping criterion is satisfied, such as **Tolerance**.

`Tolerance` — Minimum percentage of expected improvement in error
0.01 (default) | positive integer

When the percentage of expected improvement is less than **Tolerance**, the iterations stop.

`Weighting Prefilter` — Weighting prefilter for loss function
`No filter` (default) | `Passband(s)` | `LTI Filter` | `Frequency weights vector`

Set this option when you want to apply a weighting prefilter to the loss function that the task minimizes when you estimate the model. When you select an option, you must also select the associated variable in your workspace that contains the filter information. The available options depend on the domain of the data.

Weighting Prefilter	Data Domain	Filter Information
`No Filter`	Time and frequency	
`Passbands`	Time and frequency	Passband ranges, specified as a 1-by-2 row vector or an *n*-by-2 matrix, where *n* is the number of passbands
`LTI Filter`	Time and frequency	SISO LTI model
`Frequency Weights Vector`	Frequency	Frequency weights, specified as a column vector with the same length as the frequency vector

For instance, suppose that you are performing estimation with SISO frequency-domain data and that in your MATLAB workspace, you have a column vector `W` that contains frequency weights for the prefilter. In the task, select **Weighting prefilter > Frequency weights vector** and the variable `W`.

Visualize Results

`Output Plot` — Plot comparison of model and measured outputs
on (default) | off

Plot a comparison of the model output and the original measured data, along with the fit percentage. If you have separate validation data, a second plot compares the model response to the validation input data with the measured output from the validation data set.

See Also
`iddata` | `procest` | `procestOptions` | `compare` | `idproc` | `idfrd` | `frd`

Introduced in R2019b

Estimate State-Space Model

Estimate state-space model using time or frequency data in the Live Editor

Description

The **Estimate State-Space Model** task lets you interactively estimate and validate a state-space model using time or frequency data. You can define and vary the model structure and specify optional parameters, such as initial condition handling and search method. The task automatically generates MATLAB code for your live script. For more information about Live Editor tasks generally, see “Add Interactive Tasks to a Live Script”.

State-space models are models that use state variables to describe a system by a set of first-order differential or difference equations, rather than by one or more *n*th-order differential or difference equations. State variables can be reconstructed from the measured input-output data, but are not themselves measured during the experiment.

The state-space model structure is a good choice for quick estimation because it requires you to specify only one input, the model order. For more information about state-space estimation, see “What Are State-Space Models?”

The **Estimate State-Space Model** task is independent of the more general **System Identification** app. Use the **System Identification** app when you want to compute and compare estimates for multiple model structures.

To get started, load experiment data that contains input and output data into your MATLAB workspace and then import that data into the task. Then specify a model structure to estimate. The task gives you controls and plots that help you experiment with different model parameters and compare how well the output of each model fits the measurements.

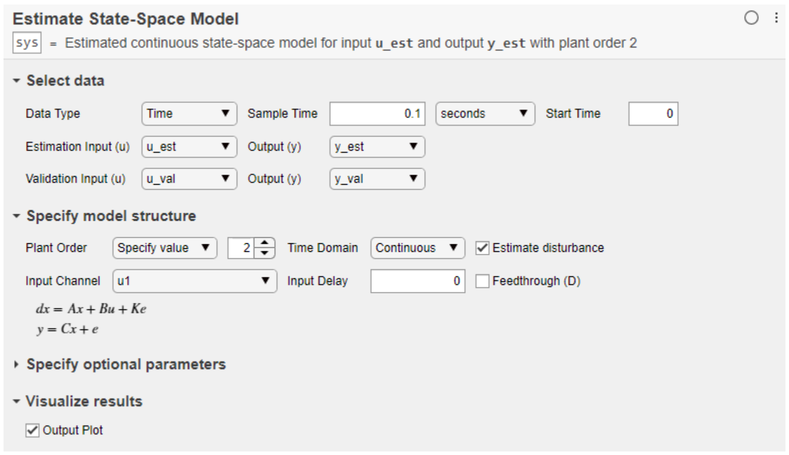
Related Functions

The code that **Estimate State-Space Model** generates uses the following functions.

- `iddata`
- `idfrd`

  `frd`
- `ssest`
- `ssestOptions`
- `compare`

The task estimates an `idss` state-space model.



Estimate State-Space Model

sys = Estimated continuous state-space model for input `u_est` and output `y_est` with plant order 2

Select data

Data Type: Time; Sample Time: 0.1 seconds; Start Time: 0

Estimation Input (u): u_est; Output (y): y_est

Validation Input (u): u_val; Output (y): y_val

Specify model structure

Plant Order: Specify value 2; Time Domain: Continuous; ✓ Estimate disturbance

Input Channel: u1; Input Delay: 0; ☐ Feedthrough (D)

$dx = Ax + Bu + Ke$

$y = Cx + e$

Specify optional parameters

Visualize results

✓ Output Plot

Description (Collapsed Portion)

Related Functions

The code that **Estimate State-Space Model** generates uses the following functions.

- `iddata`
- `idfrd`

  `frd`
- `ssest`
- `ssestOptions`
- `compare`

The task estimates an `idss` state-space model.

Open the Task

To add the **Estimate State-Space Model** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Estimate State-Space Model**.
- In a code block in your script, type a relevant keyword, such as `state`, `space`, or `estimate`. Select `Estimate State Space Model` from the suggested command completions.

Examples

Estimate State-Space Model with Live Editor Task

Use the **Estimate State-Space Model** Live Editor Task to estimate a state-space model and compare the model output to the measurement data.

Open this example to see a preconfigured script containing the task.

Set Up Data

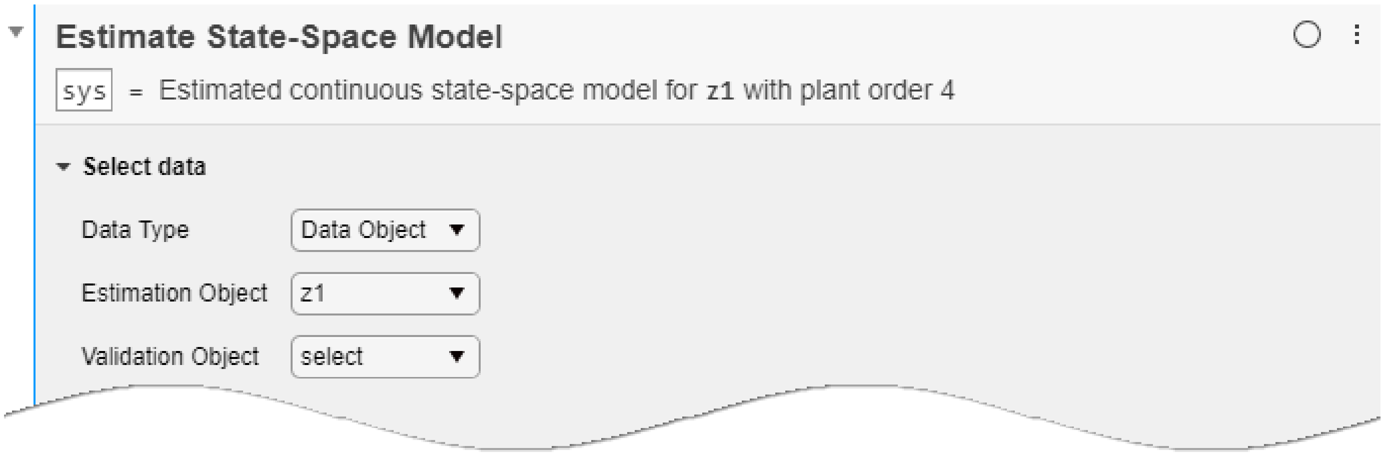
Load the measurement data `iddata1` into your MATLAB workspace.

```
load iddata1 z1
z1
```

```
z1 =

Time domain data set with 300 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Import Data into the Task

In the **Select data** section, set **Data Type** to `Data Object` and set **Estimation Object** to `z1`.



Estimate State-Space Model

sys = Estimated continuous state-space model for z1 with plant order 4

Select data

Data Type: Data Object

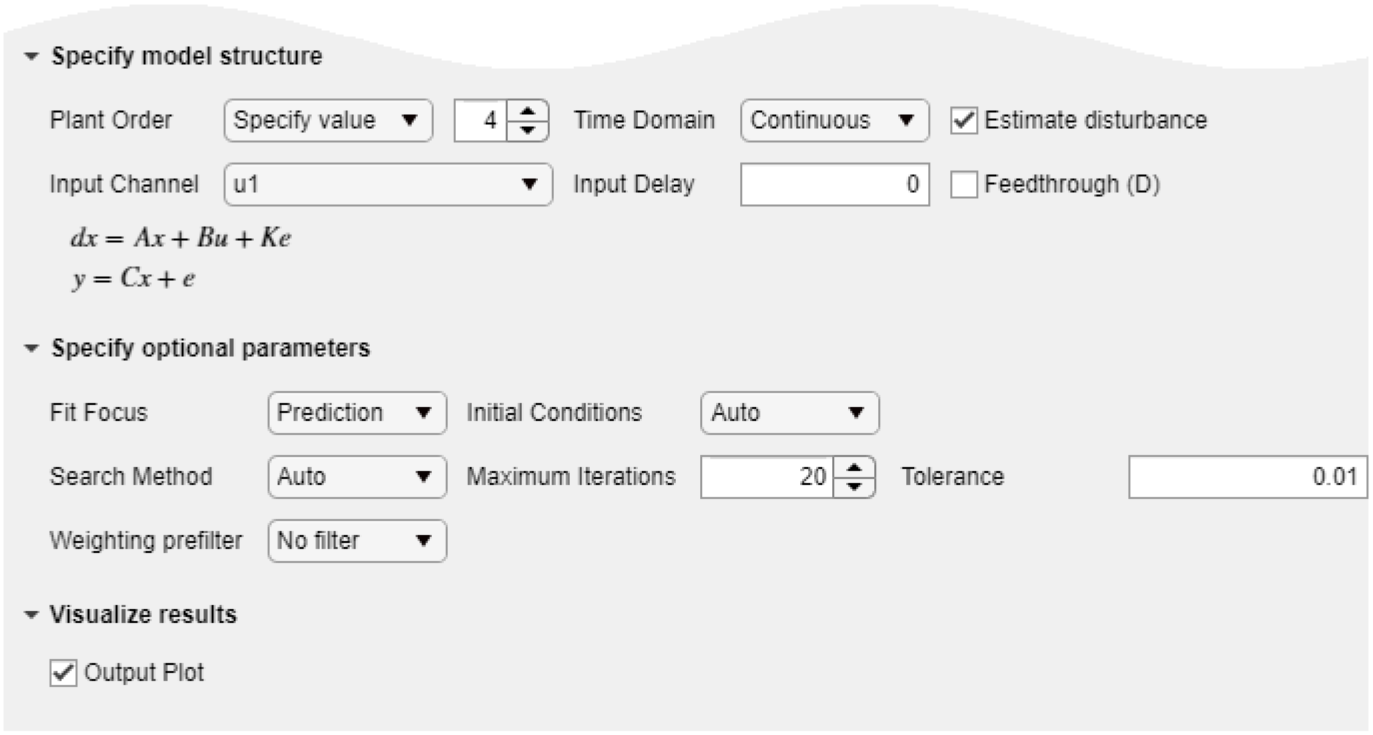
Estimation Object: z1

Validation Object: select

The data object contains the input and output variable names as well as the sample time, so you do not have to specify them.

Estimate the Model Using Default Settings

Examine the model structure and optional parameters.



Specify model structure

Plant Order: Specify value, 4 — Time Domain: Continuous — ✓ Estimate disturbance

Input Channel: u1 — Input Delay: 0 — ☐ Feedthrough (D)

$$dx = Ax + Bu + Ke$$
$$y = Cx + e$$

Specify optional parameters

Fit Focus: Prediction — Initial Conditions: Auto

Search Method: Auto — Maximum Iterations: 20 — Tolerance: 0.01

Weighting prefilter: No filter

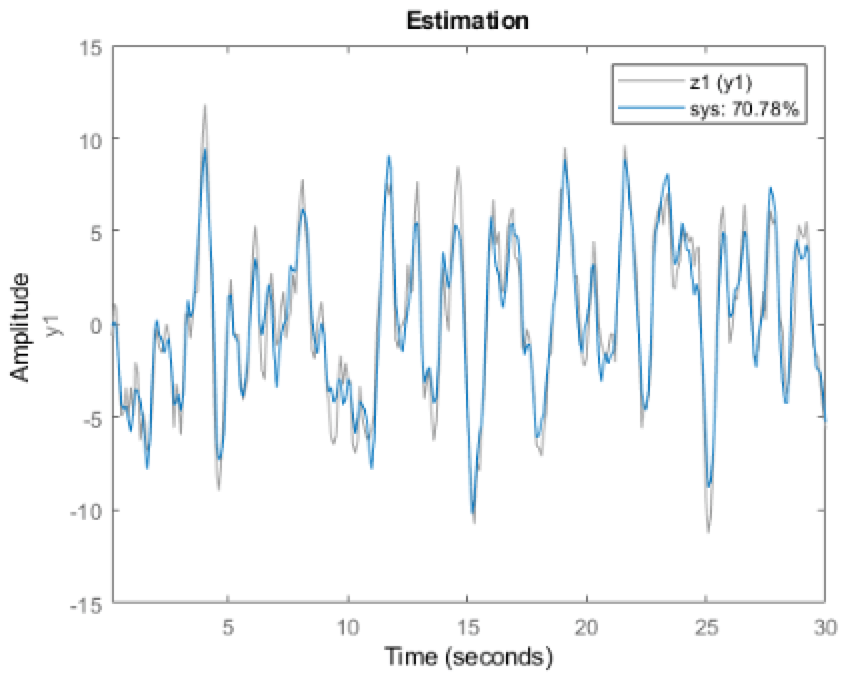
Visualize results

✓ Output Plot

In the **Specify model structure** section, the plant order is set to its default value of 4 and the model is in the continuous-time domain. Equations below the parameters in this section display the specified structure.

In the **Specify optional parameters** section, parameters display the default options for state-space estimation.

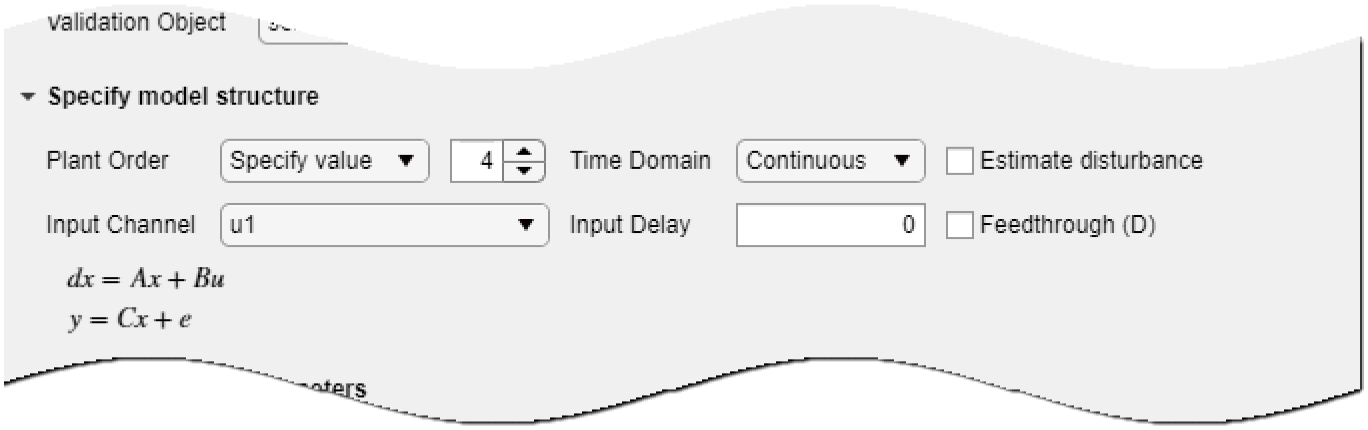
Execute the task from the **Live Editor** tab using **Run**. A plot displays the estimation data, the estimated model output, and the fit percentage.



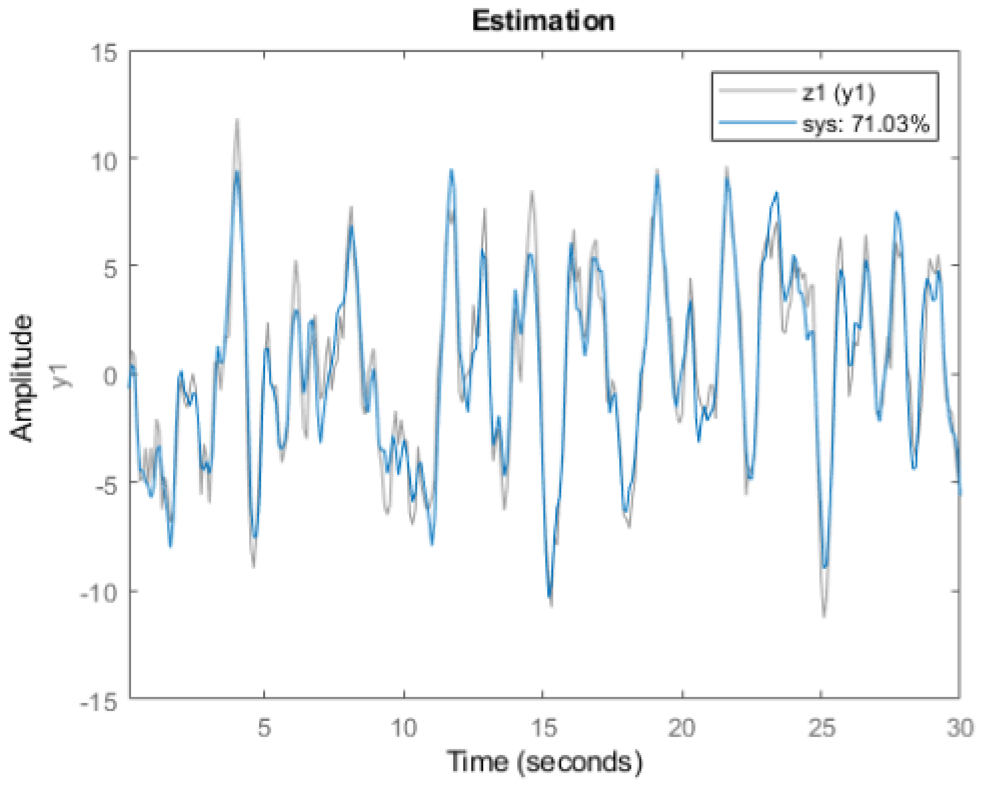
Experiment with Parameter Settings

Experiment with the parameter settings and see how they influence the fit.

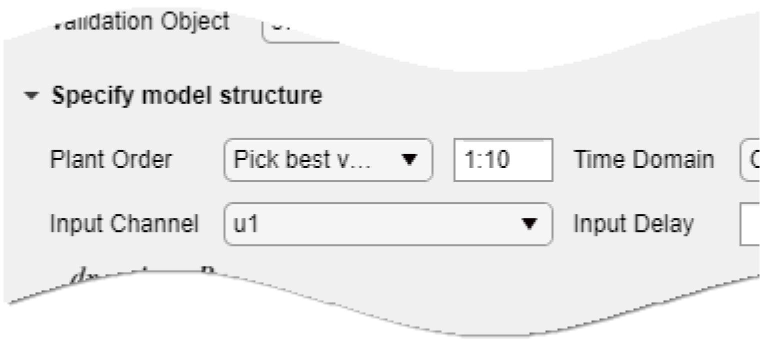
For instance, in **Specify model structure**, the **Estimate disturbance** box is selected, so the disturbance matrix **K** is present in the equations. If you clear the box, the **K** term disappears. Run the updated configuration, and see how the fit changes.



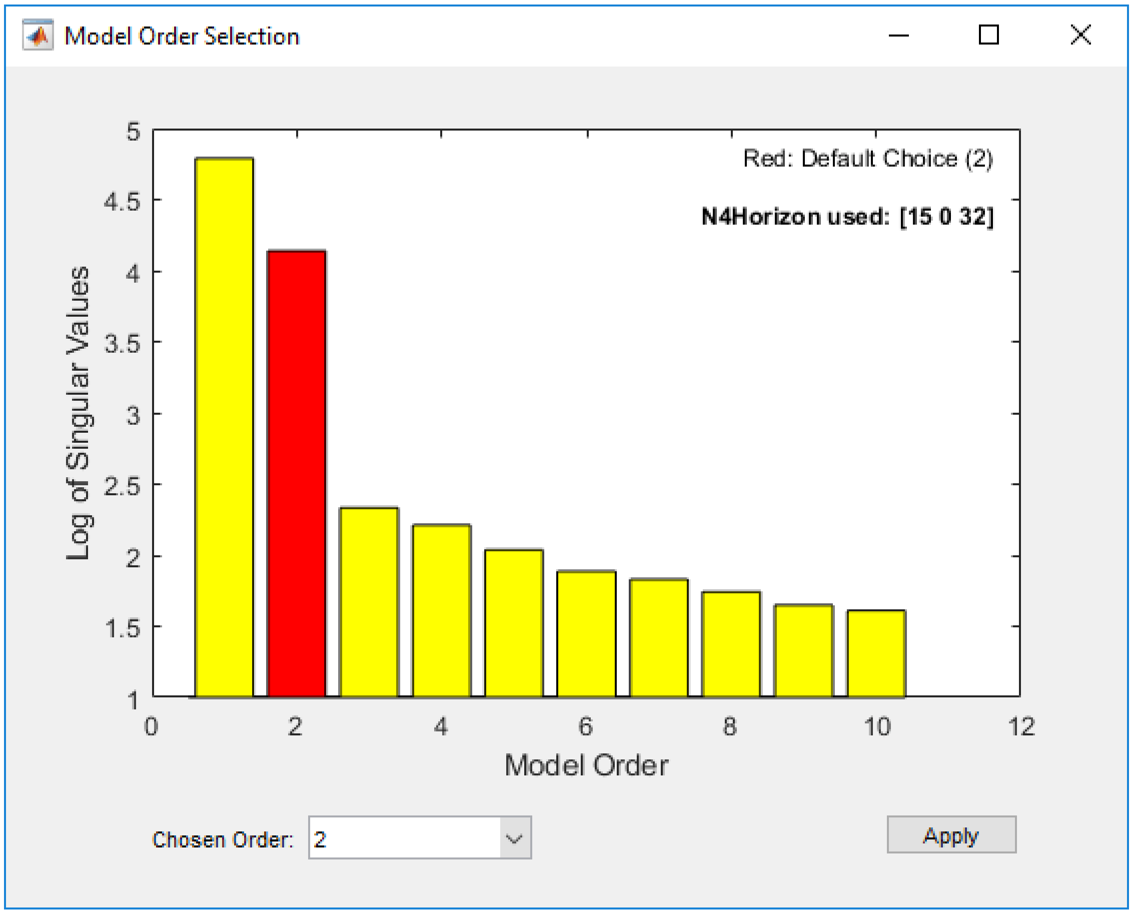
$$dx = Ax + Bu$$
$$y = Cx + e$$



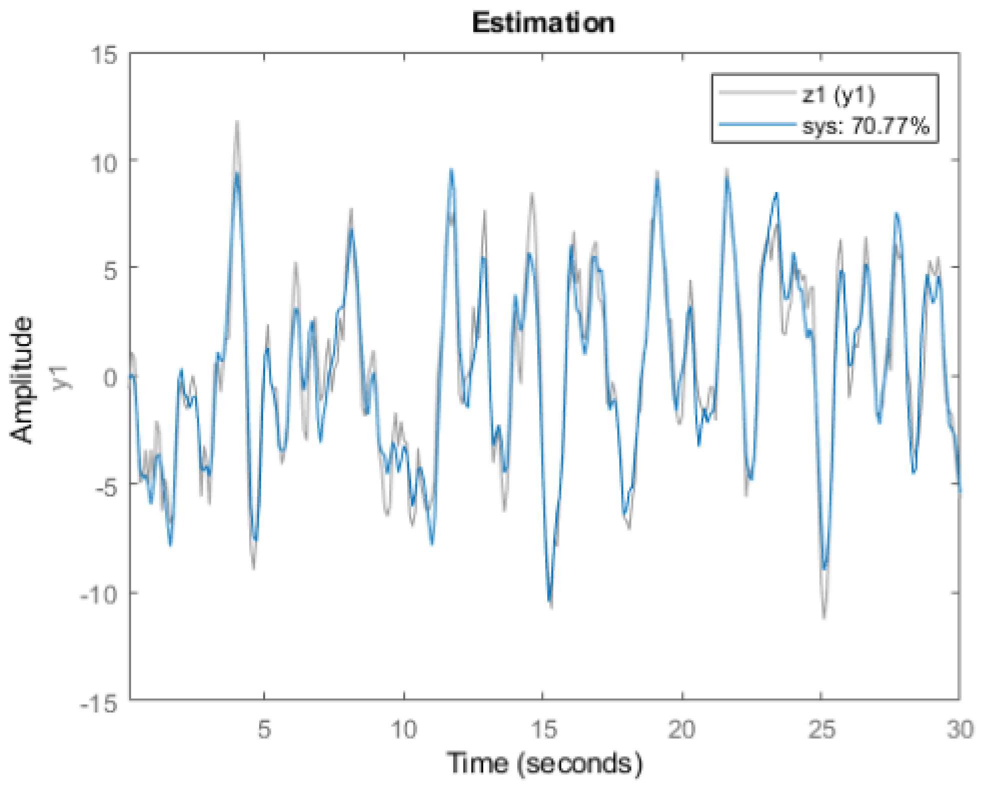
Change the **Plant Order** setting to **Pick best value in range**. The default setting is `1:10`.



When you run the model, a **Model Order Selection** plot displays the contribution of each state to the model dynamic behavior. With the initial task settings for the other parameters, the plot displays a recommendation of 2 for the model order.

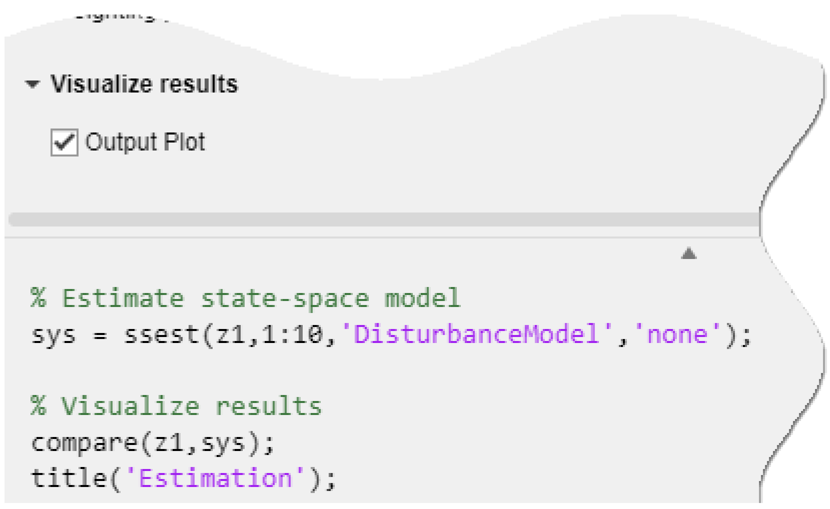


Accept this recommendation by clicking **Apply**, and see how this change affects the fit.



Generate Code

To display the code that the task generates, click ▼ at the bottom of the parameter section. The code that you see reflects the current parameter configuration of the task.



```
% Estimate state-space model
sys = ssest(z1,1:10,'DisturbanceModel','none');

% Visualize results
compare(z1,sys);
title('Estimation');
```

Use Validation Data Set to Validate Estimated Model

Use separate estimation and validation data so that you can validate the estimated state-space model.

Open this example to see a preconfigured script containing the task.

Set Up Data

Load measurement data `iddata1` into your MATLAB workspace and examine its contents.

```
load iddata1 z1
z1
```

```
z1 =

Time domain data set with 300 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Extract the input and output measurements.

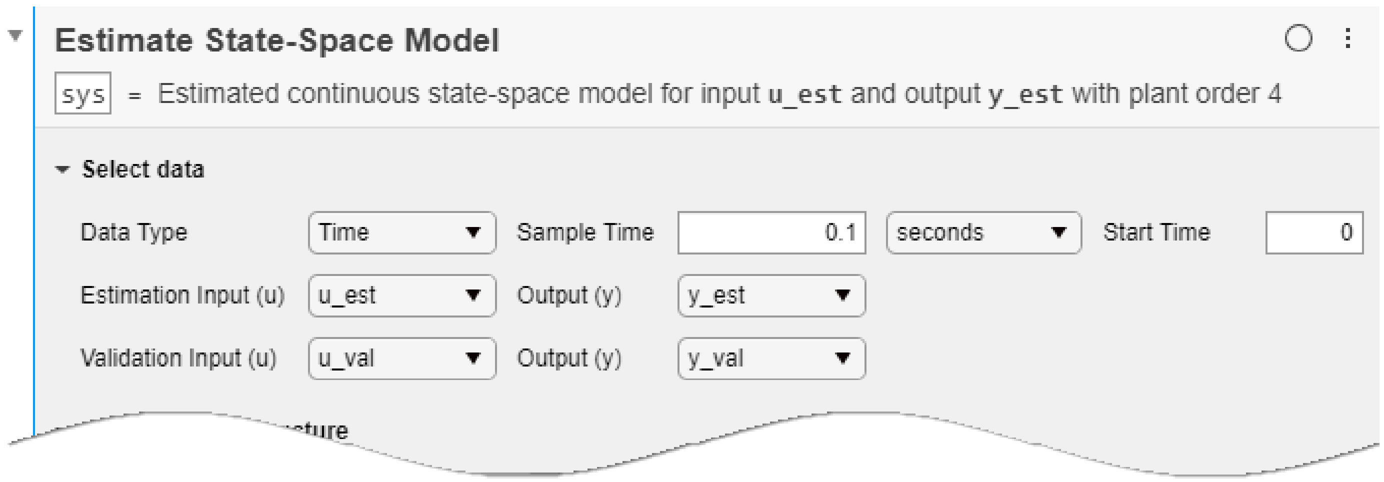
```
u = z1.u;
y = z1.y;
```

Split the data into two sets, with one half for estimation and one half for validation. The original data set has 300 samples, so each new data set has 150 samples.

```
u_est = u(1:150);
u_val = u(151:300);
y_est = y(1:150);
y_val = y(151:300);
```

Import Data into Task

In the **Select data** section, set **Data Type** to `Time`. Set **Sample Time** to `0.1` seconds, which is the sample time in the original `iddata` object `z1`. Select the appropriate data sets for estimation and validation.



Estimate State-Space Model

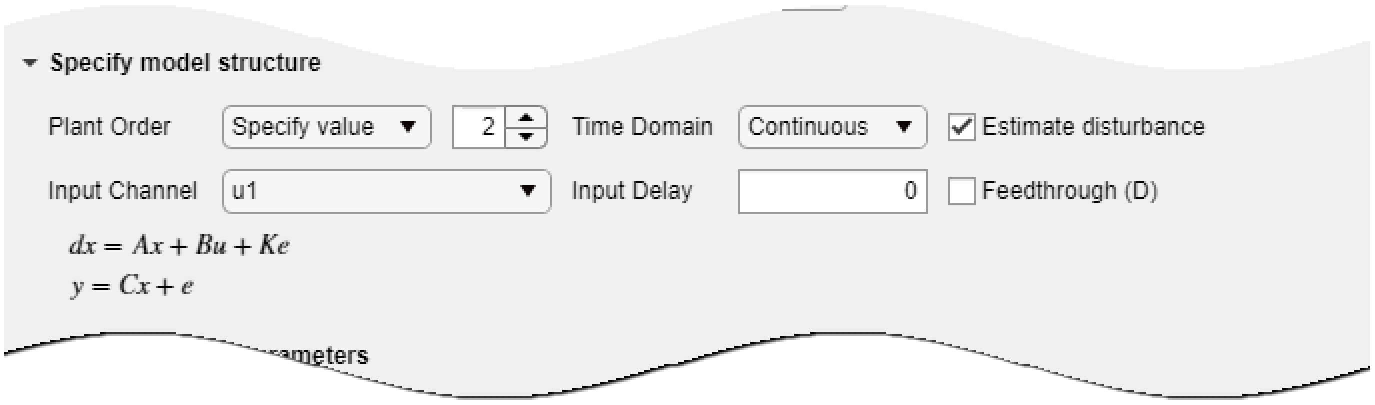
sys = Estimated continuous state-space model for input u_est and output y_est with plant order 4

Select data

Data Type	Time	Sample Time	0.1	seconds	Start Time	0
Estimation Input (u)	u_est	Output (y)	y_est			
Validation Input (u)	u_val	Output (y)	y_val			

Estimate and Validate Model

The example “Estimate State-Space Model with Live Editor Task” on page 1-302 recommends a model order of 2. Use that value for **Plant Order**. Leave other parameters at their default values. Note that **Input Channel** refers not to the input data set, but to the channel index within the input data set, which for a single-input system is always `u1`.

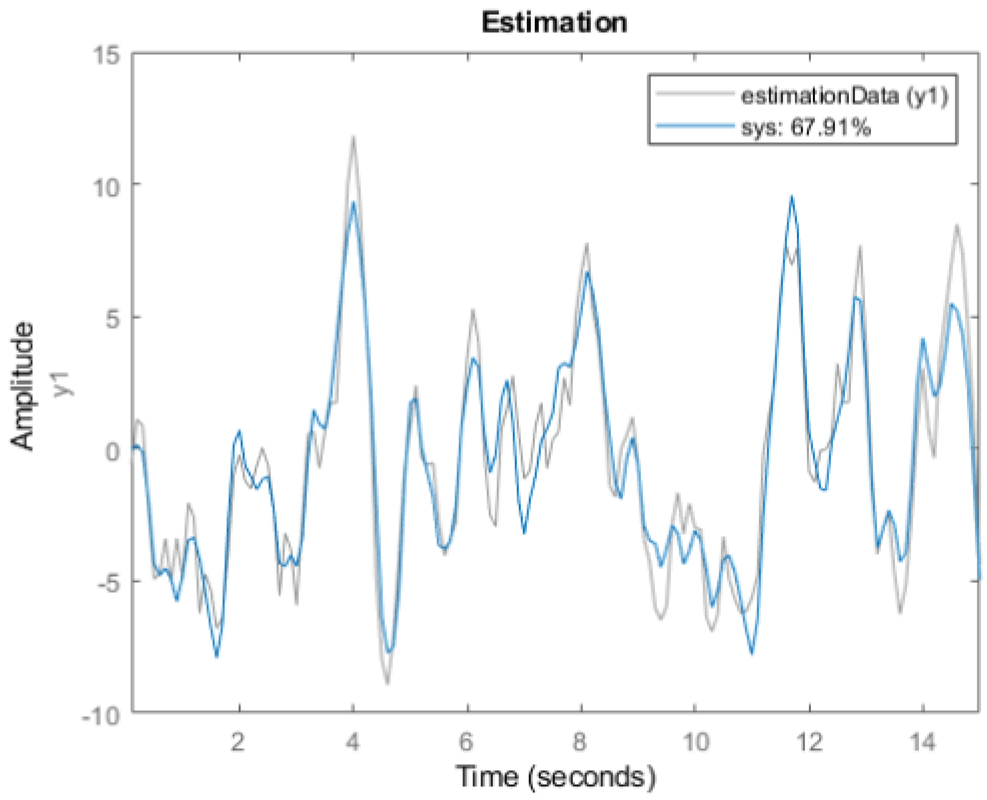


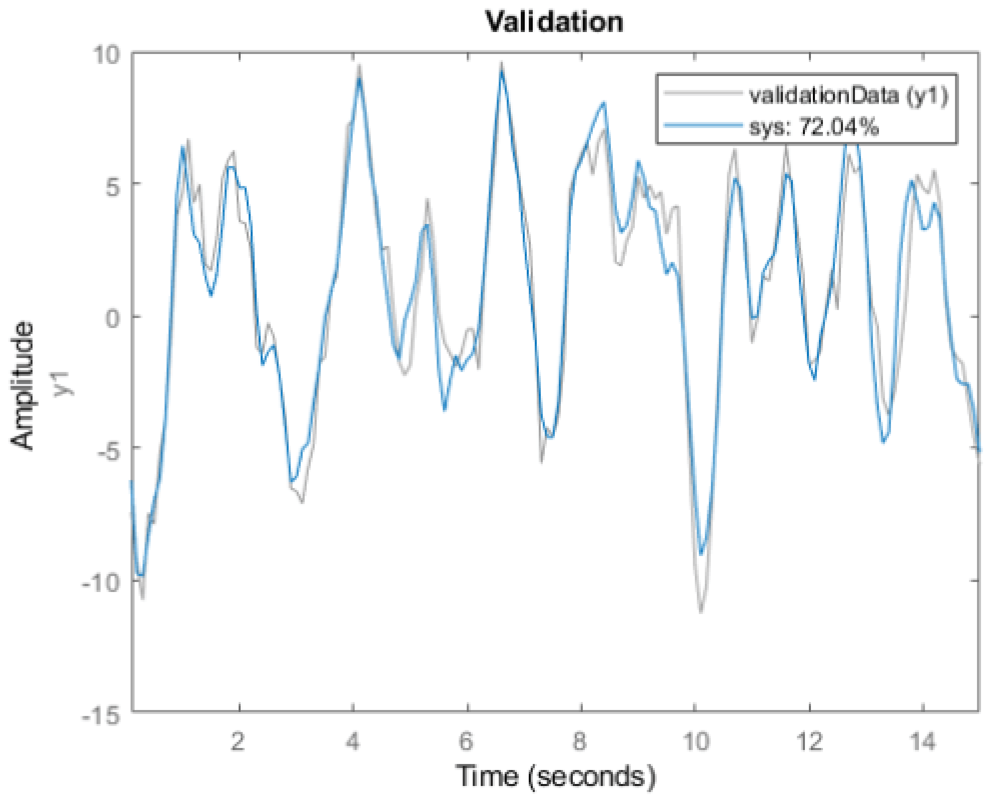
Specify model structure

Plant Order	Specify value 2	Time Domain	Continuous	✓ Estimate disturbance
Input Channel	u1	Input Delay	0	☐ Feedthrough (D)

$$dx = Ax + Bu + Ke$$
$$y = Cx + e$$

Execute the task from the **Live Editor** tab using **Run**. Executing the task creates two plots. The first plot shows the estimation results and the second plot shows the validation results.


Estimation
estimationData (y1)
sys: 67.91%
Amplitude
y1
Time (seconds)


Validation
validationData (y1)
sys: 72.04%
Amplitude
y1
Time (seconds)

The fit to the estimation data is somewhat worse than in “Estimate State-Space Model with Live Editor Task” on page 1-302. Estimation in the current example has only half the data with which to estimate the model. The fit to the validation data, which represents the goodness of the model more generally, is better than the fit to the estimation data.

Parameters

Select Data

`Data Type` — Data type for input and output data

`Time` (default) | `Frequency` | `Data Object`

The task accepts numeric measurement values that are uniformly sampled in time. Input and output signals can contain multiple channels. Data can be packaged as numeric arrays (for `Time` or `Frequency`) or in a data object, such as an `iddata` or `idfrd` object.

The data type you choose determines whether you must specify additional parameters.

- `Time` — Specify **Sample Time** and **Start Time** in the time units that you select.
- `Frequency` — Specify **Frequency** by selecting the variable name of a frequency vector in your MATLAB workspace. Specify the units for this frequency vector. Specify **Sample Time** in seconds.
- `Data Object` — Specify no additional parameters because the data object already contains information on time or frequency sampling.

`Estimation Input (u) and Estimation Output (y)` — Variable names of input and output data for estimation

valid variable names

Select the input and output variable names from the MATLAB workspace choices. Use these parameters when **Data Type** is `Time` or `Frequency`.

`Estimation Object` — Variable name of data object containing input and output data for estimation

valid variable name

Select the data object variable name from the MATLAB workspace choices. Use this parameter when **Data Type** is `Data Object`.

`Validation Input (u) and Validation Output (y)` — Variable names of input and output data for validation

valid variable names

Select the input and output variable names from the workspace choices. Use these parameters when **Data Type** is `Time` or `Frequency`. Specifying validation data is optional but recommended.

`Validation Object` — Variable name of data object containing input and output data for validation

valid variable name

Select the data object variable name from the MATLAB workspace choices. Use this parameter when **Data Type** is `Data Object`. Specifying validation data is optional but recommended.

Specify Model Structure

`Plant Order` — Order of model to estimate
4 (default) | integer scalar | integer range

The task allows you to specify a single value or a range of values for the order of the model to estimate.

- `Specify value` — Specify the order of the model explicitly.
- `Pick best value in range` — Specify a range of values, such as `1:10`. When you run the task, the Hankel singular-value plot visualizes the relative energy contribution of each state in the estimated model and recommends the lowest order that reproduces critical dynamic behavior. Proceed with this recommendation or select another order in **Chosen Order**. Click **Apply** to accept the model order and proceed.

`Time Domain` — Continuous or discrete time domain
`Continuous` (default) | `Discrete`

Select a continuous-time or discrete-time model.

`Estimate Disturbance` — Include disturbance in estimation model
off (default) | on

Select this option to estimate the disturbance model. When you select this option, the model equations update to show the *K* matrix and *e* term.

`Input Channel` — Set input channel delay and feedthrough options
u1 (default) | u2 | ...

For each input channel, assign values for **Input Delay** and **Feedthrough**.

- **Input Channel** — Select an input channel. The input channel is always of the form `u`*i*, where *i* is the *i*th channel of the input `u`.
- **Input Delay** — Enter the input delay in number of samples (discrete-time model) or number of time units (continuous-time model) for the channel. For instance, to specify a 0.2-second input delay for a continuous-time system for which the time unit is `milliseconds`, enter `200`.
- **Feedthrough** — Select this option to estimate channel feedthrough from input to output. When you select this option, the model equations update to show the *Du* term.

Specify Optional Parameters

`Fit Focus` — Minimize prediction error or simulation error
`Prediction` (default) | `Simulation`

Fit focus specifies what error to minimize in the loss function during estimation.

- `Prediction` — Minimize the one-step-ahead prediction error between measured and predicted outputs. This estimation approach focuses on producing a good predictor model for the estimation inputs and outputs. Prediction focus generally produces the best estimation results because it uses both input and output measurements, thus accounting for disturbances.
- `Simulation` — Minimize the error between measured and simulated outputs. This estimation approach focuses on producing a simulated model response that has a good fit with the estimation inputs and outputs. Simulation focus is generally best for validation, especially with data sets not used for the original estimation.

`Initial Conditions` — Handling of initial states
`Auto` (default) | `Zero` | `Estimate` | `Backcast`

Set this option when you want to choose a specific method for initializing the model states. With the default setting of `Auto`, the software chooses the method based on the estimation data. Choices are:

- `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.

`Input Intersampling` — Intersampling behavior for input signal
`Zero-order hold` (default) | `Triangle approximation` | `Band-limited`

Input intersampling is a property of the input data. The task uses this property when estimating continuous models. Specify **Input Intersampling** when your data type is `Time` or `Frequency`. If you are using an `iddata` object, the object already contains the intersampling information. Choices for this property are:

- `Zero-order hold` — Piecewise-constant input signal between samples
- `Triangle approximation` — Piecewise-linear input signal between samples, also known as first-order hold
- `Band-limited` — Input signal has zero power above the Nyquist frequency

`Search Method` — Numerical search mode for iterative parameter estimation
`Auto` (default) | `Gauss-Newton` | `Adaptive Gauss-Newton` | `Levenberg-Marquardt` | `Gradient Search`

- `Auto` — For each iteration, the software cycles through the methods until it finds the first direction descent that leads to a reduction in estimation cost.
- `Gauss-Newton` — Subspace Gauss-Newton least-squares search.
- `Levenberg-Marquardt` — Levenberg-Marquardt least-squares search.
- `Adaptive Gauss-Newton` —Adaptive subspace Gauss-Newton search.
- `Gradient Search` — Steepest descent least-squares search.

`Max. Iterations` — Maximum number of iterations during error minimization
20 (default) | positive integer

Set the maximum number of iterations during error minimization. The iterations stop when **Max. Iterations** is reached or another stopping criterion is satisfied, such as **Tolerance**.

`Tolerance` — Minimum percentage of expected improvement in error
0.01 (default) | positive integer

When the percentage of expected improvement is less than **Tolerance**, the iterations stop.

`Weighting Prefilter` — Weighting prefilter for loss function
`No filter` (default) | `Passband(s)` | `LTI Filter` | `Frequency weights vector` | `Inverse of magnitude of the frequency response` | `Inverse of square root of magnitude of the frequency response`

Set this option when you want to apply a weighting prefilter to the loss function that the task minimizes when you estimate the model. When you select an option, you must also select the associated variable in your workspace that contains the filter information. The available options depend on the domain of the data.

Weighting Prefilter	**Data Domain**	**Filter Information**
`No Filter`	Time and frequency	
`Passbands`	Time and frequency	Passband ranges, specified as a 1-by-2 row vector or an *n*-by-2 matrix, where *n* is the number of passbands.
`LTI Filter`	Time and frequency	SISO LTI model.
`Frequency Weights Vector`	Frequency	Frequency weights, specified as a column vector with the same length as the frequency vector.
`Inverse of magnitude of the frequency response`	Frequency response	The weighting filter is$1/\lvert G(\omega)\rvert$, where $G(\omega)$ is the complex frequency-response data. SISO and SIMO systems only.
`Inverse of square root of magnitude of the frequency response`	Frequency response	The weighting filter is $1/\sqrt{\lvert G(\omega)\rvert}$. SISO and SIMO systems only.

For instance, suppose that you are performing estimation with SISO frequency-domain data and that in your MATLAB workspace, you have a column vector `W` that contains frequency weights for the prefilter. In the task, select **Weighting prefilter > Frequency weights vector** and the variable `W`.

Visualize Results

`Output Plot` — Plot comparison of model and measured outputs

on (default) | off

Plot a comparison of the model output and the original measured data, along with the fit percentage. If you have separate validation data, a second plot compares the model response to the validation input data with the measured output from the validation data set.

See Also

`ssest` | `ssestOptions` | `iddata` | `compare` | `idss` | `idfrd` | `frd`

Topics

"What Are State-Space Models?"

Introduced in R2019b

Estimate Spectral Model

Estimate spectral model using time-domain data in the live editor

Description

The **Estimate Spectral Model** task lets you interactively estimate and plot a spectral model using time data. You can specify one of three estimation algorithms and modify the size of the window size that determines frequency resolution. You can also specify the frequency vector, including the number of frequencies and whether those frequencies are evenly spaced on a linear or a logarithmic scale. The task automatically generates MATLAB code for your live script. For more information about Live Editor tasks in general, see “Add Interactive Tasks to a Live Script”.

A frequency-response model is the frequency response of a linear system evaluated over a range of frequency values. The model is represented by an `idfrd` model object that stores the frequency response, sample time, and input-output channel information. For more information about frequency-response models, see “What is a Frequency-Response Model?”.

The **Estimate Spectral Model** task is independent of the more general **System Identification** app. Use the **System Identification** app when you want to compute and compare estimates for multiple models.

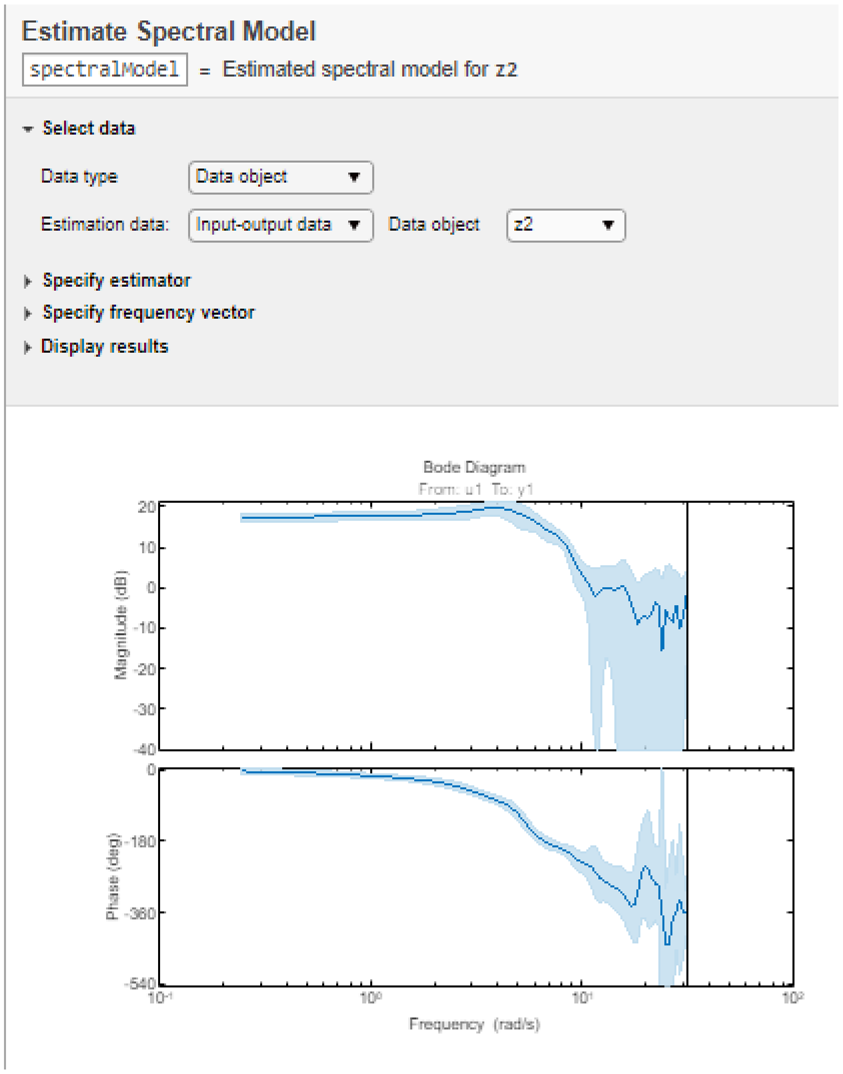
To get started, load experiment data that contains input and output data into your MATLAB workspace and then import that data into the task. Then, specify a model structure to estimate. The task gives you controls and plots that help you experiment with different model parameters and compare how well the output of each model fits the measurements.

Related Functions

The code that **Estimate Spectral Model** generates uses the following functions.

- Data objects:
  - `iddata` — Contains input-output data
- Algorithms for estimating frequency response:
  - `spa`
  - `spafdr`
  - `etfe`
- Frequency Plots:
  - `bode` for input-output data
  - `spectrum` for time series data

The task estimates an `idfrd` frequency-response model.



Estimate Spectral Model

spectralModel = Estimated spectral model for z2

Select data

Data type: Data object

Estimation data: Input-output data   Data object: z2

Specify estimator

Specify frequency vector

Display results

Description (Collapsed Portion)

Related Functions

The code that **Estimate Spectral Model** generates uses the following functions.

- Data objects:
  - `iddata` — Contains input-output data
- Algorithms for estimating frequency response:
  - `spa`
  - `spafdr`
  - `etfe`
- Frequency Plots:

- `bode` for input-output data
- `spectrum` for time series data

The task estimates an `idfrd` frequency-response model.

Open the Task

To add the **Estimate Spectral Model** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Estimate Spectral Model**.
- In a code block in your script, type a relevant keyword, such as `spectral` or `estimate`. Select `Estimate Spectral Model` from the suggested command completions.

Examples

Estimate Spectral Model in the Live Editor

Use the **Estimate Spectral Model** Live Editor Task to estimate a frequency-response model and plot the response.

Open this example to see a preconfigured script containing the task.

Set Up Data

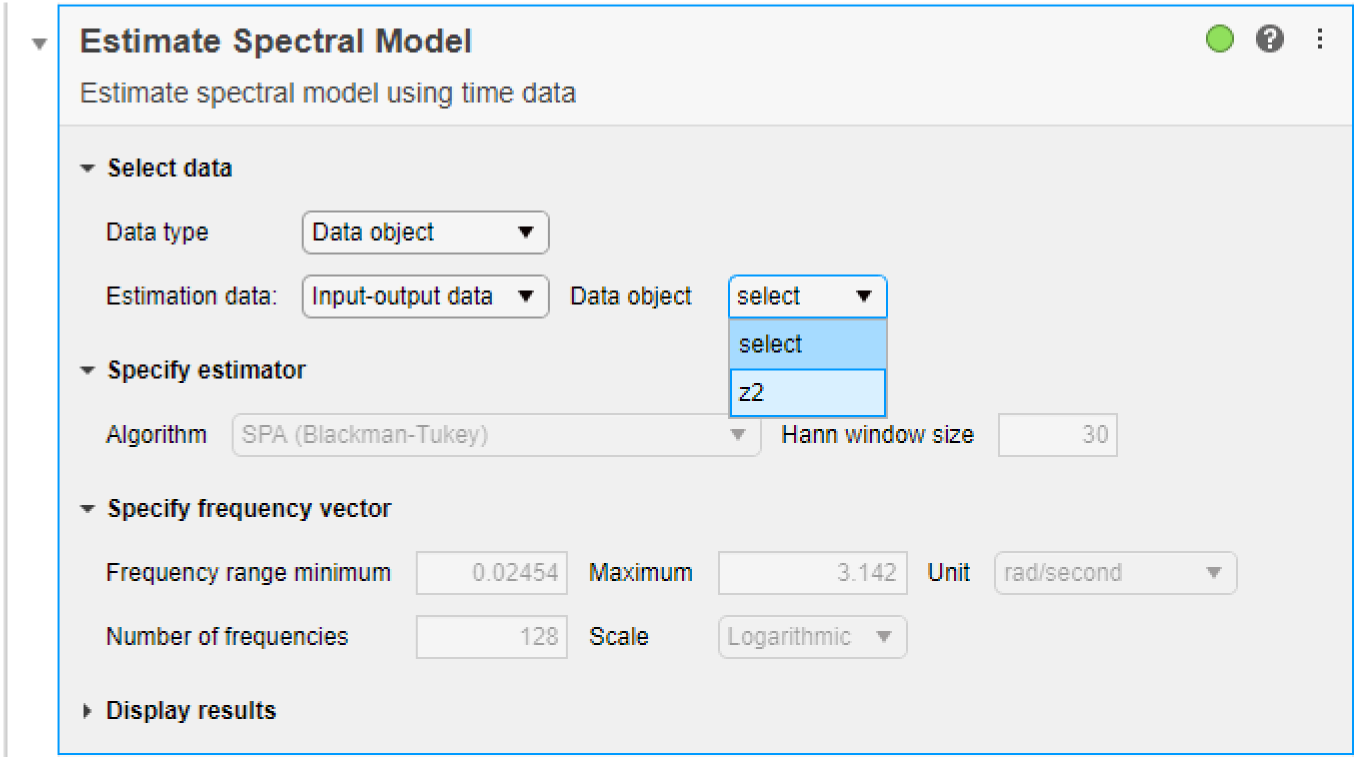
Load the measurement data `iddata2` into your MATLAB workspace.

```
load iddata2 z2
z2
```

```
z2 =

Time domain data set with 400 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Import Data into Task

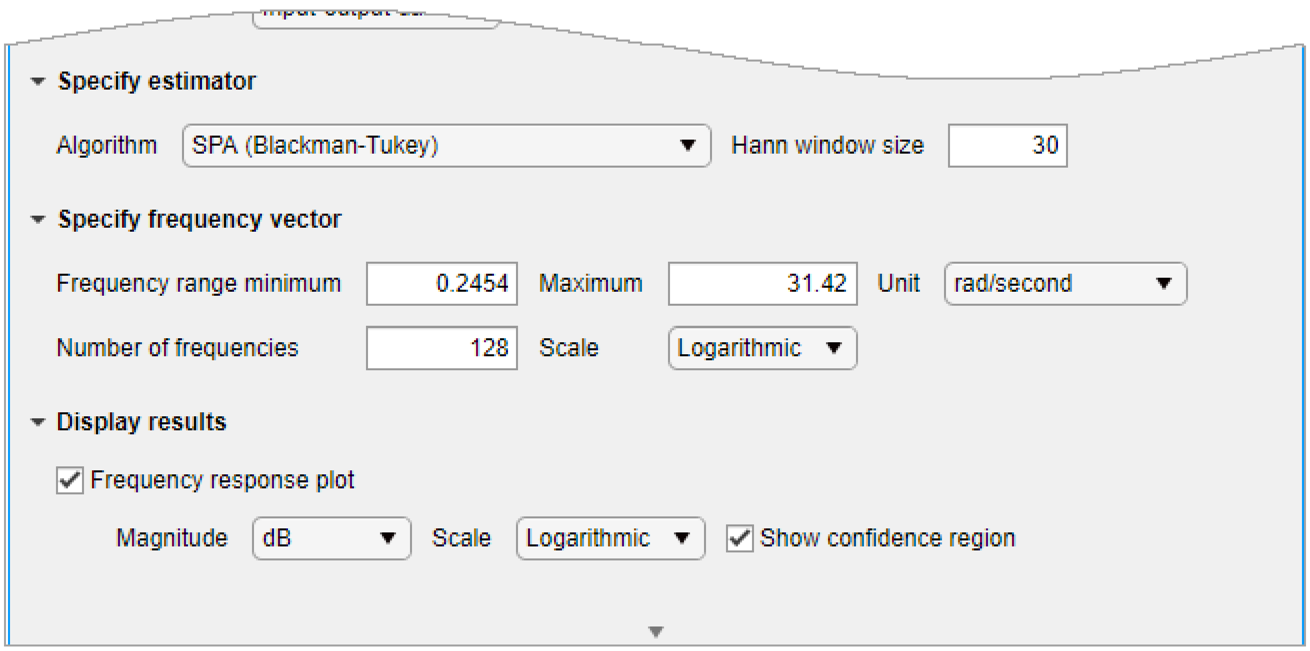
In the **Select data** section, for **Data type**, select `Data object`. For **Estimation data**, select `Input-output data`. In `Data object`, the task displays the workplace variables that meet the criteria that you set. Select `z2`.



A data object contains the input and output variable names as well as the sample time, so you do not need to specify them.

Estimate Model Using Default Settings

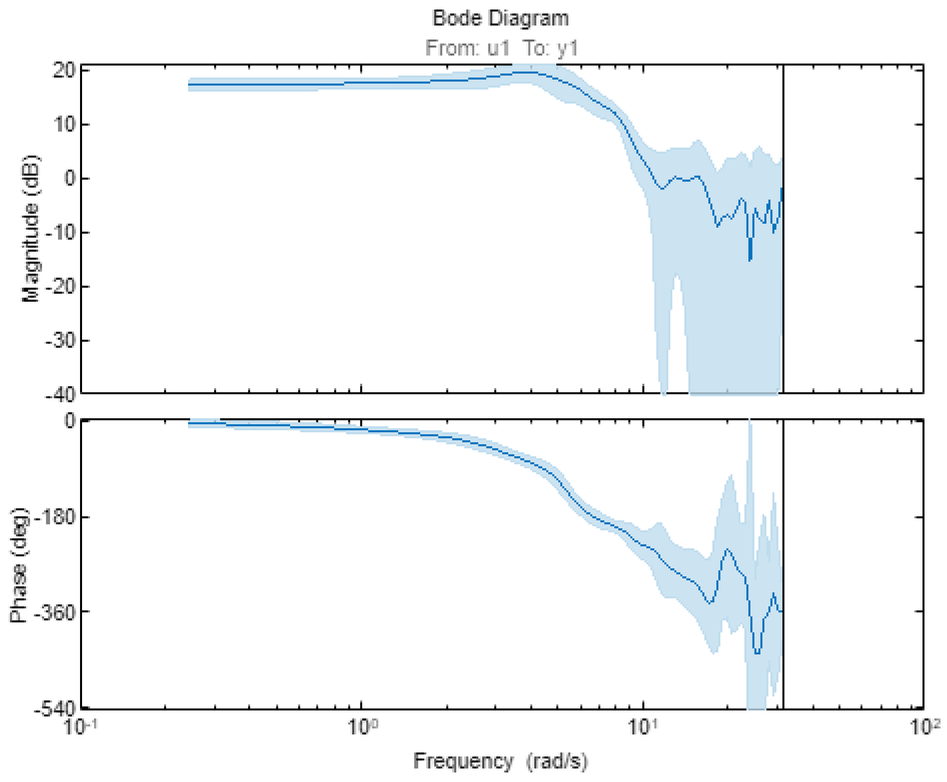
The default algorithm is `SPA (Blackman-Tukey)`.



Run the task using this algorithm and the default settings for **Specify frequency vector** and **Display results**.

Examine Plot

The task displays a Bode plot that includes a confidence region of three standard deviations.



Parameters

Select Data

`Data Type` — Data type for input and output data

`Time` (default) | `Data object`

The task accepts numeric measurement values that are uniformly sampled in time. Input and output signals can contain multiple channels. Data can be packaged either as numeric arrays (for `Time`) or in an `iddata` object (for `Data object`).

The data type you choose determines whether you must specify additional parameters.

- `Time` — Specify **Sample Time** in the time unit that you select.
- `Data Object` — Specify no additional parameters because the data object already contains information on time sampling.

`Estimation Data` — Estimation data input and output content

`Input-output data` (default) | `Time series`

The task accepts input-output data and time series data that has no input array.

The estimation data content you select, along with your selection of **Data Type**, determines your options for accessing variables from your MATLAB workspace.

- `Time series` and `Input-output data` — Select the variable names of your input and output vectors for **Input (u)** and **Output (y)**, respectively. If **Data Type** is `Time series`, then you can select only a single vector, using **Output (y)**.
- `Data object` — Select the variable name of your data object.

Specify estimator

`Algorithm` — Algorithm to use

`SPA (Blackman-Tukey)` (default) | `SPAFDR (Frequency-dependent resolution)` | `ETFE (Smoothed Fourier transform)`

The task provides three algorithms to choose from.

- `SPA` — Blackman-Tukey Spectral analysis (SPA) method. Takes the Fourier transform of windowed versions of the covariance function.
- `SPAFDR` — Variant of the SPA method that uses frequency-dependent resolution.
- `ETFE` — Empirical transfer function estimate. This method computes the ratio of the Fourier transform of the output to the Fourier transform of the input. For time series, which have no input, this method computes a periodogram as the normalized absolute squares of the Fourier transform of the time series.

For more information on these algorithms, see `spa`, `spafdr`, and `etfe`. For information on selecting an algorithm, see "Selecting the Method for Computing Spectral Models".

`Window Size or Resolution` — Window size parameter

method-dependent resolution value

Each estimation algorithm uses a unique parameter for determining and using the window size.

- `SPA` — **Hann window size**. Specify this parameter as a positive integer greater than 2. The default value is equal to 30 for data arrays with lengths of 300 or more, or, for smaller arrays, *arraylength*/10.
- `SPAFDR` — **Resolution**. Specify this parameter in rad/`TimeUnit`, where `TimeUnit` is the unit you specify for **Sample Time**. The resolution is the size of the smallest detail in the frequency function and the spectrum that is resolved by the estimate. Setting the resolution is a tradeoff between obtaining estimates with fine, reliable details, and suffering from spurious, random effects. The default value in the task is `default`, which uses the resolution that `spafdr` calculates based on the frequencies. If you want to view this resolution value for the SISO model `spectralModel`, at the command line, enter `spectralModel.Report.WindowSize`.
- `ETFE` — **Hamming window size**. Specify this parameter, which represents frequency resolution, as a positive integer greater than 2. The value of the parameter determines the amount of smoothing that the function applies to the raw spectral estimates. The default value in the task is `default`, which uses the resolution that `etfe` calculates based on the frequencies. If you want to view this resolution value for the SISO model `spectralModel`, at the command line, enter `spectralModel.Report.WindowSize`.

Specify frequency vector

`Frequency range parameters` — Frequency range minimum, maximum, and units

numeric values | unit string

Specify the frequency vector minimum and maximum, and select the unit, such as the default `rad/second`, from the **Unit** list. By default, the task sets the frequency to span the range bounded at the upper end by the Nyquist frequency, which is a function of the sample time. The task sets the default value of the lower end of the range to the first frequency value.

`Number of frequencies and scale` — Number of frequency divisions and linear or logarithmic scale selection

128 | integer | `Logarithmic` | `Linear`

Specify the number of frequency divisions and whether to use a linear or a logarithmic scale. The default number of divisions is `128`. The default scale is `Logarithmic`.

Display Results

`Frequency response plot` — Plot the frequency response

on (default) | off

Select **Frequency response plot** to create a frequency plot of your model. If you specify your data type as `Input-output data`, then the task creates the frequency response using `bode`. If your data type is `Time series`, then the task plots the power spectrum using `spectrum`.

You can plot only one model at a time in the task. If you want to compare responses, do one of the following:

- Open multiple tasks and visually compare plots for different models.
- Use unique model IDs for each model you want to compare, and then create Bode plots for them at the command line.

`Frequency response plot parameters` — Magnitude units, scale, confidence region

`dB` | `Absolute` | `Logarithmic` | `Linear` | on | off

Specify the parameters for the Bode or power spectrum plot. You can specify that the units in **Magnitude** are dB or absolute value. For **Scale**, you can specify a logarithmic or a linear scale for the frequency axis. If you are creating a Bode plot by using input-output data, you can select **Show confidence region** to display a confidence region of three standard deviations. If you are creating a power spectrum plot by using a time series, no **Show confidence region** option exists.

See Also

`idfrd` | `iddata` | `spafdr` | `spa` | `etfe` | `bode` | `spectrum`

Topics

"Frequency Response Plots for Model Validation"
"Selecting the Method for Computing Spectral Models"

Introduced in R2021b

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.

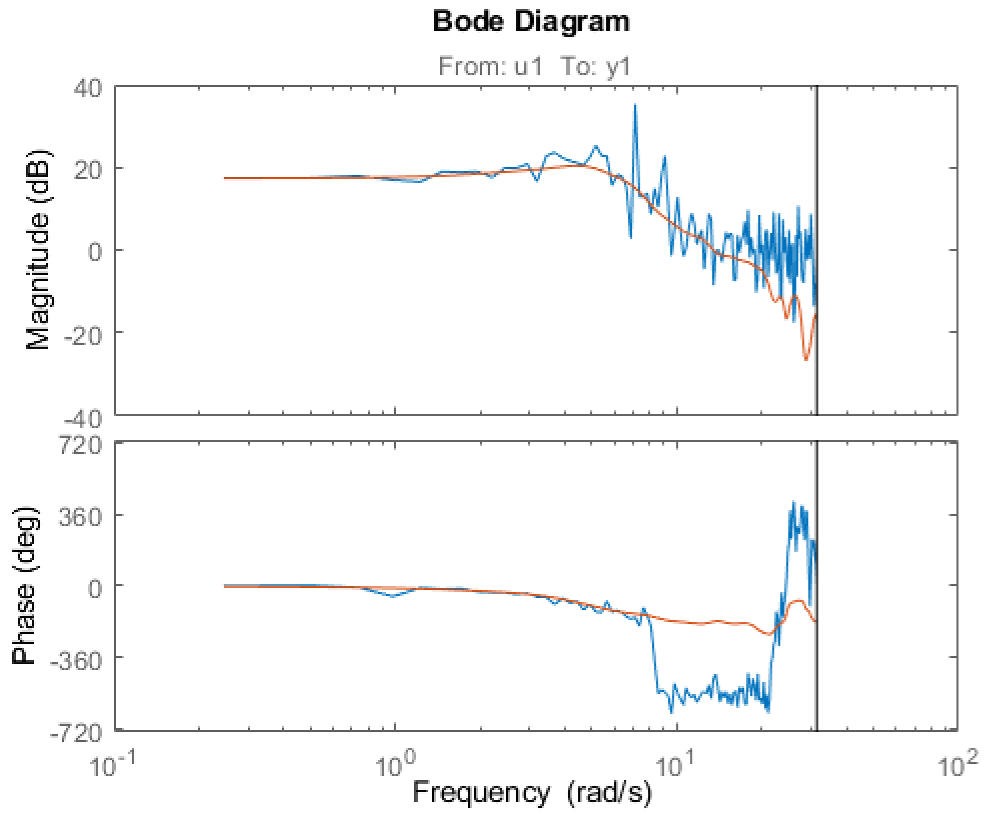
```
load iddata1 z1;
```

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.

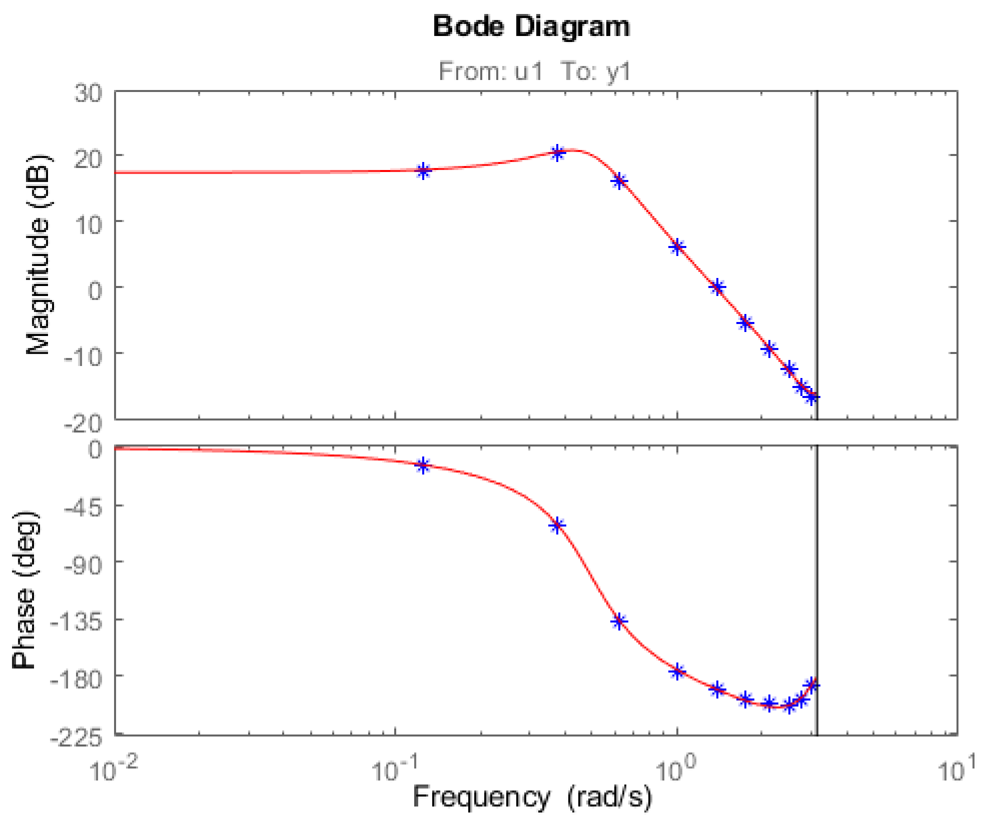
```
m = idpoly([1 -1.5 0.7],[0 1 0.5]);
u = iddata([],idinput([50,1,10],'sine'));
u.Period = 50;
y = sim(m,u);
```

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
```

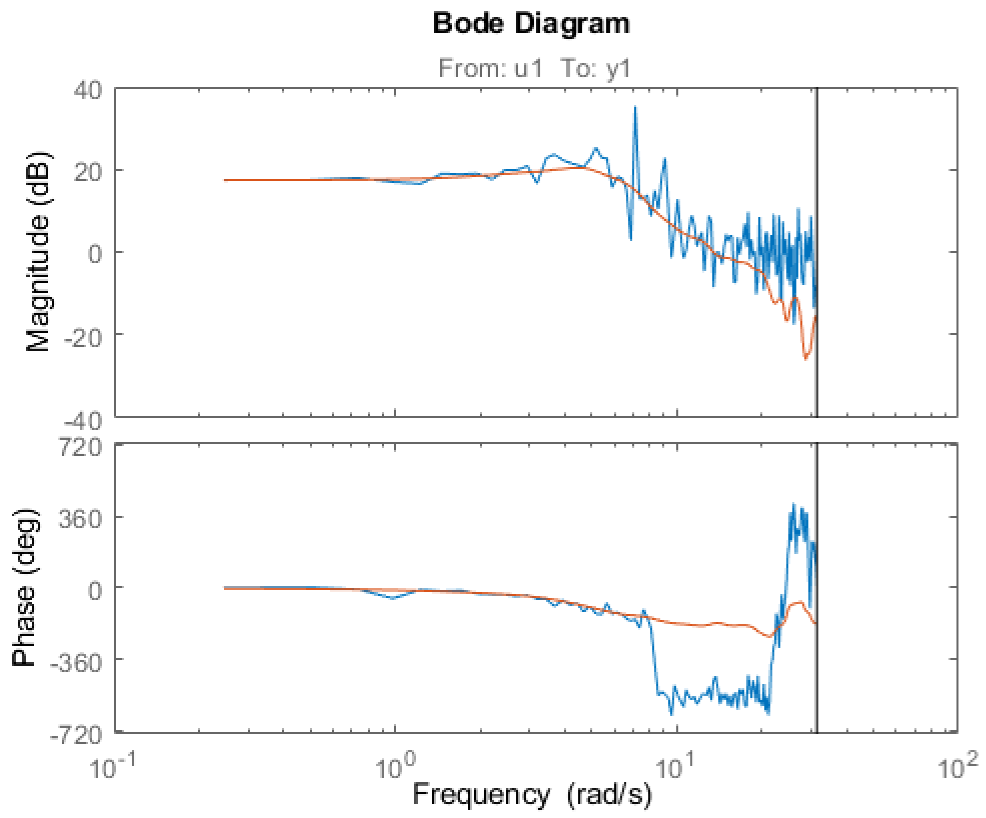
Estimate empirical transfer functions with and without the smoothing operation.

```
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.

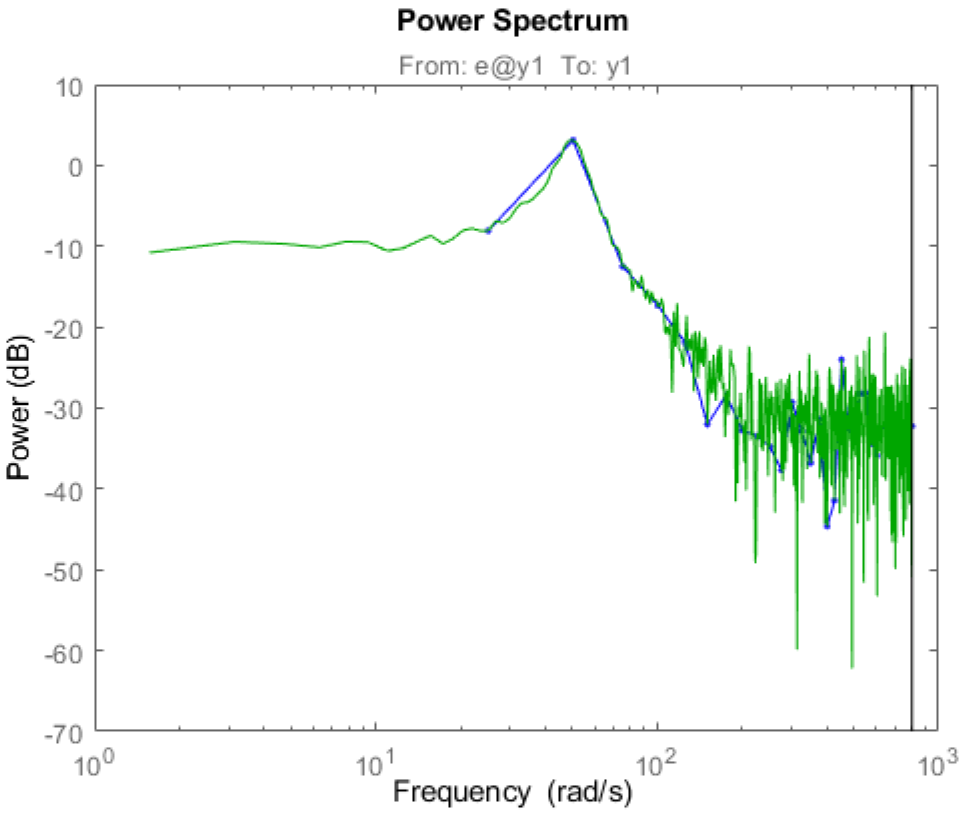
```
load iddata9
```

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')
```



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:

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.
`WindowSize`	Size of the Hamming window.
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields.

Field	Description
`Name`	Name of the data set.
`Type`	Data type.
`Length`	Number of data samples.
`Ts`	Sample time.
`InterSample`	Input intersample behavior, returned as one of the following values: • `'zoh'` — 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.
`InputOffset`	Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`.
`OutputOffset`	Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.

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

See Also

`bode` | `freqresp` | `idfrd` | `nyquist` | `spa` | `spafdr` | `impulseest` | `spectrum`

Topics

“Estimate Frequency-Response Models at the Command Line”
“What is a Frequency-Response Model?”

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

Evaluate Discrete-Time Transfer Function

Create the following discrete-time transfer function.

$$H(z) = \frac{z-1}{z^2+z+1}$$

```
H = tf([1 -1],[1 1 1],-1);
```

Evaluate the transfer function at `z = 1+j`.

```
z = 1+j;
evalfr(H,z)
```

```
ans = 0.2308 + 0.1538i
```

Evaluate Frequency Response of Identified Model at Given Frequency

Create the following continuous-time transfer function model:

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

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

Evaluate the transfer function at frequency 0.1 rad/second.

```
w = 0.1;
s = j*w;
evalfr(sys,s)
```

```
ans = 0.9705 - 0.1961i
```

Alternatively, use the `freqresp` command.

```
freqresp(sys,w)
```

```
ans = 0.9705 - 0.1961i
```

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(StateTransitionFcn,MeasurementFcn)
obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,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(StateTransitionFcn,MeasurementFcn)` creates an extended Kalman filter object using the specified state transition and measurement functions. Before using the `predict` and `correct` commands, specify the initial state values using dot notation. For example, for a two-state system with initial state values `[1;0]`, specify `obj.State = [1;0]`.

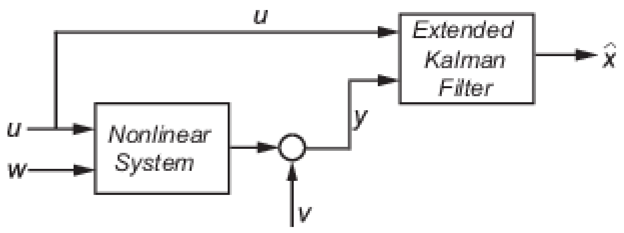
`obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,Name,Value)` specifies additional attributes of the extended Kalman filter object using one or more `Name,Value` pair arguments. Before using the `predict` and `correct` commands, specify the initial state values using `Name,Value` pair arguments or dot notation.

`obj = extendedKalmanFilter(Name,Value)` creates an extended Kalman filter object with properties specified using one or more `Name,Value` pair arguments. Before using the `predict` and `correct` commands, specify the state transition function, measurement function, and initial state values using `Name,Value` pair arguments or dot notation.

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 $\mathtt{u}_s$ and $\mathtt{u}_m$ 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.
`residual`	Return the difference between the actual and predicted measurements.
`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 1-338.

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.

Copyright 2012 The MathWorks, Inc

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 Jacobian 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;
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.MeasurementFcn = @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 following 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 the next time step, given `u[k]=0.2`.

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

Retrieve the error, or *residual*, between the prediction and the measurement.

```
[Residual, ResidualCovariance] = residual(obj,0.8,0.2);
```

Input Arguments

`StateTransitionFcn` — State transition function
function handle

State transition function *f*, specified as a function handle. The function calculates the *Ns*-element state vector of the system at time step *k*, given the state vector at time step *k*-1. *Ns* 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 *Ns*-element vector, where *Ns* 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 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 on page 1-338 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,'ProcessNoise',0.01);
```

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
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-*Ns* Jacobian matrix, where *N* is the number of measurements of the system and *Ns* 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 *Ns*-by-*Ns* matrix, where *Ns* 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 *Ns*-by-*Ns* 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 *Ns*, where *Ns* 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 *Ns*-by-*Ns* matrix, where *Ns* is the number of states of the system. If you specify a scalar, the software uses the scalar value to create an *Ns*-by-*Ns* 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
function handle

State transition function *f*, specified as a function handle. The function calculates the *Ns*-element state vector of the system at time step *k*, given the state vector at time step *k*-1. *Ns* 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 state transition 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 *Ns*-by-*Ns* Jacobian matrix, where *Ns* 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 process noise terms ($\partial f/\partial w$). The second output is returned as an *Ns*-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 on page 1-338. 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*.

Compatibility Considerations

Numerical Changes
Behavior changed in R2020b

Starting in R2020b, numerical improvements in the `extendedKalmanFilter` algorithm might produce results that are different from the results you obtained in previous versions.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

For more information, see “Generate Code for Online State Estimation in MATLAB”.

Generated code uses an algorithm that is different from the algorithm that the `extendedKalmanFilter` function uses. You might see some numerical differences in the results obtained using the two methods.

Supports MATLAB Function block: No

See Also

Functions
`predict` | `correct` | `residual` | `clone` | `unscentedKalmanFilter`

Blocks
Kalman Filter | Extended Kalman Filter | Unscented Kalman Filter

Topics
“Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter”

"Generate Code for Online State Estimation in MATLAB"
"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"

External Websites
Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

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 `idCustomNetwork`, `idDeadZone`, `idLinear`, `idFeedforwardNetwork`, `idPolynomial1D`, `idUnitGain`, `idPiecewiseLinear`, `idSaturation`, `idSigmoidNetwork`, `idTreePartition`, or `idWaveletNetwork`.

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

`fdel` | `fselect` | `interp` | `frd` | `idfrd`

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

Delete Specified Data from Frequency Response Data Model

Create a frequency response data (FRD) model at specified frequencies from a transfer function model.

```
w = logspace(0,1,10);
sys = frd(tf([1],[1 1]),w)
```

```
sys =

    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.
```

`w` is a logarithmically-spaced grid of 10 frequency points between 1 and 10 rad/second.

Remove the data nearest 2, 3.5, and 6 rad/s from `sys`.

```
freq = [2, 3.5, 6];
sys2 = fdel(sys,freq)
```

```
sys2 =

    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.
```

Note that you do not have to specify the exact frequency of the data to remove. The `fdel` command removes the data corresponding to frequencies that are nearest to the specified frequencies.

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
`fcat` | `fselect` | `frd` | `idfrd`

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)+1` 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`

Topics

“Representing Data in MATLAB Workspace”

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.SearchOptions.MaxIterations = 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: 11
               FinalCost: 0
    FirstOrderOptimality: 0
            SignalLevels: [1x1 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 1-623.

`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
`SearchMethod`	Search method used for iterative parameter estimation. See `SearchMethod` in `findopOptions` for more information.
`WhyStop`	Search algorithm termination condition.
`Iterations`	Number of estimation iterations performed.
`FinalCost`	Final value of the minimization objective function (sum of the squared errors).
`FirstOrderOptimality`	∞-norm of the search gradient vector when the search algorithm terminates.
`SignalLevels`	Structure containing the fields `Input` and `Output`, which are the operating point input and output signal levels respectively.

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 $e(t) = y(t)\text{-}f(x(t), u(t))$, 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 1-623.

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.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `findopOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = findopOptions(idnlarx);
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

`idnlarx` | `findopOptions` | `idnlarx/operspec` | `data2state` | `sim` | `idnlhw/findop`

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');
```

```
Warning: "unitgain" is deprecated and may be removed in a future release. Use "idUnitGain" instea

Warning: "pwlinear" is deprecated and may be removed in a future release. Use "idPiecewiseLinear"
```

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.SearchOptions.MaxIterations = 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,[],idPiecewiseLinear);
```

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: 0
    FirstOrderOptimality: 0
            SignalLevels: [1x1 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 1-654.

`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
`SearchMethod`	Search method used for iterative parameter estimation. See `SearchMethod` in `findopOptions` for more information.
`WhyStop`	Search algorithm termination condition.
`Iterations`	Number of estimation iterations performed.
`FinalCost`	Final value of the minimization objective function (sum of the squared errors).
`FirstOrderOptimality`	∞-norm of the search gradient vector when the search algorithm terminates.
`SignalLevels`	Structure containing the fields `Input` and `Output`, which are the operating point input and output signal levels respectively.

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 = 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 = 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 = 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.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `findopOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = findopOptions(idnlhw);
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

`idnlhw` | `findopOptions` | `idnlhw/operspec` | `sim` | `idnlarx/findop`

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.SearchOptions.MaxIterations = 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 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'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.

- Model structures where the loss function is a nonlinear or non smooth function of the parameters.
- Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOptions` Structure When `SearchMethod` is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Tolerance`	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`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	1e-6
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar.	1e-6
MaxIterations	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance.	100

To specify field values in SearchOptions, 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.SearchOptions.MaxIterations = 15;
opt.SearchOptions.Advanced.RelImprovement = 0.5;
```

Output Arguments

opt — Option set for findop command
findopOptions object

Option set for findop command, returned as a findopOptions object.

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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. 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.

```
FileName = 'dcmotor_m';
Order = [2 1 2];
Parameters = [0.24365;0.24964];
nlgr = idnlgrey(FileName,Order,Parameters);
nlgr = setinit(nlgr, 'Fixed', false(2,1)); % set initial states free
```

Load data for initial state estimation.

```
load(fullfile(matlabroot,'toolbox','ident',...
     'iddemos','data','dcmotordata'));
z = iddata(y,u,0.1);
```

Estimate the initial states such that the model's response using the estimated states X0 and measured input u is as close as possible to the measured output y.

```
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.

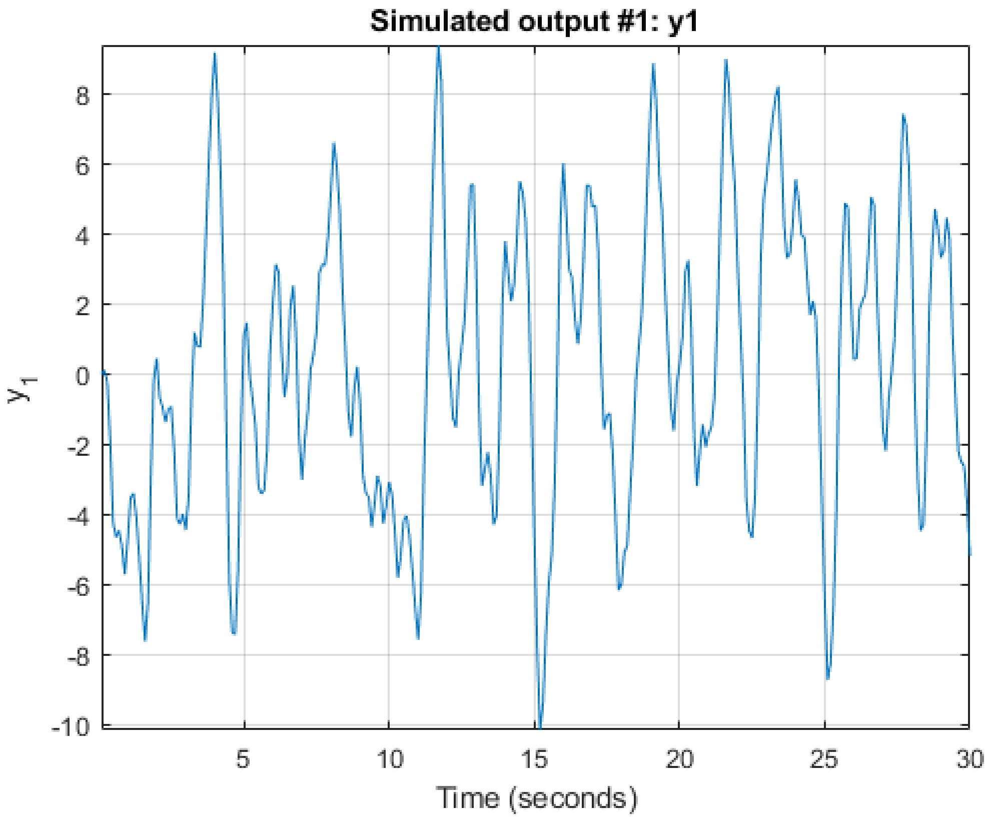
```
model = ssest(z1,2);
```

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 = 2×1

         0
    0.0061


Report = 
         Status: 'Estimated by simulation error minimization'
         Method: 'lsqnonlin'
     Covariance: [2x2 double]
       DataUsed: [1x1 struct]
    Termination: [1x1 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 = 2×1

    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

Scenario	**Horizon**
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:

Report Field	**Description**
`Status`	Summary of how the initial state were estimated.
`Method`	Search method used.
`Covariance`	Covariance of state estimates, returned as a *Ns*-by-*Ns* matrix, where *Ns* is the number of states.

Report Field	Description	
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields.	
	Field	**Description**
	`Name`	Name of the data set.
	`Type`	Data type.
	`Length`	Number of data samples.
	`Ts`	Sample time.
	`InterSample`	Input intersample behavior, returned as one of the following values: • `'zoh'` — 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.
	`InputOffset`	Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`.
	`OutputOffset`	Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.
`Termination`	Termination conditions for the iterative search used for initial state estimation of nonlinear models. Structure with the following fields:	
	Field	**Description**
	`WhyStop`	Reason for terminating the numerical search.
	`Iterations`	Number of search iterations performed by the estimation algorithm.
	`FirstOrderOptimality`	∞-norm of the gradient search vector when the search algorithm terminates.
	`FcnCount`	Number of times the objective function was called.
	`UpdateNorm`	Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`.
	`LastImprovement`	Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`.
	`Algorithm`	Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used.
	`Termination` is empty for linear models.	

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `findstatesOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = findstatesOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

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 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.

  For nonlinear grey-box models, only those initial states `i` that are designated as free in the model (`sys.InitialStates(i).Fixed = false`) are estimated. To estimate all the states of the model, first specify all the `Nx` states of the `idnlgrey` model `sys` as free.

  ```
  for i = 1:Nx
  sys.InitialStates(i).Fixed = false;
  end
  ```

  Similarly, to fix all the initial states to values specified in `sys.InitialStates`, first specify all the states as fixed in the `sys.InitialStates` property of the nonlinear grey-box model.
- `'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'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.

- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm

search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions '` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOptions` Structure When `SearchMethod` is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Tolerance`	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`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see "Large-Scale vs. Medium-Scale Algorithms" (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see "Constrained Nonlinear Optimization Algorithms" (Optimization Toolbox) and "Choosing the Algorithm" (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	`1e-6`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`.	`100`

To specify field values in `SearchOptions` , 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.SearchOptions.Tolerance = 0.02;
opt.SearchOptions.Advanced.MaxBisections = 30;
```

Output Arguments

opt — Option set for `findstates`
`findstatesOptions` option set

Option set for `findstates`, returned as an `findstatesOptions` option set.

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note "Renaming of Estimation and Analysis Options".

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 1-402.

`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.

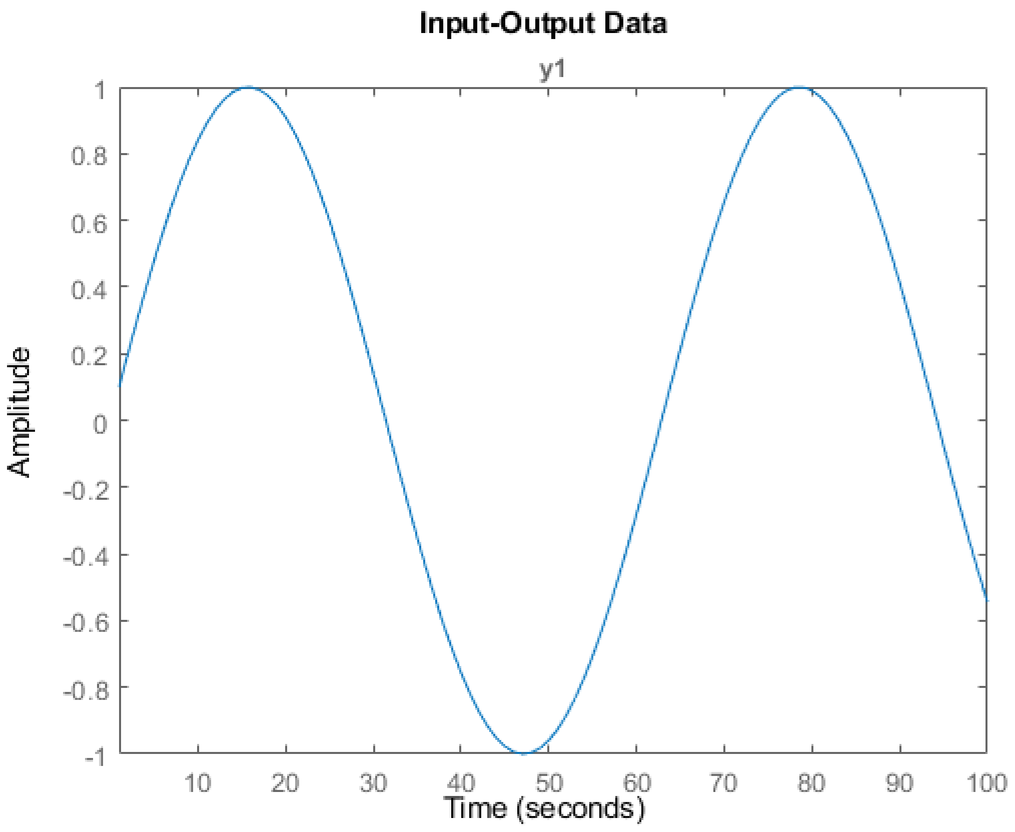
Examples

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);
```

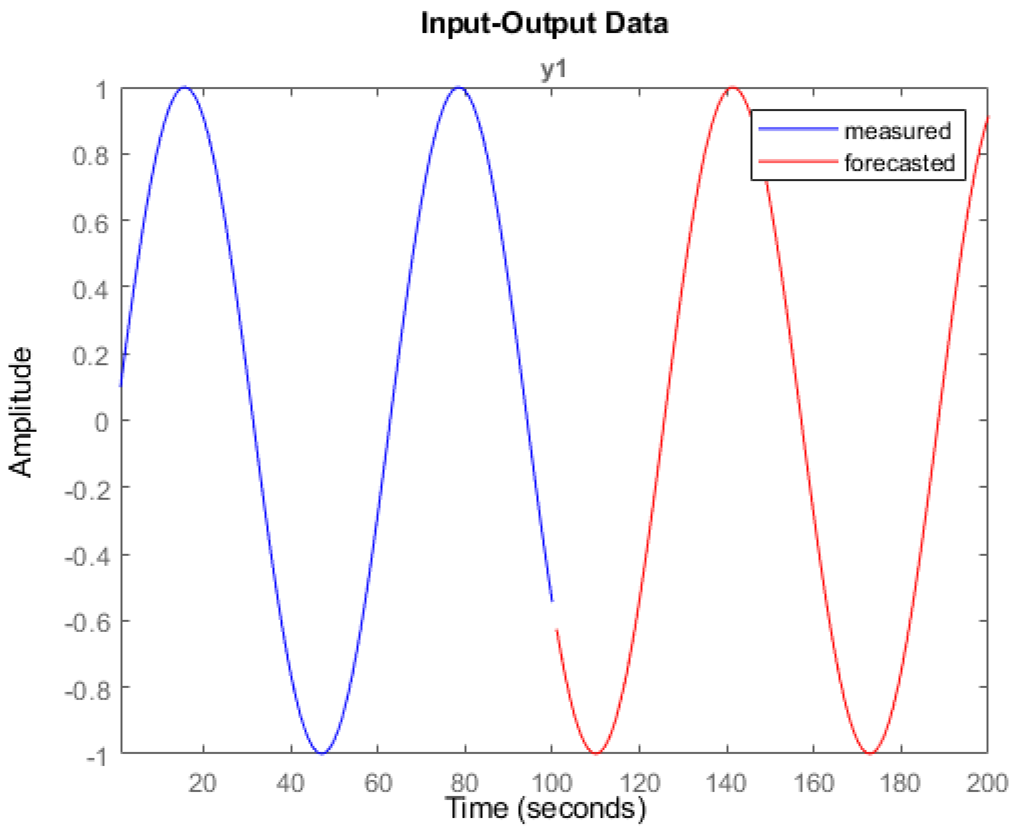
Forecast the values into the future for a given time horizon.

```
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.

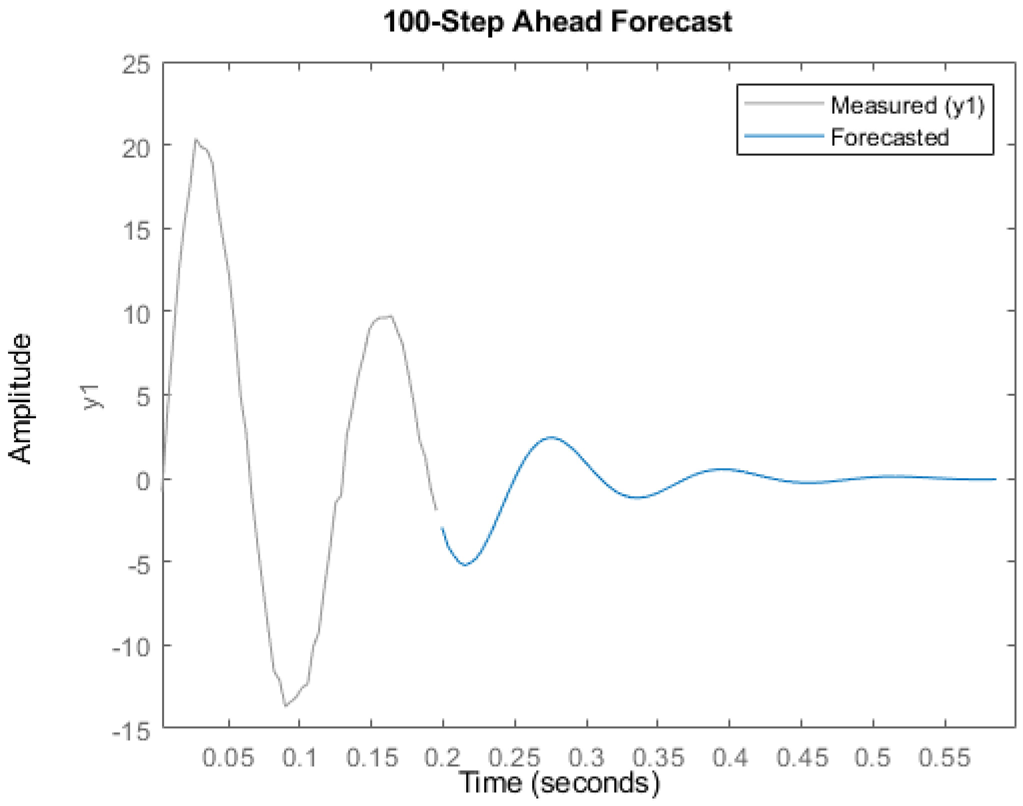
`model` is an `idpoly` time series model.

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

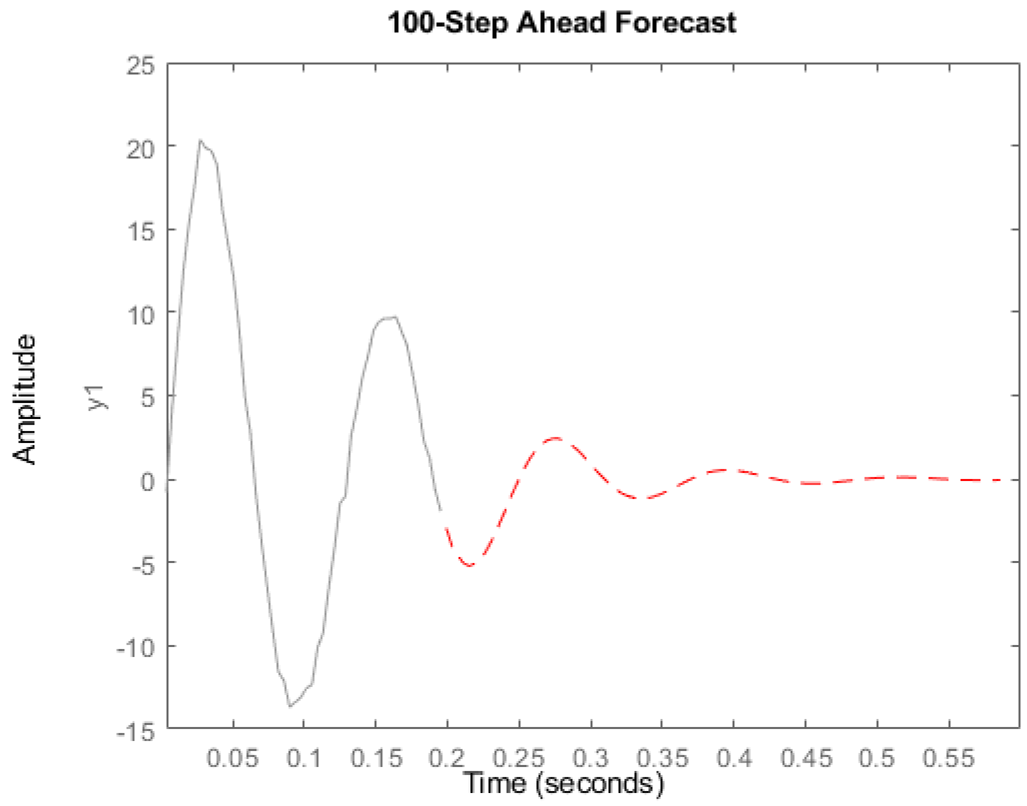
Obtain past data, and identify a time series model.

```
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`.

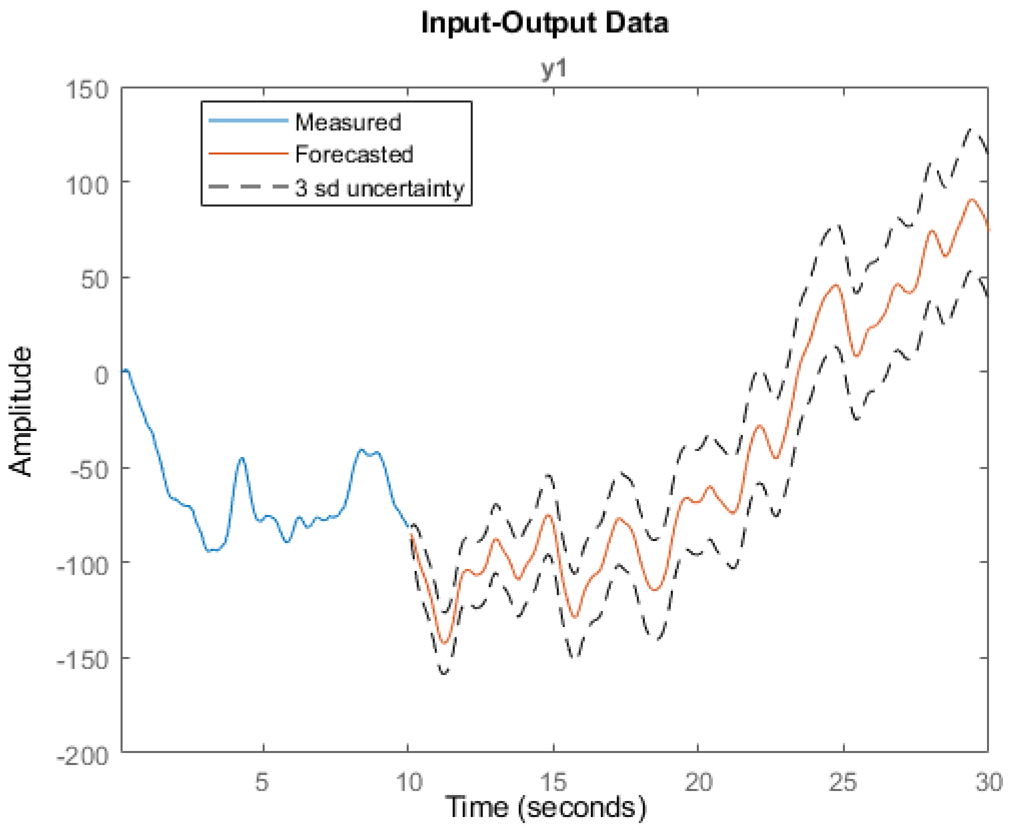
Forecast the system response into the future for a given time horizon and future inputs.

```
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`.

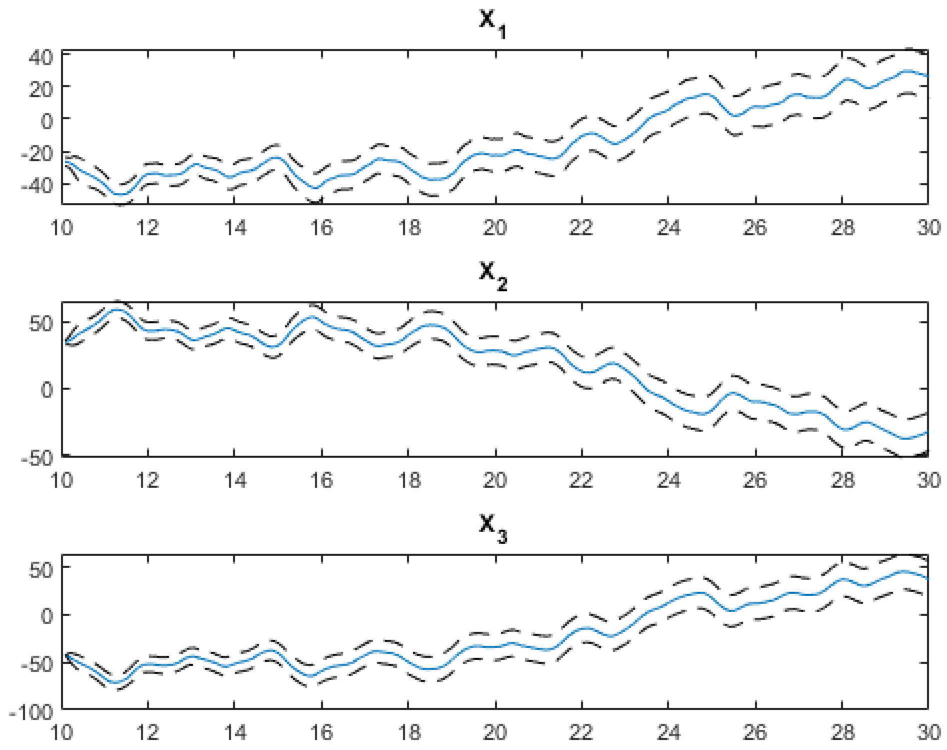
Plot the forecasted response.

```
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.

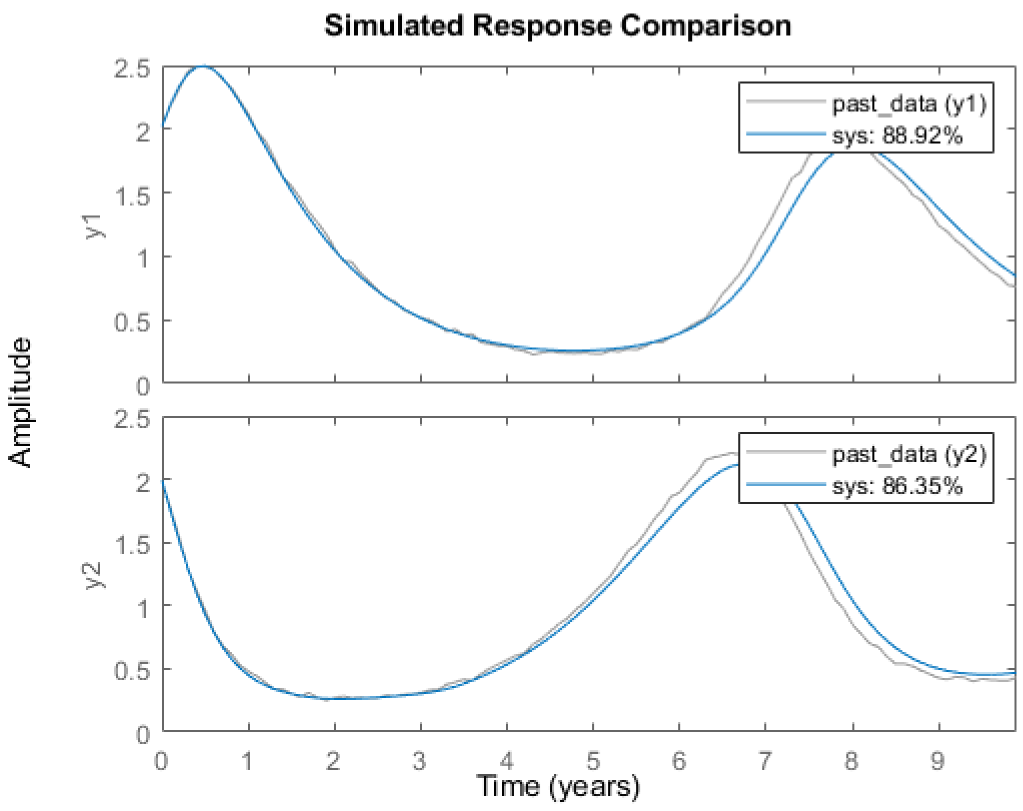
```
C = {{'y1(t-1)*y2(t-1)'};{'y1(t-1)*y2(t-1)','y2(t-1)^2'}};
```

Estimate a nonlinear ARX model using `past_data` as estimation data.

```
sys = nlarx(past_data,[na nb nk],'idWaveletNetwork','CustomRegressors',C);
```

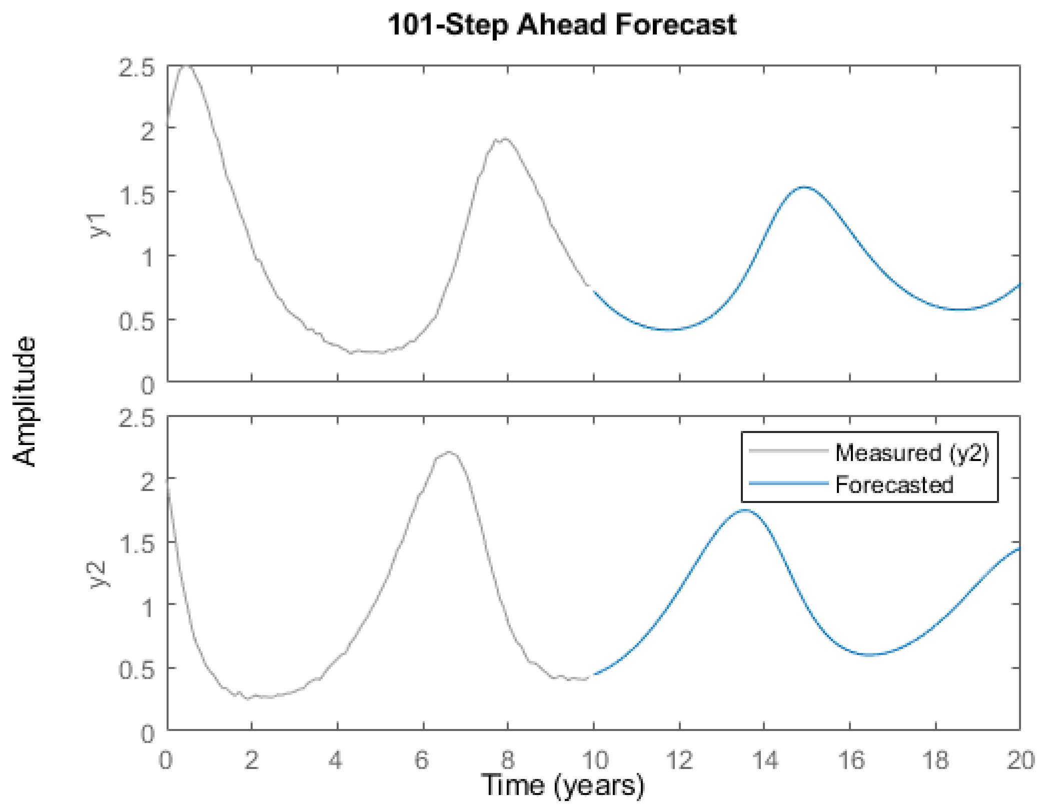
Compare the simulated output of `sys` with measured data to ensure it is a good fit.

```
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);
```

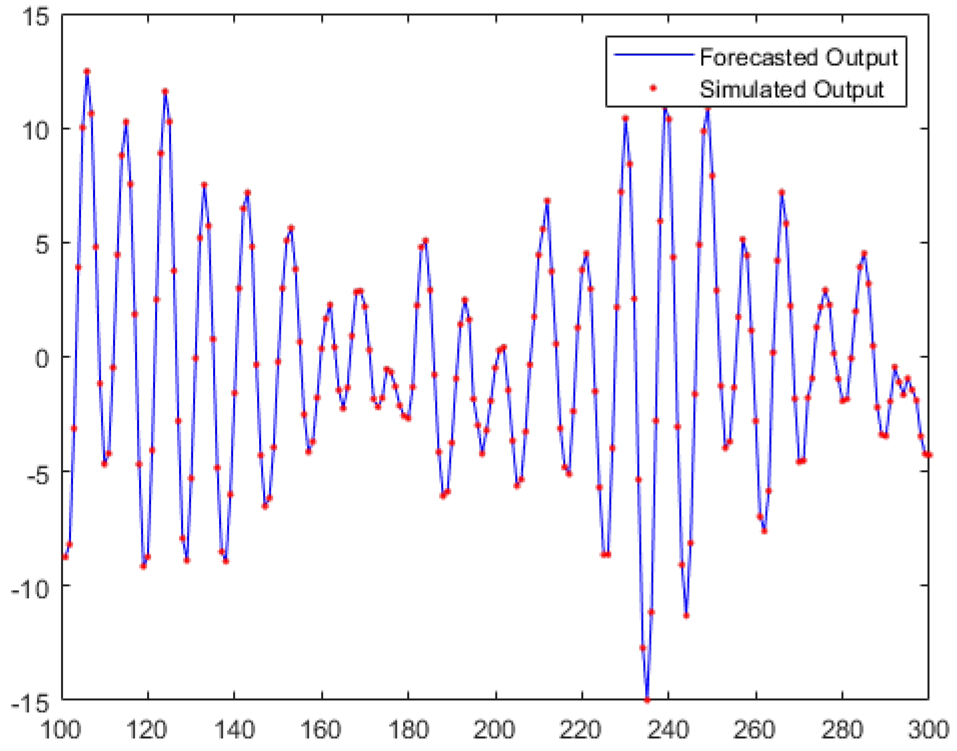
`yf` is the forecasted model response, `x0` is the estimated value for initial states, and `sysf` is the forecasting state-space model.

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`.

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 the Linespec argument of `plot`.

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 multi-experiment, `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 1-623, and “Definition of idnlhw States” on page 1-654.

`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. The software computes the standard deviation by taking into account the model parameter covariance, initial state covariance, and additive noise covariance. The additive noise covariance is stored in the `NoiseVariance` property of the model.

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. The software computes the standard deviation by taking into account the model parameter covariance, initial state covariance, and additive noise covariance. The additive noise covariance is stored in the NoiseVariance property of the model.

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).

Tips

- Right-clicking the plot opens the context menu, where you can access the following options:
  - **Systems** — Select systems to view forecasted output. 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.

See Also
`forecastOptions` | `predict` | `compare` | `sim` | `ar` | `arx` | `ssest` | `iddata`

Topics
“Forecast Output of Dynamic System”
“Forecast Multivariate Time Series”
“Time Series Prediction and Forecasting for Prognosis”
“Introduction to Forecasting of Dynamic System Response”

Introduced in R2012a

forecastOptions

Option set for `forecast`

Syntax

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

Description

`opt = forecastOptions` creates the default option set for `forecast`. Use dot notation to modify this option set. Any options that you do not modify retain their default values.

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

Examples

Specify Input Signal Offset for Model Forecasting

Create a default option set for `forecast`.

```
opt = forecastOptions;
```

Specify the input offset for a single-input data set as 5.

```
opt.InputOffset = 5;
```

You can now use this option set for forecasting. Before forecasting model response, the `forecast` command subtracts this offset value from the past input data signal.

Specify Handling of Initial Conditions During Model Forecasting

Create an option set for `forecast` using zero initial conditions.

```
opt = forecastOptions('InitialCondition','z');
```

Specify Output Offset for Forecasting Multi-Experiment Data

Load past measured data from two experiments.

```
load iddata1
load iddata2
```

`z1` and `z2` are `iddata` objects that store SISO input-output data. Create a two-experiment data set from `z1` and `z2`.

```
z = merge(z1,z2);
```

Estimate a transfer function model with 2 poles using the multi-experiment data.

```
sys = tfest(z,2);
```

Specify the offset as -1 and 1 for the output signals of the two experiments.

```
opt = forecastOptions('OutputOffset',[-1 1]);
```

`OutputOffset` is specified as an *Ny-by-Ne* matrix where *Ny* is the number of outputs in each experiment, and *Ne* is the number of experiments. In this example, *Ny* is 1 and *Ne* is 2.

Using the option set `opt`, forecast the response of the model 10 time steps into the future. The software subtracts the offset value `OutputOffset(i,j)` from the output signal *i* of experiment *j* before using the data in the forecasting algorithm. The removed offsets are added back to generate the final result.

```
y = forecast(sys,z,10,opt)
```

```
y =
Time domain data set containing 2 experiments.

Experiment   Samples      Sample Time
   Exp1         10            0.1
   Exp2         10            0.1

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

`y` is an `iddata` object that returns the forecasted response corresponding to each set of past experimental data.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `forecastOptions('InitialCondition','e')` specifies that the software estimates the initial conditions of the measured input-output data such that the 1-step prediction error for observed output is minimized.

`InitialCondition` — Handling of initial conditions

`'e'` (default) | `'z'` | `idpar` object `x0Obj`

Handling of initial conditions, specified as the comma-separated pair consisting of `'InitialCondition'` and one of the following values:

- `'z'` — Zero initial conditions.
- `'e'` — Estimate initial conditions such that the 1-step prediction error is minimized for the observed output.

  For nonlinear grey-box models, only those initial states `i` that are designated as free in the model (`sys.InitialStates(i).Fixed = false`) are estimated. To estimate all the states of the model, first specify all the `Nx` states of the `idnlgrey` model `sys` as free.

  ```
  for i = 1:Nx
  sys.InitialStates(i).Fixed = false;
  end
  ```

  Similarly, to fix all the initial states to values specified in `sys.InitialStates`, first specify all the states as fixed in the `sys.InitialStates` property of the nonlinear grey-box model.
- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space models only (`idss`, `idgrey`, and `idnlgrey`). Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum or maximum bounds.

`InputOffset` — Input signal offset
[ ] (default) | column vector | matrix

Input signal offset for time-domain data, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following values:

- `[]` — No input offsets.
- A column vector of length *Nu*, where *Nu* is the number of inputs. When you use the `forecast` command, the software subtracts the offset value `InputOffset(i)` from the *i*th input signals in the past and future input values. You specify these values in the `PastData` and `FutureInputs` arguments of `forecast`. The software then uses the offset subtracted inputs to forecast the model response.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *Nu*-by-*Ne* matrix, where *Ne* is the number of experiments. The software subtracts the offset value `InputOffset(i,j)` from the *i*th input signal of the *j*th experiment in the `PastData` and `FutureInputs` arguments of `forecast` before forecasting.

`OutputOffset` — Output signal offset
[ ] (default) | column vector | matrix

Output signal offset for time-domain data, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following values:

- `[]` — No output offsets.
- A column vector of length *Ny*, where *Ny* is the number of outputs. When you use the `forecast` command, the software subtracts the offset value `OutputOffset(i)` from the *i*th past output signal that you specify in the `PastData` argument of `forecast`. The software then uses the offset subtracted output to compute the detrended forecasts. The removed offsets are added back to the detrended forecasts to generate the final result.
- *Ny*-by-*Ne* matrix — For multi-experiment data, specify `OutputOffset` as an *Ny*-by-*Ne* matrix, where *Ne* is the number of experiments. Before forecasting, the software subtracts the offset value `OutputOffset(i,j)` from the *i*th output signal of the *j*th experiment in the `PastData` argument of `forecast`. For an example, see “Specify Output Offset for Forecasting Multi-Experiment Data” on page 1-404.

Output Arguments

`opt` — Option set for `forecast`
`forecastOptions` option set

Option set for `forecast`, retuned as a `forecastOptions` option set.

See Also

`forecast` | `idpar`

Topics
“Introduction to Forecasting of Dynamic System Response”

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:

$$FPE = \det\left(\frac{1}{N}\sum_{1}^{N} e\left(t,\widehat{\theta}_N\right)\left(e\left(t,\widehat{\theta}_N\right)\right)^T\right)\left(\frac{1 + d/N}{1 - d/N}\right)$$

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.

References

[1] Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hall PTR, 1999. See sections 7.4 and 16.4.

See Also

`aic` | `goodnessOfFit`

Topics
"Loss Function and Model Quality Metrics"

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` | `idfrd` | `freqresp` | `spectrum`

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 on page 1-417 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 page 1-417 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:

$$sys = \begin{bmatrix} 0 & \dfrac{1}{s+1} \\ \dfrac{s-1}{s+2} & 1 \end{bmatrix}$$

```
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)`.

Compute Frequency Response and Associated Covariance

Compute the frequency response and associated covariance for an identified process model at its peak response frequency.

Load estimation data `z1`.

```
load iddata1 z1
```

Estimate a SISO process model using the data.

```
model = procest(z1,'P2UZ');
```

Compute the frequency at which the model achieves the peak frequency response gain. To get a more accurate result, specify a tolerance value of `1e-6`.

```
[gpeak,fpeak] = getPeakGain(model,1e-6);
```

Compute the frequency response and associated covariance for `model` at its peak response frequency.

```
[H,wout,covH] = freqresp(model,fpeak);
```

`H` is the response value at `fpeak` frequency, and `wout` is the same as `fpeak`.

`covH` is a 5-dimensional array that contains the covariance matrix of the response from the input to the output at frequency `fpeak`. Here `covH(1,1,1,1,1)` is the variance of the real part of the response, and `covH(1,1,1,2,2)` is the variance of the imaginary part. The `covH(1,1,1,1,2)` and `covH(1,1,1,2,1)` elements are the covariance between the real and imaginary parts of the response.

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

$$D + C(j\omega - A)^{-1}B,\ \omega = \omega_1, \ldots, \omega_N$$

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.

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.

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

`interp` | `fcat` | `fdel` | `frd` | `idfrd`

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

Display Model Property Values

Create the following discrete-time SISO transfer function model:

$$H(z) = \frac{1}{z + 2}$$

Specify the sample time as 0.1 seconds and input channel name as `Voltage`.

```
h = tf(1,[1 2],0.1,'InputName','Voltage')
```

```
h =
 
  From input "Voltage" to output:
    1
  -----
  z + 2
 
Sample time: 0.1 seconds
Discrete-time transfer function.
```

Display all the properties of the transfer function.

```
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]
          Notes: [0x1 string]
       UserData: []
           Name: ''
   SamplingGrid: [1x1 struct]
```

Display the numerator of the transfer function.

```
num = get(h,'Numerator')
```

```
num = 1x1 cell array
    {[0 1]}
```

The numerator data is stored as a cell array, thus the `Numerator` property is a cell array containing the row vector `[0 1]`.

```
num{1}
```

```
ans = 1×2

     0     1
```

Display the sample time `Ts` of the transfer function.

```
get(h,'Ts')
```

```
ans = 0.1000
```

Alternatively, use dot notation to access the property value.

```
h.Ts
```

```
ans = 0.1000
```

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

`frdata` | `set` | `ssdata` | `tfdata` | `idssdata` | `polydata` | `getpvec` | `getcov`

Introduced before R2006a

getcov

Parameter covariance of identified 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 an identified 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

Obtain 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 = 5×5

    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.

Obtain 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
    {5x5 double}
    {7x7 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.

Obtain 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 = 4×4
10^5 ×
```

```
    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.

Obtain 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);
```

Obtain 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`.

Obtain 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 — Identified model

`idtf`, `idss`, `idgrey`, `idpoly`, `idproc`, `idnlarx`, `idnlhw`, or `idnlgrey` object | model array

Identified model, specified as an `idtf`, `idss`, `idgrey`, `idpoly`, `idproc`, `idnlarx`, `idnlhw`, or `idnlgrey` model or an array of such models.

The `getcov` command returns `cov_data` as `[]` for `idnlarx` and `idnlhw` models because these models do not store parameter covariance data.

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. `cov_data` is `[]` for `idnlarx` and `idnlhw` models.

- 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, 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,...)`.

See Also
`nparams` | `setcov` | `rsample` | `sim` | `simsd` | `getpvec`

Topics
“What Is Model Covariance?”
“Types of Model Uncertainty Information”

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 1-623

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],'idLinear');
```

Compute the maximum delays for each output variable individually.

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

```
Del = 2×5

     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 = 1×5

     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 plot options handle or plot options property

Syntax

```
p = getoptions(h)
p = getoptions(h,propertyName)
```

Description

You can use `getoptions` to obtain the plot handle options or properties list and use it to customize the plot, such as modify the axes labels, limits and units. For a list of the properties and values available for each plot type, see “Properties and Values Reference” (Control System Toolbox). To customize an existing plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox).

`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.

Examples

Impulse Plot with Specified Grid Color

For this example, consider a MIMO state-space model with 3 inputs, 3 outputs and 3 states. Create a impulse plot with red colored grid lines.

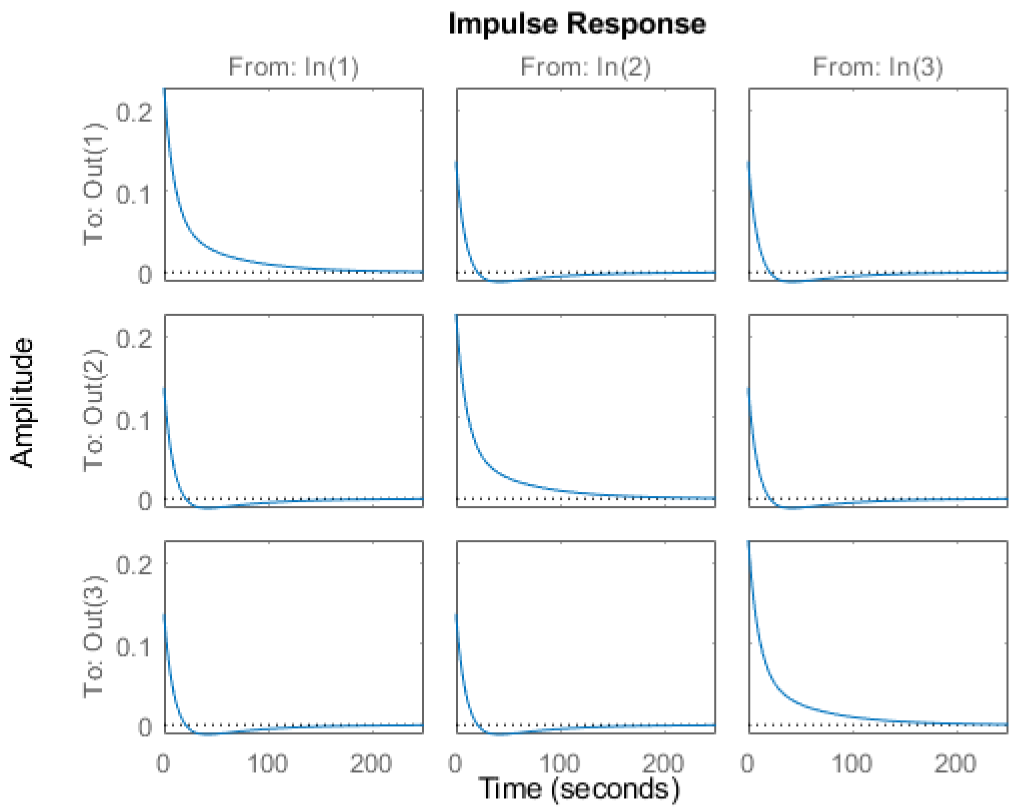
Create the MIMO state-space model `sys_mimo`.

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
size(sys_mimo)
```

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

Create an impulse plot with plot handle `h` and use `getoptions` for a list of the options available.

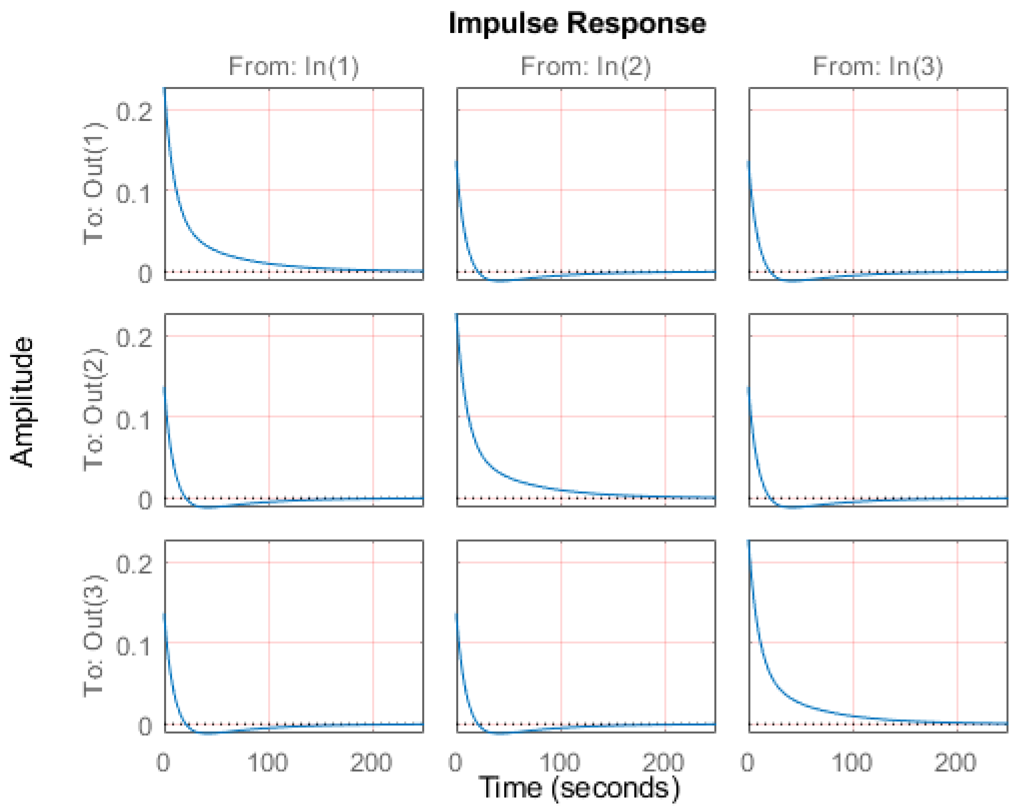
```
h = impulseplot(sys_mimo)
```



```
h =

    resppack.timeplot

p = getoptions(h)

p =

                 Normalize: 'off'
       SettleTimeThreshold: 0.0200
            RiseTimeLimits: [0.1000 0.9000]
                 TimeUnits: 'seconds'
    ConfidenceRegionNumberSD: 1
                IOGrouping: 'none'
               InputLabels: [1x1 struct]
              OutputLabels: [1x1 struct]
              InputVisible: {3x1 cell}
             OutputVisible: {3x1 cell}
                     Title: [1x1 struct]
                    XLabel: [1x1 struct]
                    YLabel: [1x1 struct]
                 TickLabel: [1x1 struct]
                      Grid: 'off'
                 GridColor: [0.1500 0.1500 0.1500]
                      XLim: {3x1 cell}
                      YLim: {3x1 cell}
                  XLimMode: {3x1 cell}
                  YLimMode: {3x1 cell}
```

Use `setoptions` to update the plot with the requires customization.

```
setoptions(h,'Grid','on','GridColor',[1 0 0]);
```



The impulse plot automatically updates when you call `setoptions`. For MIMO models, `impulseplot` produces a grid of plots, each plot displaying the impulse response of one I/O pair.

Bode Plot with Specified Frequency Scale and Units

For this example, consider a MIMO state-space model with 3 inputs, 3 outputs and 3 states. Create a Bode plot with linear frequency scale, specify frequency units in Hz and turn the grid on.

Create the MIMO state-space model `sys_mimo`.

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
size(sys_mimo)
```

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

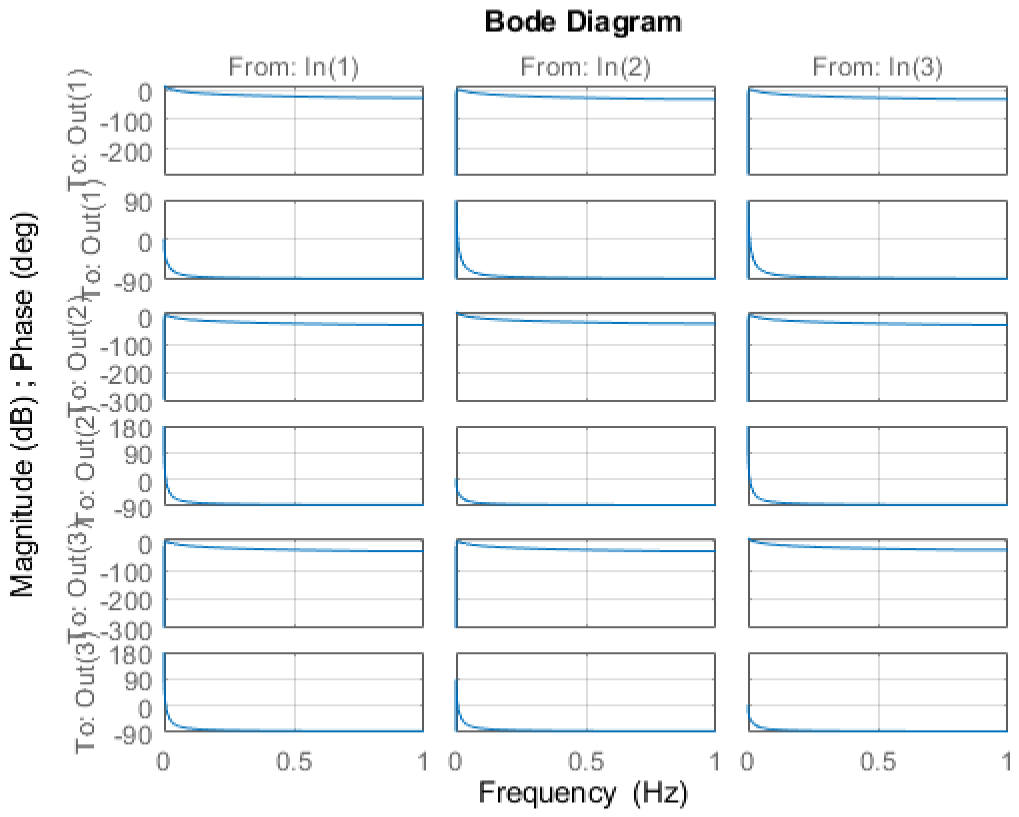
Create a Bode plot with plot handle `h` and use `getoptions` for a list of the options available.

```
h = bodeplot(sys_mimo);
p = getoptions(h)
```

```
p = 

                   FreqUnits: 'rad/s'
                   FreqScale: 'log'
                    MagUnits: 'dB'
                    MagScale: 'linear'
                  MagVisible: 'on'
             MagLowerLimMode: 'auto'
                  PhaseUnits: 'deg'
                PhaseVisible: 'on'
               PhaseWrapping: 'off'
               PhaseMatching: 'off'
           PhaseMatchingFreq: 0
    ConfidenceRegionNumberSD: 1
                 MagLowerLim: 0
          PhaseMatchingValue: 0
         PhaseWrappingBranch: -180
                  IOGrouping: 'none'
                 InputLabels: [1x1 struct]
                OutputLabels: [1x1 struct]
                InputVisible: {3x1 cell}
               OutputVisible: {3x1 cell}
                       Title: [1x1 struct]
                      XLabel: [1x1 struct]
                      YLabel: [1x1 struct]
                   TickLabel: [1x1 struct]
                        Grid: 'off'
                   GridColor: [0.1500 0.1500 0.1500]
                        XLim: {3x1 cell}
                        YLim: {6x1 cell}
                    XLimMode: {3x1 cell}
                    YLimMode: {6x1 cell}
```

Use `setoptions` to update the plot with the requires customization.

```
setoptions(h,'FreqScale','linear','FreqUnits','Hz','Grid','on');
```



The Bode plot automatically updates when you call `setoptions`. For MIMO models, `bodeplot` produces an array of Bode plots, each plot displaying the frequency response of one I/O pair.

Input Arguments

h — Plot handle
plot handle object

Plot handle, specified as a plot handle object. For example, `h` is a `mpzplot` object for a pole-zero or I/O pole-zero plot.

`propertyName` — Specific property name
string | character vector

Specific property name, specified as a string or character vector. For a list of the properties and values available for each plot type, see “Properties and Values Reference” (Control System Toolbox).

Output Arguments

p — Plot options handle
plot options handle object

Plot options handle, returned as a plot options handle object. For example, `p` is a `PZMapOptions` object for a pole-zero or I/O pole-zero plot.

See Also

`setoptions`

Topics

"Properties and Values Reference" (Control System Toolbox)
"Customizing Response Plots from the Command Line" (Control System Toolbox)

Introduced in R2012a

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 = 5×1

   -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 = 5×1

    1.0000
  100.0000
    0.3000
    0.4000
   10.0000
```

Get the free statuses of the parameters.

```
Free = getpar(m,'Free')
```

```
Free = 5x1 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 = 5×2

  -Inf   Inf
```

```
     0   Inf
     0   Inf
     0   Inf
  -Inf   Inf
```

Extract the values of the free parameters.

```
FreeValues = Val(Free)
```

```
FreeValues = 4×1

    1.0000
  100.0000
    0.3000
   10.0000
```

Extract the bounds on the free parameters.

```
FreeValBounds = MinMax(Free,:)
```

```
FreeValBounds = 4×2

  -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 = 5x1 cell
    {'a2'    }
    {0x0 char}
    {0x0 char}
    {0x0 char}
    {0x0 char}
```

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

`setpar` | `getpvec` | `getcov` | `tfdata` | `polydata` | `idssdata`

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

Obtain 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. For `idnlarx` and `idnlhw` models, all parameters are treated as free.

Input Arguments

`sys`

Identified model.

Identified model, specified as an `idtf`, `idss`, `idgrey`, `idpoly`, `idproc`, `idnlarx`, `idnlhw`, or `idnlgrey` model or an array of such models.

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` is `[]` for `idnlarx` and `idnlhw` models that have not been estimated.

`pvec_sd`

1 standard deviation value of the parameters of `sys`.

If the model covariance information for `sys` is not available, `pvec_sd` is `[]`. Thus, `pvec_sd` is always `[]` for `idnlarx` and `idnlhw` models because these models do not store parameter covariance information.

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

`setpvec` | `getcov` | `idssdata` | `tfdata` | `zpkdata`

Introduced in R2012a

getreg

Regressor expressions and numerical values in nonlinear ARX model

Syntax

```
Rs = getreg(model)
Rm = getreg(model,data)
Rm = getreg(model,data,init)
Rm = getreg( ___ ,'Type',regressorType)
```

Description

`Rs = getreg(model)` returns expressions for computing regressors in the nonlinear ARX model. `model` is an `idnlarx` object. A typical use of the regression matrices built by `getreg` is to generate input data when you want to evaluate the output of a mapping function such as `idWaveletNetwork` using `evaluate`. For example, the following pair of commands evaluates the output of a mapping function `model`.

```
Regressor_Value = getreg(model,data,'z')
y = evaluate(model.OutputFcn,RegressorValue)
```

These commands are equivalent to the command:

```
y = predict(model,data,1,predictOptions('InitialCondition','z'))
```

`Rm = getreg(model,data)` returns regressor values as a `timetable` for the specified input/output data set `data`.

`Rm = getreg(model,data,init)` uses the initial conditions that are specified in `init`. 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`).

`Rm = getreg( ___ ,'Type',regressorType)` returns the names of the regressors of the specified `regressorType`. For example, use the command `Rm = getreg(model,'Type','input')` to return the names of only the input regressors.

Input Arguments

`data`

`iddata` object containing measured data or numeric matrix that contains the values of the output and input variables in the order `[model.OutputName model.InputName]`.

`init`

Initial conditions of your data:

- `'z'` (default) specifies zero initial state.
- `NaN` denotes unknown initial conditions.
- Real column vector containing the initial state values. For more information on initial states, see Definition of idnlarx States in `idnlarx`. 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 number of samples required is equal to `max(getDelayInfo(model))`.

`model`

`iddata` object representing nonlinear ARX model.

`regressorType`

Type of regressor to return, specified as one of the following:

- `'all'` (default) — All regressors
- `'input'` — Only input regressors
- `'output'` — Only output regressors
- `'standard'` — Only linear and polynomial regressors
- `'custom'` — Only custom regressors

Output Arguments

`Rm`

`timetable` of regressor values for all or a specified subset of regressors. Each column in `Rm` contains as many rows as there are data samples. For a model with `nr` regressors, `Rm` contains one column for each regressor. When `data` contains multiple experiments, `Rm` is a cell array where each element corresponds to a timetable of regressor values for an experiment.

`Rs`

Regressor expressions represented as a 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.RegressorUsage`.

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 sec.

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)
```

```
Rs = 5x1 cell
    {'y1(t-1)'}
    {'y1(t-2)'}
    {'y1(t-3)'}
    {'u1(t-1)'}
    {'u1(t-2)'}
```

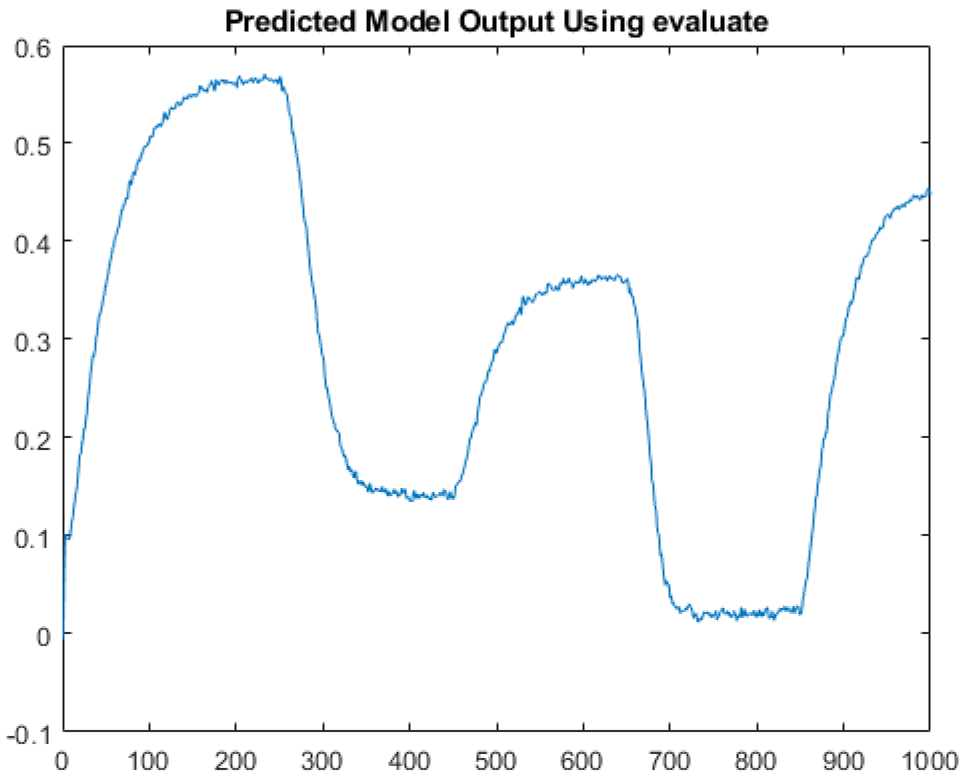
Get regressor values.

```
Rm = getreg(model,ze)
```

```
Rm=1000×5 timetable
     Time      y1(t-1)     y1(t-2)     y1(t-3)     u1(t-1)    u1(t-2)
    _______    ________    ________    ________    _______    _______

    0.2 sec           0           0           0       0          0
    0.4 sec      0.1003           0           0      10          0
    0.6 sec    0.094621      0.1003           0      10         10
    0.8 sec    0.084424    0.094621      0.1003      10         10
    1 sec      0.081449    0.084424    0.094621      10         10
    1.2 sec     0.08546    0.081449    0.084424      10         10
    1.4 sec    0.083002     0.08546    0.081449      10         10
    1.6 sec     0.08443    0.083002     0.08546      10         10
    1.8 sec    0.092793     0.08443    0.083002      10         10
    2 sec      0.099804    0.092793     0.08443      10         10
    2.2 sec     0.10559    0.099804    0.092793      10         10
    2.4 sec      0.1081     0.10559    0.099804      10         10
    2.6 sec     0.12108      0.1081     0.10559      10         10
    2.8 sec     0.12404     0.12108      0.1081      10         10
    3 sec       0.13551     0.12404     0.12108      10         10
    3.2 sec     0.13405     0.13551     0.12404      10         10
      ⋮
```

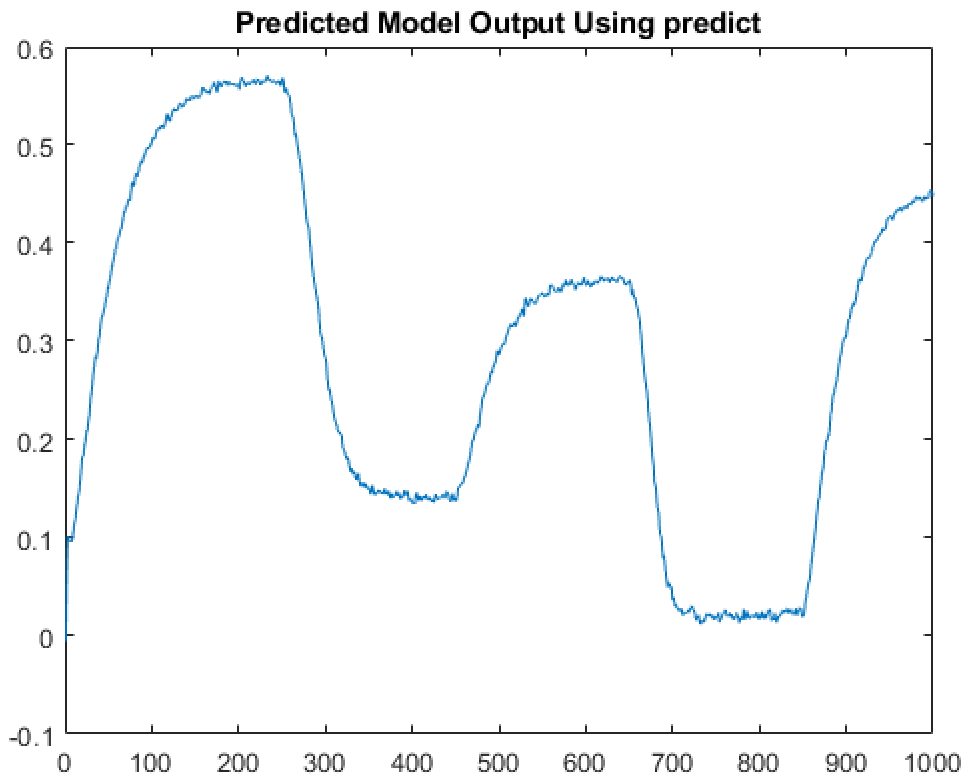
Evaluate and plot model output for one-step-prediction.

```
Y = evaluate(model.OutputFcn,Rm.Variables);
plot(1:1000,Y)
title('Predicted Model Output Using evaluate')
```



The previous result is equivalent to the result obtained by using `predict` in the following commands.

```
Y_p = predict(model,ze,1,'z');
Y = Y_p.OutputData;
plot(Y)
title('Predicted Model Output Using predict')
```



Compatibility Considerations

`getreg` returns regressor values in `timetable`
Behavior changed in R2021a

Starting in R2021a, `getreg` returns single-experiment data in a `timetable` of regressor values instead of a matrix or a cell array of values as in previous versions.

See Also

`idnlarx` | `linearRegressor` | `polynomialRegressor` | `customRegressor` | `evaluate`

Topics
“Identifying Nonlinear ARX Models”

Introduced in R2007a

getStateEstimate

Extract best state estimate and covariance from particles

Syntax

```
State = getStateEstimate(pf)
[State,StateCovariance] = getStateEstimate(pf)
```

Description

`State = getStateEstimate(pf)` returns the best state estimate based on the current set of particles. The estimate is extracted based on the `StateEstimationMethod` property from the `particleFilter` object, `pf`.

`[State,StateCovariance] = getStateEstimate(pf)` also returns the covariance of the state estimate. The covariance is a measure of the uncertainty of the state estimate. Not all state estimation methods support covariance output. In this case, `getStateEstimate` returns `StateCovariance` as [ ].

The `State` and `StateCovariance` information can directly be accessed as properties of the particle filter object pf, as `pf.State` and `pf.StateCovariance`. However, when both these quantities are needed, using the `getStateEstimation` method with two output arguments is more computationally efficient.

Examples

State Estimation using Particle Filter

Create a particle filter, and set the state transition and measurement likelihood functions.

```
myPF = particleFilter(@vdpParticleFilterStateFcn,@vdpMeasurementLikelihoodFcn);
```

Initialize the particle filter at state `[2; 0]` with unit covariance, and use `1000` particles.

```
initialize(myPF, 1000, [2;0], eye(2));
```

Pick the `mean` state estimation and `systematic` resampling methods.

```
myPF.StateEstimationMethod = 'mean';
myPF.ResamplingMethod = 'systematic';
myPF
```

```
myPF = 
  particleFilter with properties:

           NumStateVariables: 2
                NumParticles: 1000
          StateTransitionFcn: @vdpParticleFilterStateFcn
    MeasurementLikelihoodFcn: @vdpMeasurementLikelihoodFcn
     IsStateVariableCircular: [0 0]
```

```
             ResamplingPolicy: [1x1 particleResamplingPolicy]
             ResamplingMethod: 'systematic'
        StateEstimationMethod: 'mean'
             StateOrientation: 'column'
                    Particles: [2x1000 double]
                      Weights: [1.0000e-03 1.0000e-03 1.0000e-03 ... ]
                        State: 'Use the getStateEstimate function to see the value.'
              StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Assuming a measurement `2.1`, run one predict and correct step.

```
[PredictedState, PredictedStateCovariance] = predict(myPF);
[CorrectedState, CorrectedStateCovariance] = correct(myPF,2.1);
```

Get the best state estimate and covariance based on the `StateEstimationMethod` property.

```
[State, StateCovariance] = getStateEstimate(myPF)
```

```
State = 2×1

    2.1018
   -0.1413


StateCovariance = 2×2

    0.0175   -0.0096
   -0.0096    0.5394
```

Input Arguments

`pf` — Particle filter
`particleFilter` object

Particle filter, specified as an object. See `particleFilter` for more information.

Output Arguments

`State` — Best state estimate
[ ] (default) | vector

Best state estimate, defined as a vector based on the condition of the `StateOrientation` property:

- If `StateOrientation` is `'row'` then `State` is a 1-by-`NumStateVariables` vector
- If `StateOrientation` is `'column'` then `State` is a `NumStateVariables`-by-1 vector

`StateCovariance` — Current estimate of state estimation error covariance
`NumStateVariables`-by-`NumStateVariables` array (default) | [ ] | array

Current estimate of state estimation error covariance, defined as an `NumStateVariables`-by-`NumStateVariables` array. `StateCovariance` is calculated based on the `StateEstimationMethod`. If you specify a state estimation method that does not support covariance, then the function returns `StateCovariance` as [ ].

See Also
`correct` | `particleFilter` | `initialize` | `predict`

Topics
"Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter"
"What Is Online Estimation?"

Introduced in R2017b

getTrend

Create trend information object to store offset, mean, and trend information for time-domain signals stored in `iddata` object

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`. You can then apply the trend information in `T` to either `data` or to other `iddata` objects by using `detrend` or `retrend`.

`T = getTrend(data,0)` computes the means of input and output signals and stores them as the `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`. The following code represents the straight line:

```
ULine = Tr.InputOffset + (time-t0)*Tr.InputSlope
YLine = Tr.OutputOffset + (time-t0)*Tr.OutputSlope
```

Here, `time` is equal to `Z.SamplingInstants` and `t0` is equal to `data.Tstart`.

Examples

Remove Offsets from Data

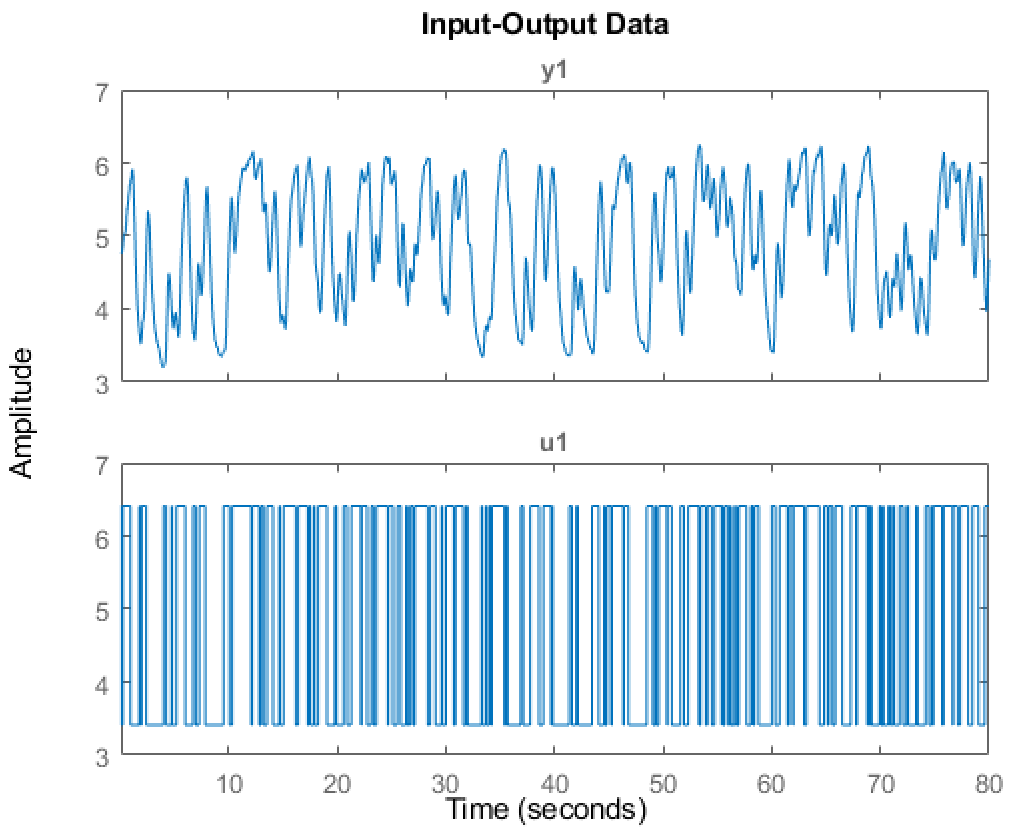
Remove a specified offset from input and output signals.

Load SISO data containing vectors `u2` and `y2`.

```
load dryer2
```

Create a data object with a 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.

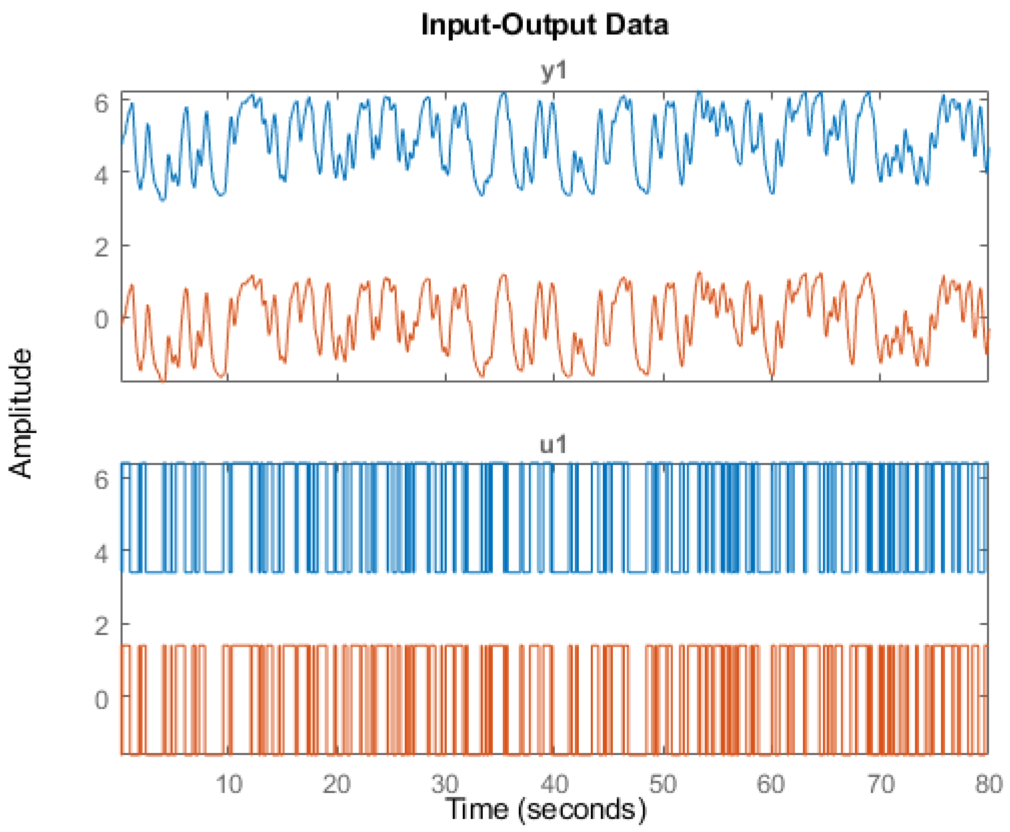
```
T.InputOffset = 5;
T.OutputOffset = 5;
```

Subtract the offsets 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 a sample time of 0.08 seconds.

```
data = iddata(y2,u2,0.08);
```

Compute the mean of the data.

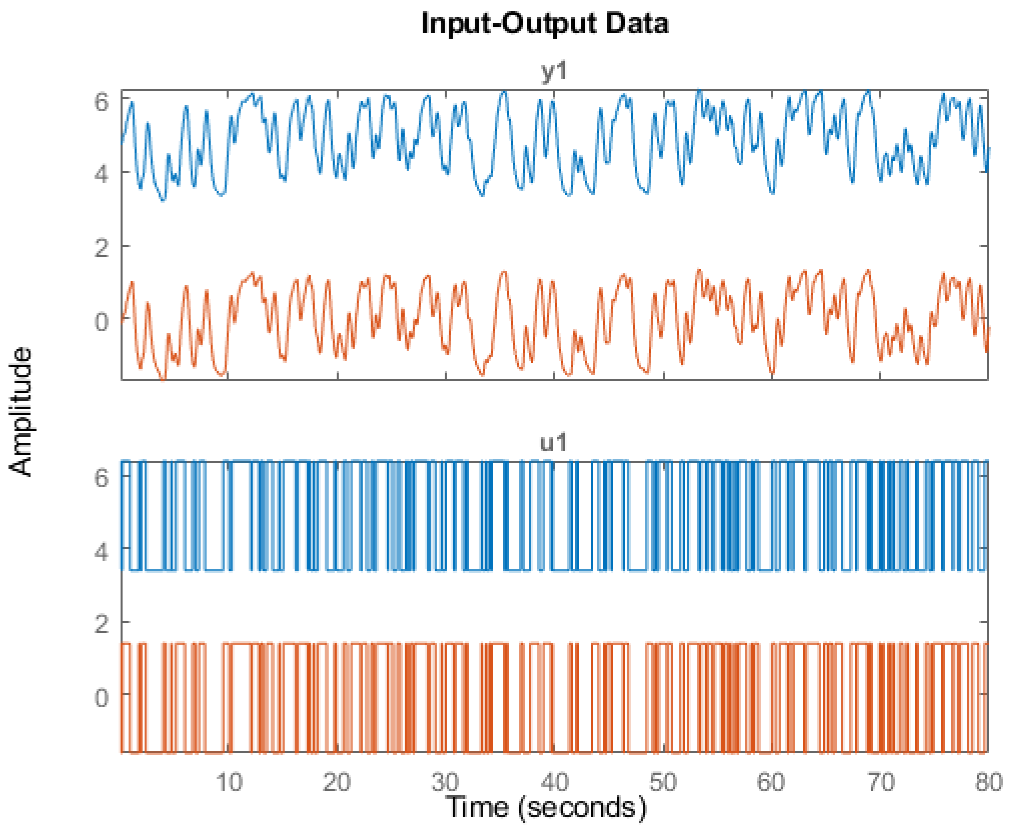
```
T = getTrend(data,0);
```

Remove the mean from the data.

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

Plot the original and detrended data on the same plot.

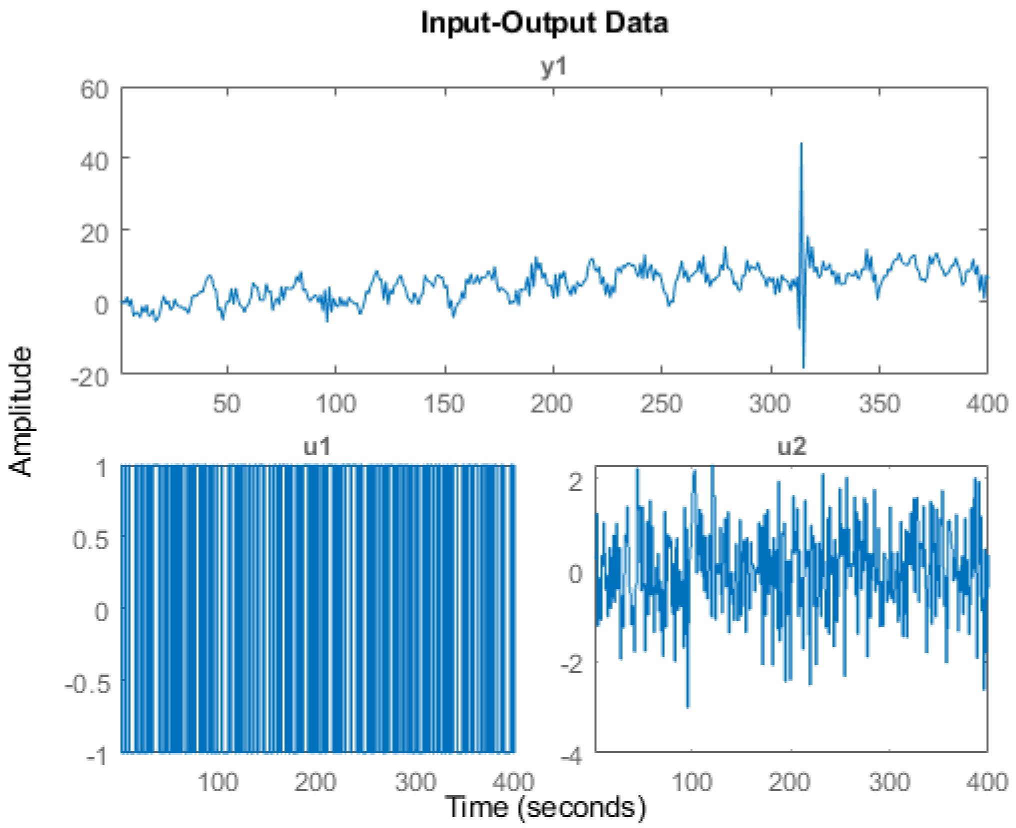
```
plot(data,data_d)
```



Combine Computed and Customized Trend Information for MISO Data

Load and plot data that contains two input channels and one output channel.

```
load z7lintrend z7L
plot(z7L)
```



The output channel of `z7L` contains a linear trend that is not present in the input channels. Compute the trend information.

```
T = getTrend(z7L,1)
```

```
Trend specifications for data "z7L" with 2 input(s), 1 output(s), 1 experiment(s):
        DataName: 'z7L'
     InputOffset: [-0.0764 -0.0683]
    OutputOffset: -0.2642
      InputSlope: [4.8338e-04 3.1642e-04]
     OutputSlope: 0.0268
```

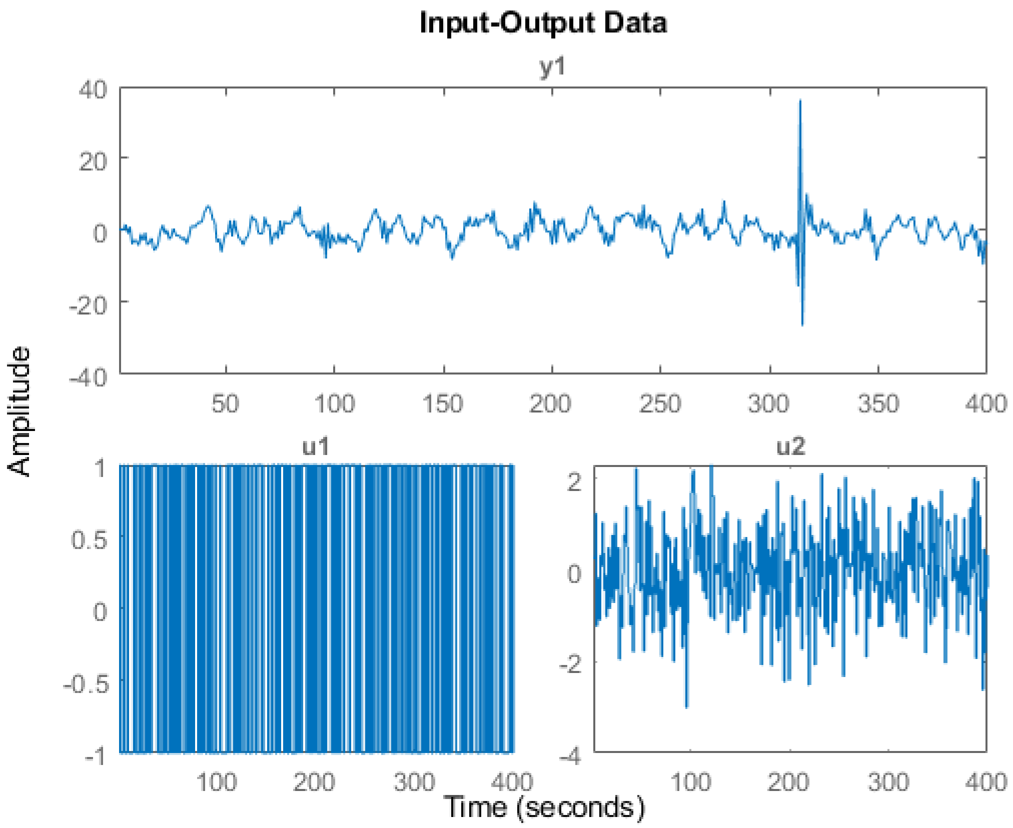
Limit the trend information to the output channel only by setting the input trend values to 0.

```
T.InputOffset = [0 0];
T.InputSlope = [0 0];
T
```

```
Trend specifications for data "z7L" with 2 input(s), 1 output(s), 1 experiment(s):
        DataName: 'z7L'
     InputOffset: [0 0]
    OutputOffset: -0.2642
      InputSlope: [0 0]
     OutputSlope: 0.0268
```

Remove the linear trend from the data.

```
z7d = detrend(z7L,T);
plot(z7d)
```



The trend is no longer in the output data and the input data is unchanged.

Input Arguments

data — Time-domain input-output data
`iddata` object

Time-domain input-output data, specified as an `iddata` object containing one or more sets of time-domain signals. The `iddata` object can contain SISO, MIMO, or multiexperiment data. The signal sets can contain either input and output data or output data only.

Output Arguments

T — Trend information
`trendInfo` object

Trend information, returned as a `TrendInfo` object.

See Also

`detrend` | `retrend` | `TrendInfo`

Topics
"Handling Offsets and Trends in Data"

Introduced in R2009a

goodnessOfFit

Goodness of fit between test and reference data for analysis and validation of identified models

Syntax

```
fit = goodnessOfFit(x,xref,cost_func)
```

Description

`goodnessOfFit` returns fit values that represent the error norm between test and reference data sets. If you want to compare and visualize simulated model output with measurement data, see also `compare`.

`fit = goodnessOfFit(x,xref,cost_func)` returns the goodness of fit between the test data `x` and the reference data `xref` using the cost function `cost_func`. `fit` is a quantitative representation of the closeness of `x` to `xref`. To perform multiple test-to-reference fit comparisons, you can specify `x` and `xref` as cell arrays of equal size that contain multiple test and reference data sets. With cell array inputs, `fit` returns an array of fit values.

Examples

Calculate Goodness of Fit Between Estimated and Measured Data

Find the goodness of fit between measured output data and the simulated output of an estimated model.

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.

Estimate a second-order transfer function model and simulate the model output `y_est`.

```
sys = tfest(z1,2);
y_est = sim(sys,z1(:,[],:));
```

Calculate the goodness of fit, or error norm, between the measured and estimated outputs. Specify the normalized root mean squared error (NRMSE) as the cost function.

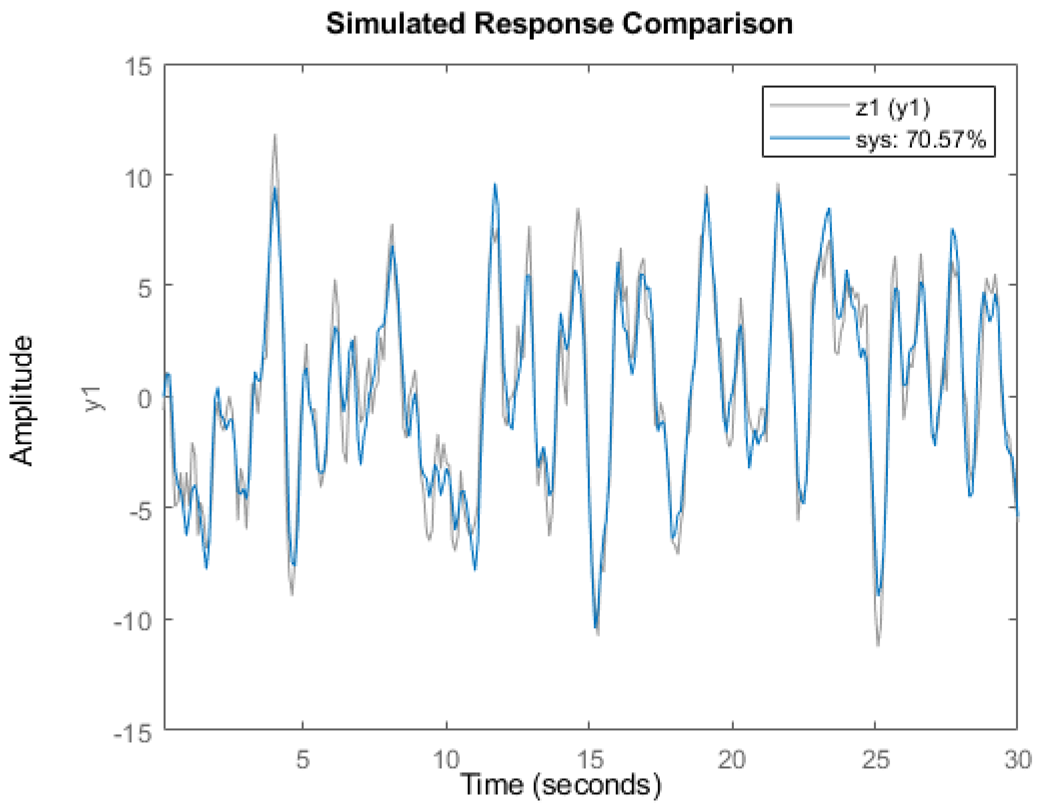
```
cost_func = 'NRMSE';
y = y_est.y;
fit = goodnessOfFit(y,yref,cost_func)
```

```
fit = 0.2943
```

Alternatively, you can use `compare` to calculate the fit. `compare` uses the NRMSE cost function, and expresses the fit percentage using the one's complement of the error norm. The fit relationship between `compare` and `goodnessOfFit` is therefore $\text{fit}_{\text{compare}} = \left(1 - \text{fit}_{\text{gof}}\right) * 100$. A `compare` result of 100% is equivalent to a `goodnessOfFit` result of 0.

Specify an initial condition of zero to match the initial condition that `goodnessOfFit` assumes.

```
opt = compareOptions('InitialCondition','z');
compare(z1,sys,opt);
```



The fit results are equivalent.

Goodness of Fit for Multiple Data Sets

Find the goodness of fit between measured and estimated outputs for two models.

Obtain the input-output measurements `z2` from `iddata2`. Copy the measured output into reference output `yref`.

```
load iddata2 z2
yref = z2.y;
```

Estimate second-order and fourth-order transfer function models using `z2`.

```
sys2 = tfest(z2,2);
sys4 = tfest(z2,4);
```

Simulate both systems to get estimated outputs.

```
y_sim2 = sim(sys2,z2(:,[],:));
y2 = y_sim2.y;
y_sim4 = sim(sys4,z2(:,[],:));
y4 = y_sim4.y;
```

Create cell arrays from the reference and estimated outputs. The reference data set is the same for both model comparisons, so create identical reference cells.

```
yrefc = {yref yref};
yc = {y2 y4};
```

Compute `fit` values for the three cost functions.

```
fit_nrmse = goodnessOfFit(yc,yrefc,'NRMSE')
```

```
fit_nrmse = 1×2

    0.1429    0.1439
```

```
fit_nmse = goodnessOfFit(yc,yrefc,'NMSE')
```

```
fit_nmse = 1×2

    0.0204    0.0207
```

```
fit_mse = goodnessOfFit(yc,yrefc,'MSE')
```

```
fit_mse = 1×2

    1.0811    1.0967
```

A fit value of 0 indicates a perfect fit between reference and estimated outputs. The fit value rises as fit goodness decreases. For all three cost functions, the fourth-order model produces a better fit than the second-order model.

Input Arguments

x — Data to test
matrix (default) | cell array

Data to test, specified as a matrix or cell array.

- For a single test data set, specify an N_s-by-N matrix, where N_s is the number of samples and N is the number of channels. You must specify `cost_fun` as `'NRMSE'` or `'NMSE'` to use multiple-channel data.
- For multiple test data sets, specify a cell array of length N_d, where N_d is the number of test-to-reference pairs and each cell contains one data matrix.

`x` must not contain any `NaN` or `Inf` values.

`xref` — Reference data
matrix (default) | cell array

Reference data with which to compare `x`, specified as a matrix or cell array.

- For a single reference data set, specify an N_s-by-N matrix, where N_s is the number of samples and N is the number of channels. `xref` must be the same size as `x`. You must specify `cost_fun` as `'NRMSE'` or `'NMSE'` to use multiple-channel data.
- For multiple reference data sets, specify a cell array of length N_d, where N_d is the number of test-to-reference pairs and each cell contains one reference data matrix. As with the individual data matrices, the cell array sizes for `x` and `xref` must match. Each *i*th element of `fit` corresponds to the pairs of the *i*th cells of `x` and `xref`.

`xref` must not contain any `NaN` or `Inf` values.

`cost_func` — Cost function
`'MSE'` | `'NRMSE'` | `'NMSE'`

Cost function to determine goodness of fit, specified as one of the following values. In the equations, the value *fit* applies to a single pairing of test and reference data sets.

Value	Description	Equation	Notes
`'MSE'`	Mean squared error	$fit = \frac{\lVert x - xref \rVert^2}{Ns}$ where N_s is the number of samples and ‖ indicates the 2-norm of a vector.	*fit* is a scalar.
`'NRMSE'`	Normalized root mean squared error	$fit(i) = \frac{\lVert xref(:,i) - x(:,i) \rVert}{\lVert xref(:,i) - mean(xref(:,i)) \rVert}$ where ‖ indicates the 2-norm of a vector. `fit` is a row vector of length N and $i = 1,\ldots,N$, where N is the number of channels.	*fit* is a row vector. `'NRMSE'` is the cost function used by `compare`.
`'NMSE'`	Normalized mean squared error	$fit(i) = \frac{\lVert xref(:,i) - x(:,i) \rVert^2}{\lVert xref(:,i) - mean(xref(:,i)) \rVert^2}$	*fit* is a row vector.

Output Arguments

`fit` — Goodness of fit
scalar | row vector | cell array

Goodness of fit between test and reference data pairs, returned as a scalar, a row vector, or a cell array.

- For a single test and reference data set pair, `fit` is returned as a scalar or row vector.
  - If `cost_fun` is `'MSE'`, then `fit` is a scalar.
  - If `cost_fun` is `'NRMSE'` or `'NMSE'`, then `fit` is a column vector of length N, where N is the number of channels.
- For multiple test and data set and reference pairs, where `x` and `xref` are cell arrays of length N_D, `fit` is returned as a vector or a matrix.
  - If `cost_fun` is `'MSE'`, then `fit` is a row vector of length N_D.
  - If `cost_fun` is `'NRMSE'` or `'NMSE'`, then `fit` is a matrix of size N-by- N_d, where N is the number of channels (data columns) and N_d represents the number of test pairs. Each element of `fit` contains the goodness of fit values for the corresponding test data and reference pair.

  Each element of `fit` contains the goodness of fit values for the corresponding test data and reference pair.

Possible values for individual fit elements depend on the selection of `cost_func`.

- If `cost_func` is `'MSE'`, each `fit` value is a positive scalar that grows with the error between test and reference data. A `fit` value of `0` indicates a perfect match between test and reference data.
- If `cost_func` is `'NRMSE'` or `'NMSE'`, `fit` values vary between `-Inf` and 1.
  - `0` — Perfect fit to reference data (zero error)
  - `-Inf` — Bad fit
  - `1` — `x` is no better than a straight line at matching `xref`

Compatibility Considerations

`goodnessOfFit`: Fit result represents the error norm for all three cost functions, with a value of zero indicating a perfect fit
Behavior changed in R2020a

`goodnessOfFit` now returns the error norm E as the fit value for all three cost functions (MSE, NRMSE, and NMSE). Previously, `goodnessOfFit` returned the one's complement of the error norm, 1-E, for fit values that used the NRMSE or NMSE cost functions. This change allows consistent fit-value interpretation across the three cost functions, with the ideal fit value of zero representing a perfect fit.

Previously computed NRMSE and NMSE fit values are the one's complements of the fit values computed with the current software. Similarly, the NRMSE fit value is now the one's complement of the fit used in the percentage value that `compare` computes. For example, if the previous `goodnessOfFit` fit value was 0.8, the current fit value is 0.2. A `goodnessOfFit` fit value of 0.2 is equivalent to a `compare` fit percentage of 80%.

See Also

`compare` | `fpe` | `aic` | `resid`

Introduced in R2012a

greyest

Linear grey-box model estimation

Syntax

```
sys = greyest(data,init_sys)
sys = greyest(data,init_sys,opt)
[sys,x0] = greyest( ___ )
```

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.

`[sys,x0] = greyest( ___ )` returns the value of the initial states computed during estimation. You can use this syntax with any of the previous input-argument combinations.

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.
InitialState	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. • 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'.
DisturbanceModel	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'.

Report Field	Description
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:

Field	Description
FitPercent	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
LossFcn	Value of the loss function when the estimation completes.
MSE	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
FPE	Final prediction error for the model.
AIC	Raw Akaike Information Criteria (AIC) measure of model quality.
AICc	Small-sample-size corrected AIC.
nAIC	Normalized AIC.
BIC	Bayesian Information Criteria (BIC).

Report Field	Description
Parameters	Estimated values of model parameters.
OptionsUsed	Option set used for estimation. If no custom options were configured, this is a set of default options. See greyestOptions for more information.
RandState	State of the random number stream at the start of estimation. Empty, [], if randomization was not used during estimation. For more information, see rng.

Report Field	Description
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields.

Field	Description
`Name`	Name of the data set.
`Type`	Data type.
`Length`	Number of data samples.
`Ts`	Sample time.
`InterSample`	Input intersample behavior, returned as one of the following values: • `'zoh'` — 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.
`InputOffset`	Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`.
`OutputOffset`	Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.

Report Field	Description
`Termination`	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields:

Field	Description
`WhyStop`	Reason for terminating the numerical search.
`Iterations`	Number of search iterations performed by the estimation algorithm.
`FirstOrderOptimality`	∞-norm of the gradient search vector when the search algorithm terminates.
`FcnCount`	Number of times the objective function was called.
`UpdateNorm`	Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`.
`LastImprovement`	Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`.
`Algorithm`	Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used.

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, returned as a matrix containing a column vector corresponding to each experiment.

This array is also stored in the `Parameters` field of the model `Report` property.

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.

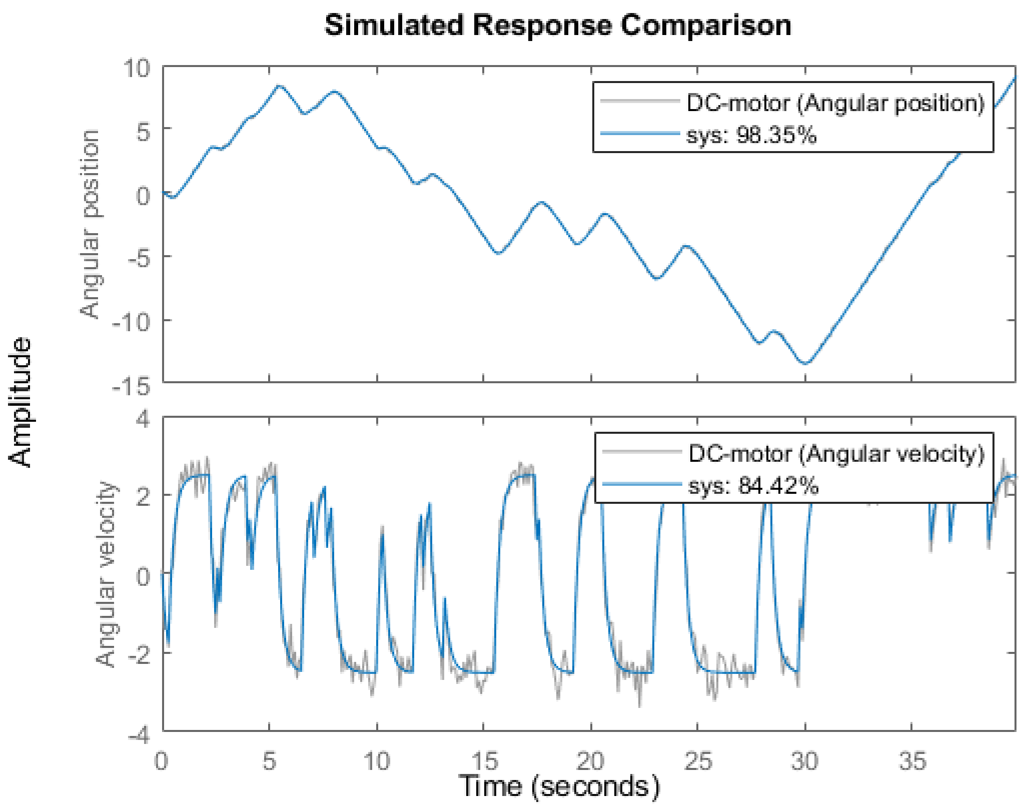
```
sys = greyest(data,init_sys);
```

`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 $\theta*$.

```
opt.Regularization.Nominal = 'model';
```

Estimate the regularized grey-box model.

```
sys = greyest(motorData, mi, opt);
```

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `greyestOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = greyestOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

`idgrey` | `greyestOptions` | `iddata` | `idfrd` | `ssest` | `idnlgrey` | `pem`

Topics

“Estimate Model Using Zero/Pole/Gain Parameters”
“Regularized Estimates of Model Parameters”

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 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`

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

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

If `EstimateCovariance` 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'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOptions` Structure When `SearchMethod` is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Tolerance`	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`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in "Optimization Options" (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	`1e-6`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`.	`100`

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

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

  - 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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`

Topics

“Loss Function and Model Quality Metrics”

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`

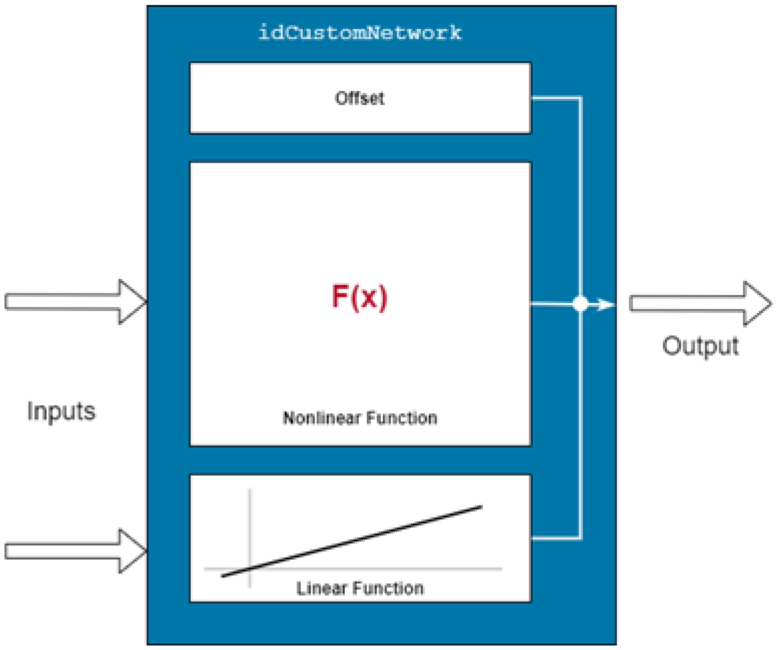
Introduced in R2012a

idCustomNetwork

Custom network function for nonlinear ARX and Hammerstein-Wiener models

Description

An `idCustomNetwork` object implements a custom network function, and is a nonlinear mapping function for estimating nonlinear ARX and Nonlinear Hammerstein-Wiener models. The mapping function, which is also referred to as a nonlinearity, uses a combination of linear weights, an offset and a nonlinear function to compute its output. The nonlinear function contains custom unit functions that operate on a ridge combination (weighted linear sum) of inputs.



Mathematically, `idCustomNetwork` is a function that maps m inputs $X(t) = [x(t_1),x_2(t),\ldots,x_m(t)]^T$ to a scalar output $y(t)$ using the following relationship:

$$y(t) = y_0 + (X(t) - \bar{X})^T PL + S(X(t))$$

Here:

- $X(t)$ is an m-by-1 vector of inputs, or regressors, with mean $\bar{X}$.
- y_0 is the output offset, a scalar.
- P is an m-by-p projection matrix, where m is the number of regressors and is p is the number of linear weights. m must be greater than or equal to p.
- L is a p-by-1 vector of weights.
- $S(X)$ is a sum of dilated and translated custom unit functions. The total number of unit functions is referred to as the number of units n of the network.

For the definition of the unit function term $S(X)$, see "More About" on page 1-494.

Use `idCustomNetwork` as the value of the `OutputFcn` property of an `idnlarx` model or the `InputNonlinearity` and `OutputLinearity` properties of an `idnlhw` object. For example, specify `idCustomNetwork` when you estimate an `idnlarx` model with the following command.

```
sys = nlarx(data,regressors,idCustomNetwork)
```

When `nlarx` estimates the model, it essentially estimates the parameters of the `idCustomNetwork` function.

You can configure the `idCustomNetwork` function to disable components and fix parameters. To omit the linear component, set `LinearFcn.Use` to `false`. To omit the offset, set `Offset.Use` to `false`. To specify known values for the linear function and the offset, set their `Value` attributes directly and set the corresponding `Free` attributes to `False`. Use `evaluate` to compute the output of the function for a given vector of inputs.

Creation

Syntax

```
C = idCustomNetwork(H)
S = idCustomNetwork(H,numUnits)
S = idCustomNetwork(H,numUnits,UseLinearFcn)
S = idCustomNetwork(H,numUnits,UseLinearFcn,UseOffset)
```

Description

`C = idCustomNetwork(H)` creates a nonlinear mapping 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 `[-a a]` where `g` is significantly nonzero. Hammerstein-Wiener models require that your custom function have only one input and one output.

`S = idCustomNetwork(H,numUnits)` specifies the number of unit functions `numUnits`.

`S = idCustomNetwork(H,numUnits,UseLinearFcn)` specifies whether the function uses a linear function as a subcomponent.

`S = idCustomNetwork(H,numUnits,UseLinearFcn,UseOffset)` specifies whether the function uses an offset term y_0 parameter.

Input Arguments

`H` — Function handle
function handle

Function handle that points to a custom function of the form `[f,g,a] = function_name(x)`, specified as a function handle. The function that `H` points to must be vectorized. That is, for a vector or matrix *x*, the output arguments *f* and *g* must have the same size as *x* when computed element by element.

This argument sets the `C.NonlinearFcn.UnitFcn` property.

`numUnits` — Number of units
`10` (default) | positive integer

Number of units, specified as a positive integer. `numUnits` determines the number of custom unit functions.

This argument sets the `C.NonlinearFcn.NumberOfUnits` property.

`UseLinearFcn` — Option to use linear function
`true` (default) | `false`

Option to use the linear function subcomponent, specified as `true` or `false`. This argument sets the value of the `C.LinearFcn.Use` property.

`UseOffset` — Option to use offset term
`true` (default) | `false`

Option to use an offset term, specified as `true` or `false`. This argument sets the value of the `C.Offset.Use` property.

Properties

`Input` — Input signal information
signal property values

Input signal information for signals used for estimation, specified as vectors of *m* property-specific values, where *m* is the number of input signals. The `Input` properties for each input signal are as follows:

- `Name` — Names of the input signals, specified as a 1-by-*m* string or character array, where *m* is the number of inputs
- `Mean` — Mean of the input signals, specified as a numeric scalar
- `Range` — Ranges of the input signals, specified as a 2-by-*m* numeric array that contains the minimum and maximum values

`Output` — Output signal information
signal property values

Output signal information, specified as property-specific values. The `Output` properties are as follows:

- `Name` — Name of the output signal, specified as a string or a character array
- `Mean` — Mean of the output signal, specified as a numeric scalar
- `Range` — Range of the output signal, specified as 2-by-1 numeric array that contains the minimum and maximum values.

`LinearFcn` — Parameters of linear function
linear function property values (default)

Parameters of the linear function, specified as follows:

- `Use` — Option to use the linear function in the custom network, specified as a scalar logical. The default value is `true`.

- `Value` — Linear weights that compose *L*', specified as a 1-by-*p* vector.
- `InputProjection` — Input projection matrix *P*, specified as an *m*-by-*p* matrix, that transforms the detrended input vector of length *m* into a vector of length *p*. For Hammerstein-Wiener models, `InputProjection` is equal to `1`.
- `Free` — Option to update entries of `Value` during estimation, specified as a 1-by-*p* logical vector. The software honors the `Free` specification only if the starting value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a 1-by-*p* vector. If `Minimum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that minimum bound during model estimation.
- `Maximum` — Maximum bound on `Value`, specified as a 1-by-*p* vector. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation.
- `SelectedInputIndex` — Indices of `idCustomNetwork` inputs (see `Input.Name`) that are used as inputs to the linear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. For nonlinear ARX models, the `RegressorUsage` property determines these indices. For Hammerstein-Wiener models, `SelectedInputIndex` is always `1`.

The software computes the output of `LinearFcn` as $F_L(t) = (X(t) - \bar{X})^T PL)$.

`Offset` — Parameters of offset term

offset property values

Parameters of the offset term, specified as follows:

- `Use` — Option to use the offset in the custom network, specified as a scalar logical. The default value is `true`.
- `Value` — Offset value, specified as a scalar.
- `Free` — Option to update `Value` during estimation, specified as a scalar logical. The software honors the `Free` specification of `false` only if the value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a numeric scalar or –`Inf`. If `Minimum` is specified with a finite value and the value of `Value` is finite, then the software enforces that minimum bound during model estimation. The default value is `-Inf`.
- `Maximum` — Maximum bound on `Value`, specified as a numeric scalar or `Inf`. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation. The default value is `Inf`.

`NonlinearFcn` — Parameters of nonlinear function

nonlinear function property values

Parameters of the nonlinear function, specified as follows:

- `UnitFcn` — Function handle that points to a custom function of the form `[f,g,a] = function_name(x)`. The function that `UnitFcn` points to must be vectorized. That is, for a vector or matrix *x*, the output arguments *f* and *g* must have the same size as *x* when computed element by element.
- `NumberOfUnits` — Number of units, specified as a positive integer. `NumberOfUnits` determines the number *n* of custom functions.

- `Parameters` — Parameters of `idCustomNetwork`, specified as in the following table:

Field Name	Description	Default
`InputProjection`	Projection matrix Q, specified as an m-by-q matrix. Q transforms the detrended input vector $(X - \bar{X})$ of length m into a vector of length q. Typically, Q has the same dimensions as the linear projection matrix P. In this case, q is equal to p, which is the number of linear weights. For Hammerstein-Wiener models, `InputProjection` is equal to `1`.	`[]`
`OutputCoefficient`	custom function output coefficients s_i, specified as an n-by-1 vector.	`[]`
`Translation`	Translation matrix, specified as an n-by-q matrix of translation row vectors c_i.	`[]`
`Dilation`	Dilation coefficients b_i, specified as an n-by-1 vector.	`[]`

- `Free` — Option to estimate parameters, specified as a logical scalar. If all the parameters have finite values, such as when the `idCustomNetwork` object corresponds to a previously estimated model, then setting `Free` to `false` causes the parameters of the nonlinear function $S(X)$ to remain unchanged during estimation. The default value is `true`.
- `SelectedInputIndex` — Indices of `idCustomNetwork` inputs (see `Input.Name`) that are used as inputs to the nonlinear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. For nonlinear ARX models, the `RegressorUsage` property determines these indices. For Hammerstein-Wiener models, `SelectedInputIndex` is always `1`.

Examples

Estimate Nonlinear ARX Model with `customnet` as Output Function

Load the data

```
load iddata1 z1
```

Create a `customnet` object that uses `gaussunit` as the unit function.

```
H = @gaussunit;
C = idCustomNetwork(@gaussunit);
```

Set properties of `C` using dot notation. Fix the value of the offset to 0.2 and set the number of unit functions to 15.

```
C.Offset.Value = 0.2;
C.Offset.Free = false;
C.NonlinearFcn.NumberOfUnits = 15
```

```
C = 
Custom Network

 Nonlinear Function: Custom Network with 15 units and "gaussunit" unit function
 Linear Function: uninitialized
 Output Offset: fixed to 0.2

           Input: 'Function inputs'
          Output: 'Function output'
       LinearFcn: 'Linear function parameters'
    NonlinearFcn: 'Custom units and their parameters'
          Offset: 'Offset parameters'
```

Create model regressors.

```
Reg = linearRegressor([z1.OutputName z1.InputName],{1:4 0:4});
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z1,Reg,C)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Custom Network with 15 units
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 76.15% (prediction focus)
FPE: 4.311, MSE: 1.104
```

Estimate Hammerstein-Wiener Model with `customnet` as Output Nonlinearity

Load the data.

```
load throttledata
```

Create a `customnet` object `C` that uses `gaussunit` as the unit function. Include input arguments that specify 10 units and exclude the linear function and the offset.

```
H = @gaussunit;
C = idCustomNetwork(@gaussunit,10,false,false)
```

```
C =
Custom Network

 Nonlinear Function: Custom Network with 10 units and "gaussunit" unit function
 Linear Function: not in use
 Output Offset: not in use

           Input: 'Function inputs'
          Output: 'Function output'
       LinearFcn: 'Linear function parameters'
```

```
    NonlinearFcn: 'Custom units and their parameters'
          Offset: 'Offset parameters'
```

Estimate the Hammerstein-Wiener Model using orders `[4 4 1]`, no input nonlinearity, and `C` as the output nonlinearity.

```
sys = nlhw(ThrottleData,[4 4 1],[],C)
```

```
sys =
Hammerstein-Wiener model with 1 output and 1 input
 Linear transfer function corresponding to the orders nb = 4, nf = 4, nk = 1
 Input nonlinearity: absent
 Output nonlinearity: Custom Network with 10 units and "gaussunit" unit function
Sample time: 0.01 seconds

Status:
Estimated using NLHW on time domain data "ThrottleData".
Fit to estimation data: 76.57%
FPE: 77.59, MSE: 60.55
```

More About

Custom Nonlinear Function *S*(*X*)

The custom nonlinear function is a sum of the dilated and translated unit functions, and is described by the following equation:

$$S(X) = \sum_{i=1}^{n} s_i f((X - \bar{X})^T Q b_i + c_i)$$

Here:

- Q is an m-by-q projection matrix, where $m \geq q$.
- $s_1, s_2, \ldots, s_n$ are scalar weights called output coefficients.
- $b_1, b_2, \ldots, b_n$ are q-by-1 vectors called dilation coefficients .
- $c_1, c_2, \ldots, c_n$ are scalars called translations.
- f(z) is a function, also called a unit function, that takes a scalar input and returns a scalar output. Here, z is a scalar of the form $b_i(X - \bar{X})^T Q + c_i$.

Algorithms

`idCustomNetwork` uses an iterative search technique for estimating parameters.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
wavenet	idWaveletNetwork
sigmoidnet	idSigmoidNetwork
treepartition	idTreePartition
customnet	idCustomNetwork
saturation	idSaturation
deadzone	idDeadZone
pwlinear	idPiecewiseLinear
poly1d	idPolynomial1D
unitgain	idUnitGain
linear	idLinear
neuralnet	idFeedforwardNetwork

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

Use of previous nonlinearity estimator properties is not recommended
Not recommended starting in R2021a

Starting in R2021a, the properties of the mapping objects, previously known as nonlinearity estimators, have been reorganized. These objects are `wavenet` (W), `sigmoidnet` (S), `treepartition` (T), `customnet` (C), and `linear` (L). The property changes do not apply to `neuralnet`. The use of the pre-R2021a properties in the following table is discouraged. However, the software still accepts commands that set these properties. There are no plans to exclude such commands at this time.

Pre-R2021a Property	R2021a Property	Applicable Mapping Objects
NumberOfUnits	NonlinearFcn.NumberOfUnits	W,S,T,C
LinearTerm	LinearFcn.Use, Offset.Use	W,S,C
Parameters	Split into three pieces: • LinearFcn.Value • Offset.Value • NonlinearFcn.Parameters	W,S,T,C,L linear (L) excludes NonlinearFcn.Parameters.
Options	NonlinearFcn.Structure	W,T

See Also
nlhw | nlarx | idLinear | idPolynomial1D | idTreePartition | idSigmoidNetwork | idSaturation | idPiecewiseLinear | idUnitGain | idDeadZone | idFeedforwardNetwork | idWaveletNetwork | idnlhw | idnlarx | evaluate | linearRegressor

Introduced in R2007a

iddata

Input-output data and its properties for system identification in the time or frequency domain

Description

Use the `iddata` object to encapsulate input and output measurement data for the system you want to identify. System identification functions use these measurements to estimate a model. Model validation functions use the input measurements to provide the input for simulations, and the output measurements to compare how well the estimated model response fits the original data.

`iddata` objects can contain a single set of measurements or multiple sets. Each set of data corresponds to an experiment. The objects have the following characteristics, which are encoded in the object properties on page 1-498:

- Data can be in the frequency domain or the time domain. You can convert objects from one domain to the other.
- In the time domain, the data can be uniformly or nonuniformly sampled. To use the `iddata` object for estimation, however, the data must be uniformly sampled, and the input and output data for each experiment must be recorded at the same time instants.
- You can specify data properties, such as the sample time, start time, time points, frequency sample points, and intersample behavior.
- You can provide labels and comments to differentiate and annotate data components, experiments, and the object as a whole.

.

Creation

Syntax

```
data = iddata(y,u,Ts)
data = iddata(y,[],Ts)
data = iddata(y,u,Ts,'Frequency',W)
data = iddata(___,Name,Value)
```

Description

`data = iddata(y,u,Ts)` creates an `iddata` object containing a time-domain output signal `y` and input signal `u`. `Ts` specifies the sample time of the experimental data.

You can use `iddata` to create a multiexperiment `iddata` object by specifying `y` and `u` as cell arrays. Alternatively, you can create single-experiment `iddata` objects and use `merge (iddata)` to combine the objects into one multiexperiment `iddata` object. For more information on multiexperiment `iddata` objects, see “Create Multiexperiment Data at the Command Line”.

`data = iddata(y,[],Ts)` creates an `iddata` object for time-series data. The object contains a time-domain output signal `y` and an empty input signal `[]`. `Ts` specifies the sample time of the experimental data.

`data = iddata(y,u,Ts,'Frequency',W)` creates an `iddata` object containing frequency-domain data. `W` sets the `iddata` property `Frequency` to a vector of frequencies. Typically, `y` and `u` are the discrete Fourier transform of time-domain signals.

`data = iddata(___,Name,Value)` sets additional properties using name-value pair arguments. Specify `Name,Value` after any of the input argument combinations in the previous syntaxes.

Input Arguments

y — Output signal from system

column vector | matrix | cell array | `[]`

Output signal from a system, specified as one of the following:

- An N-by-1 vector for a single output system, where N is the number of observations
- An N-by-Ny matrix for a multiple-output system, where Ny is the number of output channels
- An N_e element cell array for a multiexperiment data set, where N_e is the number of experiments and each cell contains the output signals for one experiment
- `[]` for a system that has no output signal, such as when only the input signal is recorded

`y` must be in the same domain as the input data `u`. If the data is in the time domain, `y` and `u` must be recorded at the same time instants.

If you use the `iddata` object for estimation, `y` and `u` must be uniformly sampled. If the nonuniformity is small, you may be able to able to convert your data into a uniformly sampled set with enough integrity that the converted data supports estimation. For more information on techniques you can try, see `interp1` and “Missing Data in MATLAB”.

`y` sets the `OutputData` property of the `iddata` object.

u — Input signal to system

column vector | matrix | cell array | `[]`

Input signal to a system, specified as one of the following:

- An N-by-1 vector for a single input system, where N is the number of observations
- An N-by-N_u matrix for a multiple-input system, where N_u is the number of input channels
- An N_e element cell array for a multiexperiment data set, where N_e is the number of experiments and each cell contains the input signals for one experiment
- `[]` for a system that has no input signal, such as a time series

`u` must be in the same domain as output data `y`. If the data is in the time domain, `y` and `u` must be recorded at the same time instants.

If you use the `iddata` object for estimation, `y` and `u` must be uniformly sampled. If the nonuniformity is small, you may be able to able to convert your data into a uniformly sampled set with enough integrity that the converted data supports estimation. For more information on techniques you can try, see `interp1` and “Missing Data in MATLAB”.

`u` sets the `InputData` property of the `iddata` object.

`Ts` — Sample time
`1` (default) | scalar | `0` | `[]`

Sample time in the units specified by the property `TimeUnit`, specified as one of the following:

- A scalar when `y` and `u` are uniformly sampled.
- `0` for continuous-time data in the frequency domain.
- `[]` when `y` and `u` are not uniformly sampled and you specify the time values in the property `SamplingInstants`. For nonuniform sampling, `y` and `u` must be in the time domain.

`Ts` sets the `Ts` property of the `iddata` object.

Properties

`Domain` — Data time or frequency domain
`'Time'` (default) | `'Frequency'`

Data time or frequency domain, specified as either:

- `'Time'` — Data is in the time domain
- `'Frequency'` — Data is in the frequency domain

`ExperimentName` — Name of each data set
`{'Exp1';'Exp2';…}` (default) | cell array

Name of each data set contained in the `iddata` object, specified as an N_e-by-1 cell array of character vectors, where N_e is the number of experiments. Each cell contains the name of the corresponding experiment. For instance, `{'MyMeas1';'MyMeas2';'MyMeas3'}` contains experiment names for a three-experiment `iddata` object.

`Frequency` — Frequency values
column vector | cell array

Frequency values for frequency-domain data, specified as either:

- An N-by-1 vector, where N is the number of frequency values in a single experiment
- A 1-by-N_e cell array, where N_e is the number of experiments and each cell contains the frequency vector for the corresponding experiment. The frequency vectors must all be in the same units.

`FrequencyUnit` — Frequency units for frequency-domain data
`'rad/TimeUnit'` (default) | `'cycles/TimeUnit'` | `'rad/s'` | `'Hz'` | `'kHz'` | `'MHz'` | `'GHz'` | `'rpm'`

Frequency units for frequency-domain data, specified as one of the following:

1. A scalar for a single experiment.

   A 1-by-N_e cell array, where N_e is the number of experiments. Because all `Frequency` vectors must have the same units, all values of `FrequencyUnit` must be the same.

Changing this property does not scale or convert the data. Modifying the property changes only the interpretation of the existing data.

`InputData` — Input signal values
vector | matrix | cell array of matrices

Input signal values to the system, specified as one of the following:

- For a single experiment, an N-by-N_u matrix, where N is the number of data samples and N_u is the number of input channels
- For multiple experiments, a cell array containing N_e single-experiment matrices, where N_e is the number of experiments

When accessing `InputData` from the command line, you can use the shorthand form `u`. For example, `u1 = data.InputData` is equivalent to `u1 = data.u`.

`InputName` — Input channel names
`{'u1';'u2';…}` (default) | cell array of character vectors

Input channel names, specified as an N_u-by-*1* cell array, where N_u is the number of input channels.

`InputUnit` — Input channel units
cell array

Input channel units, specified as an N_u-by-*1* cell array, where N_u is the number of input channels. Each cell contains the units of the corresponding input channel.

Example: `{'rad';'rad/s'}`

`InterSample` — Intersample behavior
`'zoh'` (default) | `'foh'` | `'bl'` | cell array of character vectors

Intersample behavior for transformations between discrete time and continuous time, specified as a character vector or as a cell array of character vectors. For each experiment, the possible values for each input channel are:

- `zoh` — 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 a single experiment with a single input channel, `InterSample` contains one of the values in the previous list. For multiple experiments, `InterSample` is an N_u-by-N_e cell array, where N_u is the number of input channels and N_e is the number of experiments. Each cell contains the behavior value associated with the experiment and input channel that the cell represents.

`Name` — Name of data set
`''` (default) | character vector

Name of the data set, specified as a character vector.

Example: `'dryer data'`

`Notes` — Comments about data set
`''` (default) | character vector | cell array

Comments about the data set, specified as a character vector or, for multiexperiment data sets, an N_e-by-1 cell array of character vectors, where N_e is the number of experiments.

Example: `{'data from experiment 1';data from experiment 2'}`

`OutputData` — Output signal values
vector | matrix

Output signal values from the system, specified as one of the following:

- For a single experiment, an N-by-N_y matrix, where N is the number of data samples and N_y is the number of output channels
- For multiple experiments, a cell array containing N_e single-experiment matrices, where N_e is the number of experiments

When accessing `OutputData` from the command line, you can use the shorthand form `y`. For example, `y1 = data.InputData` is equivalent to `y1 = data.y`.

`OutputName` — Output channel names
`{'y1';'y2';…}` (default) | cell array of character vectors

Output channel names, specified as an N_y-by-1 cell array, where N_y is the number of output channels.

`OutputUnit` — Output channel units
cell array

Output channel units, specified as an N_y-by-1 cell array, where N_u is the number of output channels. Each cell contains the units of the corresponding input channel.

Example: `{'rad';'rad/s'}`

`Period` — Period of input signal
`Inf` (default) | double | vector | cell array | cell array of character vectors

Period of the input signal, specified as a double for each experiment. The value is either `Inf` for nonperiodic input signals or the period in the units specified by the property `TimeUnit` for periodic input signals.

- For a single experiment with a single input channel, `Period` contains a single value.
- For a multiple-input system, `Period` is an N_u-by-1 vector, where N_u is the number of input channels and the kth entry contains the period of the kth input.
- For multiple-experiment data, `Period` is a 1-by-N_e cell array, where N_e is the number of experiments and each cell contains a scalar or vector of periods for the corresponding experiment.

`SamplingInstants` — Time values for time-domain data
vector | cell array

Time values for time-domain data in units specified by `TimeUnit`, specified as:

- An N-by-1 vector, where N is the number of data points
- A 1-by-N_e cell array, where N_e is the number of experiments and each cell contains the sampling instants for the corresponding experiment.

The values in `SamplingInstants` can be uniform or nonuniform. If you specify the `Ts` property, the software computes uniform time values in `SamplingInstants` from `Ts` and `Tstart`. If you have nonuniform sample points, specify the time values in `SamplingInstants`. The software then sets the `Ts` property to empty. Estimation functions do not support nonuniform sampling.

`TimeUnit` — Units for time variable and sample time `Ts`
`'seconds'` (default) | `'nanoseconds'` | `'microseconds'` | `'milliseconds'` | `'minutes'` | `'hours'` | `'days'` | `'weeks'` | `'months'` | `'years'`

Units for the time variable and the sample time, specified as a scalar. This property applies to all experiments in the data set.

Changing this property does not resample or convert the data. Modifying the property changes only the interpretation of the existing data.

`Ts` — Sample time
`1` (default) | positive scalar | `0` | `[]` | cell array

Sample time in units specified by `TimeUnit`, specified as a scalar or a cell array. For each experiment, the value is one of the following:

- A scalar, when `y` and `u` are uniformly sampled
- `0` for continuous-time data in the frequency domain
- `[]` when `y` and `u` are non uniformly sampled and in the time domain, because the `SamplingInstants` property sets the time values for such data.

For a single experiment, `Ts` is a scalar. For multiexperiment data, `Ts` is a 1-by-N_e cell array, where N_e is the number of experiments and each cell contains the sample time for the corresponding experiment.

For frequency-domain data, the software uses `Ts` to interpret the data.

- If `Ts` is 0, the software interprets inputs and outputs as continuous-time Fourier transforms (CTFTs) of the corresponding signals.
- If `Ts` is a scalar, the software interprets inputs and outputs as discrete-time Fourier transforms (DTFTs) of the corresponding signals with `Ts` as sample time.

`Tstart` — Start time for time-domain data
`0` (default) | scalar | cell array

Start time for time-domain data, specified as:

- A scalar for a single experiment
- A 1-by-N_e cell array for multiple experiments, where N_e is the number of experiments and each cell contains the start time for the corresponding experiment

`UserData` — Additional comments
`[]` (default) | any MATLAB data type

Additional comments on the data set, specified as any MATLAB data type.

Object Functions

In general, any function applicable to system identification data is applicable to an `iddata` object. These functions are of three general types.

1. Functions that both operate on and return `iddata` objects enable you to manipulate and process `iddata` objects.

- Use `fft` and `ifft` to transform existing `iddata` objects to and from the time and frequency domains. For example:

  ```
  datafd = fft(Data);
  datatd = ifft(Dataf);
  ```

- Use `merge (iddata)` to merge `iddata` objects into a single `iddata` object containing multiple experiments. To extract an experiment from a multiexperiment `iddata` object, use `getexp`. For example:

  ```
  data123 = merge(data1,data2,data3);
  data2 = getexp(data123,2);
  ```

  For a more detailed example, see “Extract and Model Specific Data Segments”.

- Use preprocessing functions such as `detrend` or `idfilt` to filter data in `iddata` objects and to remove bad data. For example:

  ```
  data_d = detrend(data);
  data_f = idfilt(data,filter);
  ```

2. Functions that perform analytical processing on `iddata` objects and create plots or return specific parameters or values let you analyze data and determine inputs to use for estimation.

   - Use analysis functions such as `delayest` and `spa` to compute variables such as time delay and frequency spectrum.

3. Functions that use the data in `iddata` objects to estimate, simulate, and validate models let you create dynamic models and evaluate how closely the model response matches validation data.

   - Use estimation functions such as `ssest` and `tfest` to estimate models with specific structures.
   - Use validation functions such as `compare` and `sim` to simulate estimated models and compare the simulated outputs with validation data and with other models.

The following lists contain a representative subset of the functions you can use with `iddata` objects.

Data Visualization

plot	Plot input and output channels of iddata object

Data Selection

getexp	Specific experiments from multiple-experiment data set
merge (iddata)	Merge data sets into iddata object

Data Preprocessing

detrend	Subtract offset or trend from time-domain signals contained in iddata objects
retrend	Add offsets or trends to time-domain data signals stored in iddata objects
idfilt	Filter data using user-defined passbands, general filters, or Butterworth filters
diff	Difference signals in iddata objects
misdata	Reconstruct missing input and output data
idresamp	Resample time-domain data by decimation or interpolation

Data Transformation

fft	Transform iddata object to frequency domain data
ifft	Transform iddata objects from frequency to time domain

Data Analysis

realdata	Determine whether iddata is based on real-valued signals
delayest	Estimate time delay (dead time) from data
isreal	Determine whether model parameters or data values are real
impulseest	Nonparametric impulse response estimation
pexcit	Level of excitation of input signals
feedback	Identify possible feedback data
etfe	Estimate empirical transfer functions and periodograms
spafdr	Estimate frequency response and spectrum using spectral analysis with frequency-dependent resolution
spa	Estimate frequency response with fixed frequency resolution using spectral analysis
iddataPlotOptions	Option set for plot when plotting data contained in an iddata object

Model Estimation, Simulation, and Validation

ssest	Estimate state-space model using time-domain or frequency-domain data
tfest	Estimate transfer function
ar	Estimate parameters of AR model or ARI model for scalar time series
sim	Simulate response of identified model
findstates	Estimate initial states of model
compare	Compare identified model output and measured output
predict	Predict K-step-ahead model output
goodnessOfFit	Goodness of fit between test and reference data for analysis and validation of identified models
procest	Estimate process model using time or frequency data
resid	Compute and test residuals

Examples

Time-Domain Data

Create an `iddata` object using single-input/single-output (SISO) time-domain data. The input and output each contain 1000 samples with a sample time of 0.08 seconds.

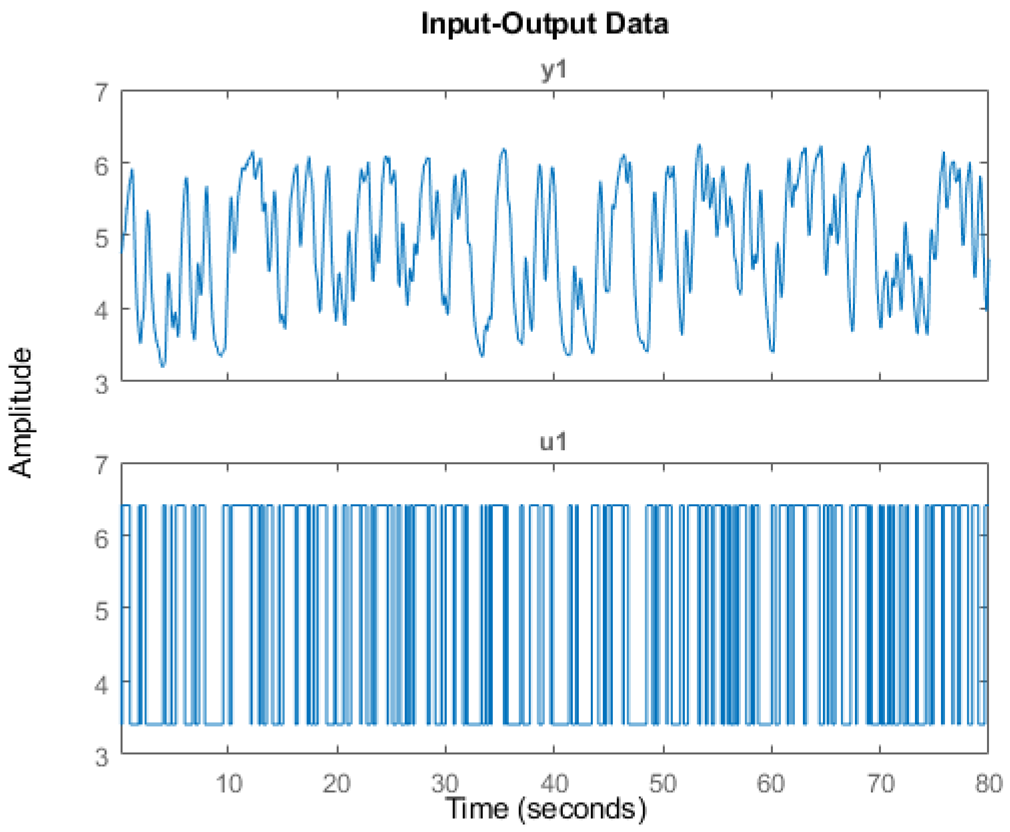
```
load dryer2_data output input;
data = iddata(output,input,0.08)
```

```
data =

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

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

The software assigns the default channel name `'y1'` to the first and only output channel. When the output signal contains several channels, the software assigns the default names `'y1','y2',...,'yn'`. Similarly, the software assigns the default channel name `'u1'` to the first and only input channel. For more information about naming channels, see “Naming, Adding, and Removing Data Channels”.

Plot the data.

```
plot(data)
```



Adjacent plots display output data and input data.

Time-Series Data

Create an `iddata` object from time-series data. Time-series data has no input channel.

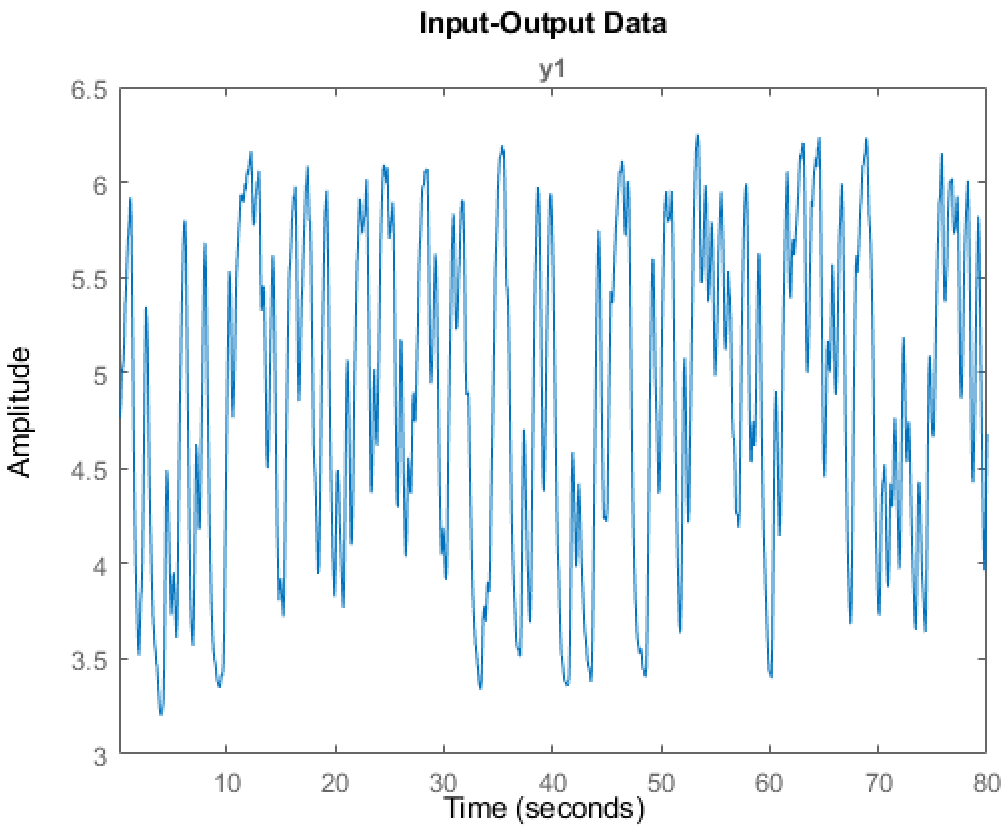
Load the output channel of a data set, and create an `iddata` object that has a sample time of 0.08 seconds.

```
load dryer2_data output
data = iddata(output,[],0.08)
```

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

Outputs      Unit (if specified)
   y1
```

Plot the data.

```
plot(data)
```



You can use `data` for time-series model estimation.

Frequency-Domain Data

Create and examine an `iddata` object from complex-valued frequency-domain input-output data. Convert the object into the time domain.

Input and output data is sometimes expressed in the form of the Fourier transforms of time-domain input-output signals. You can encapsulate this data in a frequency-domain `iddata` object.

Load the data, which consists of the complex-valued input-output frequency-domain data `U` and `Y`, frequency vector `W`, and sample time `Ts`.

```
load demofr1 U Y W Ts
```

Create the frequency-domain `iddata` object `data_fr`.

```
data_fr = iddata(Y,U,Ts,'Frequency',W)
```

```
data_fr =

Frequency domain data set with responses at 501 frequencies.
Frequency range: 0 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Examine the properties. Frequency-domain `iddata` objects include frequency-specific properties, such as `Frequency` for the frequency vector and `FrequencyUnit` for frequency units. In contrast, time-domain `iddata` objects include time-specific properties such as `Tstart` and `SamplingInstants` for time-domain data.

```
get(data_fr)
```

```
ans = struct with fields:
            Domain: 'Frequency'
              Name: ''
        OutputData: [501x1 double]
                 y: 'Same as OutputData'
        OutputName: {'y1'}
        OutputUnit: {''}
         InputData: [501x1 double]
                 u: 'Same as InputData'
         InputName: {'u1'}
         InputUnit: {''}
            Period: Inf
       InterSample: 'zoh'
                Ts: 0.1000
     FrequencyUnit: 'rad/TimeUnit'
         Frequency: [501x1 double]
          TimeUnit: 'seconds'
    ExperimentName: 'Exp1'
             Notes: {}
          UserData: []
```

Assign the contents of the frequency property to the variable `F`.

```
F = data_fr.Frequency;
```

Get the frequency units of the data. The property `TimeUnit` sets the units of the sample time.

```
frequ = data_fr.FrequencyUnit
```

```
frequ = 
'rad/TimeUnit'
```

```
timeu = data_fr.TimeUnit
timeu =
'seconds'
```

Convert `data_fr` back into the time domain by using the inverse Fourier transform function `ifft`.

```
data_t = ifft(data_fr)
```

```
data_t =

Time domain data set with 1000 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

```
get(data_t)
```

```
ans = struct with fields:
              Domain: 'Time'
                Name: ''
          OutputData: [1000x1 double]
                   y: 'Same as OutputData'
          OutputName: {'y1'}
          OutputUnit: {''}
           InputData: [1000x1 double]
                   u: 'Same as InputData'
           InputName: {'u1'}
           InputUnit: {''}
              Period: Inf
         InterSample: 'zoh'
                  Ts: 0.1000
              Tstart: []
    SamplingInstants: []
            TimeUnit: 'seconds'
      ExperimentName: 'Exp1'
               Notes: {}
            UserData: []
```

View and Modify Properties

View properties of an `iddata` object. Modify the properties both during and after object creation.

Load input and output data.

```
load dryer2_data input output
```

Create an `iddata` object.

```
data = iddata(output,input,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
```

View all properties of the `iddata` object.

```
get(data)
```

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

You can specify properties when you create an `iddata` object using name-value pair arguments. Create an `iddata` object from the same data inputs, but change the experiment name from its default setting to `Dryer2`.

```
data = iddata(output,input,0.08,'ExperimentName','Dryer2')
```

```
data =

Experiment Dryer2.Time domain data set with 1000 samples.
Sample time: 0.08 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

To change property values for an existing `iddata` object, use dot notation. Change the sample time property `Ts` to 0.05 seconds.

```
data.Ts = 0.05
```

```
data =

Experiment Dryer2.Time domain data set with 1000 samples.
Sample time: 0.05 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

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

```
data.exp = "Dryer2 January 2015"
```

```
data =

Experiment Dryer2 January 2015.Time domain data set with 1000 samples.
Sample time: 0.05 seconds

Outputs                   Unit (if specified)
   y1

Inputs                    Unit (if specified)
   u1
```

You can use `data.y` as a shorthand for `data.OutputData` to access the output values, or use `data.u` as a shorthand for `data.InputData` to access the input values.

```
y_data = data.y;
u_data = data.u;
```

See Also

`idfrd` | `idinput`

Topics

"Representing Data in MATLAB Workspace"
"Create Multiexperiment Data at the Command Line"
"Representing Time- and Frequency-Domain Data Using iddata Objects"
"Managing iddata Objects"

Introduced before R2006a

iddataPlotOptions

Option set for `plot` when plotting data contained in an `iddata` object

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.

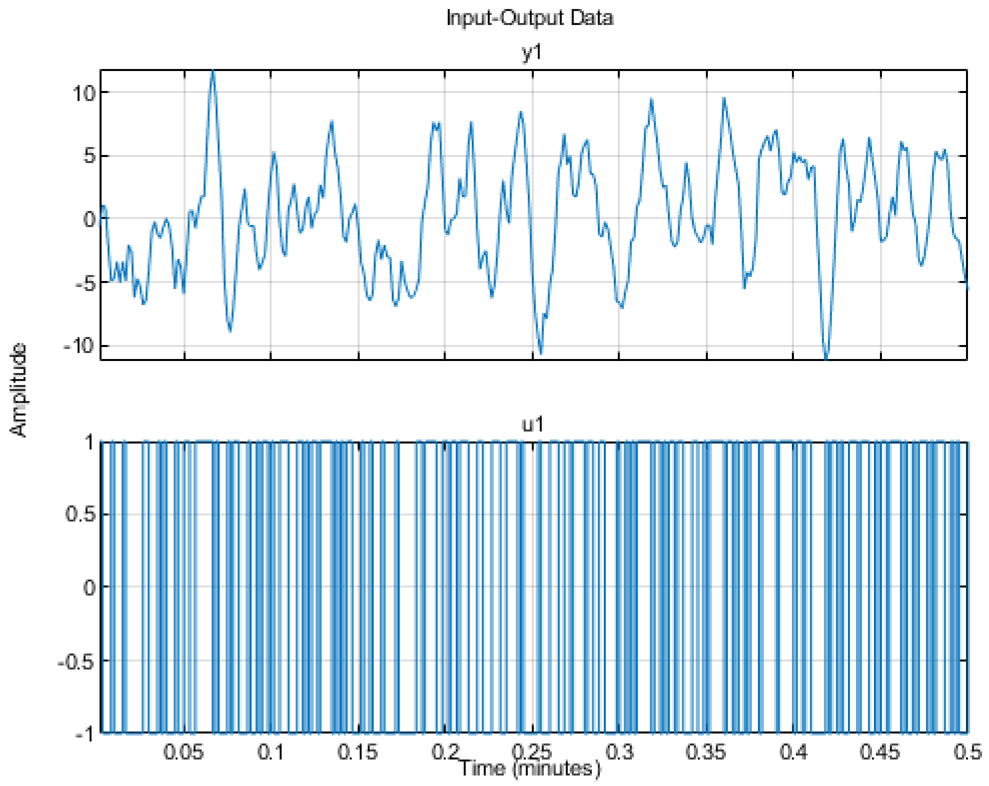
```
opt = iddataPlotOptions('time');
```

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.

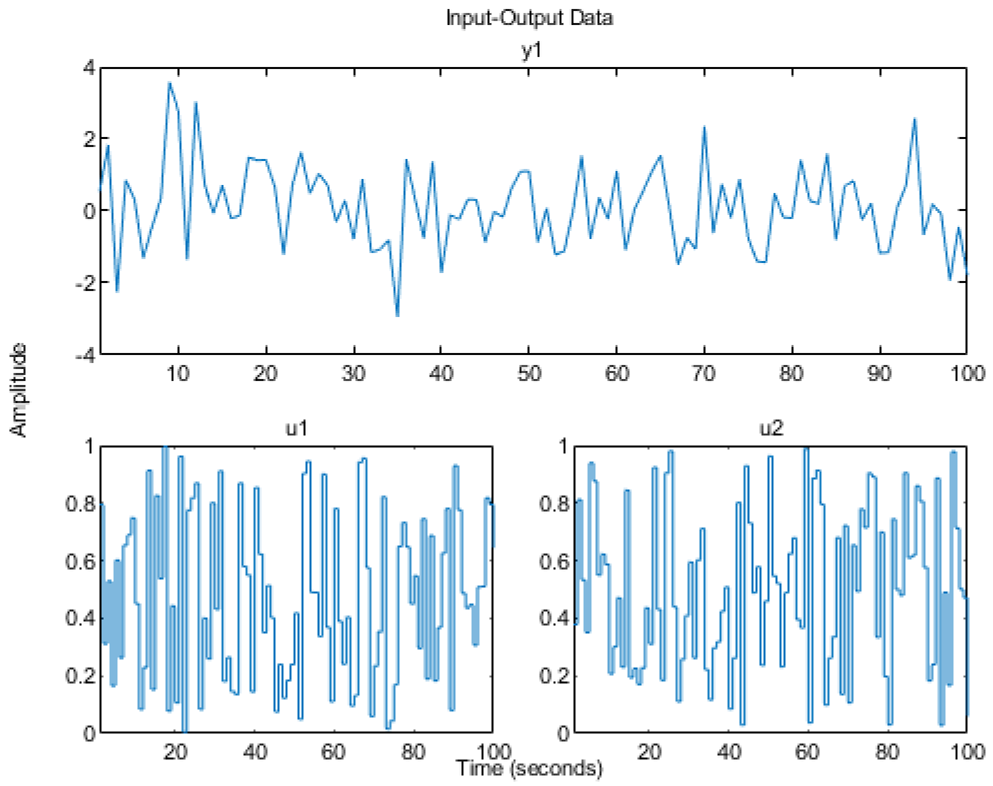
```
z = iddata(randn(100,1),rand(100,2));
```

Configure a time plot.

```
opt = iddataPlotOptions('time');
```

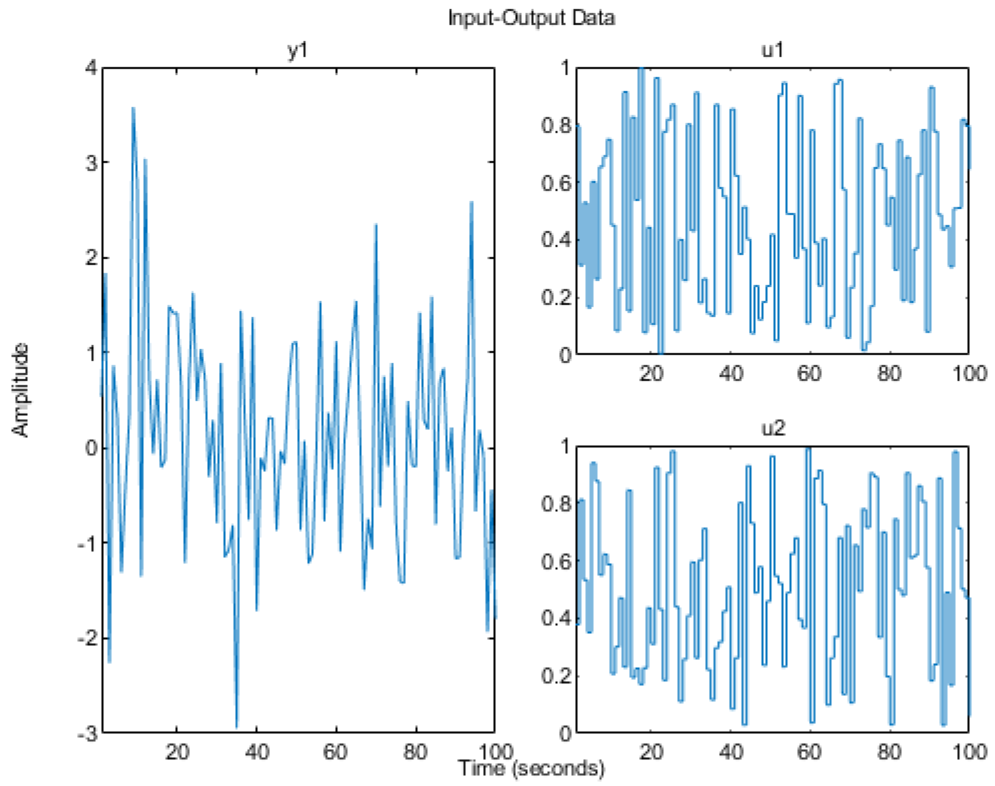
Plot the data.

```
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.

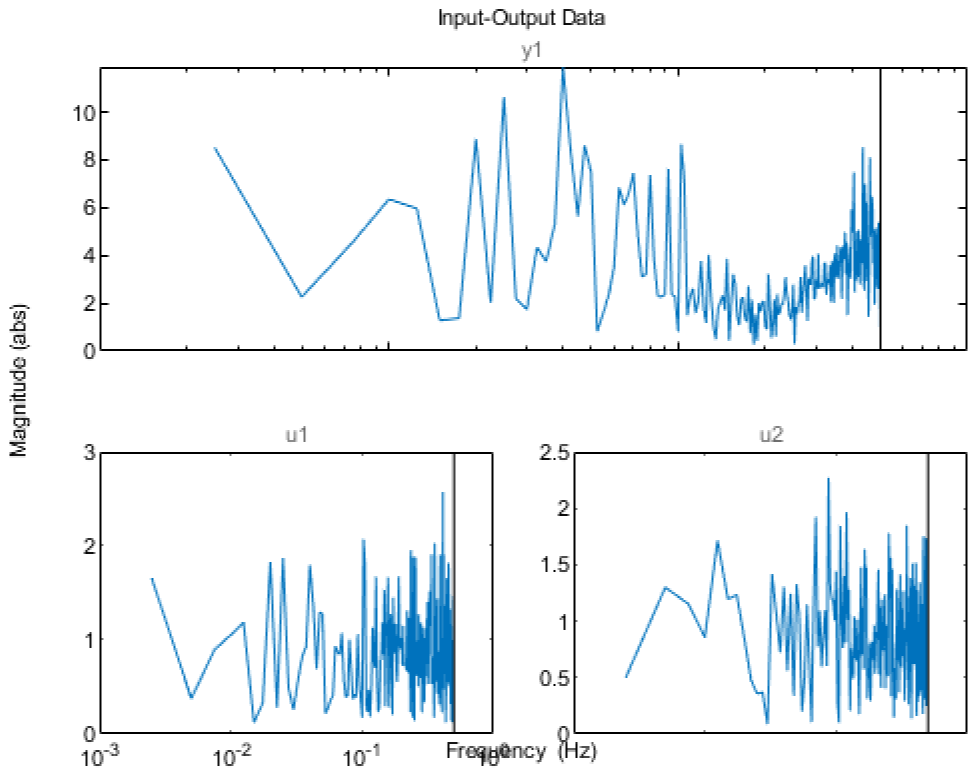
```
opt = iddataPlotOptions('frequency');
```

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

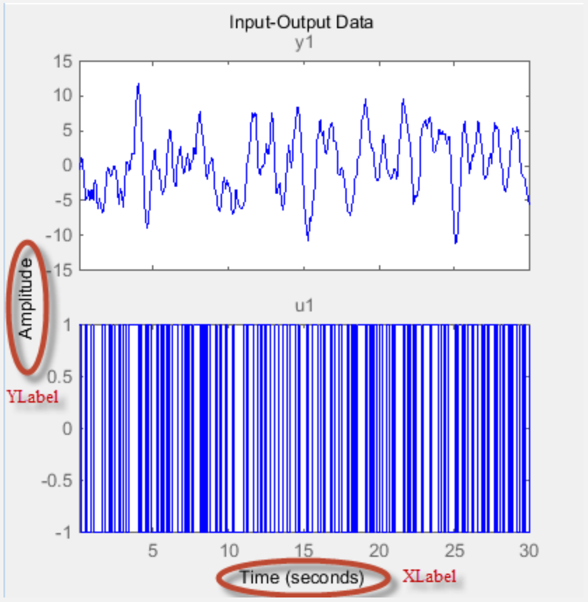
```
opt = iddataPlotOptions('time','identpref');
```

Output Arguments

opt — Option set for `iddata/plot`

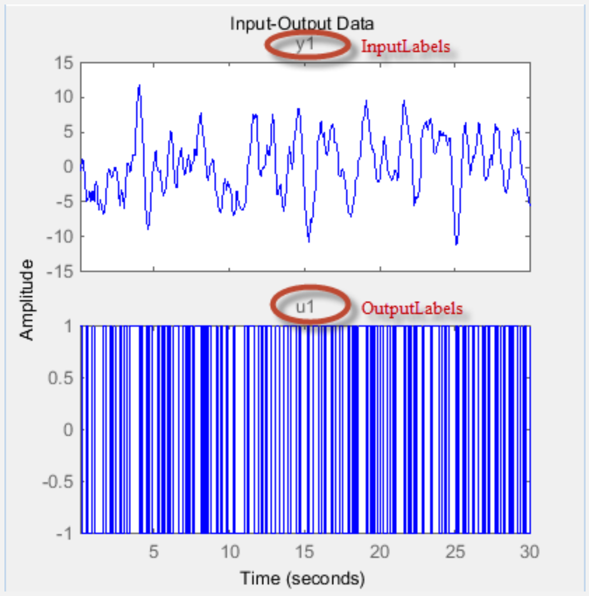
`iddataPlotOptions` option set

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'` **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'`

Field	Description
	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]`
`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]`

Field	Description
`IOGrouping`	Grouping of input-output pairs, specified as one of the following values: `'none'` | `'inputs'` | `'outputs'`|`'all'` **Default**: `'none'`

Field	Description
`InputLabels`, `OutputLabels`	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 | `'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'`

Field	Description
`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'`.
For time-domain data plots only:	

Field	Description
`TimeUnits`	Time units, specified as one of the following values: • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'` You can also specify `'auto'` which uses time units specified in the `TimeUnit` property of the data. For multiple systems with different time units, the units of the first system is used.
`Normalize`	Normalize responses, specified as one of the following values: `'on'` |`'off'` **Default**: `'off'`

Field	Description
For frequency-domain data plots only:	

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 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'`

Field	Description
Field	**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`	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'`
`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

`plot` | `identpref`

Introduced in R2014a

idDeadZone

Create a dead-zone nonlinearity estimator object

Syntax

```
NL = idDeadZone
NL = idDeadZone('ZeroInterval',[a,b])
```

Description

`NL = idDeadZone` 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 = idDeadZone('ZeroInterval',[a,b])` creates a dead-zone nonlinearity estimator object initialized with zero interval, `[a,b]`.

Alternatively, use `NL = idDeadZone([a,b])`.

Object Description

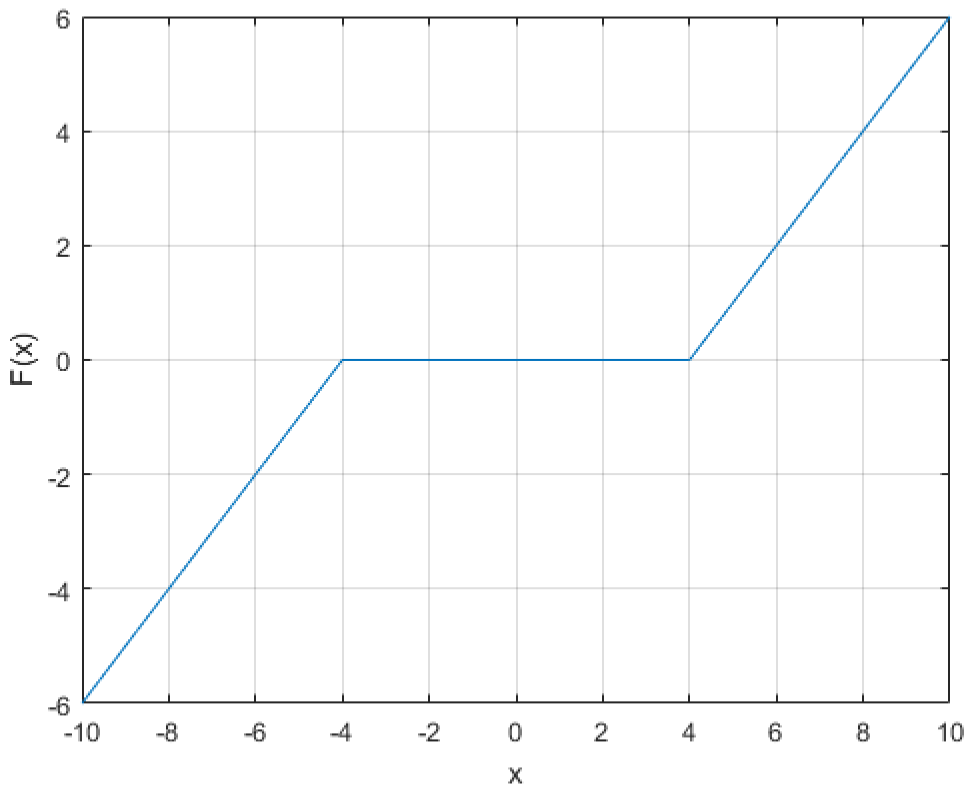
`idDeadZone` is an object that stores the dead-zone nonlinearity estimator for estimating Hammerstein-Wiener models.

Use `idDeadZone` 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 `idDeadZone` object.

For `idDeadZone` object properties, see “Properties” on page 1-527.

Examples

Create a Default Dead-Zone Nonlinearity Estimator

```
NL = idDeadZone;
```

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 an `idDeadZone` object, and specify the initial guess for the zero-interval.

```
OutputNL = idDeadZone('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 = idSaturation;
```

Specify different nonlinearity estimators for each output channel.

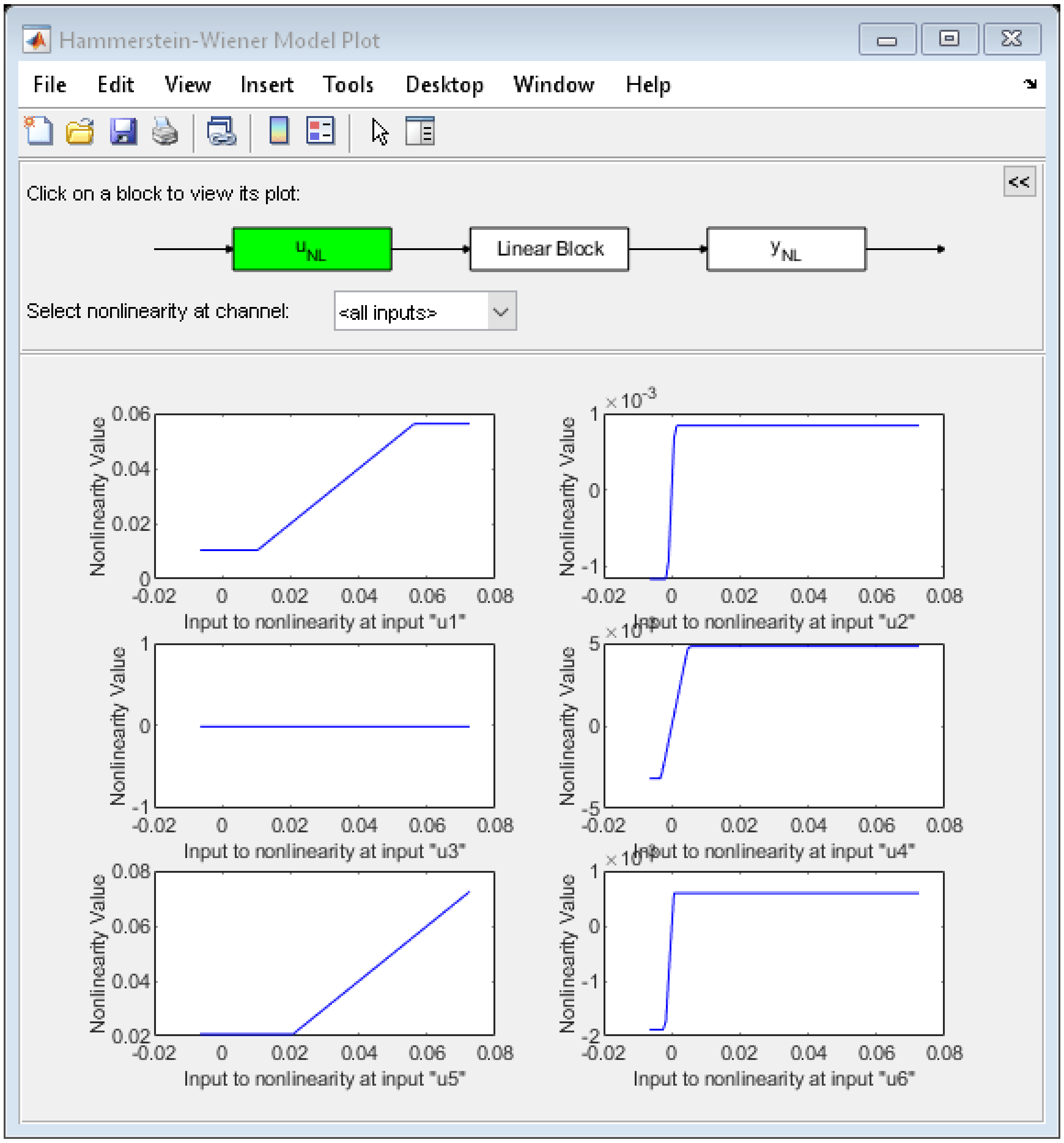
```
 OutputNL = [idDeadZone,idWaveletNetwork];
```

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
<<
u_{NL}
Linear Block
y_{NL}
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"

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 nonlinearities.

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]`

`Free`

Option to fix or free the parameters of `ZeroInterval`, specified as a 2–element logical row vector. When you set an element of `Free` to `false`, the corresponding value in `ZeroInterval` remains fixed during estimation to the initial value that you specify.

Default: `[true true]`

Output Arguments

`NL` — Dead-zone nonlinearity estimator object
`idDeadZone` object

Dead-zone nonlinearity estimator object, returned as an `idDeadZone` object.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`

Pre-R2021b Name	R2021b Name
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

See Also
`nlhw`

Introduced in R2007a

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.

See Also

Topics
"Toolbox Preferences Editor"

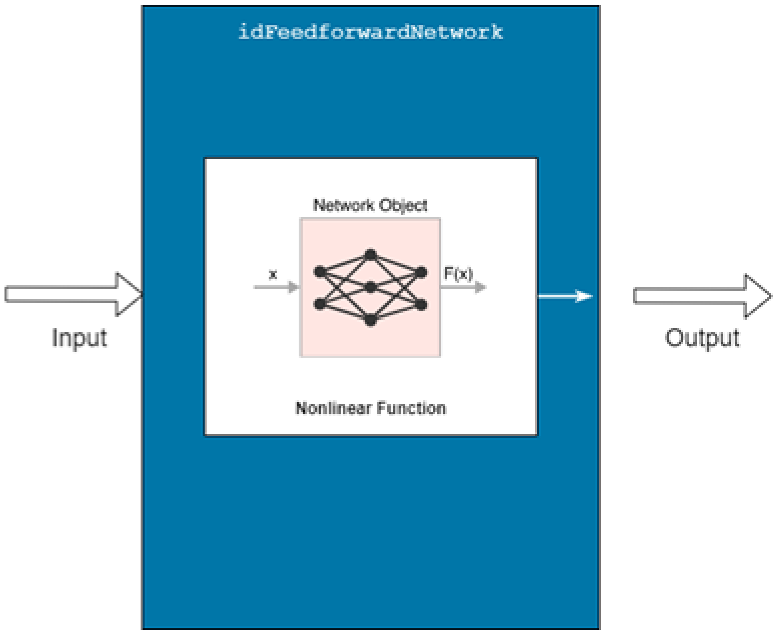
Introduced in R2012a

idFeedforwardNetwork

Multilayer feedforward neural network mapping function for nonlinear ARX models (requires Deep Learning Toolbox)

Description

An `idFeedforwardNetwork` object implements a neural network function, and is a nonlinear mapping object for estimating nonlinear ARX models. This mapping object lets you use `network` objects that are created using Deep Learning Toolbox™ in nonlinear ARX models.



Mathematically, `idFeedforwardNetwork` is a function that maps *m* inputs $X(t) = [x(t_1),x_2(t),\ldots,x_m(t)]^T$ to a scalar output *y*(*t*), using a multilayer feedforward (static) neural network, as defined in Deep Learning Toolbox.

You create multi-layer feedforward neural networks by using 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 and 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 1-0 for more information.

Use `evaluate(net_estimator,x)` to compute the value of the function defined by the `idFeedforwardNetwork` 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 `idFeedforwardNetwork` object is assigned as the nonlinearity estimator.

You cannot use `idFeedforwardNetwork` when the `Focus` option in `nlarxOptions` is `'simulation'` because the underlying `network` object is considered to be nondifferentiable for estimation. Minimization of simulation error requires differentiable nonlinear functions.

Use `idFeedforwardNetwork` as the value of the `OutputFcn` property of an `idnlarx` model. For example, specify `idFeedforwardNetwork` when you estimate an `idnlarx` model with the following command.

```
sys = nlarx(data,regressors,idFeedforwardNetwork)
```

When `nlarx` estimates the model, it essentially estimates the parameters of the `idFeedforwardNetwork` function.

Creation

Syntax

`net_estimator = idFeedforwardNetwork(Network)`

Description

`net_estimator = idFeedforwardNetwork(Network)` creates a feedforward neural network mapping object that is based on the feedforward (static) network object `Network` that has been created using one of the neural network commands `feedforwardnet`, `cascadeforwardnet`, or `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 set to one. For a multiple-output nonlinear ARX models, create a separate `idFeedforwardNetwork` object for each output—that is, each element of the output function must represent a single-output network object.

Properties

`Network` — Feedforward neural network
`Network` object

Feedforward neural network object, typically created using `feedforwardnet`, `cascadeforwardnet` or `linearlayer`.

`Input` — Input signal information
signal property values

Input signal information for signals used for estimation, specified as vectors of *m* property-specific values, where *m* is the number of input signals. The `Input` properties for each input signal are as follows:

- `Name` — Names of the input signals, specified as a 1-by-*m* string or character array, where *m* is the number of inputs
- `Mean` — Mean of the input signals, specified as a numeric scalar
- `Range` — Ranges of the input signals, specified as a 2-by-*m* numeric array that contains the minimum and maximum values

Free — Option to train the network

true (default) | false

Option to train the neural network, specified as true or false. Set Free to false when the neural network you are using has already been trained and is known to provide good fit results. The Free property is especially useful when your idnlarx model has multiple outputs that each use a neural network. Setting Free to false for well trained networks allows processing time to be focused on the networks that do need training.

Output — Output signal information

signal property values

Output signal information, specified as property-specific values. The Output properties are as follows:

- Name — Name of the output signal, specified as a string or a character array
- Mean — Mean of the output signal, specified as a numeric scalar
- Range — Range of the output signal, specified as a 2-by-1 numeric array that contains the minimum and maximum values

Examples

Create Nonlinear Mapping Object Using Feedforward Neural Network

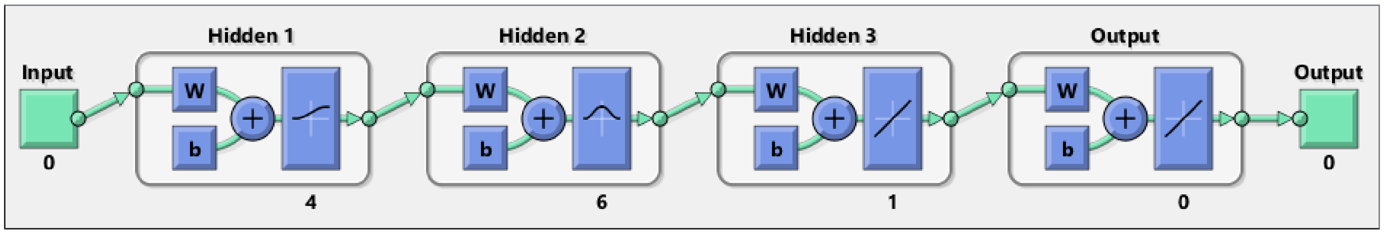
Create a neural network mapping object that uses a feedforward neural network with three hidden layers, transfer functions of types logsig, radbas,and purelin, and unknown input and output sizes.

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 mapping object.

```
net_estimator = idFeedforwardNetwork(net);
```

Estimate Nonlinear ARX Model Using Feedforward Neural Network Mapping Object

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 feedforward neural network mapping object.

```
net_estimator = idFeedforwardNetwork(net);
```

Load the estimation data.

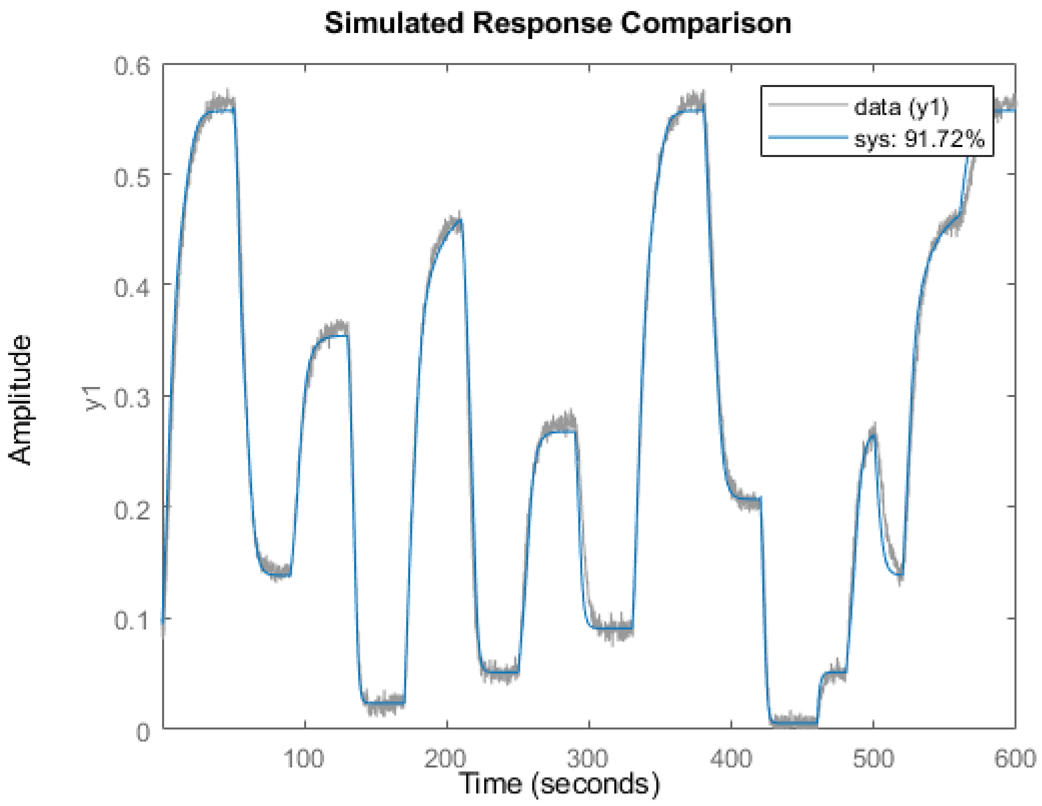
```
load twotankdata
data = iddata(y,u,0.2);
```

Estimate a nonlinear ARX model `sys`.

```
sys = nlarx(data,[2 2 1],net_estimator);
```

Compare the model response to the measured output signal.

```
compare(data,sys)
```



The plot shows good agreement between the measured signal and the simulated model output signal.

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);
R = R.Variables;
```

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 and the choice of training function.

```
net.outputs{net.outputConnect}.range = OutputRange;
net.trainFcn = 'trainbfg';
```

Create a neural network nonlinearity estimator.

```
net_estimator = idFeedforwardNetwork(net);
```

Specify the nonlinearity estimator in the model.

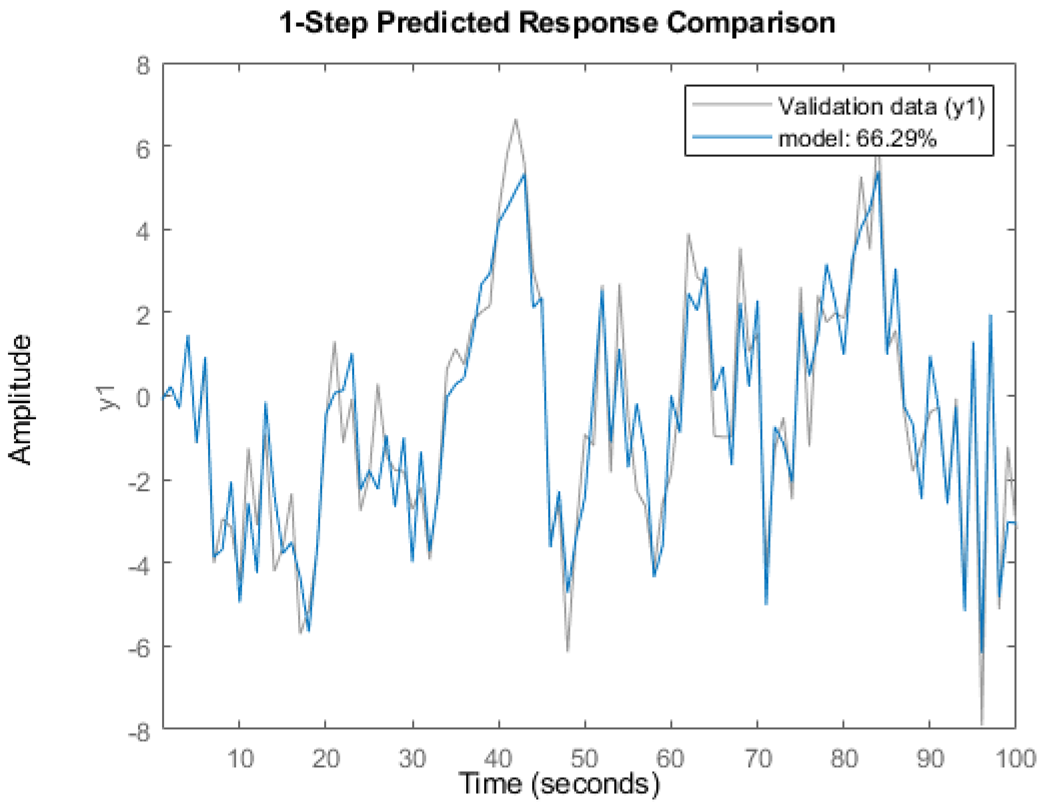
```
model.Nonlinearity = net_estimator;
```

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)
```



Algorithms

The `nlarx` command uses the `train` method of the `network` object, defined in the Deep Learning Toolbox software, to compute the network parameter values.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`

Pre-R2021b Name	R2021b Name
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

See Also

`idnlarx` | `nlarx` | `idLinear` | `idSigmoidNetwork` | `idTreePartition` | `idWaveletNetwork` | `idCustomNetwork` | `evaluate`

Introduced in R2007a

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`.

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

Description

An `idfrd` object stores frequency response data over a range of frequency values. You can use an `idfrd` object in two ways. You can use the object as estimation data for estimating a time-domain or frequency-domain model, similarly to an `iddata` object. Or, you can use the object as a linear model, similarly to how you use an `idss` state-space model or any other identified linear model. Use the `idfrd` command to encapsulate frequency response data or to convert a linear time-domain or frequency-domain dynamic model into a frequency response model.

Commands that accept `iddata` objects, such as the model estimation command `ssest`, generally also accept `idfrd` objects. However, an `idfrd` object can contain data from only one experiment. It does not have the multiexperiment capability that an `iddata` object has.

Commands that accept identified linear models, such as the analysis and validation commands `compare`, `sim`, and `bode`, generally also accept `idfrd` models.

For a model of the form

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

the transfer function estimate is $G\left(e^{i\omega}\right)$ and the additive noise spectrum Φ_v at the output is

$$\Phi_v(\omega) = \lambda T|H(e^{i\omega T})|^2$$

Here, λ is the estimated variance of $e(t)$ and T is the sample time.

For a continuous-time system, the noise spectrum is

$$\Phi_v(\omega) = \lambda|H(e^{i\omega})|^2$$

An `idfrd` object stores $G\left(e^{i\omega}\right)$ and Φ_v.

Creation

You can obtain an `idfrd` model in one of three ways.

- Create the model from frequency response data using the `idfrd` command. For example, create an `idfrd` model that encapsulates frequency response data taken at specific frequencies using the sample time `Ts`.

  ```
  sysfr = idfrd(ResponseData,Freq,Ts)
  ```

  For an example, see “Create idfrd Object from Frequency Response Data” on page 1-548.
- Estimate the model using a frequency response estimation command such as `spa`, using time-domain, frequency-domain, or frequency response data.

```
sysfr = spa(data)
```

For more information about frequency response estimation commands, see `spa`, `spafdr`, and `etfe`.

- Convert a linear model such as an `idss` model into an `idfrd` model by computing the frequency response of the model.

  ```
  sysfr = idfrd(sys)
  ```

  For an example of linear model conversion, see “Convert Time-Domain Model to Frequency Response Model” on page 1-551.

For information on functions you can use to extract information from or transform `idfrd` model objects, see “Object Functions” on page 1-548.

Syntax

```
sysfr = idfrd(ResponseData,Frequency,Ts)
sysfr = idfrd(___,Name,Value)

sysfr = idfrd(sys)
sysfr = idfrd(sys,Frequency)
sysfr = idfrd(sys,Frequency,FrequencyUnits)
```

Description

Create Frequency Response Object

`sysfr = idfrd(ResponseData,Frequency,Ts)` creates a discrete-time `idfrd` object that stores the frequency response `ResponseData` of a linear system at frequency values `Frequency`. `Ts` is the sample time. For a continuous-time system, set `Ts` to `0`.

`sysfr = idfrd(___,Name,Value)` sets additional properties using one or more name-value arguments. Specify the name-value arguments after the first three arguments. For instance, to specify the frequency units as MHz, use `sysfr = idfrd(ResponseData,Frequency,Ts,'FrequencyUnits','MHz')`.

Convert Linear Identified Model to Frequency Response Model

`sysfr = idfrd(sys)` converts a System Identification Toolbox or Control System Toolbox linear model to frequency response data at default frequencies, including the output noise spectra and spectra covariance.

`sysfr = idfrd(sys,Frequency)` computes the frequency response at frequencies `Frequency`, where `Frequency` is expressed in radians/`TimeUnit`.

`sysfr = idfrd(sys,Frequency,FrequencyUnits)` interprets frequencies in the `Frequency` vector in the units specified by `FrequencyUnit`.

Input Arguments

sys — Linear dynamic system model

linear dynamic system model

Linear dynamic system model, specified as a System Identification Toolbox or Control System Toolbox linear model.

Properties

`ResponseData` — Frequency response data
3-D array of complex numbers

Frequency response data, specified 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 N_u inputs and N_y outputs, `ResponseData` is an N_y-by-N_u-by-N_f array, where N_f is the number of frequency points.

  `ResponseData(ky,ku,kf)` represents the frequency response from the input `ku` to the output `ky` at the frequency `Frequency(kf)`.

`Frequency` — Frequency points
column vector

Frequency points corresponding to `ResponseData`, specified as a column vector that contains N_f points in the units specified by `FrequencyUnit`.

`FrequencyUnit` — Units for frequency vector
`'rad/TimeUnit'` (default) | `'cycles/TimeUnit'` | `'rad/s'` | `'Hz'` | `'kHz'` | `'MHz'` | `'GHz'` | `'rpm'`

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 does not resample or convert the data. Modifying the property changes only the interpretation of the existing data. Use `chgTimeUnit` to convert the data to different frequency units.

`SpectrumData` — Power spectra and cross spectra
vector of complex numbers | 3-D array of complex numbers

Power spectra and cross spectra of the system output disturbances (noise), specified as a vector (single-output system) or a 3-D array of complex numbers (multiple-output system). For response data with N_y outputs and N_f frequency points, specify `SpectrumData` as an N_y-by-N_y-by-N_w array.

`SpectrumData(ky1,ky2,kf)` is the cross spectrum between the noise at output `ky1` and the noise at output `ky2` at the frequency `Frequency(kf)`. The power spectrum is the subset of the cross spectrum where `ky1` and `ky2` are equal.

`CovarianceData` — Covariance of response
5-D numeric array

Covariance of `SpectrumData`, specified as a 5-D array with dimensions N_y-by-N_u-by-N_f-by-2-by-2, where N_y is the number of outputs, N_u is the number of inputs, and N_f is the number of frequency points.

`CovarianceData(ky,ku,kf,:,:)` is the 2-by-2 covariance matrix of `SpectrumData(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(CovarianceData(ky,ku,kf,:,:))` thus gives the covariance matrix of the corresponding response.

If you obtain `sysfr` by converting a model `sys`, the value of `CovarianceData` depends on how you obtained `sys`.

- If you obtained `sys` by identification, the software computes the estimated covariance for `sysfr` from the uncertainty information in `sys`. The software uses Gauss' approximation formula for this calculation for all model types, except grey-box models. For grey-box models (`idgrey`), the software applies numerical differentiation.
- If you created `sys` by using commands such as `idss`, `idtf`, `idproc`, `idgrey`, or `idpoly`, then the software sets `CovarianceData` for `sysfr` to `[]`.

`NoiseCovariance` — Power spectra variance
numeric vector | 3-D numeric array | `0`

Power spectra variance, specified as a vector (single-output system) or a 3-D array (multiple-output system). For response data with N_y outputs and N_w frequency points, specify `NoiseCovariance` as an N_y-by-N_y-by-N_w array. `NoiseCovariance(ky1,ky2,kw)` is the variance of the corresponding power spectrum.

To eliminate the influence of the noise component from the model, specify `NoiseCovariance` as `0`. With zero covariance, the predicted output is the same as the simulated output.

`InterSample` — Intersample behavior
`'zoh'` | `'foh'` | `'bl'` | cell array of character vectors

Intersample behavior of the input signal for transformations between discrete time and continuous time, specified as a character vector or as an N_u-by-1 cell array of character vectors, where N_u is the number of input channels. This property is meaningful only when you are estimating continuous-time models (sample time `Ts` > 0) from discrete-time data.

For each input channel, the possible values of `InterSample` are:

- `'zoh'` — 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 (`pi/sys.Ts` rad/s). This behavior typically occurs when the input signal is measured experimentally using an antialiasing 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.

If you obtain `sysfr` by conversion of a model `sys`, then `InterSample` is equal to the `Intersample` property of the `iddata` object that you used to estimate `sys`.

For more information on this property, see “Effect of Input Intersample Behavior on Continuous-Time Models”.

`IODelay` — Transport delays
`0` (default) | numeric array

Transport delays, specified as a numeric array containing a separate transport delay for each input-output pair.

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 delays of a multiple of the sample time `Ts`.

For a MIMO system with N_y outputs and N_u inputs, set `IODelay` as an N_y-by-N_u 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.

`InputDelay` — Input delay for each input channel
`0` (default) | scalar | vector

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, setting `InputDelay` to `3` specifies a delay of three sample times.

For a system with N_u inputs, set `InputDelay` to an N_u-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.

`OutputDelay` — Output delay for each output channel
`0` (default)

For identified systems such as `idfrd`, `OutputDelay` is fixed to zero.

`Ts` — Sample time
`1` (default) | `0` | positive scalar | -1

Sample time, specified as one of the following.

- Discrete-time model with a specified sampling time — a positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model
- Continuous-time model — `0`
- Discrete-time model with an unspecified sample time — `-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.

TimeUnit — Units for time variable
'seconds' (default) | 'nanoseconds' | 'microseconds' | 'milliseconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'months' | 'years'

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 does not resample or convert the data. Modifying the property changes only the interpretation of the existing data. Use `chgTimeUnit` to convert data to different time units

InputName — Input channel names
'' (default) | character vector | cell array

Input channel names, specified as a character vector or cell array.

- Single-input model — Character vector. For example, `'controls'`.
- Multi-input model — Cell array of character vectors.

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`.

You can use input channel names in several ways, including:

- To identify channels on model display and plots
- To extract subsystems of MIMO systems
- To specify connection points when interconnecting models

InputUnit — Input channel units
'' (default) | character vector | cell array

Input channel units, specified as a character vector or cell array:

- Single-input model — Character vector
- Multi-input Model — Cell array of character vectors

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

`InputGroup` — Input channel groups
`structure` with no fields (default) | `structure`

Input channel groups, specified as a structure. The `InputGroup` property lets you divide the input channels of MIMO systems into groups so that you can refer to each group by name. In the `InputGroup` structure, set field names to the group names, and field values to the input channels belonging to each group.

For example, create input groups named `controls` and `noise` that include input channels 1 and 2 and channels 3 and 5, respectively.

```
sys.InputGroup.controls = [1 2];
sys.InputGroup.noise = [3 5];
```

You can then extract the subsystem from the `controls` inputs to all outputs using the following syntax:

```
sys(:,'controls')
```

`OutputName` — Output channel names
`''` (default) | character vector | cell array

Output channel names, specified as a character vector or cell array.

- Single-input model — Character vector. For example, `'measurements'`
- Multi-input model — Cell array of character vectors

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`.

You can use output channel names in several ways, including:

- To identify channels on model display and plots
- To extract subsystems of MIMO systems
- To specify connection points when interconnecting models

`OutputUnit` — Output channel units
`''` (default) | character vector | cell array

Output channel units, specified as a character vector or cell array.

- Single-input model — Character vector. For example, `'seconds'`.
- Multi-input model — Cell array of character vectors.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

`OutputGroup` — Output channel groups
`structure` with no fields (default) | `structure`

Output channel groups, specified as a structure. The `OutputGroup` property lets you divide the output channels of MIMO systems into groups and refer to each group by name. In the `OutputGroup` structure, set field names to the group names, and field values to the output channels belonging to each group.

For example, create output groups named `temperature` and `measurement` that include output channel 1, and channels 3 and 5, respectively.

```
sys.OutputGroup.temperature = [1];
sys.OutputGroup.measurement = [3 5];
```

You can then extract the subsystem from all inputs to the `measurement` outputs using the following syntax.

```
sys('measurement',:)
```

Name — System name
`''` (default) | character vector

System name, specified as a character vector. For example, `'system_1'`.

Notes — Notes on system
`0`-by-1 string (default) | string | character vector

Any text that you want to associate with the system, specified as a string or a cell array of character vectors. The property stores whichever data type you provide. For instance, if `sys1` and `sys2` are dynamic system models, you can set their `Notes` properties as follows.

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

`UserData` — Data to associate with system
`[]` (default) | any MATLAB data type

Data to associate with the system, specified as any MATLAB data type.

SamplingGrid — Sampling grid

[] (default) | struct

Sampling grid for model arrays, specified as a structure.

For arrays of identified linear (IDLTI) models that you derive by sampling one or more independent variables, this property tracks the variable values associated with each model. This information appears when you show 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 must be numeric and scalar valued, and all arrays of sampled values must match the dimensions of the model array.

For example, suppose that 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)
```

Here, `sys` is an array containing three identified models obtained at 1000, 5000, and 10,000 rpm, respectively.

For model arrays that you generate 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.

Report — Summary report

report field values

This property is read-only.

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

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);
sysfr = idfrd(response,f,0.08);
sysfr.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;
sysfr = spa(z3);sysfr.Report.Method

ans =

SPA
```

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”.

Object Functions

Many functions applicable to “Dynamic System Models” are also applicable to an `idfrd` model object. These functions are of three general types.

- Functions that operate on and return `idfrd` model objects enable you to convert and manipulate `idfrd` models.
- Functions that perform analytical and simulation functions on `idfrd` objects, such as `bode` and `sim`
- Functions that retrieve or interpret model information, such as `getcov`

Unlike other identified linear models, you cannot directly convert an `idfrd` model into another model type using commands such as `idss` or `idtf`. Instead, use the estimation command for the model you want, using the `idfrd` object as the estimation data. For instance, use `sys = ssest(sysfr,2)` to estimate a second-order state-space model from the frequency response data in `idfrd` model `sysfr`. For an example of using an `idfrd` object as estimation data, see “Estimate Time-Domain Model Using Frequency Response Data” on page 1-553.

The following lists contain a representative subset of the functions that you can use with `idss` models.

Transformation and Manipulation

chgTimeUnit	Change time units of dynamic system
chgFreqUnit	Change frequency units of frequency-response data model
fselect	Select frequency points or range in FRD model
frdata	Access data for frequency response data (FRD) object
fcat	Concatenate FRD models along frequency dimension

Analysis and Simulation

bode	Bode plot of frequency response, or magnitude and phase data
spectrum	Plot or return output power spectrum of time series model or disturbance spectrum of linear input-output model

Information Extraction and Interpretation

get	Access model property values
getcov	Parameter covariance of identified model

Examples

Create `idfrd` Object from Frequency Response Data

Create an `idfrd` object from frequency response data.

Load the magnitude data `AMP`, the phase data `PHA`, and the frequency vector `W`. Set sample time `Ts` to 0.1.

```
load demofr AMP PHA W
Ts = 0.1;
```

Use the values of `AMP` and `PHA` to compute the complex-valued response `response`.

```
response = AMP.*exp(1j*PHA*pi/180);
```

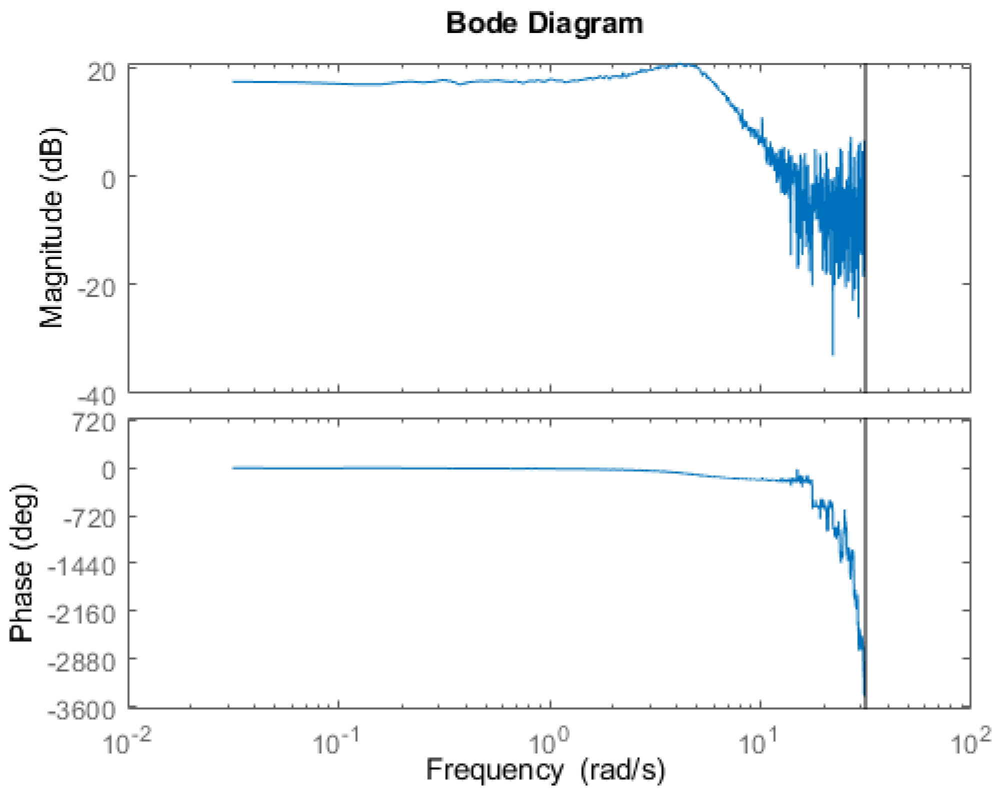
Create an `idfrd object` to store `response` in the `idfrd` object `frdata`.

```
frdata = idfrd(response,W,Ts)
```

```
frdata =
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s).
Response data is available at 1000 frequency points, ranging from 0.03142 rad/s to 31.42 rad/s.

Sample time: 0.1 seconds
Status:
Created by direct construction or transformation. Not estimated.
```

Plot the data.

```
bode(frdata)
```



`frdata` is a complex `idfrd` object with object properties that you can access using dot notation. For example, confirm the value of `Ts`.

```
tsproperty = frdata.Ts
```

```
tsproperty = 0.1000
```

You can also set property values. Set the `Name` property to `'DC_Converter'`.

```
frdata.Name = 'DC_Converter';
```

If you import `frdata` into the `System Identification` app, the app names this data `DC_Converter`, and not the variable name `frdata`.

Use `get` to obtain the full set of property settings.

```
get(frdata)
```

```
      FrequencyUnit: 'rad/TimeUnit'
             Report: [1x1 idresults.frdest]
       SpectrumData: []
     CovarianceData: []
    NoiseCovariance: []
        InterSample: {'zoh'}
       ResponseData: [1x1x1000 double]
            IODelay: 0
         InputDelay: 0
        OutputDelay: 0
```

```
              Ts: 0.1000
        TimeUnit: 'seconds'
       InputName: {''}
       InputUnit: {''}
      InputGroup: [1x1 struct]
      OutputName: {''}
      OutputUnit: {''}
     OutputGroup: [1x1 struct]
           Notes: [0x1 string]
        UserData: []
            Name: 'DC_Converter'
    SamplingGrid: [1x1 struct]
       Frequency: [1000x1 double]
```

Convert Time-Domain Model to Frequency Response Model

Convert a state-space model to a frequency response model using the `idfrd` command.

Load the data `z2` and estimate a second-order state-space model `sys`.

```
load iddata2 z2
sys = ssest(z2,2);
```

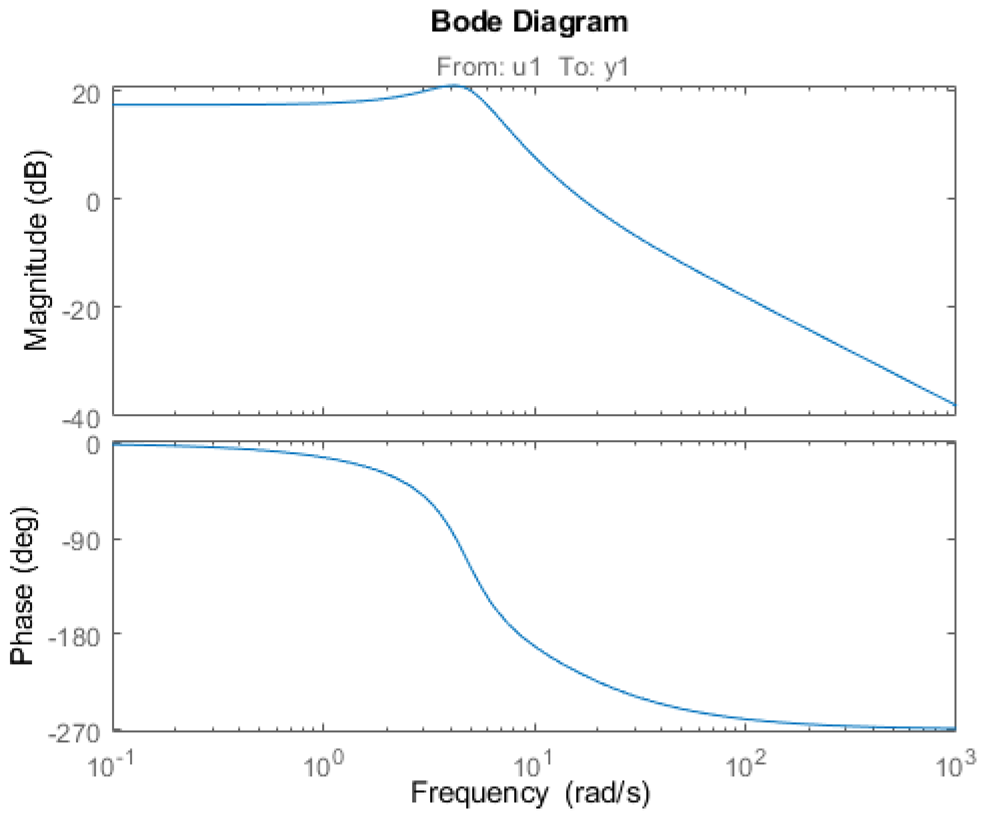
Convert `sys` to the `idfrd` model `frsys`.

```
frsys = idfrd(sys)
```

```
frsys =
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the spectra for disturbances
Response data and disturbance spectra are available at 68 frequency points, ranging from 0.1 rad/

Output channels: 'y1'
Input channels: 'u1'
Status:
Created by conversion from idss model.
```

Plot `frsys`.

```
bode(frsys)
```



`frsys` is an `idfrd` model that you can use as a dynamic system model or as estimation data for a time-domain or frequency-domain model.

Create `idfrd` Object from Frequency Response of Time-Domain Model

Obtain the frequency response of a transfer function model and convert the response into an `idfrd` object.

Construct a transfer function model with one zero and three poles.

```
systf = idtf([1 .2],[1 2 1 1]);
```

Use `bode` to obtain the frequency response of `systf`, in terms of magnitude and phase, for the frequency vector `f`.

```
f = logspace(-1,1,100);
[mag,phase] = bode(systf,f);
```

Use the values of `mag` and `phase` to compute the complex-valued response `response`.

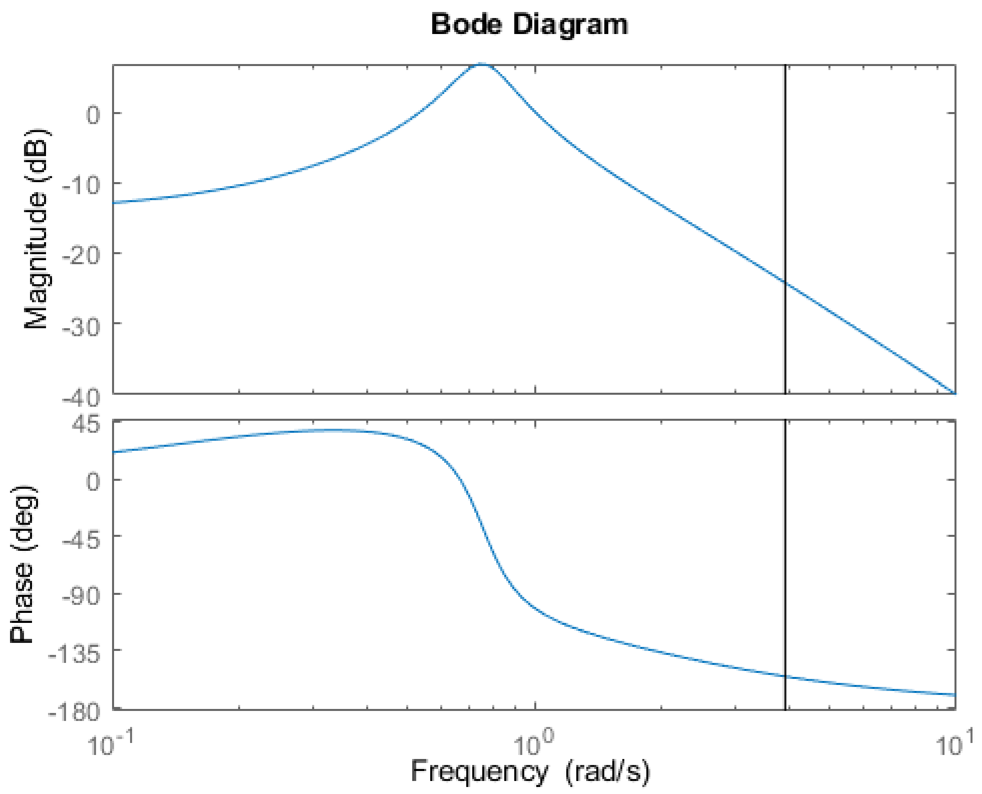
```
response = mag.*exp(1j*phase*pi/180);
```

Create an `idfrd` object `frdata` to store `response`, specifying a sample rate `Ts` of 0.8.

```
Ts = 0.8;
frdata = idfrd(response,f,Ts)

frdata =
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s).
Response data is available at 100 frequency points, ranging from 0.1 rad/s to 10 rad/s.

Sample time: 0.8 seconds
Status:
Created by direct construction or transformation. Not estimated.
```

Plot the data.

```
bode(frdata)
```



`frdata` is a complex `idfrd` object.

Estimate Time-Domain Model Using Frequency Response Data

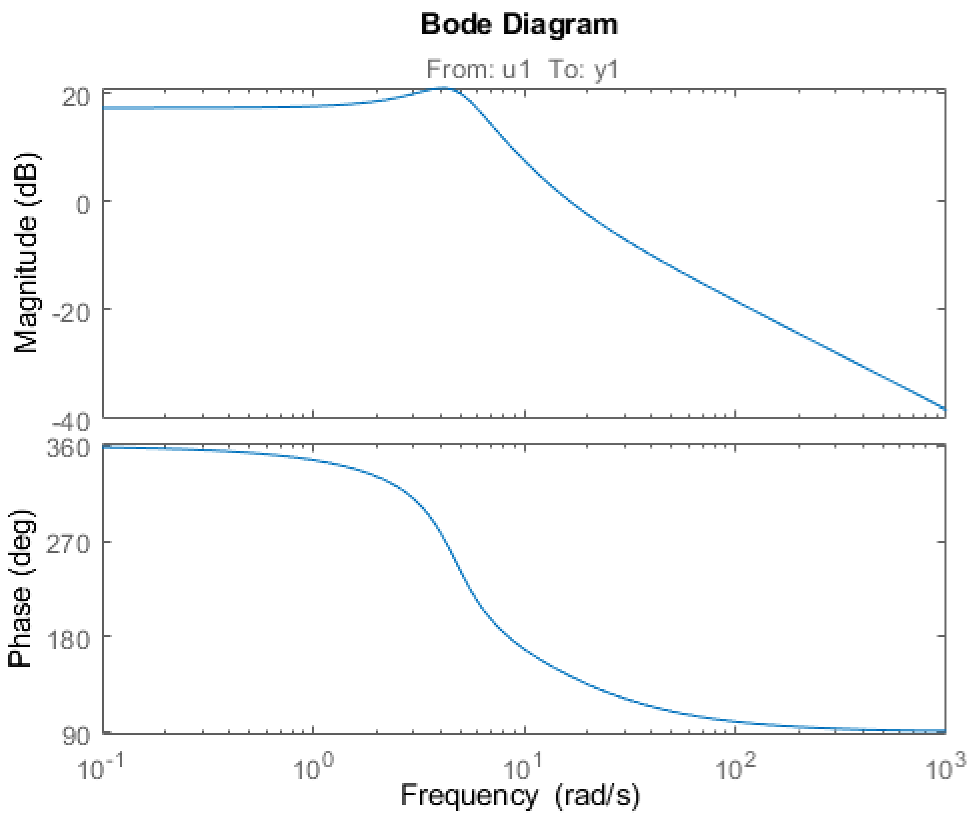
Estimate a transfer function model from time-domain data and convert the resulting `idtf` model to an `idfrd` model. Estimate a new transfer function model from the frequency response data in the `idfrd` model. Compare the model responses with the original data.

Load time-domain data z2 and use it to estimate a transfer function sys that has two poles and one zero.

```
load iddata2 z2
sys = tfest(z2,2,1);
```

Convert sys to an idfrd model and plot the frequency response.

```
frsys = idfrd(sys);
bode(sys)
```

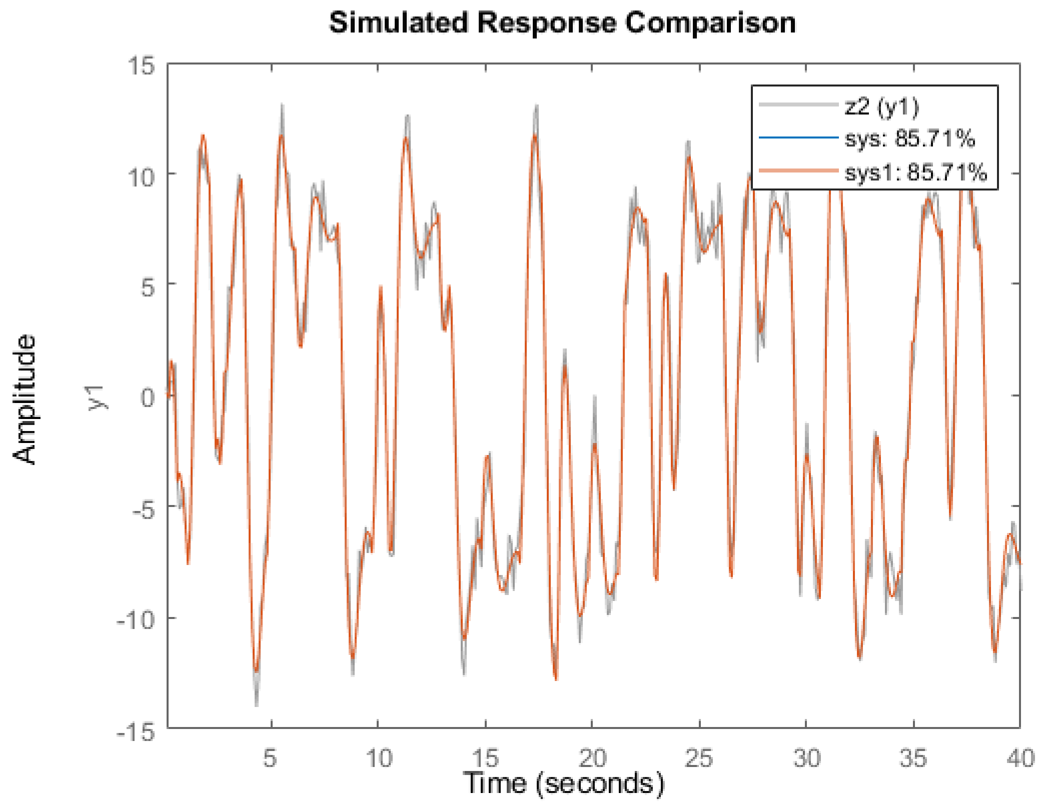


Estimate a new transfer function sys1 using the data from frsys as the estimation data.

```
sys1 = tfest(frsys,2,1);
```

Compare the responses of sys and sys1 with the original estimation data z2.

```
compare(z2,sys,sys1)
```



The model responses are identical.

See Also

`bode` | `etfe` | `freqresp` | `nyquist` | `spa` | `spafdr` | `tfest`

Topics

"Representing Frequency-Response Data Using idfrd Objects"
"Estimating Models Using Frequency-Domain Data"
"Frequency Domain Identification: Estimating Models Using Frequency Domain Data"

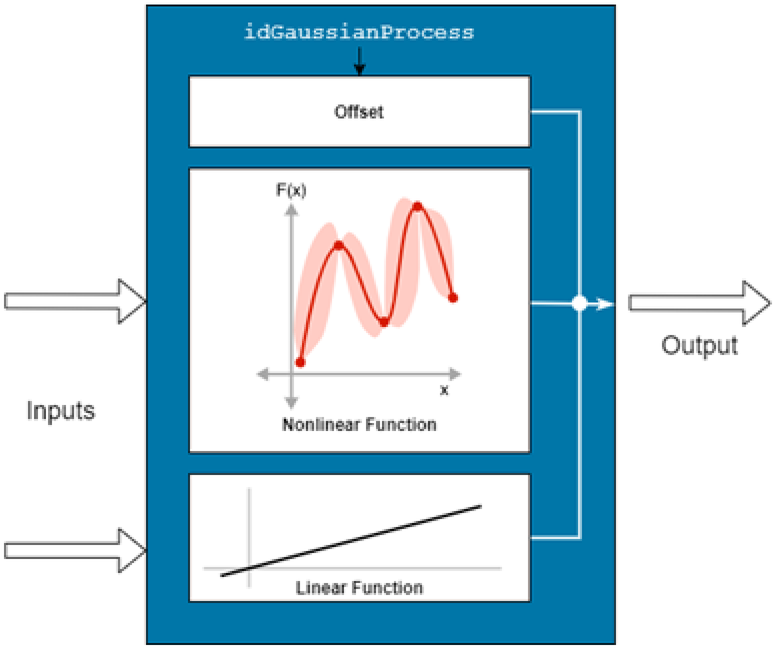
Introduced before R2006a

idGaussianProcess

Gaussian process regression mapping function for nonlinear ARX models (requires Statistics and Machine Learning Toolbox)

Description

An `idGaussianProcess` object implements a Gaussian process regression model, and is a nonlinear mapping function for estimating nonlinear ARX models. This mapping object, which is also referred to as a nonlinearity, incorporates `RegressionGP` objects that the mapping function creates using Statistics and Machine Learning Toolbox™. The mapping object contains three components: a linear component that uses a combination of linear weights, an offset, and a nonlinear component.



Mathematically, `idGaussianProcess` is a function that maps m inputs $X(t) = [x(t_1),x_2(t),\dots,x_m(t)]^T$ to a scalar output $y(t)$ using the following relationship:

$$y(t) = y_0 + (X(t) - \bar{X})^T PL + G(X(t), \theta))$$

Here,

- $X(t)$ is an m-by-1 vector of inputs, or regressors, with mean $\bar{X}$.
- y_0 is the output offset, a scalar.
- P is an m-by-p projection matrix, where m is the number of regressors and p is the number of linear weights. m must be greater than or equal to p.
- L is a p-by-1 vector of weights.
- $G(X,\theta)$ is the regressive Gaussian process that constitutes the nonlinear component of the `idGaussianProcess` object. G has a mean of zero and a covariance that the user specifies by choosing a kernel, and can be expressed generally as

$$G(X) = GP(0, K(X_{test}, X_{train}, \theta))$$

The Gaussian process G can be defined more precisely as the predicted value of a test input for a given value of θ. In this case, the relationship becomes:

$$G(X) = K(X_{test}, X_{train})[K(X_{train}, X_{train}) + \sigma_n 2I]^{-1} Y_{train}$$

Here:

- $K(X_{test}, X_{train})$ is the covariance kernel function.
- X_{train} is a matrix representing the set of training inputs.
- X_{test} is a matrix representing the set of test inputs.
- Y_{train} is the vector of outputs from the training set.
- σ_n is the standard deviation of the additive measurement noise.

For more information about creating Gaussian process regression models, see `fitrgp`.

Use `idGaussianProcess` as the value of the `OutputFcn` property of an `idnlarx` model. For example, specify `idGaussianProcess` when you estimate an `idnlarx` model with the following command.

```
sys = nlarx(data,regressors,idGaussianProcess)
```

When `nlarx` estimates the model, it essentially estimates the parameters of the `idGaussianProcess` function.

You can configure the `idGaussianProcess` function to disable components and fix parameters. To omit the linear component, set `LinearFcn.Use` to `false`. To omit the offset, set `Offset.Use` to `false`. To specify known values for the linear function and the offset, set their `Value` attributes directly and set the corresponding `Free` attributes to `False`. To modify the estimation options, set the option property in `EstimationOptions`. For example, to change the fit method to `'exact'`, use `G.EstimationOptions.FitMethod = 'exact'`. Use `evaluate` to compute the output of the function for a given vector of inputs.

Creation

Syntax

```
G = idGaussianProcess
G = idGaussianProcess(kernel)
G = idGaussianProcess(kernel,kernelParameters)
G = idGaussianProcess(kernel,kernelParameters,UseLinearFcn)
G = idGaussianProcess(kernel,kernelParameters,UseLinearFcn,UseOffset)
```

Description

`G = idGaussianProcess` creates an `idGaussianProcess` object `G` with the kernel function `'SquaredExponential'` and default kernel parameters. The number of inputs is determined during model estimation and the number of outputs is 1.

`G = idGaussianProcess(kernel)` specifies a specific kernel.

`G = idGaussianProcess(kernel,kernelParameters)` initializes the parameters of the specified kernel to the values in `kernelParameters`.

`G = idGaussianProcess(kernel,kernelParameters,UseLinearFcn)` specifies whether the function uses a linear function as a subcomponent.

`G = idGaussianProcess(kernel,kernelParameters,UseLinearFcn,UseOffset)` specifies whether the function uses an offset term y_0 parameter.

Input Arguments

`kernel` — Kernel covariance function

`'SquaredExponential'` (default) | `'Exponential'` | `'Matern32'` | `'Matern52'` | `'RationalQuadratic'` | `'ARDSquaredExponential'` | `'ARDExponential'` | `'ARDMatern32'` | `'ARDMatern52'` | `'ARDRationalquadratic'`

Kernel covariance function, specified as character array or string. For information about the options, see Kernel (Covariance) Function in `fitrgp`.

This argument sets the `G.NonlinearFcn.KernelFunction` property.

`kernelParameters` — Initial values for kernel parameters

vector

Initial values for the kernel parameters, specified as a vector. The size of the vector and the values depend on the choice of `kernel`. For more information, see Kernel Parameters in `fitrgp`.

This argument sets the `G.NonlinearFcn.Parameters` property.

`UseLinearFcn` — Option to use linear function

`true` (default) | `false`

Option to use the linear function subcomponent, specified as `true` or `false`. This argument sets the value of the `G.LinearFcn.Use` property.

`UseOffset` — Option to use offset term

`true` (default) | `false`

Option to use an offset term, specified as `true` or `false`. This argument sets the value of the `G.Offset.Use` property.

Properties

`Input` — Input signal information

signal property values

Input signal information for signals used for estimation, specified as vectors of *m* property-specific values, where *m* is the number of input signals. The `Input` properties for each input signal are as follows:

- `Name` — Names of the input signals, specified as a 1-by-*m* string or character array, where *m* is the number of inputs
- `Mean` — Mean of the input signals, specified as a numeric scalar

- `Range` — Ranges of the input signals, specified as a 2-by-*m* numeric array that contains the minimum and maximum values

`Output` — Output signal information
signal property values

Output signal information, specified as property-specific values. The `Output` properties are as follows:

- `Name` — Name of the output signal, specified as a string or a character array
- `Mean` — Mean of the output signal, specified as a numeric scalar
- `Range` — Range of the output signal, specified as a 2-by-1 numeric array that contains the minimum and maximum values

`LinearFcn` — Parameters of linear function
linear function property values

Parameters of the linear function, specified as follows:

- `Use` — Option to use the linear function in the `idGaussianProduct` model, specified as a scalar logical. The default value is `true`.
- `Value` — Linear weights that compose *L*', specified as a 1-by-*p* vector.
- `InputProjection` — Input projection matrix *P*, specified as an *m*-by-*p* matrix, that transforms the detrended input vector of length *m* into a vector of length *p*.
- `Free` — Option to update entries of `Value` during estimation, specified as a 1-by-*p* logical vector. The software honors the `Free` specification only if the starting value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a 1-by-*p* vector. If `Minimum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that minimum bound during model estimation.
- `Maximum` — Maximum bound on `Value`, specified as a 1-by-*p* vector. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation.
- `SelectedInputIndex` — Indices of `idGaussianProcess` inputs (see `Input.Name`) that are used as inputs to the linear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. For nonlinear ARX models, the `RegressorUsage` property determines these indices..

`Offset` — Parameters of offset term
offset property values

Parameters of the offset term, specified as follows:

- `Use` — Option to use the offset in the `idGaussianProcess` model, specified as a scalar logical. The default value is `true`.
- `Value` — Offset value, specified as a scalar.
- `Free` — Option to update `Value` during estimation, specified as a scalar logical. The software honors the `Free` specification of `false` only if the value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a numeric scalar or `–Inf`. If `Minimum` is specified with a finite value and the value of `Value` is finite, then the software enforces that minimum bound during model estimation. The default value is `-Inf`.

- `Maximum` — Maximum bound on `Value`, specified as a numeric scalar or `Inf`. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation. The default value is `Inf`.

`NonlinearFcn` — Parameters of nonlinear function
nonlinear function property values

Parameters of the nonlinear function, specified as follows:

- `KernelFunction` — Kernel covariance kernel function, specified as one of the values listed in the `kernel` argument description. For more information about these options, see Kernel (Covariance) Function in `fitrgp`.
- `Parameters` — Parameters of the `idGaussianProcess` kernel function, specified as a vector. The size of the vector and the values depend on the choice of `KernelFunction`. For more information, see Kernel Parameters in `fitrgp`.
- `Free` — Option to estimate parameters, specified as a logical scalar. If all the parameters have finite values, such as when the `idGaussianProcess` object corresponds to a previously estimated model, then setting `Free` to `false` causes the parameters of the nonlinear function $S(X)$ to remain unchanged during estimation. The default value is `true`.
- `SelectedInputIndex` — Indices of `idGaussianProcess` inputs (see `Input.Name`) that are used as inputs to the nonlinear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. For nonlinear ARX models, the `RegressorUsage` property determines these indices.

`Estimation Options` — Estimation options
estimation option property values

Estimation options for the nonlinear block of the `idGaussianProcess` model, specified as follows. For more information on any of these options, see `fitrgp`.

- `FitMethod` — Method to use for estimating the parameters of the `idGaussianProcess` nonlinear model, specified as one of the items in the following table.

Option	Description
`'auto'`	Software selects the method automatically (default)
`'exact'`	Exact Gaussian process regression
`'sd'`	Subset of data points approximation
`'sr'`	Subset of regressors approximation
`'fic'`	Fully independent conditional approximation

- `ActiveSetMethod` — Active set selection method, specified as one of the items in the following table.

Option	Description
`'random'`	Random selection (default)
`'sgma'`	Sparse greedy matrix approximation
`'entropy'`	Differential entropy-based selection
`'likelihood'`	Subset of regressors log likelihood-based selection

- `SparseFitRegularization` — Regularization standard deviation for the sparse methods subset of regressors (`'sr'`) and the fully independent conditional approximation (`'fic'`), specified as a positive scalar value.
- `Optimizer` — Optimizer to use for parameter estimation, specified as one of the items in the following table.

Option	Description
`'quasinewton'`	Dense, symmetric rank-1-based, quasi-Newton approximation to the Hessian (default)
`'lbfgs'`	LBFGS-based quasi-Newton approximation to the Hessian
`'fminsearch'`	Unconstrained nonlinear optimization using the simplex search method of Lagarias et al. [see `fitrgp`]
`'fminunc'`	Unconstrained nonlinear optimization (requires an Optimization Toolbox license)
`'fmincon'`	Constrained nonlinear optimization (requires an Optimization Toolbox license)

- `OptimizerOptions` — Options for the optimizer, specified as a structure or object. When `Optimizer` is set or changed, the software automatically updates the value of `OptimizerOptions` to match the defaults for the corresponding optimizer. Use the properties of the `OptimizerOptions` option set to change the values from their defaults.

Examples

Estimate Nonlinear ARX Model with `idGaussianProcess` as Output Function

Load the input/output data from `twotankdata` and construct an `iddata` object `z`.

```
load twotankdata u y
z = iddata(y,u,0.8,'timeunit','hours');
```

Create an `idGaussianProduct` mapping object `g` that uses a Matern kernel with the parameter 3/2.

```
g = idGaussianProcess('Matern32');
```

Estimate a nonlinear ARX model that uses `g` as the output function.

```
sys = nlarx(z,[4 4 1],g)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Gaussian process function using a Matern32 kernel.
Sample time: 0.8 hours

Status:
```

```
Estimated using NLARX on time domain data "z".
Fit to estimation data: 97.08% (prediction focus)
FPE: 2.945e-05, MSE: 2.92e-05
```

Display the postestimation properties of `g`.

```
disp(sys.OutputFcn.Input)
```

```
Function inputs

     Name: {1x8 cell}
     Mean: [0.2700 0.2699 0.2697 0.2696 6.4286 6.4286 6.4286 6.4286]
    Range: [2x8 double]
```

```
disp(sys.outputFcn.Offset)
```

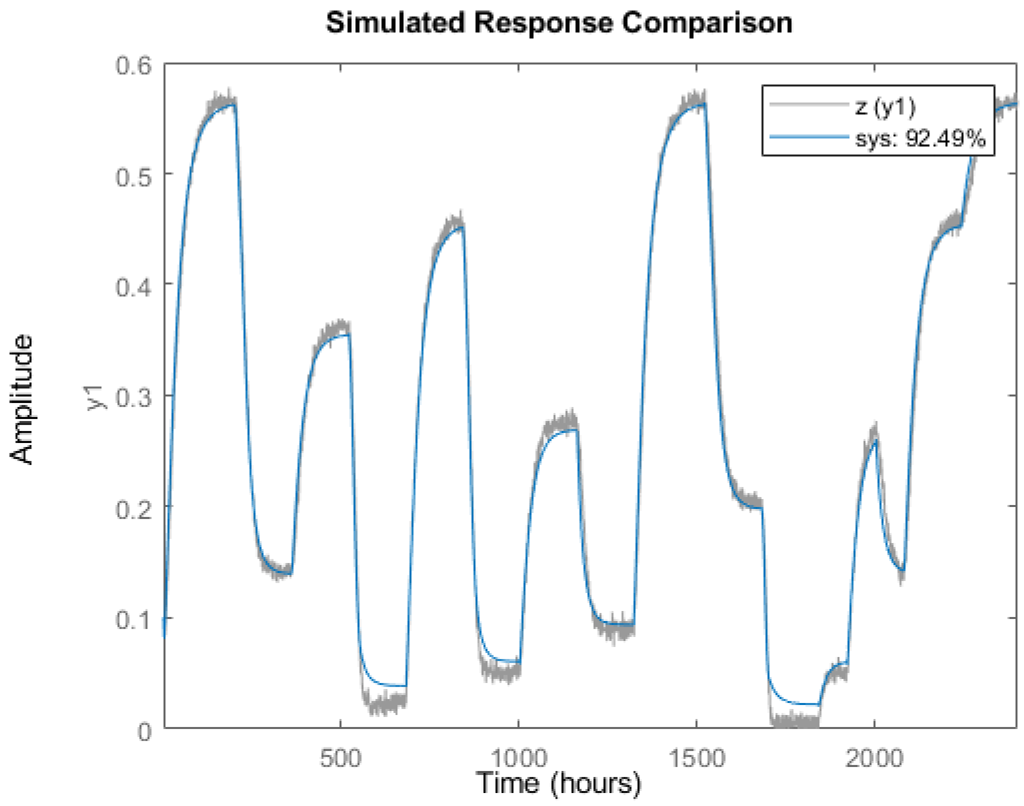
```
Output Offset: initialized to 0.27
      Use: 1
    Value: 0.2702
     Free: 1
```

```
disp(sys.outputFcn.NonlinearFcn)
```

```
GP kernel and its parameters

        KernelFunction: 'Matern32'
            Parameters: '<Kernel parameters>'
                  Free: 1
    SelectedInputIndex: [1 2 3 4 5 6 7 8]
```

Compare the output of `sys` with the measured output `z`.

```
compare(z,sys)
```



The nonlinear model shows a good fit to the estimation data.

See Also

`nlarx` | `RegressionGP` | `fitrgp` | `idLinear` | `idTreePartition` | `idWaveletNetwork` | `idSigmoidNetwork` | `idFeedforwardNetwork` | `idCustomNetwork` | `idnlarx` | `evaluate`

Topics

“Gaussian Process Regression Models” (Statistics and Machine Learning Toolbox)

Introduced in R2021b

idgrey

Linear ODE (grey-box model) with identifiable parameters

Description

An `idgrey` model represents a linear system as a continuous-time or discrete-time state-space model with identifiable (estimable) coefficients. Use an `idgrey` model when you want to capture complex relationships, constraints, and prior knowledge that structured state-space (`idss`) models cannot encapsulate. To create an `idgrey` model, you must know explicitly the system of equations (ordinary differential or difference equations) that govern the system dynamics.

An `idgrey` model allows you to incorporate conditions such as the following:

- Parameter constraints that the `idss`/`ssest` framework cannot handle, such as linear or equality constraints on parameters, or prior knowledge about the variance of the states, inputs, outputs, or any combination of the three, that you want to include as known information
- A linear model of an arbitrary form, such as a transfer function or polynomial model, with parameter constraints such as a known DC gain, limits on pole locations, a shared denominator across multiple inputs, or nonzero input/output delays in MIMO models
- Differential or difference equations with known and unknown coefficients

In these and similar cases, you can create an ODE (ordinary differential or difference equation) function in MATLAB that implements a state-space realization of the linear model and that specifies constraints and prior knowledge.

A simple example of creating an ODE for `idgrey` uses the following equations to describe motor dynamics.

$$\begin{aligned}\dot{x}(t) &= \begin{bmatrix} 0 & 1 \\ 0 & -\frac{1}{\tau} \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \frac{G}{\tau} \end{bmatrix} u(t) \\ y(t) &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t)\end{aligned}$$

In these equations, τ is the single estimable parameter and G represents the known static gain.

These equations fit the state-space form:

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t)\end{aligned}$$

For this case, both the A and B matrices contain the estimable parameter τ, and B also includes the known gain G. You can write a MATLAB function that accepts τ and G as input arguments and returns the state-space matrices A, B, and C as its output arguments. For example, you can code a function `motorFcn` as follows.

```
function [A,B,C] = motorFcn(tau,G)
% ODE function for computing state-space matrices as functions of parameters
A = [0 1; 0 -1/tau];
```

```
B = [0; G/tau];
C = eye(2);
```

After creating a function such as `motorFcn`, create an `idgrey` model by specifying that function as the value of its `odefun` input argument, as the following command shows.

```
sys = idgrey(@motorFcn,tau0,'c',G)
```

Here, `tau0` is the initial guess for the parameter τ and `G` specifies the fixed constant. Additionally, `'c'` indicates to `idgrey` that `odefun` returns matrices corresponding to a continuous-time system. For more information, see `function_type`.

For an executable example that creates an `idgrey` model from these motor dynamics equations, see “Create Grey-Box Model with Estimable Parameters” on page 1-576.

More generally, the following equations describe state-space forms for continuous-time and discrete-time systems.

A state-space model of a system with input vector *u*, output vector *y*, and disturbance *e*, takes the following form in continuous time:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$
$$x(0) = x0$$

In discrete time, the state-space model takes the form:

$$x[k+1] = Ax[k] + Bu[k] + Ke[k]$$
$$y[k] = Cx[k] + Du[k] + e[k]$$
$$x[1] = x0$$

Your MATLAB ODE function incorporates the user-defined parameters into the *A*, *B*, *C*, and *D* matrices that the function returns. The associated `idgrey` model references this function, and the estimation functions `greyest` and `pem` use these matrix definitions when estimating the parameters.

For more information on creating an ODE function for `idgrey`, see “Estimate Linear Grey-Box Models”.

Creation

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. You can pass additional input arguments, such as a time constant or gain, that are not parameters but that the ODE uses in the expressions for the output arguments.

In addition to the *A*, *B*, *C*, and *D* matrices, your MATLAB function can return the *K* matrix if you want the *K* values to be functions of your input parameters. Your function can also return the initial state vector *x0*. However, the alternative and recommended approach for parameterizing *x0* is to use the `InitialState` estimation option of `greyestOptions`.

Note that you can write your ODE function to represent either the continuous time dynamics or the discrete-time dynamics regardless of the nature of the `idgrey` model itself. For example, you can specify a discrete-time `idgrey` model (`sys.Ts>0`) that uses a continuous-time parameterization of the ODE function. Similarly, you can specify a discrete-time parameterization of the ODE function and use it with a continuous-time idgrey model (`sys.Ts=0`). The `idgrey` input argument `fcn_type` informs the idgrey model what type of parameterization the ODE function uses. For more information, see “Estimate Linear Grey-Box Models”.

Use the estimating functions `pem` or `greyest` to obtain estimated values for the unknown parameters of an `idgrey` model. Unlike other estimation functions such as `ssest`, which can create a new model object, `greyest` can estimate parameters only for an `idgrey` model that already exists and is specified as an input argument. You can access estimated parameters using `sys.Structures.Parameters`, where `sys` is an `idgrey` model.

You can convert an `idgrey` model into other dynamic systems, such as `idpoly`, `idss`, `tf`, or `ss`. You cannot convert a dynamic system into an `idgrey` model.

Syntax

```
sys = idgrey(odefun,parameters,fcn_type)
sys = idgrey(odefun,parameters,fcn_type,extra_args)
sys = idgrey(odefun,parameters,fcn_type,extra_args,Ts)
sys = idgrey(odefun,parameters,fcn_type,extra_args,Ts,Name,Value)
```

Description

`sys = idgrey(odefun,parameters,fcn_type)` creates a linear grey-box model `sys` with identifiable parameters. `odefun` specifies the user-defined function that relates the model parameters `parameters` to their state-space representation. `fcn_type` specifies whether the model is parameterized in continuous-time, discrete-time, or both.

`sys = idgrey(odefun,parameters,fcn_type,extra_args)` specifies additional arguments `extra_args` that `odefun` requires.

`sys = idgrey(odefun,parameters,fcn_type,extra_args,Ts)` specifies the sample time `Ts`.

`sys = idgrey(odefun,parameters,fcn_type,extra_args,Ts,Name,Value)` incorporates additional options specified by one or more name-value arguments.

Input Arguments

`odefun` — MATLAB function

function handle | character array | string

MATLAB function (.m, .p, or .mex* file) that relates the model parameters `parameters` to their state-space representation, specified as a function handle or as a character array or string that contains the name of the function. As an option, `odefun` can also relate the model parameters to the disturbance matrix and initial states. For information about creating the ODE function, see “Estimate Linear Grey-Box Models”. The parameters that the ODE function defines are the same parameters that you specify in the `parameters` input argument to `idgrey`.

If `odefun` is not on the MATLAB path, then specify the full file name, including the path.

If `odefun` does not return the disturbance matrix *K* and the initial state values *x0*, then these values are not estimable parameters in the `idgrey` object. Instead, during estimation, the software determines these values using the `DisturbanceModel` and `InitialState` estimation options, respectively. You can fix the value of *K* to zero by setting the `DisturbanceModel` option to `'none'`.

Doing so generally provides the best match between the simulation results and the measured data. For more information about the *K* values, see `K`. For more information about the estimation options, see `greyestOptions`.

The `idgrey` model stores the ODE function name or handle in the `sys.Structures.Function` property.

For more information on creating an ODE function, see “Estimate Linear Grey-Box Models”.

`parameters` — Initial values of parameters
cell array | matrix

Initial values of the parameters required by `odefun`, specified as a cell array or a matrix:

- If your model requires multiple parameters, `parameters` must be a cell array.
- If your model requires only a single parameter, which itself might be a vector or a matrix, `parameters` can be a matrix.

You can 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 instance, the following command specifies parameters named `'mass'`, `'stiffness'`, and `'damping'`.

```
parameters = {'mass',par1;'stiffness',par2;'damping',par3}
```

For an example of configuring parameters, see “Configure Estimable Parameter of Grey-Box Model” on page 1-577.

The `idgrey` model stores the estimated parameters in the `sys.Structures.Parameters` property.

`fcn_type` — Function type
`'c'` | `'d'` | `'cd'`

Function type that indicates whether the model is parameterized in continuous-time, discrete-time, or both, specified as a character array or string that contains 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 might or might not depend on the value of `Ts`.
- `'cd'` — `odefun` returns matrices corresponding to a continuous-time system if `Ts` = 0 or a discrete time system if `Ts` > 0.

  If `Ts` > 0, select `'cd'` rather than `'d'` when you want the software to sample your model using the values returned by `odefun` rather using the software’s internal sample time conversion routines.

For an example of setting this argument, see “Create Grey-Box Model with Estimable Parameters” on page 1-576.

The `idgrey` model stores the function type in the `sys.Structures.FunctionType` property.

`extra_args` — Extra arguments
`{}` (default) | cell array

Extra input arguments that are required by `odefun`, specified as a cell array. If `odefun` does not require extra input arguments, specify `extra_args` as `{}`.

For an example of using this argument, see “Create Grey-Box Model with Estimable Parameters” on page 1-576.

Properties

`A,B,C,D` — Values of state-space matrices
matrices

This property is read-only.

Values of the state-space matrices that the ODE function represented by `odefun` returns, specified as the following:

- `A` — State matrix *A*, an *Nx*-by-*Nx* matrix, where *Nx* is the number of states.
- `B` — Input-to-state matrix *B*, an *Nx*-by-*Nu* matrix, where *Nu* is the number of inputs.
- `C` — State-to-output matrix *C*, an *Ny*-by-*Nx* matrix, where *Ny* is the number of outputs.
- `D` — Feedthrough matrix *D*, an *Ny*-by-*Nu* matrix.

For an example of this property, see “Create Grey-Box Model with Estimable Parameters” on page 1-576.

`K` — Value of state disturbance matrix K
matrix

Values of the state disturbance matrix *K*, specified as an *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. The zero value is treated as a fixed value of the *K* matrix during estimation. To make the value of *K* estimable, use the `DisturbanceModel` estimation option.
- Regardless of whether the *K* matrix is parameterized by `odefun` or not, you can set the values of the `K` property explicitly. 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
`{''}` (default) | character vector | cell array of character vectors

State names, specified as one of these values:

- Character vector — For first-order models
- Cell array of character vectors — For models with two or more states

- `''` — For unnamed states

You can specify `StateName` using a string, such as `"velocity"`, but the state name is stored as a character vector, `'velocity'`.

Example: `'velocity'`

Example: `{'x1','x2'}`

`StateUnit` — State units
`{''}` (default) | character vector | cell array of character vectors

State units, specified as one of these values:

- Character vector — For first-order models
- Cell array of character vectors — For models with two or more states
- `''` — For states without specified units

Use `StateUnit` to keep track of the units each state is expressed in. `StateUnit` has no effect on system behavior.

You can specify `StateUnit` using a string, such as `"mph"`, but the state units are stored as a character vector, `'mph'`.

Example: `'mph'`

Example: `{'rpm','rad/s'}`

`Structure` — Information about estimable parameters
`LinearODE` structure

Information about the estimable parameters of the `idgrey` model, specified as a `LinearODE` structure.

- `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` — Extra input arguments the ODE function requires.
- `Structure.StateName` — Names of the model states.
- `Structure.StateUnit` — Units of the model states.

`Noise Variance` — Noise variance of model innovations
scalar | matrix

Noise variance of the model innovations *e*, specified as a scalar or a covariance matrix. For SISO models, `NoiseVariance` is a scalar. For MIMO models, `NoiseVariance` is an *Ny*-by-*Ny* matrix, where *Ny* is the number of outputs in the system.

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.

`Report` — Summary report
report field values

This property is read-only.

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'
  EstimateCovariance: 1
```

```
             Display: 'off'
         InputOffset: []
        OutputOffset: []
      Regularization: [1x1 struct]
        OutputWeight: []
        SearchMethod: 'auto'
       SearchOptions: [1x1 idoptions.search.identsolver]
            Advanced: [1x1 struct]
```

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 — Delay at inputs
0 (default) | scalar | vector

Delay at each input, specified as a scalar or a vector. 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. For continuous-time models, specify input delays in the time unit stored in the `TimeUnit` property of the model object. For discrete-time models, specify input delays in integer multiples of the sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

Set `InputDelay` to a scalar value to apply the same delay to all channels.

OutputDelay — Output delays
0 (default)

For identified systems like `idgrey`, `OutputDelay` is fixed to zero.

Ts — Sample time
`0` | `-1` | positive scalar

Sample time, specified as one of the following.

- Continuous-time model — `0`
- Discrete-time model with a specified sampling time — Positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model
- Discrete-time model with unspecified sample time — `-1`

For `idgrey` models, `Ts` has no unique default value. `Ts` depends on the value of `fcn_type`.

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.

TimeUnit — Model time units
`'seconds'` (default) | `'minutes'` | `'milliseconds'` | ...

Model time units, specified as one of these values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`

- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

You can specify TimeUnit using a string, such as "hours", but the time units are stored as a character vector, 'hours'.

Model properties such as sample time Ts, InputDelay, OutputDelay, and other time delays are expressed in the units specified by TimeUnit. 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.

InputName — Names of input channels
{''} (default) | character vector | cell array of character vectors

Names of input channels, specified as one of these values:

- Character vector — For single-input models
- Cell array of character vectors — For models with two or more inputs
- '' — For inputs without specified names

You can 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)'}.

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

You can specify InputName using a string, such as "voltage", but the input name is stored as a character vector, 'voltage'.

When you estimate a model using an iddata object, data, the software automatically sets InputName to data.InputName.

InputUnit — Units of input signals
{''} (default) | character vector | cell array of character vectors

Units of input signals, specified as one of these values:

- Character vector — For single-input models
- Cell array of character vectors — For models with two or more inputs
- `''` — For inputs without specified units

Use `InputUnit` to keep track of the units each input signal is expressed in. `InputUnit` has no effect on system behavior.

You can specify `InputUnit` using a string, such as `"voltage"`, but the input units are stored as a character vector, `'voltage'`.

Example: `'voltage'`

Example: `{'voltage','rpm'}`

`InputGroup` — Input channel groups
structure with no fields (default) | structure

Input channel groups, specified as a structure where the fields are the group names and the values are the indices of the input channels belonging to the corresponding group. When you use `InputGroup` to assign the input channels of MIMO systems to groups, you can refer to each group by name when you need to access it. For example, suppose you have a five-input model `sys`, where the first three inputs are control inputs and the remaining two inputs represent noise. Assign the control and noise inputs of `sys` to separate groups.

```
sys.InputGroup.controls = [1:3];
sys.InputGroup.noise = [4 5];
```

Use the group name to extract the subsystem from the control inputs to all outputs.

```
sys(:,'controls')
```

Example: `struct('controls',[1:3],'noise',[4 5])`

`OutputName` — Names of output channels
`{''}` (default) | character vector | cell array of character vectors

Names of output channels, specified as one of these values:

- Character vector — For single-output models
- Cell array of character vectors — For models with two or more outputs
- `''` — For outputs without specified names

You can 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)'}`.

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

You can specify `OutputName` using a string, such as `"rpm"`, but the output name is stored as a character vector, `'rpm'`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

`OutputUnit` — Units of output signals
`{''}` (default) | character vector | cell array of character vectors

Units of output signals, specified as one of these values:

- Character vector — For single-output models
- Cell array of character vectors — For models with two or more outputs
- `''` — For outputs without specified units

Use `OutputUnit` to keep track of the units each output signal is expressed in. `OutputUnit` has no effect on system behavior.

You can specify `OutputUnit` using a string, such as `"voltage"`, but the output units are stored as a character vector, `'voltage'`.

Example: `'voltage'`

Example: `{'voltage','rpm'}`

`OutputGroup` — Output channel groups
structure with no fields (default) | structure

Output channel groups, specified as a structure where the fields are the group names and the values are the indices of the output channels belonging to the corresponding group. When you use `OutputGroup` to assign the output channels of MIMO systems to groups, you can refer to each group by name when you need to access it. For example, suppose you have a four-output model `sys`, where the second output is a temperature, and the rest are state measurements. Assign these outputs to separate groups.

```
sys.OutputGroup.temperature = [2];
sys.InputGroup.measurements = [1 3 4];
```

Use the group name to extract the subsystem from all inputs to the measurement outputs.

```
sys('measurements',:)
```

Example: `struct('temperature',[2],'measurement',[1 3 4])`

`Name` — Model name
`''` (default) | character vector

Model name, stored as a character vector. You can specify `Name` using a string, such as `"DCmotor"`, but the output units are stored as a character vector, `'DCmotor'`.

Example: `'system_1'`

`Notes` — Text notes about model
`[0×1 string]` (default) | string | cell array of character vector

Text notes about the model, stored as a string or a cell array of character vectors. The property stores whichever of these two data types you provide. For instance, suppose that `sys1` and `sys2` are dynamic system models, and set their `Notes` properties to a string and a character vector, respectively.

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

UserData — Data associated with model

[ ] (default) | any data type

Data of any kind that you want to associate and store with the model, specified as any MATLAB data type.

SamplingGrid — Sampling grid for model arrays

structure with no fields (default) | structure

Sampling grid for model arrays, specified as a 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.

Object Functions

For information about functions that are applicable to an `idgrey` object, see “Linear Grey-Box Models”.

Examples

Create Grey-Box Model with Estimable Parameters

Create and configure an `idgrey` model that incorporates an ODE function with one estimable parameter.

This example uses the shipped file `motorDynamics.m`, which represents the linear dynamics of a DC motor in the following form:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{1}{\tau} \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \frac{G}{\tau} \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t)$$

`motorDynamics` returns the$A$,B,C, and D matrices and explicitly sets the elements of the Kmatrix and the initial conditions $X0$ to `0`. `motorDynamics` defines the motor time constant τ as the single estimable parameter. The model also includes an auxiliary argument G that represents the known static gain. If you want to view the code for this model, enter `edit motorDynamics` at the command line.

Initialize τ to 1 by setting the value of the `parameters` single-element matrix to `1`. Set `fcn_type` to `'cd'` to specify that `odefun` can return either continuous-time (`Ts=0`) or discrete-time representation (`Ts>0`). Set `extra_args`, which represents G, to `0.25`. Set the sample time `Ts` to `0`.

```
odefun = 'motorDynamics';
parameters = 1;
fcn_type = 'cd';
extra_args = 0.25;
Ts = 0;
```

Create the `idgrey` model `sys`.

```
sys = idgrey(odefun,parameters,fcn_type,extra_args,Ts)
```

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

  A = 
       x1  x2
   x1   0   1
   x2   0  -1

  B = 
         u1
   x1     0
   x2  0.25

  C =
       x1  x2
   y1   1   0
   y2   0   1

  D =
       u1
   y1   0
   y2   0

  K =
       y1  y2
   x1   0   0
   x2   0   0

  Model parameters:
   Par1 = 1

Parameterization:
   ODE Function: motorDynamics
   (parametrizes both continuous- and discrete-time equations)
   Disturbance component: parameterized by the ODE function
   Initial state: parameterized by the ODE function
   Number of free coefficients: 1
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

To refine the estimate for τ, use `pem` or `greyest`.

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 and properties that you specify. Then, specify an additional property.

Use name-value arguments to specify names for the input and output channels.

```
odefun = 'motorDynamics';
parameters = 1;
fcn_type = 'cd';
extra_args = 0.25;
Ts = 0;
sys = idgrey(odefun,parameters,fcn_type,extra_args,Ts,'InputName','Voltage',...
            'OutputName',{'Angular Position','Angular Velocity'});
```

Specify `TimeUnit` using dot notation.

```
sys.TimeUnit = 'seconds';
```

Create Array of Grey-Box Models

Use the `stack` command to create an array of linear grey-box models.

Specify `odefun1` using the function handle `@motordynamics`. Set the static gain to `1`, using `extra_args1`.

```
odefun1 = @motorDynamics;
parameters1 = [1 2];
fcn_type = 'cd';
extra_args1 = 1;
sys1 = idgrey(odefun1,parameters1,fcn_type,extra_args1);
size(sys1)
```

```
Grey-box model with 2 outputs, 1 inputs, 2 states and 2 free parameters.
```

Specify `odefun2` using the function name `'motorDynamics'`. Set the static gain to 0.5, using `extra_args2`.

```
odefun2 = 'motorDynamics';
parameters2 = {[1 2]};
extra_args2 = 0.5;
sys2 = idgrey(odefun2,parameters2,fcn_type,extra_args2);
```

Use `stack` to create the 2-by-1 array `sysarr` of `idgrey` models.

```
sysarr = stack(1,sys1,sys2);
size(sysarr)
```

```
2x1 array of grey-box models.
Each model has 2 outputs, 1 inputs, 2 states and 2 free parameters.
```

See Also
`greyest` | `greyestOptions` | `pem` | `idnlgrey` | `idss` | `ssest` | `getpvec` | `setpvec` | `stack`

Topics
"Estimate Linear Grey-Box Models"
"Estimate Coefficients of ODEs to Fit Given Solution"
"Estimate Model Using Zero/Pole/Gain Parameters"
"Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance"

Introduced before R2006a

idinput

Generate input signals

Syntax

```
u = idinput(N)
u = idinput([N,Nu])
u = idinput([Period,Nu,NumPeriod])

u = idinput( ___ ,Type)
u = idinput( ___ ,Type,Band)
u = idinput( ___ ,Type,Band,Range)
[u,freq] = idinput( ___ ,'sine',Band,Range,SineData)
```

Description

The `idinput` command generates an input signal with specified characteristics for your system. You can use the generated input, and simulate the response of your system to study system behavior. For example, you can study the system response to periodic inputs. The system can be an actual physical system or a model such as a Simulink model. You can also design optimal experiments. For example, you can determine which input signals isolate faults or nonlinearities in your system. You can also use `idinput` to design an input that has sufficient bandwidth to excite the dynamic range of your system.

`u = idinput(N)` returns a single-channel random binary input signal `u` of length `N`. The generated signal values are either -1 or 1.

`u = idinput([N,Nu])` returns an `Nu`-channel random binary input signal, where each channel signal has length `N`. The signals in each channel differ from each other.

`u = idinput([Period,Nu,NumPeriod])` returns an `Nu`-channel periodic random binary input signal with specified period and number of periods. Each input channel signal is of length `NumPeriod*Period`.

`u = idinput( ___ ,Type)` specifies the type of input to be generated as one of the following:

- `'rbs'` — Random binary signal
- `'rgs'` — Random Gaussian signal
- `'prbs'` — Pseudorandom binary signal
- `'sine'` — Sum-of-sinusoids signal

Use with any of the previous input argument combinations.

`u = idinput( ___ ,Type,Band)` specifies the frequency band of the signal. For pseudorandom binary signals (PRBS), `Band` specifies the inverse of the clock period of the signal.

`u = idinput( ___ ,Type,Band,Range)` specifies the amplitude-range of the signal.

`[u,freq] = idinput( ___ ,'sine',Band,Range,SineData)` specifies the `Type` as a sum-of-sinusoids signal and specifies the characteristics of the sine waves used to generate the signal in

SineData. You can specify characteristics such as the number of sine waves and their frequency separation. The frequencies of the sine waves are returned in freq.

Examples

Generate a Random Binary Input Signal

Generate a single-channel random binary input signal with 200 samples.

```
N = 200;
u = idinput(N);
```

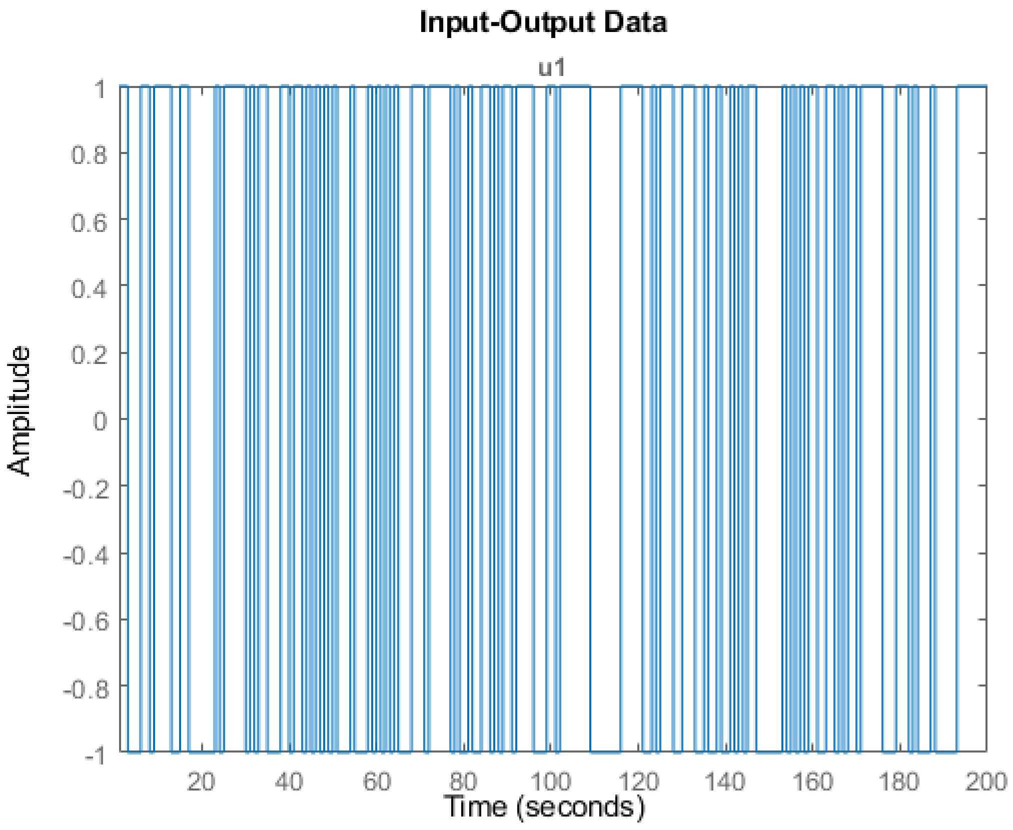
u is a column vector of length 200. The values in u are either -1 or 1.

Create an iddata object from the generated signal. For this example, specify the sample time as 1 second.

```
u = iddata([],u,1);
```

To examine the signal, plot it.

```
plot(u)
```



The generated signal is a random binary input signal with values -1 or 1. You can use the generated input signal to simulate the output of your system using the sim command.

Generate a Multichannel Random Binary Input Signal

Generate a two-channel random binary input signal with 200 samples.

```
N = 200;
u = idinput([N,2]);
```

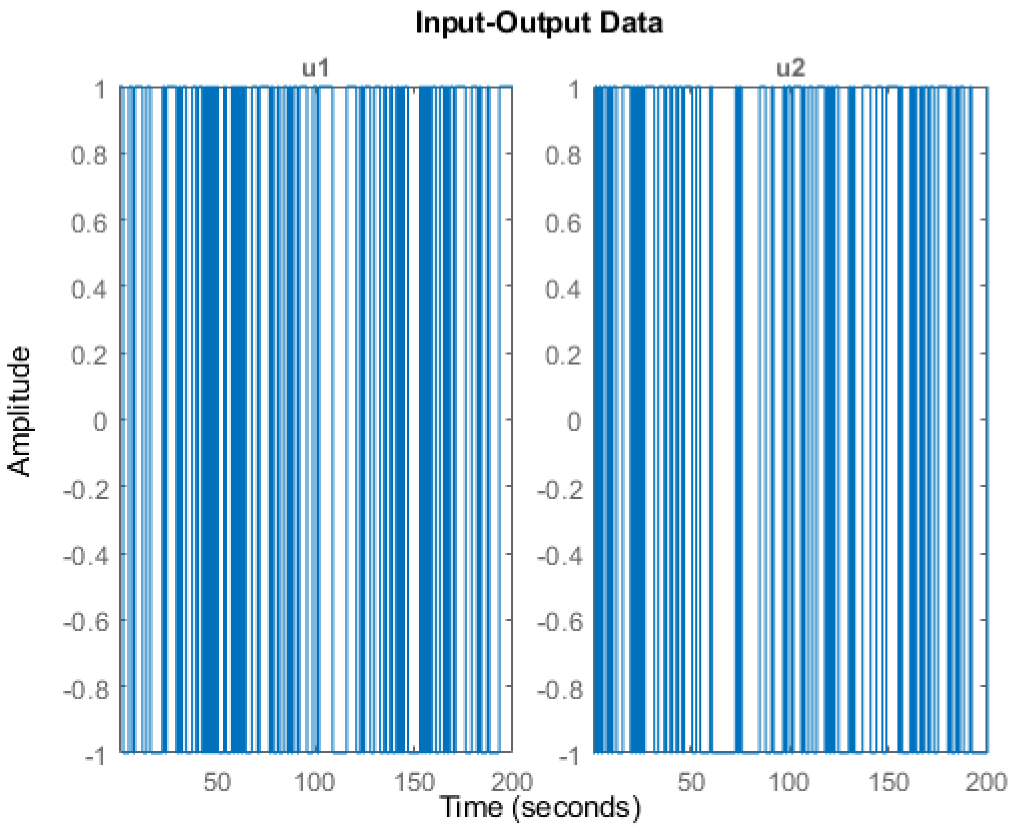
`u` is a 200-by-2 matrix with values -1 or 1.

Create an `iddata` object from the generated signal. For this example, specify the sample time as 1 second.

```
u = iddata([],u,1);
```

Plot the signals for the two channels, and examine the signals.

```
plot(u)
```



The plot shows the two generated random binary signals with values -1 or 1.

Generate a Periodic Random Binary Input Signal

Generate a single-channel periodic random binary input signal with a period of 10 samples and 5 periods in the signal.

```
NumChannel = 1;
Period = 10;
NumPeriod = 5;
u = idinput([Period,NumChannel,NumPeriod]);
```

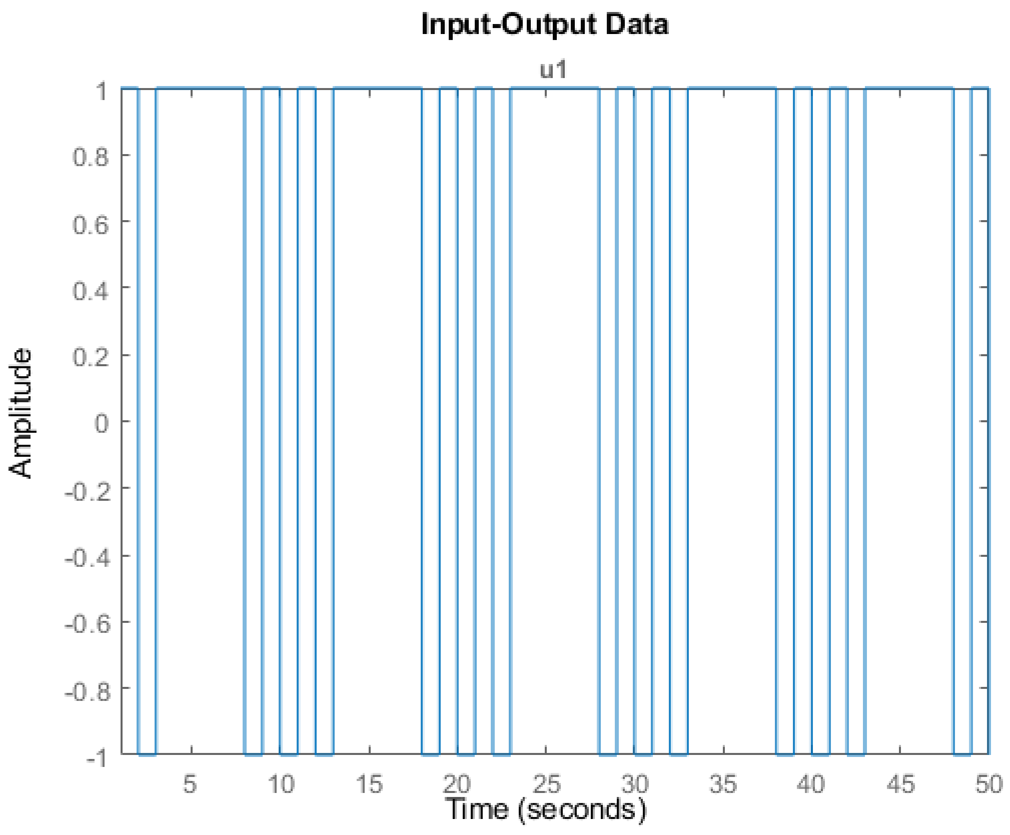
`u` is a column vector of length 50 (= Period*NumPeriod). The values in `u` are either -1 or 1.

Create an `iddata` object from the generated signal. Specify the sample time as 1 second.

```
u = iddata([],u,1);
```

Plot the signal.

```
plot(u)
```



As specified, the generated single-channel periodic random binary input signal has a period of 10 seconds, and there are 5 whole periods in the signal.

Generate a Periodic Random Gaussian Input Signal in Specified Frequency Range

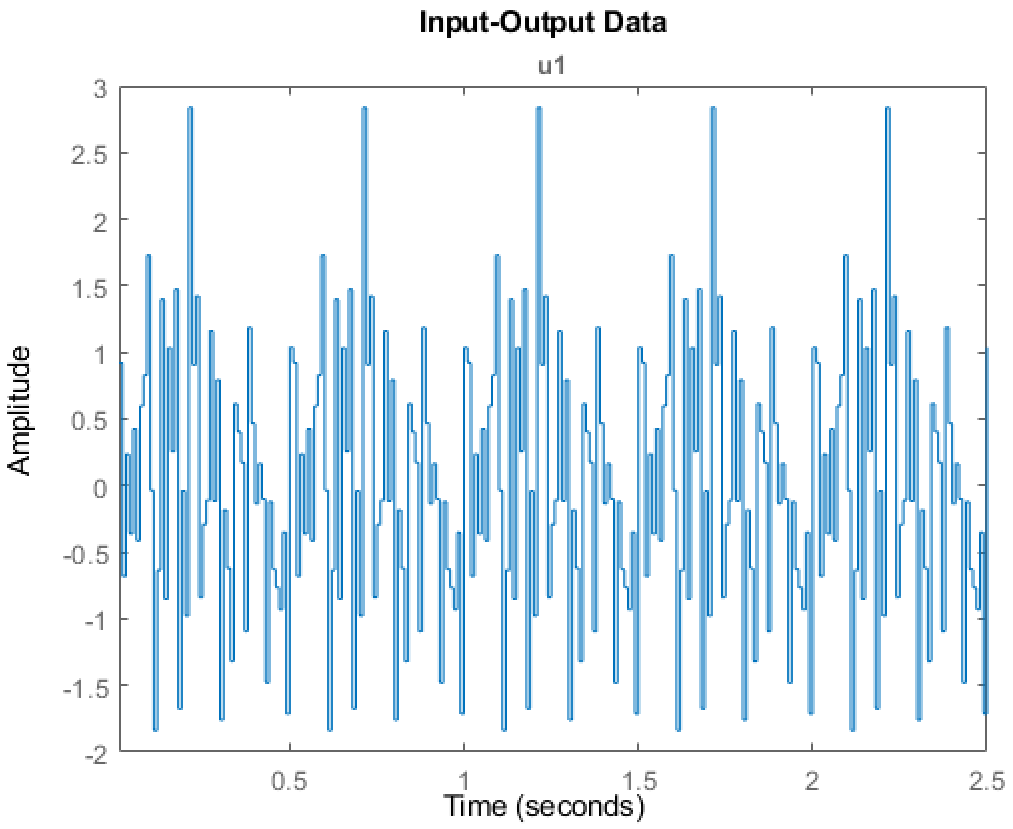
Generate a single-channel periodic random Gaussian input signal with a period of 50 samples and 5 periods in the signal. First generate the signal using the entire frequency range, then specify a passband.

```
NumChannel = 1;
Period = 50;
NumPeriod = 5;
u = idinput([Period,NumChannel,NumPeriod],'rgs');
```

u is a column vector of length 250 (= Period*NumPeriod).

Create an iddata object from the generated signal, and plot the signal. For this example, specify the sample time as 0.01 seconds.

```
u = iddata([],u,0.01);
plot(u)
```



The plot shows that u contains a random segment of 50 samples, repeated 5 times. The signal is a Gaussian white noise signal with zero mean and variance one.

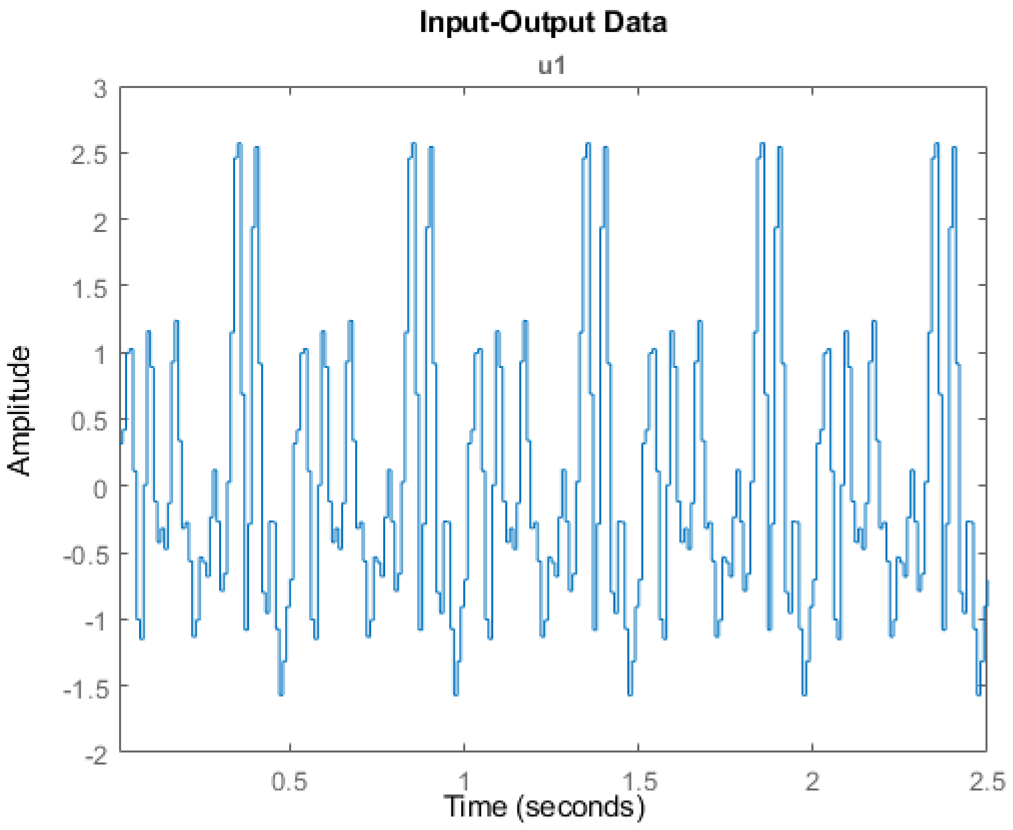
Since the sample time is 0.01 seconds, the generated signal has a period of 0.5 seconds. The frequency content of the signal spans the entire available range (0-50 Hz).

Now specify a passband between 0 and 25 Hz (= 0.5 times the Nyquist frequency).

```
Band = [0 0.5];
u2 = idinput([Period,NumChannel,NumPeriod],'rgs',Band);
```

Create an `iddata` object, and plot the signal.

```
u2 = iddata([],u2,0.01);
plot(u2)
```



The frequency content of the generated signal u2 is limited to 0-25 Hz.

Generate a Nonperiodic Pseudorandom Binary Input Signal

A pseudorandom binary input signal (PRBS) is a deterministic signal whose frequency properties mimic white noise. A PRBS is inherently periodic with a maximum period length of $2^n - 1$, where integer n is the order of the PRBS. For more information, see “Pseudorandom Binary Signals” on page 1-597.

Specify that the single-channel PRBS value switches between -2 and 2.

```
Range = [-2,2];
```

Specify the clock period of the signal as 1 sample. That is, the signal value can change at each time step. For PRBS signals, the clock period is specified in Band = [0 B], where B is the inverse of the required clock period.

```
Band = [0 1];
```

Generate a nonperiodic PRBS of length 100 samples.

```
u = idinput(100,'prbs',Band,Range);
```

```
Warning: The PRBS signal delivered is the 100 first values of a full sequence of length 127.
```

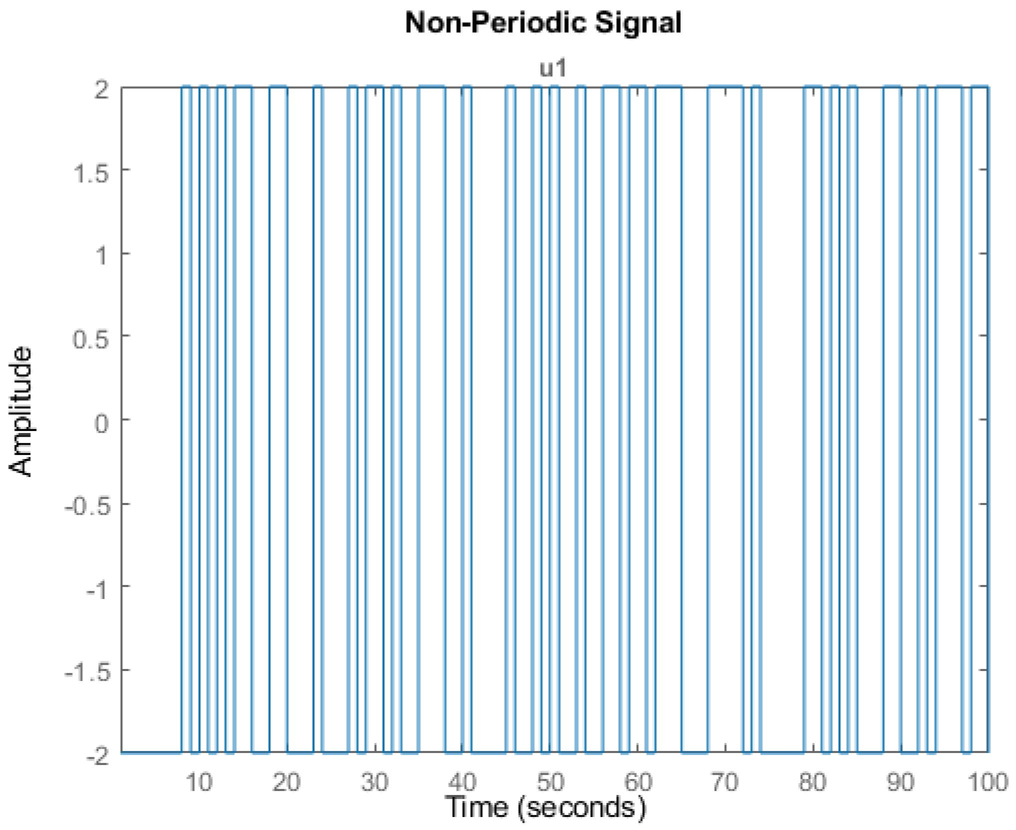
A PRBS is inherently periodic. To generate a nonperiodic signal, the software generates a maximum length PRBS of length 127 that has a period greater than the required number of samples, 100. The software returns the first 100 samples of the generated PRBS. This action ensures that the generated signal is not periodic, as indicated in the generated warning.

Create an `iddata` object from the generated signal. For this example, specify the sample time as 1 second.

```
u = iddata([],u,1);
```

Plot, and examine the generated signal.

```
plot(u);
title('Non-Periodic Signal')
```



The generated signal is a nonperiodic PRBS of length 100 that switches between -2 and 2.

Generate a Periodic Pseudorandom Binary Input Signal

Specify that the pseudorandom binary input signal (PRBS) switches between -2 and 2.

```
Range = [-2,2];
```

Specify the clock period of the signal as 1 sample. That is, the signal value can change at each time step. For PRBS signals, the clock period is specified in `Band` = [0 B], where `B` is the inverse of the required clock period.

```
Band = [0 1];
```

Generate a single-channel, periodic PRBS with a period of 100 samples and 3 periods in the signal.

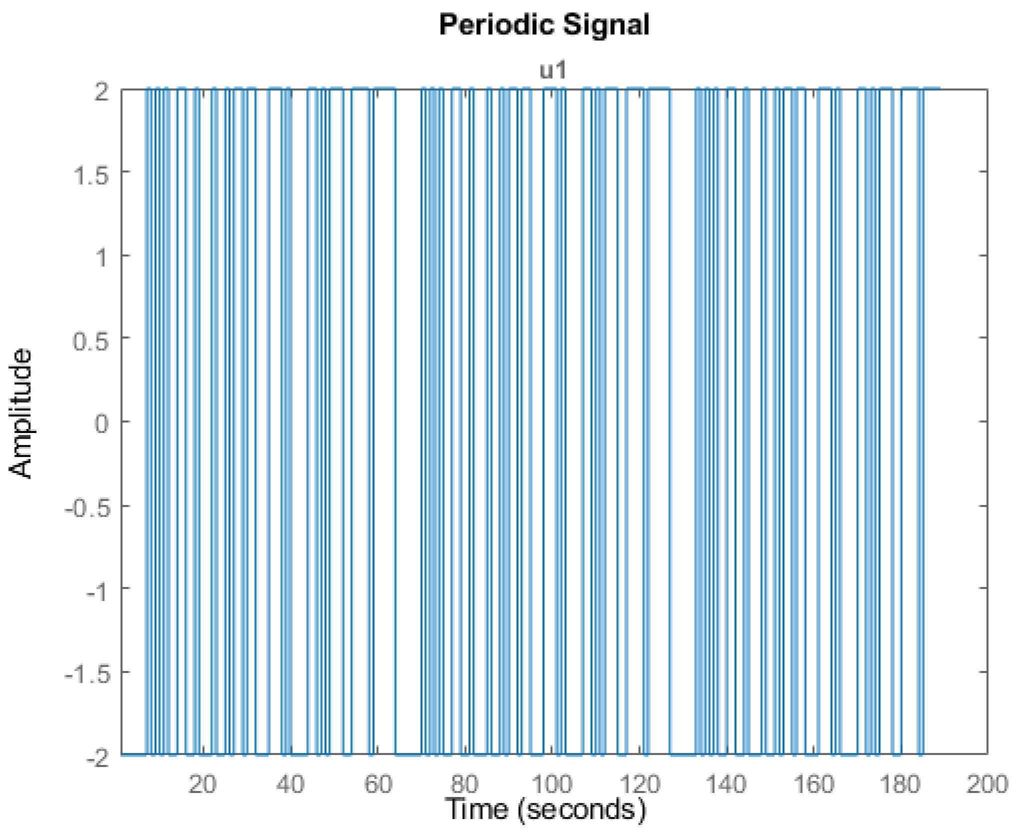
```
u1 = idinput([100,1,3],'prbs',Band,Range);
```

```
Warning: The period of the PRBS signal was changed to 63. Accordingly, the length of the generate
```

A PRBS is inherently periodic with a maximum period length of $2^n - 1$, where integer n is the order of the PRBS. If the period you specify is not equal to a maximum length PRBS, the software adjusts the period of the generated signal to obtain an integer number of maximum length PRBS, and issues a warning. For more information about maximum length PRBS, see “Pseudorandom Binary Signals” on page 1-597. In this example, the desired period, 100, is not equal to a maximum length PRBS, thus the software instead generates a maximum length PRBS of order `n = floor(log2(Period)) = 6`. Thus, the period of the PRBS signal is 63 ($= 2^6 - 1$), and the length of the generated signal is 189 (= `NumPeriod`*63). This result is indicated in the generated warning.

Create an `iddata` object from the generated signal, and plot the signal. Specify the period of the signal as 63 samples.

```
u1 = iddata([],u1,1,'Period',63);
plot(u1)
title('Periodic Signal')
```



The generated signal is a periodic PRBS with three periods.

Generate Pseudorandom Binary Input Signal with Specified Clock Period

Generate periodic and nonperiodic pseudorandom binary input signals (PRBS) with specified clock period.

Generate a single-channel PRBS that switches between -2 and 2. Specify the clock period of the signal as 4 samples. That is, the signal has to stay constant for at least 4 consecutive samples before it can change. For PRBS signals, the clock period is specified in `Band` = [0 B], where `B` is the inverse of the required clock period.

```
Range = [-2,2];
Band = [0 1/4];
```

First generate a nonperiodic signal of length 100.

```
u1 = idinput(100,'prbs',Band,Range);
```

```
Warning: The PRBS signal delivered is the 100 first values of a full sequence of length 124.
```

To understand the generated warning, first note that the code is equivalent to generating a single-channel PRBS with a 100-sample period and 1 period.

```
u1 = idinput([100,1,1],'prbs',Band,Range);
```

The generated PRBS signal has to remain constant for at least 4 samples before the value can change. To satisfy this requirement, the software first computes the order of the smallest possible maximum length PRBS as `n = floor(log2(Period*B)) = 4` and period $2^n - 1 = 15$. For information about maximum length PRBS, see "Pseudorandom Binary Signals" on page 1-597. The software then stretches this PRBS such that the period of the stretched signal is $P = (1/B)(2^n - 1) = 60$.

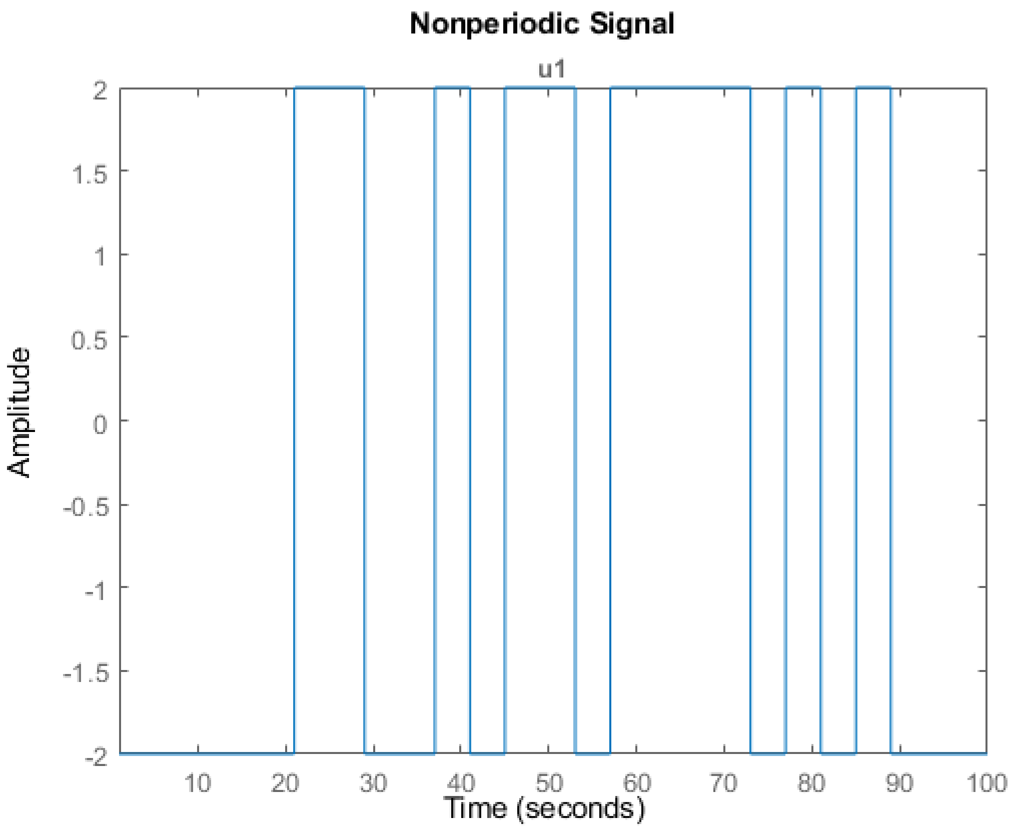
However, since this period is less than the specified length, 100, the software computes instead a maximum length PRBS of order `m = n+1 = 5`. The software then stretches this PRBS such that the period is now $P2 = (1/B)(2^m - 1) = 124$. The software returns the first 100 samples of this signal as `u1`. This result ensures that the generated signal is not periodic but is constant for every 4 samples.

Create an `iddata` object from the generated signal. For this example, specify the sample time as 1 second.

```
u1 = iddata([],u1,1);
```

Plot, and examine the signal.

```
plot(u1);
title('Nonperiodic Signal')
```



The generated signal is a nonperiodic PRBS of length 100. The signal remains constant for at least 4 samples before each change in value. Thus, the signal satisfies the clock period specified in `Band`.

Now generate a periodic signal with a 100-sample period and 3 periods.

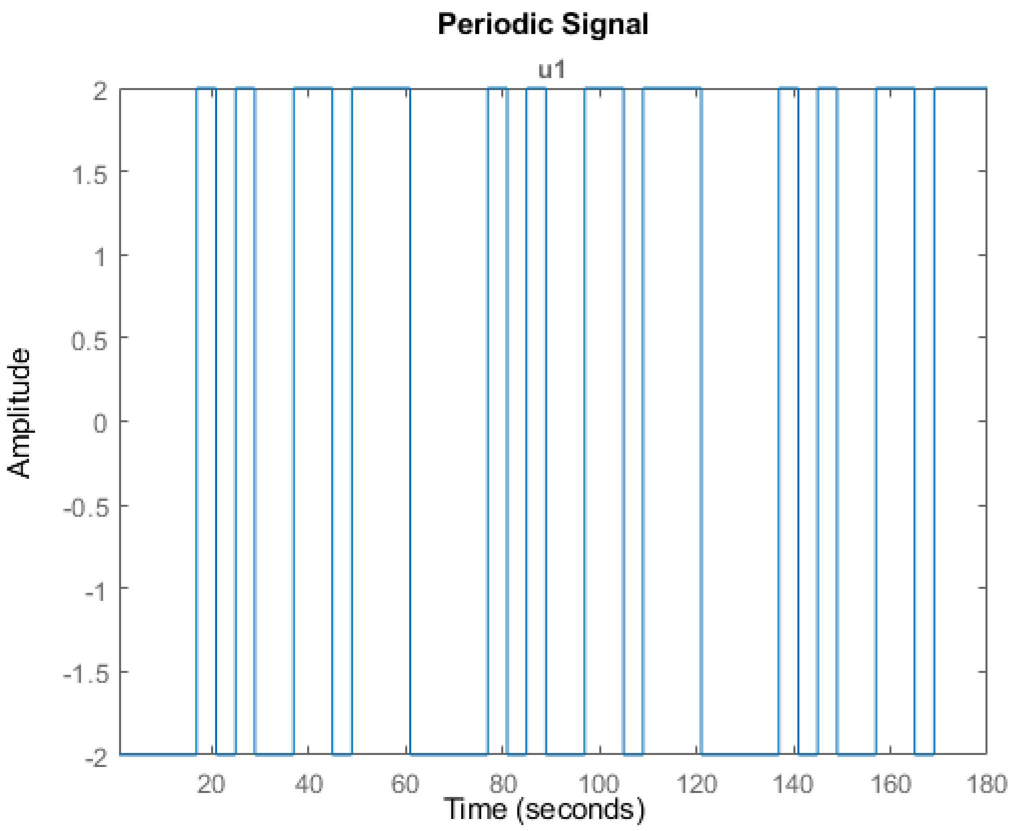
```
u2 = idinput([100,1,3],'prbs',Band,Range);
```

```
Warning: The period of the PRBS signal was changed to 60. Accordingly, the length of the generate
```

To generate a periodic signal with specified clock period, the software generates `u2` as 3 repetitions of the original stretched signal of period `P = 60`. Thus, the length of `u2` is `P*NumPeriod = 60*3 = 180`. This change in period and length of the generated signal is indicated in the generated warning.

Create an `iddata` object from the generated signal, and plot the signal. Specify the period of the signal as 60 seconds.

```
u2 = iddata([],u2,1,'Period',60);
plot(u2)
title('Periodic Signal')
```



The generated signal is a periodic PRBS with a 60-second period and 3 periods. The signal remains constant for at least 4 samples before each change in value. Thus, the signal satisfies the specified clock period.

Generate a Sum-of-Sinusoids Signal

You can generate a sum-of-sinusoids signal using default characteristics for the sine waves. Alternatively, you configure the number of sine waves, and the frequencies and phases of the sine waves. This example shows both approaches.

Specify that the signal has 50 samples in each period and 3 periods. Also specify that the signal amplitude range is between -1 and 1.

```
Period = 50;
NumPeriod = 3;
Range = [-1 1];
```

Specify the frequency range of the signal. For a sum-of-sinusoids signal, you specify the lower and upper frequencies of the passband in fractions of the Nyquist frequency. In this example, use the entire frequency range between 0 and Nyquist frequency.

```
Band = [0 1];
```

First generate the signal using default characteristics for the sine waves. By default, the software uses 10 sine waves to generate the signal. The software assigns a random phase to each sinusoid, and then changes these phases 10 times to get the smallest signal spread. The signal spread is the difference between the minimum and the maximum value of the signal over all samples.

```
[u,freq] = idinput([Period 1 NumPeriod],'sine',Band,Range);
```

The software returns the sum-of-sinusoids signal in `u` and the frequencies of the sinusoids in `freq`. The values in `freq` are scaled assuming that the sample time is 1 time unit. Suppose that the sample time is 0.01 hours. To retrieve the actual frequencies in rad/hours, divide the values by the sample time.

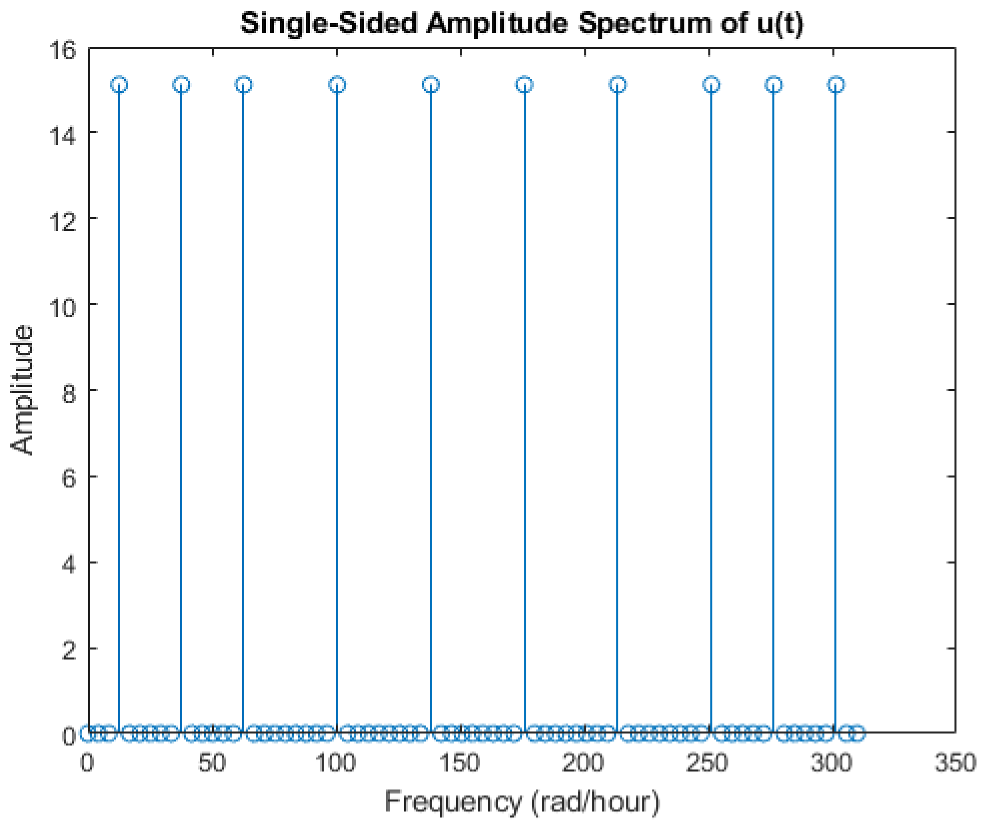
```
Ts = 0.01; % Sample time in hours
freq = freq/Ts;
freq(1)
```

```
ans = 12.5664
```

`freq(1)` is the frequency of the first sine wave. To see how the software chooses the frequencies, see the `SineData` argument description on the `idinput` reference page.

To verify that 10 sine waves were used to generate the signal, you can view the frequency content of the signal. Perform a Fourier transform of the signal, and plot the single-sided amplitude spectrum of the signal.

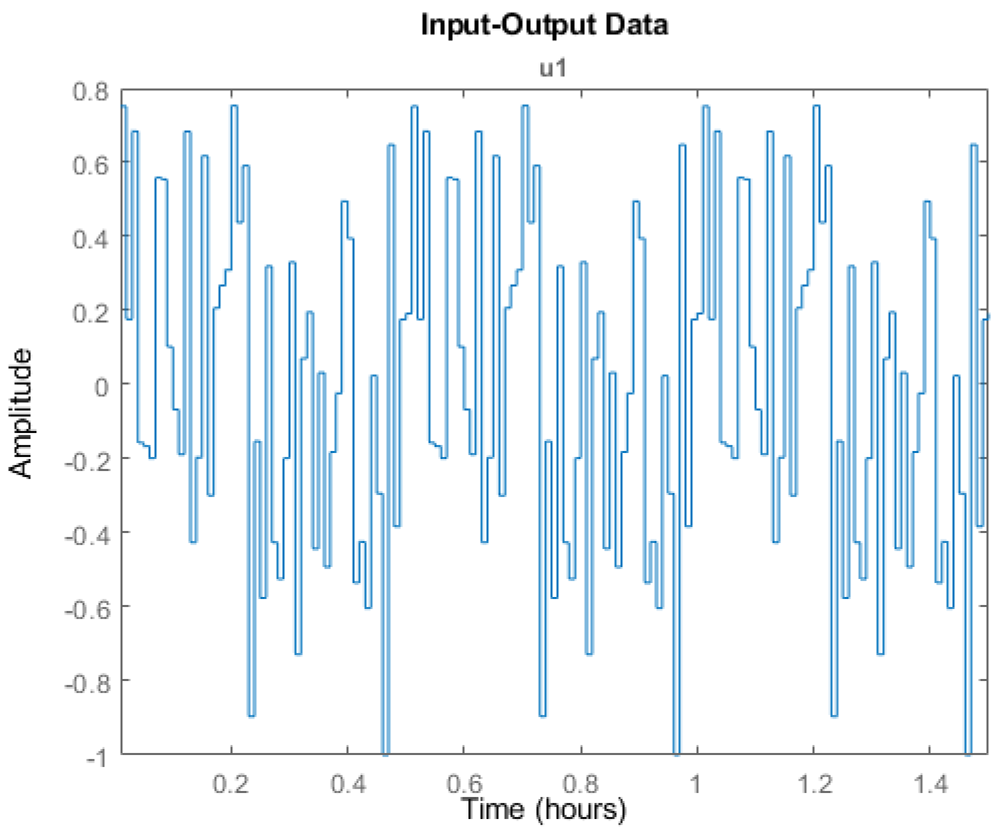
```
ufft = fft(u);
Fs = 2*pi/Ts; % Sampling frequency in rad/hour
L = length(u);
w = (0:L-1)*Fs/L;
stem(w(1:L/2),abs(ufft(1:L/2))) % Plot until Nyquist frequency
title('Single-Sided Amplitude Spectrum of u(t)')
xlabel('Frequency (rad/hour)')
ylabel('Amplitude')
```



The generated plot shows the frequencies of the 10 sine waves used to generate the signal. For example, the plot shows that the first sine wave has a frequency of 12.57 rad/hour, the same as `freq(1)`.

Convert the generated signal into an `iddata` object, and plot the signal. Specify the sample time as 0.01 hours.

```
u = iddata([],u,Ts,'TimeUnit','hours');
plot(u)
```



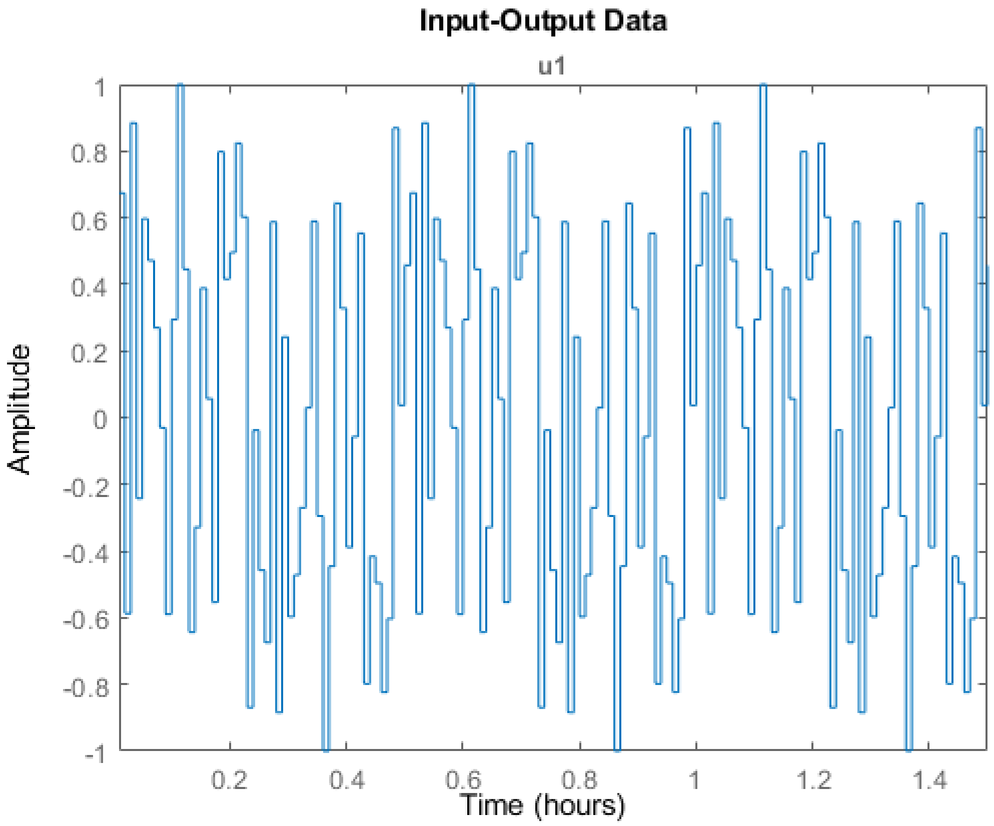
The signal `u` is generated using 10 sinusoids and has a period of 0.5 hours and 3 periods.

Now modify the number, frequency, and phase of the sinusoids that are used to generate the sum-of-sinusoids signal. Use 12 sinusoids and try 15 different sets of phases. To set the frequencies of the sinusoids, specify `GridSkip` = 2. The software selects the frequencies of the sinusoids from the intersection of the frequency grid `2*pi*[1:GridSkip:fix(Period/2)]/Period` and the passband `pi*Band`.

```
NumSinusoids = 12;
NumTrials = 15;
GridSkip = 2;
SineData = [NumSinusoids,NumTrials,GridSkip];
u2 = idinput([Period 1 NumPeriod],'sine',Band,Range,SineData);
```

Convert the generated signal into an `iddata` object, and plot the signal.

```
u2 = iddata([],u2,Ts,'TimeUnit','hours');
plot(u2)
```



The signal u2 is generated using 12 sinusoids and has a period of 0.5 hours and 3 periods.

Input Arguments

N — Number of generated input data samples
real positive integer

Number of generated input data samples, specified as a real positive integer. For a single-channel input data, the generated input u has N rows. For an Nu-channel input data, u is returned as an N-by-Nu matrix, where each channel signal has length N.

Nu — Number of input channels
1 (default) | real positive integer

Number of input channels in generated signal, specified as a real positive integer.

Period — Number of samples in each period
real positive integer

Number of samples in each period of generated signal, specified as a real positive integer. Use this input to specify a periodic signal. Also specify the number of periods in NumPeriod. Each generated input channel signal has NumPeriod*Period samples.

NumPeriod — Number of periods in generated signal
1 (default) | real positive integer

Number of periods in generated signal, specified as a real positive integer. Use this input to specify a periodic signal. Also specify the signal `Period`. Each generated input channel signal has `NumPeriod*Period` samples.

Type — Type of generated signal
`'rbs'` (default) | `'rgs'` | `'prbs'` | `'sine'`

Type of generated signal, specified as one of the following values:

- `'rbs'` — Generates a random binary signal. A random binary signal is a random process that assumes only two values. You can specify these values using `Range`. To generate a band-limited signal, specify the passband in `Band`. To generate a periodic signal, specify `Period` and `NumPeriod`.
- `'rgs'` — Generates a random Gaussian signal. The generated Gaussian signal has mean μ and standard deviation σ such that [μ-σ, μ+σ] equals `Range`. To generate a band-limited Gaussian signal, specify the passband in `Band`. To generate a periodic Gaussian signal with an `n` samples period that repeats itself `m` times, specify `Period` as `n` and `NumPeriod` as `m`.
- `'prbs'`— Generates a pseudorandom binary signal (PRBS). A PRBS is a periodic, deterministic signal with white-noise-like properties that shifts between two values. You can specify these two values using `Range`. You can also specify the clock period, the minimum number of sampling intervals for which the value of the signal does not change. You specify the inverse of the clock period in `Band`.

  The length of the generated signal is not always the same as what you specify. The length depends on whether you require a periodic or nonperiodic signal and also on the clock period you specify. For more information, see “Pseudorandom Binary Signals” on page 1-597.
- `'sine'`— Generates a signal that is a sum-of-sinusoids. The software selects the frequencies of the sinusoids to be equally spread over a chosen grid and assigns each sinusoid a random phase. The software then tries several random phases for each sinusoid and selects the phases that give the smallest signal spread. The signal spread is the difference between the minimum and the maximum value of the signal over all samples. The amplitude of the generated sum-of-sinusoids signal is scaled to satisfy the `Range` you specify.

  You can specify the characteristics of the sine waves used to generate the signal, such as the number of sine waves and their frequency separation, in the `SineData` argument.

Band — Frequency range of generated signal
`[0 1]` (default) | 1-by-2 row vector

Frequency range of generated signal, specified as a 1-by-2 row vector containing minimum and maximum frequency values.

- If `Type` is `'rgs'`, `'rbs'`, or `'sine'` — Specify `Band` as a passband `[wlow whigh]`. Where, `wlow` and `whigh` are the lower and upper frequencies of the passband, expressed in fractions of the Nyquist frequency. For example, to generate an input with white noise characteristics, use `Band = [0 1]`.

  The software achieves the frequency contents for a random Gaussian signal (`'rgs'`) using `idfilt` with an eighth-order Butterworth, noncausal filter. For generating a random binary signal (`'rbs'`), the software uses the same filter and then makes the signal binary. Thus, the frequency content in the generated random binary signal may not match the specified passband.

For `'sine'` signals, the frequencies of the sinusoids are selected to be equally spread over a chosen grid in the specified passband. For more information, see the `SineData` argument description.

- If `Type` is `'prbs'` — Specify `Band` as `[0 B]`, where `B` is the inverse of the clock period of the signal. The clock period is the minimum number of sampling intervals for which the value of the signal does not change. Thus, the generated signal is constant over intervals of length `1/B` samples. If `1/B` is not an integer, the software uses `floor(1/B)` as the clock period.

Range — Generated input signal range
`[-1,1]` (default) | two-element row vector

Generated input signal range, specified as a two-element row vector of the form `[umin,umax]`.

- If `Type` is `'rbs'` or `'prbs'`— The generated signal `u` has values `umin` or `umax`.
- If `Type` is `'sine'` — The generated signal `u` has values between `umin` and `umax`.
- If `Type` is `'rgs'` — The generated Gaussian signal has mean μ and standard deviation σ such that `umin` and `umax` are equal to μ-σ and μ+σ, respectively. For example, `Range = [-1,1]` returns a Gaussian white noise signal with zero mean and variance one.

`SineData` — Characterization of sinusoids
`[10,10,1]` (default) | three-element row vector `[NumSinusoids,NumTrials,GridSkip]`

Characterization of sinusoids used to generate a sum-of-sinusoids signal, specified as a three-element row vector `[NumSinusoids,NumTrials,GridSkip]`. Where,

- `NumSinusoids` is the number of sinusoids used to generate the signal. The default value is `10`.
- `NumTrials` is the number of different random relative phases of the sinusoids that the software tries to find the lowest signal spread. The signal spread is the difference between the minimum and the maximum value of the signal over all samples.

  The maximum amplitude of the sum-of-sinusoids signal depends on the relative phases of the different sinusoids. To find the phases that give the smallest signal spread, the software tries `NumTrials` different random choices of phases to find the best phase values. For example, suppose that `NumSinusoids` is `20` and `NumTrials` is `5`. The software tries 5 different sets of relative phases for the 20 sinusoids, and selects the phases that give the smallest signal spread. The default value for `NumTrials` is `10`.
- `GridSkip` is used to characterize the frequency of the sinusoids. The software selects the frequency of the sinusoids from the intersection of the frequency grid `2*pi*[1:GridSkip:fix(Period/2)]/Period` and the pass band `pi*[Band(1) Band(2)]`. For multichannel input signals, the software uses different frequencies from this frequency grid to generate the different input channels. You can use `GridSkip` for controlling odd and even frequency multiples, for example, to detect nonlinearities of different kinds.

  To extract the frequencies `freq` that are selected by the software to generate the signal, use the following syntax.

  ```
  [u,freq] = idinput(__)
  ```

Output Arguments

u — Generated input signal
column vector | matrix

Generated input signal, returned as a column vector of length `N` for a single-channel input or an `N`-by-`Nu` matrix for an `Nu`-channel signal. You use the generated signal to simulate the response of your system using `sim`.

You can create an `iddata` object from `u` by specifying output data as `[]`.

```
u = iddata([],u);
```

In the `iddata` object, you can also specify the properties of the signal such as sample time, input names, and periodicity.

`freq` — Frequencies of sine waves
column vector | matrix

Frequencies of sine waves used for sum-of-sinusoids signal, returned as a column vector of length equal to the number of sinusoids, `NumSinusoids`. You specify `NumSinusoids` in the `SineData` argument. The frequency values are scaled assuming the sample time is 1 time unit. To retrieve the actual frequencies, divide the values by the sample time. For an example, see “Generate a Sum-of-Sinusoids Signal” on page 1-590.

For multichannel input signals, `freq` is an `Nu`-by-`NumSinusoids` matrix where the *k*th row contains the frequencies corresponding to the *k*th channel. For information about how the software selects the frequencies, see the `SineData` argument description.

More About

Pseudorandom Binary Signals

A pseudorandom binary signal (PRBS) is a periodic, deterministic signal with white-noise-like properties that shifts between two values.

A PRBS is generated as:

$$u(t) = rem(a_1 u(t-1) + \ldots + a_n u(t-n), 2)$$

Here, $u(t-1), \ldots u(t-n)$ is the vector of past inputs, n is the PRBS order, and `rem` denotes the remainder when $(a_1 u(t-1) + \ldots a_n u(t-n))$ is divided by 2. Thus, a PRBS can only take the values 0 and 1. The software scales these values according to the `Range` you specify. In addition, the vector of past inputs $u(t-1), \ldots u(t-n)$ can only take 2^n values. Out of these values, the state with all zeros is ignored because it will result in future signals equal to zero. Thus, a PRBS is an inherently periodic signal with a maximum period length of 2^n-1. The following table lists the maximum length possible for different orders n of the PRBS.

Order *n*	**Maximum length PRBS (2^n-1)**
2	3
3	7
4	15
5	31
6	63
7	127

Order *n*	**Maximum length PRBS (2^n-1)**
⋮	⋮
32	4294967295

Note The software does not generate signals with period greater than 2^{32}-1.

Length of Generated PRBS

Since PRBS are inherently periodic, the length and period of the generated signal depends on the clock period that you specify and whether you require a periodic or nonperiodic signal. The clock period is the minimum number of sampling intervals for which the value of the signal does not change. You specify the clock period in `Band`.

Clock period = 1 sample (`Band = [0 B] = [0 1]`):

- To generate a *nonperiodic signal* of length `N`, (`NumPeriod` = 1), the software first computes a maximum length PRBS with a period greater than `N`. The software then returns the first `N` samples of the PRBS as `u`. This action ensures that `u` is not periodic. For example, if `N` is 100, the software creates a maximum length PRBS of period 127 (order 7), and returns the first 100 samples as `u`.

  For an example, see “Generate a Nonperiodic Pseudorandom Binary Input Signal” on page 1-585.
- To generate a *periodic signal* (`NumPeriod` > 1), the software adjusts the period of the signal to obtain an integer number of maximum length PRBS. To do so, the software computes a PRBS of order `n = floor(log2(Period))` and period `P =` 2^n`-1`. The signal `u` is then generated as `NumPeriod` repetitions of this PRBS signal of period `P`. Thus, the length of `u` is `P*NumPeriod`.

  For an example, see “Generate a Periodic Pseudorandom Binary Input Signal” on page 1-586.

  In the multiple-input channel case, the signals are maximally shifted. That is, the overlap between the different inputs is minimized. This means `Period/NumPeriod` is an upper bound for the model orders that you can estimate using such a signal.

Clock period > 1 sample (`Band = [0 B], where B`<1):

The generated signal has to remain constant for at least `1/B` samples. To satisfy this requirement, the software first computes the order of the smallest possible maximum length PRBS as `n = floor(log2(Period*B))` and period 2^n`-1`. The software then stretches the PRBS such that period of the stretched signal is `P =` $B^{-1}(2^n-1)$.

- To generate a *nonperiodic signal* of length `N` , if the period `P` of the stretched signal is greater than or equal to `N`, the software returns the first `N` samples of the stretched signal as `u`. This ensures that `u` is nonperiodic but constant for every `1/B` samples. Note that for a nonperiodic signal, `Period` is equal to `N`.

  If the period `P` is less than `N`, the software computes instead a maximum length PRBS of order `n2 = n+1`. The software then stretches this PRBS such that the period is now $P_2 = B^{-1}(2^{n2}-1)$. The software then returns the first `N` samples of this signal as `u`.
- To generate a *periodic signal*, the software generates `u` as `NumPeriod` repetitions of the stretched signal of period `P`. Thus, the length of `u` is `P*NumPeriod`.

For an example, see “Generate Pseudorandom Binary Input Signal with Specified Clock Period” on page 1-588.

References

[1] Söderström, T. and P. Stoica., Chapter C5.3 in *System Identification*, Prentice Hall, 1989.

[2] Ljung, L., Section 13.3 in *System Identification: Theory for the User*, Prentice Hall PTR, 1999.

See Also

`iddata` | `sim`

Topics

"Ways to Obtain Identification Data"
"Generate Data Using Simulation"
"Simulate and Predict Identified Model Output"

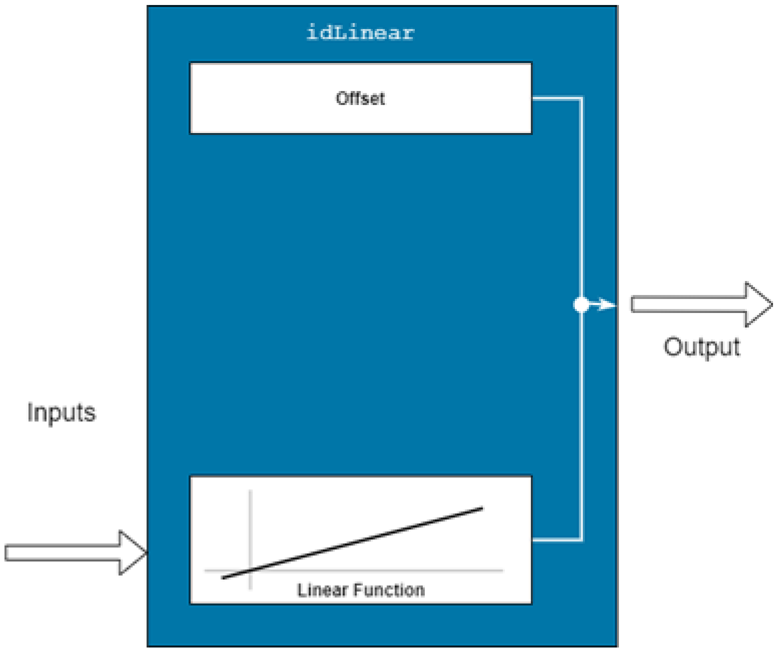
Introduced before R2006a

idLinear

Linear mapping object for nonlinear ARX models

Description

An `idLinear` object implements an affine function, and is a mapping function for estimating nonlinear ARX models. The mapping function uses a combination of linear weights and an offset. Unlike the other mapping objects for the nonlinear models, the `idLinear` object contains no accommodation for a nonlinear component.



Mathematically, `idLinear` is a linear function $y = F(x)$ that maps m inputs $X(t) = [x(t_1),x_2(t),\ldots,x_m(t)]^T$ to a scalar output $y(t)$. . F is a (affine) function of x:

$$y(t) = y_0 + (X(t) - \bar{X})^T PL$$

Here:

- $X(t)$ is an m-by-1 vector of inputs, or regressors, with mean $\bar{X}$.
- y_0 is the output offset, a scalar.
- P is an m-by-p projection matrix, where m is the number of regressors and is p is the number of linear weights. m must be greater than or equal to p.
- L is a p-by-1 vector of weights.

Set `idLinear` as the value of the `OutputFcn` property of an `idnlarx` model. For example, specify `idLinear` when you estimate an `idnlarx` model with the following command.

```
sys = nlarx(data,regressors,idLinear)
```

When `nlarx` estimates the model, it also estimates the parameters of the `idLinear` function.

Use the `idLinear` mapping object when you want to create nonlinear ARX models that operate linearly on the regressors. The regressors themselves can be nonlinear functions of the inputs and outputs. The `polynomialRegressor` and `customRegressor` commands allow you to create such regressors. When the `idnlarx` model has no custom regressors and the output function is set to `idLinear`, the model is similar to a linear ARX model. However, for the nonlinear ARX model, the offset is an estimable parameter.

You can configure the `idLinear` object to disable components and fix parameters. Use `evaluate` to compute the output of the function for a given vector of inputs.

Creation

Syntax

`Lin = idLinear`

Description

`Lin = idLinear` creates an `idLinear` object `Lin` with unknown parameters.

Properties

`Input` — Input signal information
signal property values

Input signal information for signals used for estimation, specified as vectors of *m* property-specific values, where *m* is the number of input signals. The `Input` properties for each input signal are as follows:

- `Name` — Names of the input signals, specified as a 1-by-*m* string or character array, where *m* is the number of inputs
- `Mean` — Mean of the input signals, specified as a numeric scalar
- `Range` — Ranges of the input signals, specified as a 2-by-*m* numeric array that contains the minimum and maximum values

`Output` — Output signal information
signal property values

Output signal information, specified as property-specific values. The `Output` properties are as follows:

- `Name` — Name of the output signal, specified as a string or a character array
- `Mean` — Mean of the output signal, specified as a numeric scalar
- `Range` — Range of the output signal, specified as 2-by-1 numeric array that contains the minimum and maximum values.

LinearFcn — Parameters of linear function

linear function property values (default)

Parameters of the linear function, specified as follows:

- Value — Value of L', specified as a 1-by-m vector.
- Free — Option to update entries of Value during estimation. specified as a logical scalar. The software honors the Free specification only if the starting value of Value is finite. The default value is true.
- Minimum — Minimum bound on Value, specified as a 1-by-p vector. If Minimum is specified with a finite value and the starting value of Value is finite, then the software enforces that minimum bound during model estimation.
- Maximum — Maximum bound on Value, specified as a 1-by-p vector. If Maximum is specified with a finite value and the starting value of Value is finite, then the software enforces that maximum bound during model estimation.
- SelectedInputIndex — Indices of idLinear inputs (see Input.Name) that are used as inputs to the linear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. The RegressorUsage property of the idnlarx model determines these indices.

Offset — Parameters of offset term

offset property values

Parameters of the offset term, specified as follows:

- Value — Offset value, specified as a scalar.
- Free — Option to update Value during estimation, specified as a scalar logical. The software honors the Free specification of false only if the value of Value is finite. The default value is true.
- Minimum — Minimum bound on Value, specified as a numeric scalar or –Inf. If Minimum is specified with a finite value and the value of Value is finite, then the software enforces that minimum bound during model estimation. The default value is -Inf.
- Maximum — Maximum bound on Value, specified as a numeric scalar or Inf. If Maximum is specified with a finite value and the starting value of Value is finite, then the software enforces that maximum bound during model estimation. The default value is Inf.

Examples

Estimate Nonlinear ARX Model with idLinear as Output Function

Load the data.

```
load iddata7 z7
```

Create an idLinear mapping object L.

```
L = idLinear;
```

Create model regressors that include nonlinear polynomial regressors.

```
Reg1 = linearRegressor({'y1','u1'},{1:4, 0:4});
Reg2 = polynomialRegressor({'y1','u1'},{1:2, 0:2},2,false,true,true);
Reg3 = polynomialRegressor({'y1','u1'},{2, 1:3},3,false,true);
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z7,[Reg1;Reg2;Reg3],L)
```

```
sys =
Nonlinear ARX model with 1 output and 2 inputs
  Inputs: u1, u2
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables y1, u1
  3. Order 3 regressors in variables y1, u1
  List of all regressors

Output function: None
Sample time: 1 seconds

Status:
Estimated using NLARX on time domain data "z7".
Fit to estimation data: 43.22% (prediction focus)
FPE: 5.66, MSE: 4.963
```

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

Use of previous nonlinearity estimator properties is not recommended
Not recommended starting in R2021a

Starting in R2021a, the properties of the mapping objects, previously known as nonlinearity estimators, have been reorganized. These objects are `wavenet` (W), `sigmoidnet` (S), `treepartition` (T), `customnet` (C), and `linear` (L). The property changes do not apply to `neuralnet`. The use of the pre-R2021a properties in the following table is discouraged. However, the software still accepts commands that set these properties. There are no plans to exclude such commands at this time.

Pre-R2021a Property	R2021a Property	Applicable Mapping Objects
`NumberOfUnits`	`NonlinearFcn.NumberOfUnits`	W,S,T,C
`LinearTerm`	`LinearFcn.Use`, `Offset.Use`	W,S,C
`Parameters`	Split into three pieces: • `LinearFcn.Value` • `Offset.Value` • `NonlinearFcn.Parameters`	W,S,T,C,L `linear` (L) excludes `NonlinearFcn.Parameters`.
`Options`	`NonlinearFcn.Structure`	W,T

See Also

`nlarx` | `idTreePartition` | `idSigmoidNetwork` | `idWaveletNetwork` | `idFeedforwardNetwork` | `idCustomNetwork` | `idnlarx` | `evaluate` | `linearRegressor` | `polynomialRegressor`

Introduced in R2007a

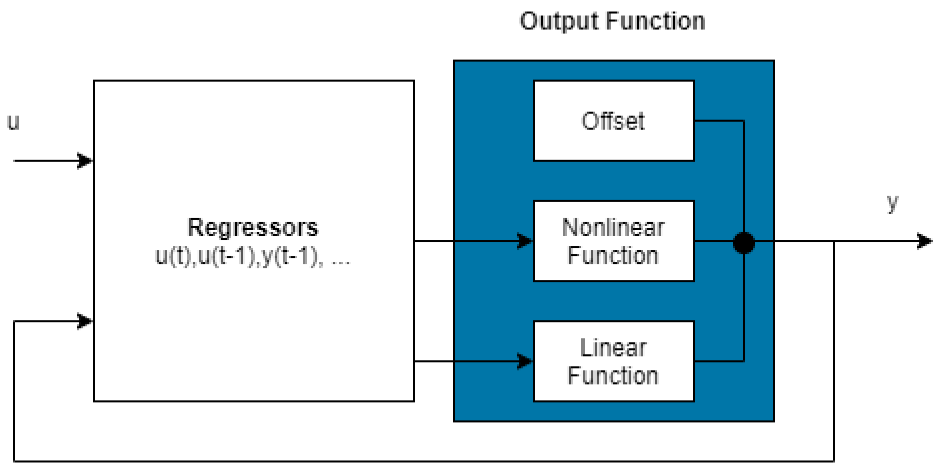
idnlarx

Nonlinear ARX model

Description

An `idnlarx` model represents a nonlinear ARX model, which is an extension of the linear ARX structure and contains linear and nonlinear functions.

A nonlinear ARX model consists of model regressors and an output function. The output function includes linear and nonlinear functions that act on the model regressors to give the model output and a fixed offset for that output. This block diagram represents the structure of a nonlinear ARX model in a simulation scenario.



The software computes the nonlinear ARX model output *y* in two stages:

1. It computes regressor values from the current and past input values and the past output data.

   In the simplest case, regressors are delayed inputs and outputs, such as *u*(*t*–1) and *y*(*t*–3). These kind of regressors are called linear regressors. You specify linear regressors using the `linearRegressor` object. You can also specify linear regressors by using linear ARX model orders as an input argument. For more information, see “Nonlinear ARX Model Orders and Delay”. However, this second approach constrains your regressor set to linear regressors with consecutive delays. To create polynomial regressors, use the `polynomialRegressor` object. You can also specify custom regressors, which are nonlinear functions of delayed inputs and outputs. For example, *u*(*t*–1)*y*(*t*–3) is a custom regressor that multiplies instances of input and output together. Specify custom regressors using the `customRegressor` object.

   You can assign any of the regressors as inputs to the linear function block of the output function, the nonlinear function block, or both.
2. It maps the regressors to the model output using an output function block. The output function block can include linear and nonlinear blocks in parallel. For example, consider the following equation:

   $$F(x) = L^T(x - r) + g(Q(x - r)) + d$$

Here, x is a vector of the regressors, and r is the mean of x. $F(x) = L^T(x - r) + y_0$ is the output of the linear function block. $g(Q(x - r)) + y_0$ represents the output of the nonlinear function block. Q is a projection matrix that makes the calculations well-conditioned. d is a scalar offset that is added to the combined outputs of the linear and nonlinear blocks. The exact form of $F(x)$ depends on your choice of output function. You can select from the available mapping objects, such as tree-partition networks, wavelet networks, and multilayer neural networks. You can also exclude either the linear or the nonlinear function block from the output function.

When estimating a nonlinear ARX model, the software computes the model parameter values, such as L, r, d, Q, and other parameters specifying g.

The resulting nonlinear ARX models are `idnlarx` objects that store all model data, including model regressors and parameters of the output function. For more information about these objects, see “Nonlinear Model Structures”.

For more information on the `idnlarx` model structure, see “What are Nonlinear ARX Models?”.

For `idnlarx` object properties, see “Properties” on page 1-607.

Creation

You can obtain an `idnlarx` object in one of two ways.

- Use the `nlarx` command to both construct an `idnlarx` object and estimate the model parameters.

  ```
  sys = nlarx(data,reg)
  ```
- Use the `idnlarx` constructor to create the nonlinear ARX model and then estimate the model parameters using `nlarx` or `pem`.

  ```
  sys = idnlarx(output_name,input_name,reg)
  ```

Syntax

```
sys = idnlarx(output_name,input_name,orders)
sys = idnlarx(output_name,input_name,Regressors)
sys = idnlarx( ___ ,OutputFcn)

sys = idnlarx(linmodel)
sys = idnlarx(linmodel,OutputFcn)

sys = idnlarx( ___ ,Name,Value)
```

Description

Specify Model Directly

`sys = idnlarx(output_name,input_name,orders)` specifies a set of linear regressors using ARX model orders. Use this syntax when you extend an ARX linear model, or when you plan to use only regressors that are linear with consecutive lags.

`sys = idnlarx(output_name,input_name,Regressors)` creates a nonlinear ARX model with the output and input names of `output_name` and `input_name`, respectively, and a regressor set in

`Regressors` that contains any combination of linear, polynomial, and custom regressors. The software constructs `sys` using the default wavelet network (`'idWaveletNetwork'`) mapping object for the output function.

`sys = idnlarx( ___ ,OutputFcn)` specifies the output function `OutputFcn` that maps the regressors to the model output. You can use this syntax with any of the previous input argument combinations.

Initialize Model Values Using Linear Model

`sys = idnlarx(linmodel)` uses a linear model `linmodel` to extract certain properties such as names, units, and sample time, and to initialize the values of the linear coefficients of the model. Use this syntax when you want to create a nonlinear ARX model as an extension of, or an improvement upon, an existing linear model.

`sys = idnlarx(linmodel,OutputFcn)` specifies the output function `OutputFcn` that maps the regressors to the model output.

Specify Model Properties

`sys = idnlarx( ___ ,Name,Value)` specifies additional properties on page 1-607 of the `idnlarx` model structure using one or more name-value arguments.

Input Arguments

`orders` — ARX model orders
`nlarx` orders `[na nb nk]`

ARX model orders, specified as the matrix `[na nb nk]`. `na` denotes the number of delayed outputs, `nb` denotes the number of delayed inputs, and `nk` denotes the minimum input delay. The minimum output delay is fixed to `1`. For more information on how to construct the `orders` matrix, see `arx`.

When you specify `orders`, the software converts the order information into a linear regressor form in the `idnlarx` `Regressors` property. For an example, see “Create Nonlinear ARX Model Using ARX Model Orders” on page 1-614.

`linmodel` — Discrete-time linear model
`idpoly` object | `idss` object | `idtf` object | `idproc` object

Discrete-time identified input/output linear model, specified as any linear model created using estimators, that is, an `idpoly` object, an `idss` object, an `idtf` object, or an `idproc` object with `Ts` > 0. Create this model using the constructor function for the object or estimate the model using the associated estimation command. For example, to create an ARX model, use `arx`, and specify the resulting `idpoly` object as `linmodel`.

Properties

`Regressors` — Regressor specification
`linearRegressor` object | `polynomialRegressor` object | `customRegressor` object | column array of regressor specification objects

Regressor specification, specified as a column vector containing one or more regressor specification objects, which are the `linearRegressor` objects, `polynomialRegressor` objects, and `customRegressor` `customRegressor` objects. Each object specifies a formula for generating regressors from lagged variables. For example:

- `L = linearRegressor({'y1','u1'},{1,[2 5]})` generates the regressors $y_1(t–1)$, $u_1(t–2)$, and $u_2(t–5)$.
- `P = polynomialRegressor('y2',4:7,2)` generates the regressors $y_2(t–4)^2$, $y_2(t–5)^2$,$y_2(t–6)^2$, and $y_2(t–7)^2$.
- `C = customRegressor({'y1','u1','u2'},{1 2 2},@(x,y,z)sin(x.*y+z))` generates the single regressor $\sin(y_1(t–1)u_1(t–2)+u_2(t–2)$

  .

For an example that implements these regressors, see “Create and Combine Regressor Types” on page 1-618.

To add regressors to an existing model, create a vector of specification objects and use dot notation to set `Regressors` to this vector. For example, the following code first creates the `idnlarx` model `sys` and then adds the regressor objects `L`, `P`, and `C` to the regressors of `sys`.

```
sys = idnlarx({'y1','y2'},{'u1','u2'});
R = [L;P;C];
sys.Regressors = R;
```

For an example of creating and using a linear regressor set, see “Create Nonlinear ARX Model Using Linear Regressors” on page 1-615.

`OutputFcn` — Output function
`'idWaveletNetwork'` (default) | `'idLinear'` | `[]` | `''` | `'idSigmoidNetwork'` | `'idTreePartition'` | `'idGaussianProcess'` | `'idTreeEnsemble'` | mapping object | array of mapping objects

Output function that maps the regressors of the `idnlarx` model into the model output, specified as a column array containing zero or more of the following strings or mapping objects:

`'idWaveletNetwork'` or `idWaveletNetwork` object	Wavelet network
`'idLinear'` or `''` or `[]` or `idLinear` object	Linear function
`'idSigmoidNetwork'` or `idSigmoidNetwork` object	Sigmoid network
`'idTreePartition'` or `idTreePartition` object	Binary tree partition regression model
`'idGaussianProcess'` or `idGaussianProcess` object	Gaussian process regression model (requires Statistics and Machine Learning Toolbox)
`'idTreeEnsemble'` or `idTreeEnsemble`	Regression tree ensemble model(Statistics and Machine Learning Toolbox)
`idFeedforwardNetwork` object	Neural network — Multilayer feedforward network of Deep Learning Toolbox.
`idCustomNetwork` object	Custom network — Similar to `idSigmoidNetwork`, but with a user-defined replacement for the sigmoid function.

The `idWaveletNetwork`, `idSigmoidNetwork`, `idTreePartition`, and `idCustomNetwork` objects contain both linear and nonlinear components. You can remove (not use) the linear components of `idWaveletNetwork`, `idSigmoidNetwork`, and `idCustomNetwork` by setting the `LinearFcn.Use` value to `false`.

The `idFeedforwardNetwork` function has only a nonlinear component that is the `network` object of Deep Learning Toolbox. The `idLinear` function, as the name implies, has only a linear component.

Specifying a character vector, for example `'idSigmoidNetwork'`, creates a mapping object with default settings. Alternatively, you can specify mapping object properties in two other ways:

- Create the mapping object using arguments to modify default properties.

  ```
  MO = idSigmoidNetwork(15)
  ```
- Create a default mapping object first and then use dot notation to modify properties.

  ```
  MO = idSigmoidNetwork;
  MO.NumberOfUnits = 15
  ```

For n_y output channels, you can specify mapping objects individually for each channel by setting `OutputFcn` to an array of n_y mapping objects. For example, the following code specifies `OutputFcn` using dot notation for a system with two input channels and two output channels.

```
sys = idnlarx({'y1','y2'},{'u1','u2'});
sys.OutputFcn = [idWaveletNetwork; idSigmoidNetwork]
```

To specify the same mapping for all outputs, specify `OutputFcn` as a character vector or a single mapping object.

`OutputFcn` represents a static mapping function that transforms the regressors of the nonlinear ARX model into the model output. `OutputFcn` is static because it does not depend on the time. For example, if $y(t) = y_0 + a_1y(t-1) + a_2y(t-2) + \ldots + b_1u(t-1) + b_2u(t-2) + \ldots$, then `OutputFcn` is a linear function represented by the `idLinear` object.

For an example of specifying the output function, see “Specify Output Function for Nonlinear ARX Model” on page 1-617.

`RegressorUsage` — Regressor assignments

table with logical entries

Regressor assignments to the linear and nonlinear components of the nonlinear ARX model, specified as an n_r-by-n_c table with logical entries that specify which regressors to use for which component. Here, n_r is the number of regressors. n_c is the total number of linear and nonlinear components in `OutputFcn`. The rows of the table correspond to individual regressors. The row names are set to regressor names. If the table value for row i and component index j is `true`, then the ith regressor is an input to the linear or nonlinear component j.

For multi-output systems, `OutputFcn` contains one mapping object for each output. Each mapping object can use both linear and nonlinear components or only one of the two components.

For an example of viewing and modifying the `RegressorUsage` property, see “Modify Regressor Assignments to Output Function Components” on page 1-621.

`Report` — Summary report

report field values

This property is read-only.

Summary report that contains information about the estimation options and results for a nonlinear ARX model obtained using the `nlarx` command. Use `Report` to find estimation information for the identified model, including:

- Estimation method

- Estimation options
- Search termination conditions
- Estimation data fit

The contents of `Report` are irrelevant if the model was constructed using `idnlarx`.

```
sys = idnlarx('y1','u1',reg);
sys.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;
sys = nlarx(z1,reg);
m.Report.OptionsUsed

Option set for the nlarx command:

  IterativeWavenet: 'auto'
             Focus: 'prediction'
           Display: 'off'
    Regularization: [1x1 struct]
      SearchMethod: 'auto'
     SearchOptions: [1x1 idoptions.search.identsolver]
      OutputWeight: 'noise'
          Advanced: [1x1 struct]
```

For more information on this property and how to use it, see "Output Arguments" on page 1-998 in the `nlarx` reference page and "Estimation Report".

TimeVariable — Independent variable

`'t'` (default) | character vector

Independent variable for the inputs, outputs, and—when available—internal states, specified as a character vector.

NoiseVariance — Noise variance

matrix

Noise variance (covariance matrix) of the model innovations *e*. The estimation algorithm typically sets this property. However, you can also assign the covariance values by specifying an `ny`-by-`ny` matrix.

Ts — Sample time

1 (default) | positive scalar

Sample time, specified as a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model.

TimeUnit — Units for time variable

`'seconds'` (default) | `'nanoseconds'` | `'microseconds'` | `'milliseconds'` | `'minutes'` | `'hours'` | `'days'` | `'weeks'` | `'months'` | `'years'`

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.

`InputName` — Input channel names
`''` for all input channels (default) | character vector | cell array of character vectors

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.

Input names in Nonlinear ARX models must be valid MATLAB variable names after you remove any spaces.

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

`InputUnit` — Input channel units
`''` for all input channels (default) | character vector | cell array of character vectors

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.

`InputGroup` — Input channel groups

structure with no fields (default) | structure

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')
```

`OutputName` — Output channel names

`''` for all output channels (default) | character vector | cell array of character vectors

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.

Output names in Nonlinear ARX models must be valid MATLAB variable names after you remove any spaces.

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

`OutputUnit` — Output channel units

`''` for all output channels (default) | character vector | cell array of character vectors

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.

`OutputGroup` — Output channel groups

structure with no fields (default) | structure

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',:)
```

Name — System Name

`''` (default) | character vector

System name, specified as a character vector. For example, `'system 1'`.

Notes — Notes on system

`0`-by-`1` string (default) | string | character vector

Any text that you want to associate with the system, specified as a string or a cell array of character vectors. The property stores whichever data type you provide. For instance, if `sys1` and `sys2` are dynamic system models, you can set their `Notes` properties as follows.

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes


ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

`UserData` — Data to associate with system

`[]` (default) | any MATLAB data type

Any data you want to associate with the system, specified as any MATLAB data type.

Object Functions

For information about object functions for `idnlarx`, see “Nonlinear ARX Models”.

Examples

Create Nonlinear ARX Model Using ARX Model Orders

Create an `idnlarx` model by specifying an ARX model order vector.

Create an order vector of the form `[na nb nk]`, where `na` and `nb` are the orders of the *A* and *B* ARX model polynomials and `nk` is the number of input/output delays.

```
na = 2;
nb = 3;
nk = 5;
orders = [na nb nk];
```

Construct a nonlinear ARX model `sys`.

```
output_name = 'y1';
input_name = 'u1';

sys = idnlarx(output_name,input_name,[2 3 5]);
```

View the output function.

```
disp(sys.OutputFcn)
```

```
Wavelet Network

 Nonlinear Function: Wavelet network with number of units chosen automatically
 Linear Function: uninitialized
 Output Offset: uninitialized

               Input: '<Function inputs>'
              Output: '<Function output>'
           LinearFcn: '<Linear function parameters>'
        NonlinearFcn: '<Wavelet and scaling function units and their parameters>'
              Offset: '<Offset parameters>'
   EstimationOptions: '<Estimation options>'
```

By default, the model uses a wavelet network, represented by a `idWaveletNetwork` object, for the output function. The `idWaveletNetwork` object includes linear and nonlinear components.

View the `Regressors` property.

```
disp(sys.Regressors)
```

```
Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1 2]  [5 6 7]}
     UseAbsolute: [0 0]
    TimeVariable: 't'
```

The `idnlarx` constructor transforms the model orders into the `Regressors` form.

- The Lags array for y1, [1 2], is equivalent to the na value of 2. Both forms specify two consecutive output regressors, y1(t-1) and y1(t-2).
- The Lags array for u1, [5 6 7], incorporates the three delays specified by the nb value of 3, and shifts them by the nk value of 5. The input regressors are therefore u1(t-5), u1(t-6), and u1(t-7).

View the regressors.

```
getreg(sys)
```

```
ans = 5x1 cell
    {'y1(t-1)'}
    {'y1(t-2)'}
    {'u1(t-5)'}
    {'u1(t-6)'}
    {'u1(t-7)'}
```

You can use the orders syntax to specify simple linear regressors. However, to create more complex regressors, use the regressor commands linearRegressor, polynomialRegressor, and customRegressor to create a combined regressor for the Regressors syntax.

Create Nonlinear ARX Model Using Linear Regressors

Construct an idnlarx model by specifying linear regressors.

Create a linear regressor that contains two output lags and two input lags.

```
output_name = 'y1';
input_name = 'u1';
var_names = {output_name,input_name};

output_lag = [1 2];
input_lag = [1 2];
lags = {output_lag,input_lag};

reg = linearRegressor(var_names,lags)
```

```
reg =
Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1 2]  [1 2]}
     UseAbsolute: [0 0]
    TimeVariable: 't'

  Regressors described by this set
```

The model contains the regressors y(t-1), y(t-2), u(t-1), and u(t-2).

Construct the idnlarx model and view the regressors.

```
sys = idnlarx(output_name,input_name,reg);
getreg(sys)
```

```
ans = 4x1 cell
    {'y1(t-1)'}
    {'y1(t-2)'}
    {'u1(t-1)'}
    {'u1(t-2)'}
```

View the output function.

```
disp(sys.OutputFcn)
```

```
Wavelet Network

 Nonlinear Function: Wavelet network with number of units chosen automatically
 Linear Function: uninitialized
 Output Offset: uninitialized

                Input: '<Function inputs>'
               Output: '<Function output>'
            LinearFcn: '<Linear function parameters>'
         NonlinearFcn: '<Wavelet and scaling function units and their parameters>'
               Offset: '<Offset parameters>'
    EstimationOptions: '<Estimation options>'
```

View the regressor usage table.

```
disp(sys.RegressorUsage)
```

```
               y1:LinearFcn    y1:NonlinearFcn
               ____________    _______________

    y1(t-1)       true              true      
    y1(t-2)       true              true      
    u1(t-1)       true              true      
    u1(t-2)       true              true      
```

All the regressors are inputs to both the linear and nonlinear components of the `wavenet` object.

Create and Configure Nonlinear ARX Model

Create a nonlinear ARX model with a linear regressor set.

Create a linear regressor that contains three output lags and two input lags.

```
output_name = 'y1';
input_name = 'u1';
var_names = {output_name,input_name};

output_lag = [1 2 3];
input_lag = [1 2];
lags = {output_lag,input_lag};

reg = linearRegressor(var_names,lags)
```

```
reg = 
Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1 2 3]  [1 2]}
```

```
     UseAbsolute: [0 0]
    TimeVariable: 't'

  Regressors described by this set
```

Construct the nonlinear ARX model.

```
sys = idnlarx(output_name,input_name,reg);
```

View the `Regressors` property.

```
disp(sys.Regressors)
```

```
Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1 2 3]  [1 2]}
     UseAbsolute: [0 0]
    TimeVariable: 't'
```

`sys` uses `idWavenetNetwork` as the default output function. Reconfigure the output function to `idSigmoidNetwork`.

```
sys.OutputFcn = 'idSigmoidNetwork';
disp(sys.OutputFcn)
```

```
Sigmoid Network

 Nonlinear Function: Sigmoid network with 10 units
 Linear Function: uninitialized
 Output Offset: uninitialized

           Input: '<Function inputs>'
          Output: '<Function output>'
       LinearFcn: '<Linear function parameters>'
    NonlinearFcn: '<Sigmoid units and their parameters>'
          Offset: '<Offset parameters>'
```

Specify Output Function for Nonlinear ARX Model

Specify the sigmoid network output function when you construct a nonlinear ARX model.

Assign variable names and specify a regressor set.

```
output_name = 'y1';
input_name = 'u1';
r = linearRegressor({output_name,input_name},{1 1});
```

Construct a nonlinear ARX model that specifies the `idSigmoidNetwork` output function. Set the number of terms in the sigmoid expansion to `15`.

```
sys = idnlarx(output_name,input_name,r,idSigmoidNetwork(15));
```

View the output function specification.

```
disp(sys.OutputFcn)
```

```
Sigmoid Network

 Nonlinear Function: Sigmoid network with 15 units
 Linear Function: uninitialized
 Output Offset: uninitialized

           Input: '<Function inputs>'
          Output: '<Function output>'
       LinearFcn: '<Linear function parameters>'
    NonlinearFcn: '<Sigmoid units and their parameters>'
          Offset: '<Offset parameters>'
```

Create Nonlinear ARX Model Without Nonlinear Mapping Function

Construct an `idnlarx` model that uses only linear mapping in the output function. An argument value of `[]` is equivalent to an argument value of `idLinear`.

```
sys = idnlarx([2 2 1],[])
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: None
Sample time: 1 seconds

Status:
Created by direct construction or transformation. Not estimated.
```

Create and Combine Regressor Types

Create a regressor set that includes linear, polynomial, and custom regressors.

Specify `L` as the set of linear regressors $y_1(t-1)$, $u_1(t-2)$, and $u_1(t-5)$.

```
L = linearRegressor({'y1','u1'},{1, [2 5]});
```

Specify `P` as the set of polynomial regressors $y_2(t-4)^2$, $y_2(t-5)^2$, $y_2(t-6)^2$, and $y_2(t-7)^2$.

```
P = polynomialRegressor('y2',4:7,2);
```

Specify `C` as the custom regressor $\sin(y_1(t-1)u_1(t-2)+u_2(t-2))$, using the `@` symbol to create an anonymous function handle.

```
C = customRegressor({'y1','u1','u2'},{1 2 2},@(x,y,z)sin(x.*y+z));
```

Combine the regressors into one regressor set `R`.

```
R = [L;P;C]
```

```
R = 
[3 1] array of linearRegressor, polynomialRegressor, customRegressor objects.
------------------------------------
1. Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1]  [2 5]}
     UseAbsolute: [0 0]
    TimeVariable: 't'

------------------------------------
2. Order 2 regressors in variables y2
               Order: 2
           Variables: {'y2'}
                Lags: {[4 5 6 7]}
         UseAbsolute: 0
    AllowVariableMix: 0
         AllowLagMix: 0
        TimeVariable: 't'

------------------------------------
3. Custom regressor: sin(y1(t-1).*u1(t-2)+u2(t-2))
    VariablesToRegressorFcn: @(x,y,z)sin(x.*y+z)
                  Variables: {'y1'  'u1'  'u2'}
                       Lags: {[1]  [2]  [2]}
                 Vectorized: 1
               TimeVariable: 't'


Regressors described by this set
```

Create a nonlinear ARX model.

```
sys = idnlarx({'y1','y2'},{'u1','u2'},R)
```

```
sys =
Nonlinear ARX model with 2 outputs and 2 inputs
  Inputs: u1, u2
  Outputs: y1, y2

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables y2
  3. Custom regressor: sin(y1(t-1).*u1(t-2)+u2(t-2))
  List of all regressors

Output functions:
  Output 1:  Wavelet network with number of units chosen automatically
  Output 2:  Wavelet network with number of units chosen automatically

Sample time: 1 seconds

Status:
Created by direct construction or transformation. Not estimated.
```

Create Nonlinear ARX Model Using Linear Model

Use a linear ARX model instead of a regressor set to construct a nonlinear ARX model.

Construct a linear ARX model using `idpoly`.

```
A = [1 -1.2 0.5];
B = [0.8 1];
LinearModel = idpoly(A, B, 'Ts', 0.1);
```

Specify input and output names for the model using dot notation.

```
LinearModel.OutputName = 'y1';
LinearModel.InputName = 'u1';
```

Construct a nonlinear ARX model using the linear ARX model.

```
m1 = idnlarx(LinearModel)
```

```
m1 =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Wavelet network with number of units chosen automatically
Sample time: 0.1 seconds

Status:
Created by direct construction or transformation. Not estimated.
```

You can create a linear ARX model from any identified discrete-time linear model.

Estimate a second-order state-space model from estimation data `z1`.

```
load iddata1 z1
ssModel = ssest(z1,2,'Ts',0.1);
```

Construct a nonlinear ARX model from `ssModel`. The software uses the input and output names that `ssModel` extracts from `z1`.

```
m2 = idnlarx(ssModel)
```

```
m2 =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Wavelet network with number of units chosen automatically
Sample time: 0.1 seconds
```

```
Status:
Created by direct construction or transformation. Not estimated.
```

Modify Regressor Assignments to Output Function Components

Modify regressor assignments by modifying the `RegressorUsage` table.

Construct a nonlinear ARX model that has two inputs and two outputs.

Create the variable names and the regressors.

```
varnames = {'y1','y2','u1','u2'};
lags = {[1 2 3],[1 2],[1 2],[1 3]};
reg = linearRegressor(varnames,lags);
```

Create an output function specification `fcn` that uses `idWaveletNetwork` for mapping regressors to output `y1` and `idSigmoidNetwork` for mapping regressors to output `y2`. Both mapping objects contain linear and nonlinear components.

```
fcn = [idWaveletNetwork;idSigmoidNetwork];
```

Construct the nonlinear ARX model.

```
output_name = {'y1' 'y2'};
input_name = {'u1' 'u2'};
sys = idnlarx(output_name,input_name,reg,fcn)
```

```
sys =
Nonlinear ARX model with 2 outputs and 2 inputs
  Inputs: u1, u2
  Outputs: y1, y2

Regressors:
  Linear regressors in variables y1, y2, u1, u2
  List of all regressors

Output functions:
  Output 1:  Wavelet network with number of units chosen automatically
  Output 2:  Sigmoid network with 10 units

Sample time: 1 seconds

Status:
Created by direct construction or transformation. Not estimated.
```

Display the `RegressorUsage` table.

```
disp(sys.RegressorUsage)
```

```
               y1:LinearFcn    y1:NonlinearFcn    y2:LinearFcn    y2:NonlinearFcn
               ____________    _______________    ____________    _______________

    y1(t-1)       true              true              true             true
    y1(t-2)       true              true              true             true
    y1(t-3)       true              true              true             true
    y2(t-1)       true              true              true             true
    y2(t-2)       true          true             true              true
    u1(t-1)       true          true             true              true
    u1(t-2)       true          true             true              true
    u2(t-1)       true          true             true              true
    u2(t-3)       true          true             true              true
```

The rows of the table represent the regressors. The first two columns of the table represent the linear and nonlinear components of the mapping to output `y1` (`idWaveletNetwork`). The last two columns represent the two components of the mapping to output `y2 (idSigmoidNetwork)`.

In this table, all the input and output regressors are inputs to all components.

Remove the `y2(t-2)` regressor from the `y2` nonlinear component.

```
sys.RegressorUsage{4,4} = false;
disp(sys.RegressorUsage)
```

```
               y1:LinearFcn    y1:NonlinearFcn    y2:LinearFcn    y2:NonlinearFcn
               ____________    _______________    ____________    _______________

    y1(t-1)       true          true             true              true
    y1(t-2)       true          true             true              true
    y1(t-3)       true          true             true              true
    y2(t-1)       true          true             true              false
    y2(t-2)       true          true             true              true
    u1(t-1)       true          true             true              true
    u1(t-2)       true          true             true              true
    u2(t-1)       true          true             true              true
    u2(t-3)       true          true             true              true
```

The table displays `false` for this regressor-component pair.

Store the regressor names in `Names`.

```
Names = sys.RegressorUsage.Properties.RowNames;
```

Determine the indices of the rows that contain `y1` or `y2` and set the corresponding values of `y1:NonlinearFcn` to `False`.

```
idx = contains(Names,'y1')|contains(Names,'y2');
sys.RegressorUsage{idx,2} = false;
disp(sys.RegressorUsage)
```

```
               y1:LinearFcn    y1:NonlinearFcn    y2:LinearFcn    y2:NonlinearFcn
               ____________    _______________    ____________    _______________

    y1(t-1)       true          false            true              true
    y1(t-2)       true          false            true              true
    y1(t-3)       true          false            true              true
    y2(t-1)       true          false            true              false
    y2(t-2)       true          false            true              true
    u1(t-1)       true          true             true              true
    u1(t-2)       true          true             true              true
    u2(t-1)       true          true             true              true
    u2(t-3)       true          true             true              true
```

The table values reflect the new assignments.

The `RegressorUsage` table provides complete flexibility for individually controlling regressor assignments.

More About

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 `sim`, `predict`, and `compare` use for simulation and prediction. `linearize` also uses this definition for linearization of nonlinear ARX models.

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*–1), *p*(*t*–2), ..., *p*(*t*–*D*).

For example, if you have a single-input, single-output `idnlarx` model.

```
m = idnlarx([2 3 0],'idWaveletNetwork','CustomRegressors',{'y1(t-10)*u1(t-1)'});
```

This model has these regressors.

```
getreg(m)
```

```
ans = 6x1 cell
    {'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)]
```
(1-1)

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],[idWaveletNetwork; idLinear]);
```

This model has these regressors.

```
getreg(m)
```

```
ans = 11x1 cell
    {'y1(t-1)'}
    {'y1(t-2)'}
    {'u1(t-1)'}
    {'u1(t-2)'}
    {'u2(t)'  }
    {'u2(t-1)'}
    {'u2(t-2)'}
    {'u2(t-3)'}
    {'u2(t-4)'}
    {'u2(t-5)'}
    {'u3(t)'  }
```

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 multi-input multi-output model `m`, `getDelayInfo` returns:

```
maxDel = getDelayInfo(m)
```

```
maxDel = 1×5

     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.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`

Pre-R2021b Name	**R2021b Name**
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

Use of previous `idnlarx` properties is not recommended.
Not recommended starting in R2021a

Starting in R2021a, several properties of `idnlarx` have been modified or replaced.

These changes affect the syntaxes in both `idnlarx` and `nlarx`. The use of the pre-R2021a properties in the following table is discouraged. However, the software still accepts calling syntaxes that include these properties. There are no plans to exclude these syntaxes at this time. The command syntax that uses ARX model orders continues be a recommended syntax.

Pre-R2021a Property	**R2021a Property**	**Usage**
ARX model orders `na,nb,nk`	`Regressors`, which can include `linearRegressor`, `polynomialRegressor`, and `customRegressor` objects.	`na,nb,nk` remains a valid `idnlarx` and `nlarx` input argument that the software converts to a `linearRegressor` object. You can no longer change order values in an existing `idnlarx` model by dot assignment or by using the `set` function. Create a new model object instead.
`customRegressors`	`Regressors`	Use `polynomialRegressor` or `customRegressor` to create regressor objects and add the objects to the `Regressors` array.
`NonlinearRegressors`	`RegressorUsage`	`RegressorUsage` is a table that contains regressor assignments to linear and nonlinear output components. Change assignments by modifying the corresponding `RegressorUsage`table entries.

Pre-R2021a Property	R2021a Property	Usage
`Nonlinearity`	`OutputFcn`	Change is in name only. Property remains an object or an array or objects that map regressor inputs to an output.

See Also

`linearRegressor` | `polynomialRegressor` | `customRegressor` | `idnlarx/findop` | `getreg` | `linearize` | `nlarx` | `pem` | `idSigmoidNetwork` | `idWaveletNetwork` | `idLinear` | `idCustomNetwork` | `idFeedforwardNetwork`

Topics

"Use nlarx to Estimate Nonlinear ARX Models"
"Estimate Nonlinear ARX Models Initialized Using Linear ARX Models"
"Initialize Nonlinear ARX Estimation Using Linear Model"
"Identifying Nonlinear ARX Models"
"Nonlinear Model Structures"

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 1-632.

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);
```

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	

Field	Description	Default
`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

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.	`''`

Field	Description	Default
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))
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 ith initial state. For example, for idnlgrey model M, the ith 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 ith 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 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 on page 1-632 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 • Finite real column vector • Two-dimensional real matrix	

Field	Description	Default
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 ith parameter. For example, for idnlgrey model M, the ith 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.	''

Field	Description	Default
`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))`
`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. • A positive real value > `MinStep`.	`'Auto'` Automatically chooses the time step.

Field	Description	Default
`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'` • General: `'Auto'` — Automatically chooses one of the previous solvers.	`'Auto'` Automatically chooses one of the 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]
 EstimateCovariance: 1
            Display: 'off'
     Regularization: [1x1 struct]
       SearchMethod: 'auto'
      SearchOptions: [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 1-1017 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, stored as a string or a cell array of character vectors. The property stores whichever data type you provide. For instance, if `sys1` and `sys2` are dynamic system models, you can set their `Notes` properties as follows:

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

Default: `[0×1 string]`

`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.

See Also

`nlgreyest` | `pem` | `get` | `set` | `getinit` | `setinit` | `getpar` | `setpar`

Topics
"Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation"
"Creating IDNLGREY Model Files"
"Estimate Nonlinear Grey-Box Models"

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 on page 1-653 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 1-647.

Examples

Create a Hammerstein-Wiener Model Structure with Default Nonlinearities

Create a Hammerstein-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],'idSigmoidNetwork','idDeadZone');
```

The above is equivalent to:

```
m = idnlhw([2 2 1],'idsig','iddead');
```

The specified nonlinearities have a default configuration.

Create Hammerstein-Wiener Model and Configure the Nonlinearities

```
m = idnlhw([2 2 1],idSigmoidNetwork('num',5),idDeadZone([-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],[],'idSaturation');
```

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,'idSaturation',[],'InputName','Control');
```

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
`'idPiecewiseLinear'` (default) | `'idSigmoidNetwork'` | `'idWaveletNetwork'` | `'idSaturation'` | `'idDeadZone'` | `'idPolynomial1D'` | `'idUnitGain'` | nonlinearity estimator object | array of nonlinearity estimators

Input static nonlinearity estimator, specified as one of the following.

`'idPiecewiseLinear'` or `idPiecewiseLinear` object (default)	Piecewise linear function
`'idSigmoidNetwork'` or `idSigmoidNetwork` object	Sigmoid network
`'idWaveletNetwork'` or `idWaveletNetwork` object	Wavelet network
`'idSaturation'` or `idSaturation` object	Saturation
`'idDeadZone'` or `idDeadZone` object	Dead zone
`'idPolynomial1D'` or `idPolynomial1D` object	One-dimensional polynomial
`'idUnitGain'` or `[]` or`idUnitGain` object	Unit gain
`idCustomNetwork` object	Custom network — Similar to `idSigmoidNetwork`, but with a user-defined replacement for the sigmoid function.

Specifying a character vector, for example `'idSigmoidNetwork'`, creates a nonlinearity estimator object with default settings. Use object representation instead to configure the properties of a nonlinearity estimator.

```
InputNL = idWaveletNetwork;
InputNL.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
InputNL = idWaveletNetwork('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 = [idSigmoidNetwork('NumberofUnits',5); idDeadZone([-1,2])]
```

To specify the same nonlinearity for all inputs, specify a single input nonlinearity estimator.

OutputNL — Output static nonlinearity
'idPiecewiseLinear' (default) | 'idSigmoidNetwork' | 'idWaveletNetwork' | 'idSaturation' | 'idDeadZone' | 'idPolynomial1D' | 'idUnitGain' | nonlinearity estimator object | array of nonlinearity estimators

Output static nonlinearity estimator, specified as one of the following:

`'idPiecewiseLinear'` or `idPiecewiseLinear` object (default)	Piecewise linear function
`'idSigmoidNetwork'` or `idSigmoidNetwork` object	Sigmoid network
`'idWaveletNetwork'` or `idWaveletNetwork` object	Wavelet network
`'idSaturation'` or `idSaturation` object	Saturation
`'idDeadZone'` or `idDeadZone` object	Dead zone
`'idPolynomial1D'` or `idPolynomial1D` object	One-dimensional polynomial
`'idUnitGain'` or `[]` or`idUnitGain` object	Unit gain
`idCustomNetwork` object	Custom network — Similar to `idSigmoidNetwork`, 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 = idSigmoidNetwork;
OutputNL.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
OutputNL = idSigmoidNetwork('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 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 on page 1-647 of `idnlhw` models during model creation. For example, `m = idnlhw([2 3 1],'idPiecewiseLinear','idWaveletNetwork','InputName','Volts','Ts',0.1)` creates an `idnlhw` model object with input nonlinearity estimator `'idPiecewiseLinear'`, output nonlinearity estimator `'idWaveletNetwork'`, 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 on page 1-653, 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 on page 1-653, 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.

`Bfree`

Option to fix or free the parameters of the *B* polynomial, 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 `Bfree(i,j)` is a row vector representing the numerator polynomial for the `j`th input to `i`th output transfer function. `Bfree(i,j) = false` causes the numerator of the linear transfer function between the input `j` and output `i` to be fixed to `B(i,j)}`. The software honors the `Bfree` specification only if the *B* polynomial contains finite values.

`Ffree`

Option to fix or free the parameters of the *F* polynomial, 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 `Ffree(i,j)` is a row vector representing the numerator polynomial for the `j`th input to `i`th output transfer function. `Ffree(i,j) = false` causes the numerator of the linear transfer function between the input `j` and output `i` to be fixed to `F(i,j)`. The software honors the `Ffree` specification only if the *F* polynomial contains finite values.

`InputNonlinearity`

Input nonlinearity estimator, specified as one of the following:

`'idPiecewiseLinear'` or `idPiecewiseLinear` object (default)	Piecewise linear function
`'idSigmoidNetwork'` or `idSigmoidNetwork` object	Sigmoid network
`'idWaveletNetwork'` or `idWaveletNetwork` object	Wavelet network
`'idSaturation'` or `idSaturation` object	Saturation
`'idDeadZone'` or `idDeadZone` object	Dead zone
`'idPolynomial1D'` or `idPolynomial1D` object	One-dimensional polynomial
`'idUnitGain'` or `[]` or`idUnitGain` object	Unit gain
`idCustomNetwork` object	Custom network — Similar to `idSigmoidNetwork`, 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.

```
InputNonlinearity = idWaveletNetwork;
InputNonlinearity.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
InputNonlinearity = idWaveletNetwork('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:

`'idPiecewiseLinear'` or `idPiecewiseLinear` object (default)	Piecewise linear function
`'idSigmoidNetwork'` or `idSigmoidNetwork` object	Sigmoid network
`'idWaveletNetwork'` or `idWaveletNetwork` object	Wavelet network
`'idSaturation'` or `idSaturation` object	Saturation
`'idDeadZone'` or `idDeadZone` object	Dead zone
`'idPolynomial1D'` or `idPolynomial1D` object	One-dimensional polynomial
`'idUnitGain'` or `[]` or`idUnitGain` object	Unit gain
`idCustomNetwork` object	Custom network — Similar to `idSigmoidNetwork`, 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.

```
OutputNonlinearity = idSigmoidNetwork;
OutputNonlinearity.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
OutputNonlinearity = idSigmoidNetwork('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 1-1038 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, stored as a string or a cell array of character vectors. The property stores whichever data type you provide. For instance, if `sys1` and `sys2` are dynamic system models, you can set their `Notes` properties as follows:

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

Default: `[0×1 string]`

`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,

- *f* is a nonlinear function that transforms input data $u(t)$ as $w(t) = f(u(t))$.

  $w(t)$, an internal variable, is the output of the Input Nonlinearity block and has the same dimension as $u(t)$.
- *B*/*F* is a linear transfer function that transforms $w(t)$ as $x(t) = (B/F)w(t)$.

$x(t)$, an internal variable, is the output of the Linear block and has the same dimension as $y(t)$.

B and F are similar to polynomials in a linear Output-Error model. For more information about Output-Error models, see “What Are Polynomial Models?”.

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`.
- h is a nonlinear function that maps the output of the linear block $x(t)$ to the system output $y(t)$ as $y(t) = h(x(t))$.

Because f acts on the input port of the linear block, this function is called the *input nonlinearity*. Similarly, because h acts on the output port of the linear block, this function is called the *output nonlinearity*. If your system contains several inputs and outputs, you must define the functions f and h for each input and output signal. You do not have to include both the input and the output nonlinearity in the model structure. When a model contains only the input nonlinearity f, it is called a Hammerstein model. Similarly, when the model contains only the output nonlinearity h, it is called a *Wiener* model.

The software computes the Hammerstein-Wiener model output y in three stages:

1. Compute $w(t) = f(u(t))$ from the input data.

   $w(t)$ is an input to the linear transfer function B/F.

   The input nonlinearity is a static (*memoryless*) function, where the value of the output a given time t depends only on the input value at time t.

   You can configure the input nonlinearity as a sigmoid network, wavelet network, saturation, dead zone, piecewise linear function, one-dimensional polynomial, or a custom network. You can also remove the input nonlinearity.
2. Compute the output of the linear block using $w(t)$ and initial conditions: $x(t) = (B/F)w(t)$.

   You can configure the linear block by specifying the orders of numerator B and denominator F.
3. Compute the model output by transforming the output of the linear block $x(t)$ using the nonlinear function h as $y(t) = h(x(t))$.

   Similar to the input nonlinearity, the output nonlinearity is a static function. You can configure the output nonlinearity in the same way as the input nonlinearity. You can also remove the output nonlinearity, such that $y(t) = x(t)$.

Resulting models are `idnlhw` objects that store all model data, including model parameters and nonlinearity estimators. For more information about these objects, see “Nonlinear Model Structures”.

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.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

See Also

`idCustomNetwork` | `idLinear` | `linearize` | `nlhw` | `findop` | `pem` | `idPolynomial1D` | `idSaturation` | `idSigmoidNetwork` | `idWaveletNetwork`

Topics
"Estimate Multiple Hammerstein-Wiener Models"
"Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models"
"Identifying Hammerstein-Wiener Models"
"Initialize Hammerstein-Wiener Estimation Using Linear Model"

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.

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

`predict` | `findstates` | `findstatesOptions` | `forecast` | `ssest` | `pem`

Introduced in R2012a

idPiecewiseLinear

Create a piecewise-linear nonlinearity estimator object

Syntax

```
NL = idPiecewiseLinear
NL = idPiecewiseLinear(Name,Value)
```

Description

`NL = idPiecewiseLinear` 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 = idPiecewiseLinear(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

`idPiecewiseLinear` is an object that stores the piecewise-linear nonlinearity estimator for estimating Hammerstein-Wiener models.

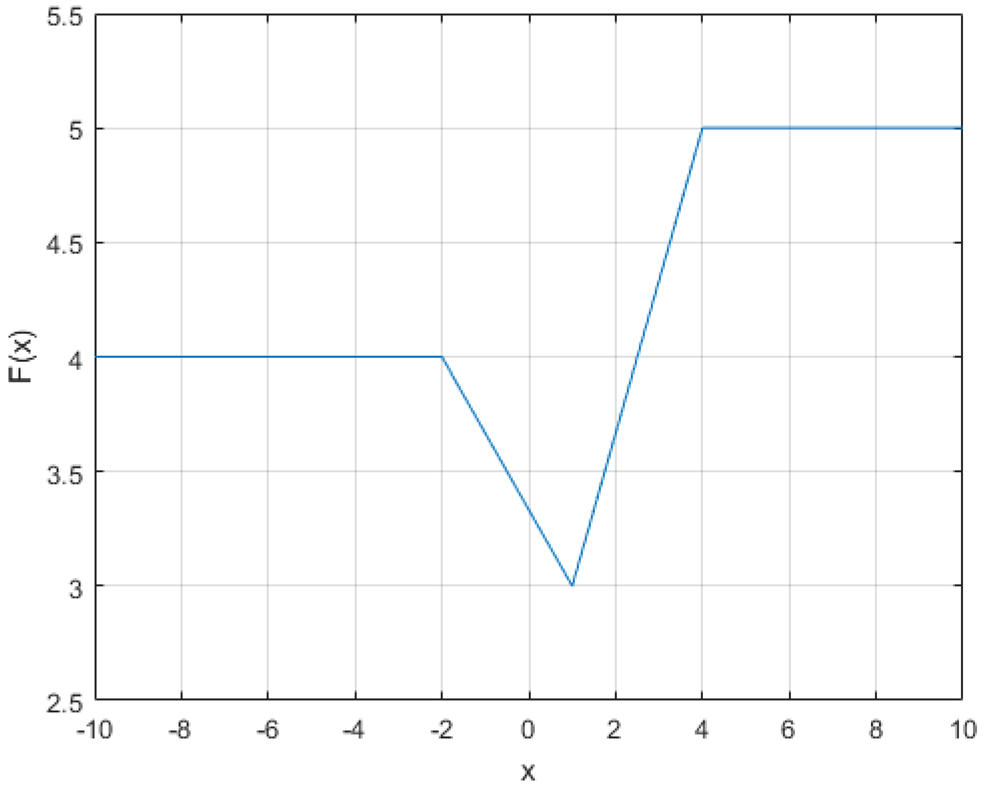
Use `idPiecewiseLinear` to define a nonlinear function $y = F(x,\theta)$, where y and x are scalars, and θ represents the parameters specifying the number of break points and the value of nonlinearity at the break points.

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 `idPiecewiseLinear` object. When using `evaluate`, the break points have to be initialized manually.

For `idPiecewiseLinear` object properties, see “Properties” on page 1-662.

Examples

Create a Default Piecewise-Linear Nonlinearity Estimator

```
NL = idPiecewiseLinear;
```

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 an `idPiecewiseLinear` object, and specify the breakpoints.

```
InputNL = idPiecewiseLinear('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 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 on page 1-647 of `idPiecewiseLinear` nonlinearity. For example, `NL= idPiecewiseLinear('NumberofUnits',5)` creates a piecewise-linear nonlinearity estimator object with 5 breakpoints.

Properties

`idPiecewiseLinear` 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: `[]`

`Free`

Option to fix or free the values in the mapping object, specified as a logical scalar. When you set an element of `Free` to `false`, the object does not update during estimation.

Default: `true`

Output Arguments

`NL` — Piecewise-linear nonlinearity estimator object

`idPiecewiseLinear` object

Piecewise-linear nonlinearity estimator object, returned as an `idPiecewiseLinear` object.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.

Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

See Also

`nlhw`

Introduced in R2007a

idpoly

Polynomial model with identifiable parameters

Description

An `idpoly` model represents a system as a continuous-time or discrete-time polynomial model with identifiable (estimable) coefficients. Use `idpoly` to create a polynomial model or to convert “Dynamic System Models” to polynomial form.

A polynomial model of a system with input vector *u*, output vector *y*, and disturbance *e* takes the following form in discrete time:

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

The variables *A*, *B*, *C*, *D*, and *F* are polynomials expressed with the time-shift operator q^{-1}. For instance, the *A* polynomial takes this form:

$$A(q) = 1 + a_1q^{-1} + a_2q^{-2} + \cdots + a_{na}q^{-na}$$

Here, *na* is the order of the *A* polynomial. $q^{-1}y(t)$ is equivalent to $y(t-1)$.

For example, if $A(q) = 1 + a_1q^{-1} + a_2q^{-2}$, then $A(y(t)) = 1 + a_1(t-1) + a_2(t-2)$.

The *C*, *D*, and *F* polynomials take the same form as the *A* polynomial, starting with 1. The *B* polynomial does not start with 1.

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*) contains the Laplace transformed inputs to `sys`. *Y*(*s*) contains the Laplace transformed outputs. *E*(*s*) contains the Laplace transform of the disturbances for each output.

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.

Although `idpoly` supports continuous-time models, `idtf` and `idproc` enable more choices for estimation of continuous-time models. Therefore, for most continuous-time applications, these other model types are preferable.

For more information about polynomial models, see “What Are Polynomial Models?”

Creation

You can obtain an `idpoly` model in one of three ways.

- Estimate the `idpoly` model based on output or input-output measurements of a system by 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 information on the 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.

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

Create Input-Output Polynomial Model

`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.

Create Time-Series Model

`sys = idpoly(A)` creates a time-series model with only an autoregressive term. In this case, `sys` represents the AR model given by $A(q)\ y(t) = e(t)$. 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` is set to `[]`, then `sys` represents the ARMA model given by

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

`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.

Convert Dynamic System Model to Polynomial Model

`sys = idpoly(sys0)` converts the dynamic system model `sys0` to `idpoly` model form. `sys0` can be any dynamic system model.

`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 `tf`, `zpk`, or `ss` model object. Also, `sys0` must have at least as many inputs as outputs.

Input Arguments

`sys0` — Dynamic system
dynamic system model

Dynamic system, specified as a dynamic system model to convert to an `idpoly` 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`.

For the syntax `sys = idpoly(sys0,'split')`, `sys0` must meet the following requirements.

- `sys0` is a numeric `tf`, `zpk`, or `ss` model object.
- `sys0` has at least as many inputs as outputs.
- The subsystem `sys0(:,Ny+1:Nu)` must be biproper.

Properties

`A,B,C,D,F` — Values of polynomial coefficients
[ ] | 1 | row vectors | array of row vectors

Values of the polynomial coefficients, specified as row vectors for SISO models or, for MIMO models, cell arrays of row vectors that correspond to each of the `A`, `B`, `C`, `D`, and `F` polynomials. For each polynomial, the coefficients are stored in the following order:

- 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` are always 1.

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 and Constraints	Relation Described

`A`	N_y-by-N_y cell array of row vectors
	
	Leading coefficients:
	• Diagonal entries — Fixed to 1
	• Off-diagonal entries — Fixed to 0
		`A{i,j}` contains coefficients that relate the output y_i to the output y_j.
		
		For example, for a two-output system, `A` is a 2-by-2 cell array, such as:
		
		`A{1,1} = [1 .1]`
		`A{1,2} = [0.4 -0.6]`
		`A{2,1} = 0`
		`A{2,2} = [1 0.2 0.3]`

`B,F`	N_y-by-N_u array of row vectors
	
	Leading coefficients:
	• B — Not constrained
	• F — Fixed to 1
		`B{i,j}` and `F{i,j}` contain coefficients that relate the output y_i to the input u_j.
		
		For example, for a two-output system, `B` and `F` are 2-by-1 cell arrays, such as:
		
		`B{1,1} = [0.1 0.2]`
		`B{2,1} = [0.5 0.3]`
		`F{1,1} = [1 0.8]`
		`F{2,1} = [1 0.4]`

`C,D`	N_y-by-1 array of row vectors
	
	Leading coefficients:
	• C — Fixed to 1
	• D — Fixed to 1
		`C{i}` and `D{i}` contain coefficients that relate the output y_i to the noise e_i.
		
		For example, for a two-output system, `C` and `D` are 2-by-1 cell arrays, such as:
		
		`C{1,1} = [1 0.3]`
		`C{2,1} = [1 0.5 0.3]`
		`D{1,1} = [1 0.7]`
		`D{2,1} = [1 0.1 0.2]`

For a time series model (a model with no measured inputs), `B = []` and `F = []`.

If you obtain an `idpoly` model `sys` by identification using a function such as `polyest` or `arx`, then `sys.A`, `sys.B`, `sys.C`, `sys.D`, and `sys.F` contain the estimated values of the 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,F` input arguments. Use `NaN` for any coefficient whose initial value is not known. Use `[]` for any polynomial that is not present in the model structure that you want to create. For example, to create an ARX model, use `[]` for `C`, `D`, and `F`. For an ARMA time series model, use `[]` for `B` and `F`. Default initial values when you create an `idpoly` model are:

- B — `[]`
- C — `1` for all outputs
- D — `1` for all outputs
- F — `[]`

For an `idpoly` model `sys`, each property `sys.A`, `sys.B`, `sys.C`, `sys.D`, and `sys.F` is an alias of the corresponding `Value` entry in the `Structure` property of `sys`. For example, `sys.A` is an alias of the value of the property `sys.Structure.A.Value`.

`Variable` — Polynomial model display variable

`'s'` (default) | `'p'` | `'z^-1'` | `'q^-1'`

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. When `Variable` is set to `'z^-1'` or `'q^-1'`, the coefficient vectors are ordered as ascending powers of the variable.

`IODelay` — Transport delays

`0` (default) | scalar | numeric array

Transport delays, specified as a numeric array containing a separate transport delay for each input-output pair or as a scalar that applies the same delay to each input-output pair.

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 a delay of a multiple of the sample time `Ts`.

For a MIMO system with N_y outputs and N_u inputs, `IODelay` is an N_y-by-N_u 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.

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 pair argument.

If you obtain an `idpoly` model `sys` by identification using a function such as `polyest` or `arx`, then `sys.IODelay` contains the estimated values of the transport delay.

For an `idpoly` model `sys`, the property `sys.IODelay` is an alias for the value of the property `sys.Structure.IODelay.Value`.

`IntegrateNoise` — Presence of integration on noise channels

logical vector of zeros (default) | logical vector

Logical vector denoting the presence or absence of integration on noise channels, specified as a logical vector with length equal to the number of outputs.

`IntegrateNoise(i) = true` indicates that the noise channel for the *i*th output contains an integrator. In this case, the corresponding *D* polynomial contains an additional term that is not represented in the property `sys.D`. This integrator term is equal to $1/s$ for continuous-time systems and $1/(1\text{-}z^{-1})$ for discrete-time systems.

`Structure` — Information about the estimable parameters

structure property values

Property-specific information about the estimable parameters of the `idpoly` model, specified as a structure.

For a system with N_y outputs and N_u inputs, the dimensions of the `Structure` elements are as follows:

- `sys.Structure.A` — N_y-by-N_y
- `sys.Structure.B` — N_y-by-N_u
- `sys.Structure.C` — N_y-by-1
- `sys.Structure.D` — N_y-by-1
- `sys.Structure.F` — N_y-by-N_u

`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` contains information about the integration terms on the noise. Each parameter in `Structure` contains the following fields.

Field	**Description**	**Examples**
Value	Parameter values. Each property is an alias of the corresponding `Value` entry in the `Structure` property of `sys`. `NaN` represents unknown parameter values.	`sys.Structure.A.Value` contains the initial or estimated values of the SISO *A* polynomial. `sys.A` is an alias of the value of this property. `sys.A{i,j}` is the alias of the MIMO property `sys.Structure.A(i,j).Value`.
Minimum	Minimum value that the parameter can assume during estimation	`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` to `false`. For fixed values, such as the leading coefficients of the values of *A* polynomial, which are always equal to 1, the corresponding value of `Free` is always `false`.	If *B* is a 3-by-3 matrix,`sys.Structure.B.Free = eye(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.
Scale	Scale of the value of the parameter. The estimation algorithm does not use `Scale`.	

Field	Description	Examples
Info	Structure array that contains the fields `Label` and `Unit` for storing parameter labels and units. Specify parameter labels and units as character vectors.	Example: `'Time'`

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` and `sys.Structure.IntegrateNoise`. However, there is no field in `Structure` corresponding to `B`, `C`, `D`, `F`, or `IODelay`.

`NoiseVariance` — Variance of model innovations

positive scalar | matrix

Variance (covariance matrix) of the model innovations *e*, specified as a scalar or a positive semidefinite matrix.

- SISO model — Scalar
- MIMO model with N_y outputs — N_y-by-N_y positive semidefinite matrix

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 `polyest`) determines this variance.

`Report` — Summary report

report field values

This property is read-only.

Summary report that contains information about the estimation options and results for a state-space model obtained using estimation commands, such as `polyest`, `armax`, `oe`, and `bj`. Use `Report` to find estimation information for the identified model, including:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit and other quality metrics

If you create the model by construction, the contents of `Report` are irrelevant.

```
m = idpoly({[1 0.5]},{[1 5]},{[1 0.01]});
m.Report.OptionsUsed
```

```
ans =

     []
```

If you obtain the 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'
  EstimateCovariance: 1
             Display: 'off'
         InputOffset: []
        OutputOffset: []
      Regularization: [1x1 struct]
        SearchMethod: 'auto'
       SearchOptions: [1x1 idoptions.search.identsolver]
            Advanced: [1x1 struct]
```

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
`0` (default) | scalar | vector

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, setting `InputDelay` to `3` specifies a delay of three sample times.

For a system with N_u inputs, set `InputDelay` to an N_u-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.

In estimation, `InputDelay` is a fixed constant of the model. The software uses the `IODelay` property for estimating time delays. To specify initial values and constraints for estimation of time delays, use `sys.Structure.IODelay`.

`OutputDelay` — Output delay for each output channel
`0` (default)

This property is read-only.

Output delay for each output channel, specified as `0`. This value is fixed for identified systems such as `idpoly`.

Ts — Sample Time
`-1` (default) | `0` | positive scalar

Sample time, specified as one of the following.

- Discrete-time model with an unspecified sample time — `-1`
- Continuous-time model — `0`
- Discrete-time model with a specified sampling time — Positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model

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.

`TimeUnit` — Units for time variable
`'seconds'` (default) | `'nanoseconds'` | `'microseconds'` | `'milliseconds'` | `'minutes'` | `'hours'` | `'days'` | `'weeks'` | `'months'` | `'years'`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as a scalar.

Changing this property does not resample or convert the data. Modifying the property changes only the interpretation of the existing data. Use `chgTimeUnit` to convert data to different time units

`InputName` — Input channel names
`''` (default) | character vector | cell array

Input channel names, specified as a character vector or cell array.

- Single-input model — Character vector, for example, `'controls'`
- Multi-input model — Cell array of character vectors

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter the following:

```
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`.

You can use input channel names in several ways, including:

- To identify channels on model display and plots.
- To extract subsystems of MIMO systems.
- To specify connection points when interconnecting models.

`InputUnit` — Input channel units
`''` (default) | character vector | cell array

Input channel units, specified as a character vector or cell array:

- Single-input model — Character vector
- Multi-input Model — Cell array of character vectors

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

`InputGroup` — Input channel groups
structure with no fields (default) | structure

Input channel groups, specified as a structure. The `InputGroup` property lets you divide the input channels of MIMO systems into groups so that you can refer to each group by name. In the `InputGroup` structure, set field names to the group names, and field values to the input channels belonging to each group.

For example, create input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively.

```
sys.InputGroup.controls = [1 2];
sys.InputGroup.noise = [3 5];
```

You can then extract the subsystem from the `controls` inputs to all outputs using the following syntax:

```
sys(:,'controls')
```

`OutputName` — Output channel names
`''` (default) | character vector | cell array

Output channel names, specified as a character vector or cell array.

- Single-input model — Character vector, for example, `'measurements'`
- Multi-input model — Cell array of character vectors

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter the following:

```
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`.

You can use output channel names in several ways, including:

- To identify channels on model display and plots.
- To extract subsystems of MIMO systems.
- To specify connection points when interconnecting models.

`OutputUnit` — Output channel units
`''` (default) | character vector | cell array

Output channel units, specified as a character vector or cell array.

- Single-input model — Character vector, for example, `'seconds'`
- Multi-input Model — Cell array of character vectors

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

`OutputGroup` — Output channel groups
`struct` with no fields (default) | `struct`

Output channel groups, specified as a structure. The `OutputGroup` property lets you divide the output channels of MIMO systems into groups and refer to each group by name. In the `OutputGroup` structure, set field names to the group names, and field values to the output channels belonging to each group.

For example, create output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively.

```
sys.OutputGroup.temperature = [1];
sys.OutputGroup.measurement = [3 5];
```

You can then extract the subsystem from all inputs to the `measurement` outputs using the following syntax:

```
sys('measurement',:)
```

Name — System name
`''` (default) | character vector

System name, specified as a character vector, for example, `'system_1'`.

Notes — Text to associate with system
`0`-by-`1` string (default) | string | string array | character vector

Any text that you want to associate with the system, specified as a string.

- For a single note, specify `Notes` as a string or a character vector
- For multiple notes, specify `Notes` as a string array.

The property preserves the string or character data type that you specify. When you specify a character vector, the software packages the character vector in a 1-by-1 cell array.

For example, if `sys1`, `sys2`, and `sys3` are dynamic system models, you can set their `Notes` properties as follows.

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = ["sys2 has a first string";"sys3 has a second string"];
sys3.Notes = 'sys3 has a character vector.';
sys1.Notes
sys2.Notes
sys3.Notes


ans = 

    "sys1 has a string."

ans = 

  2×1 string array

    "sys2 has a first string"
    "sys2 has a second string"

ans =

  1×1 cell array
```

```
    {'sys3 has a character vector'}
```

`UserData` — Data to associate with system
[ ] (default) | any MATLAB data type

Data to associate with the system, specified as any MATLAB data type.

`SamplingGrid` — Sampling grid
[ ] (default) | structure

Sampling grid for model arrays, specified as a structure.

For arrays of identified linear (IDLTI) models that you derive 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 must be numeric and scalar valued, and all arrays of sampled values must match the dimensions of the model array.

For example, suppose that 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)
```

Here, `sys` is an array containing three identified models obtained at 1000, 5000, and 10000 rpm, respectively.

For model arrays that you generate 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.

Object Functions

In general, any function applicable to “Dynamic System Models” is applicable to an `idpoly` model object. These functions are of four general types.

- Functions that operate and return `idpoly` model objects enable you to transform and manipulate `idpoly` models. For instance:
  - Use `merge` to merge estimated `idpoly` models.
  - Use `c2d` to convert an `idpoly` model from continuous to discrete time. Use `d2c` to convert an `idpoly` model from discrete to continuous time.
- Functions that perform analytical and simulation functions on `idpoly` models, such as `bode` and `sim`
- Functions that retrieve or interpret model information, such as `advice` and `getpar`
- Functions that convert `idpoly` models into a different model type, such as `idtf` for time domain or `idfrd` for frequency domain

The following lists contain a representative subset of the functions that you can use with `idpoly` models.

Transformation and Manipulation

translatecov	Translate parameter covariance across model transformation operations
setpar	Set attributes such as values and bounds of linear model parameters
chgTimeUnit	Change time units of dynamic system
d2d	Resample discrete-time model
d2c	Convert model from discrete to continuous time
c2d	Convert model from continuous to discrete time
merge	Merge estimated models

Analysis and Simulation

sim	Simulate response of identified model
predict	Predict state and state estimation error covariance at next time step using extended or unscented Kalman filter, or particle filter
compare	Compare identified model output and measured output
impulse	Impulse response plot of dynamic system; impulse response data
step	Step response plot of dynamic system; step response data
bode	Bode plot of frequency response, or magnitude and phase data

Information Extraction and Interpretation

tfdata	Access transfer function data
get	Access model property values
getpar	Obtain attributes such as values and bounds of linear model parameters
getcov	Parameter covariance of identified model
advice	Analysis and recommendations for data or estimated linear models

Conversion to Other Model Structures

idtf	Transfer function model with identifiable parameters
idss	State-space model with identifiable parameters
idfrd	Frequency response data or model

Examples

Create Polynomial Model

Create an `idpoly` model representing the single-input, single-output ARMAX model described by the following equation:

$$y(t) + 0.5y = u(t) + 5u(t-1) + 2u(t-2) + e(t) + 0.01e(t-1)$$

y is the output, u is the input, and e is the white-noise disturbance on y.

To create the `idpoly` model, define the `A`, `B`, and `C` polynomials that describe the relationships between the output, input, and noise values, respectively. Because there are no denominator terms in the system equation, `D` and `F` are 1.

```
A = [1 0.5];
B = [1 5 2];
C = [1 0.01];
```

Create an `idpoly` model with the specified coefficients.

```
sys = idpoly(A,B,C)
```

```
sys =
Discrete-time ARMAX model: A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 + 0.5 z^-1

  B(z) = 1 + 5 z^-1 + 2 z^-2

  C(z) = 1 + 0.01 z^-1

Sample time: unspecified

Parameterization:
   Polynomial orders:   na=1   nb=3   nc=1   nk=0
   Number of free coefficients: 5
   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 five free coefficients.

Create an `idpoly` model with specified noise variance `nv` and sample time `Ts`. To do so, you must also include values of 1 for `D` and `F`.

```
Ts = 0.1;
nv = 0.01;
sys = idpoly(A,B,C,1,1,nv,Ts)
```

```
sys =
Discrete-time ARMAX model: A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 + 0.5 z^-1

  B(z) = 1 + 5 z^-1 + 2 z^-2

  C(z) = 1 + 0.01 z^-1

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=1   nb=3   nc=1   nk=0
   Number of free coefficients: 5
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The display shows a sample time of 0.1 seconds.

Specify an input-output delay `iod` of one sample when you create an `idpoly` model.

```
iod = 1;
sys = idpoly(A,B,C,1,1,nv,Ts,'IODelay',1)
```

```
sys =
Discrete-time ARMAX model: A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 + 0.5 z^-1

  B(z) = 1 + 5 z^-1 + 2 z^-2

  C(z) = 1 + 0.01 z^-1

Input delays (listed by channel): 1
Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=1   nb=3   nc=1   nk=0
   Number of free coefficients: 5
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The display shows an input delay of one sample.

You can use `sys` to specify an initial parameterization for estimation with commands such as `polyest` or `armax`.

Create Polynomial Time-Series Model

Create an `idpoly` model representing the single-output ARMA model described by the following equation:

$$y(t) + 0.5y = e(t) + 0.01e(t-1)$$

Because a time series has no measured inputs, this model contains only `A` and `C` polynomials.

```
A = [1 0.5];
C = [1 0.01];
```

Create a discrete-time time-series model without specifying a sample time.

```
sys = idpoly(A,[],C)
```

```
sys =
Discrete-time ARMA model: A(z)y(t) = C(z)e(t)
  A(z) = 1 + 0.5 z^-1

  C(z) = 1 + 0.01 z^-1

Sample time: unspecified

Parameterization:
   Polynomial orders:   na=1   nc=1
   Number of free coefficients: 2
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.
```

```
Status:
Created by direct construction or transformation. Not estimated.
```

The display reflects your specifications.

Create a continuous-time time-series by specifying a sample time of `0` for the name-value pair argument `'Ts'`.

```
sys = idpoly(A,[],C,'Ts',0)
```

```
sys =
Continuous-time ARMA model: A(s)y(t) = C(s)e(t)
  A(s) = s + 0.5

  C(s) = s + 0.01

Parameterization:
   Polynomial orders:   na=1   nc=1
   Number of free coefficients: 2
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

You can also set the sample time using the `Ts` input argument rather than the name-value pair argument, but the syntax is more complex. You must specify the `D` value as `1` or empty, and set both the `F` position and the noise variance position (if you are not specifying noise variance) to empty.

```
Ts = 0;
sys = idpoly(A,[],C,1,[],[],Ts)
```

```
sys =
Continuous-time ARMA model: A(s)y(t) = C(s)e(t)
  A(s) = s + 0.5

  C(s) = s + 0.01

Parameterization:
   Polynomial orders:   na=1   nc=1
   Number of free coefficients: 2
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

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, `D` 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 parameterization for estimation with commands such as `polyest` or `armax`.

Convert Transfer Function Model into Polynomial Model

Model a dynamic system using a transfer function. Then use `idpoly` to convert the transfer-function model into polynomial form.

Using `idtf`, construct the continuous-time, single-input, single-output (SISO) transfer function model described by the following equation:

$$G(s) = \frac{s + 4}{s^2 + 20s + 5}$$

```
num = [1 4];
den = [1 20 5];
G = idtf(num,den)
```

```
G =

      s + 4
  --------------
  s^2 + 20 s + 5

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:
Created by direct construction or transformation. Not estimated.
```

Convert the transfer function into polynomial form.

```
sys = idpoly(G)
```

```
sys =
Continuous-time OE model: y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = s + 4

  F(s) = s^2 + 20 s + 5

Parameterization:
   Polynomial orders:   nb=2   nf=2   nk=0
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The display shows the polynomial form and the polynomial coefficients.

See Also

`polydata` | `arx` | `armax` | `bj` | `oe` | `ar` | `polyest` | `setPolyFormat` | `idss` | `idproc` | `idtf` | `iv4` | `ivar` | `translatecov`

Topics

"Dynamic System Models"
"What Are Polynomial Models?"
"Estimate Polynomial Models in the App"
"Estimate Polynomial Models at the Command Line"
"Polynomial Sizes and Orders of Multi-Output Polynomial Models"

Introduced before R2006a

idPolynomial1D

Class representing single-variable polynomial nonlinear estimator for Hammerstein-Wiener models

Syntax

```
t=idPolynomial1D('Degree',n)
t=idPolynomial1D('Coefficients',C)
t=idPolynomial1D(n)
```

Description

`idPolynomial1D` 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=idPolynomial1D('Degree',n)` creates a polynomial nonlinearity estimator object of `n`th degree.

`t=idPolynomial1D('Coefficients',C)` creates a polynomial nonlinearity estimator object with coefficients `C`.

`t=idPolynomial1D(n)` a polynomial nonlinearity estimator object of `n`th degree.

Use `evaluate(p,x)` to compute the value of the function defined by the `idPolynomial1D` object `p` at `x`.

idPolynomial1D 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: `idPolynomial1D('Degree',3)`
`Coefficients`	1-*by*-`(n+1)` matrix containing the polynomial coefficients.
`Free`	Option to fix or free the values in the mapping object. When you set `Free` to `false`, the object does not update during estimation.

Examples

Use `idPolynomial1D` to specify the single-variable polynomial nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,idPolynomial1D('deg',3),[]);
```

where `'deg'` is an abbreviation for the property `'Degree'`.

Tips

Use `idPolynomial1D` to define a nonlinear function $y = F(x)$, where F is a single-variable polynomial function of x:

$$F(x) = c(1)x^n + c(2)x^{(n-1)} + \ldots + c(n)x + c(n+1)$$

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

See Also
`nlhw`

Introduced in R2007b

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. T_{p1} is the time constant of the real pole, and T_d is the transport delay (dead time).

More generally, `idproc` can represent process models with up to three poles and a zero:

$$sys = K_p\frac{1 + T_z s}{(1 + T_{p1}s)(1 + T_{p2}s)(1 + T_{p3}s)}e^{-T_d s}.$$

Two of the poles can be a complex conjugate (underdamped) pair. In that case, the general form of the process model is:

$$sys = K_p\frac{1 + T_z s}{\left(1 + 2\zeta T_\omega s + (T_\omega s)^2\right)(1 + T_{p3}s)}e^{-T_d s}.$$

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 1-690 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 1-690 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:

$$sys = \frac{0.01}{1 + 2(0.1)(10)s + (10s)^2} e^{-5s}$$

Create a process model with the specified structure.

```
sys = idproc('P2DU')
```

```
sys =
Process model with transfer function:
                  Kp
  G(s) = --------------------- * exp(-Td*s)
         1+2*Zeta*Tw*s+(Tw*s)^2
         Kp = NaN
         Tw = NaN
       Zeta = NaN
         Td = NaN

Parameterization:
    {'P2DU'}
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The input `'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 parameterization 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 parameterization 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 = 1×3

     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:

  $$sys = \frac{K_p}{\left(1 + 2\zeta T_\omega s + (T_\omega s)^2\right)}.$$

  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:

  $$sys = K_p \frac{1 + T_z s}{s(1 + T_{p1}s)(1 + T_{p2}s)(1 + T_{p3}s)} e^{-T_d s}.$$

  The identifiable parameters of this model are Kp, Tz, Tp1, Tp2, Tp3, and Td.

The values of all parameters in a particular model structure are initialized to NaN. You can change them to finite values by setting the values of the corresponding idproc model properties after you create the model. For example, sys.Td = 5 sets the initial value of the time delay of sys to 5.

For a MIMO process model with Ny outputs and Nu inputs, type is an Ny-by-Nu cell array of 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 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 on page 1-690 of idproc models during model creation. For example, sys = idproc('p2z','InputName','Voltage','Kp',10,'Tz',0); creates an idproc 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'
  EstimateCovariance: 1
             Display: 'off'
         InputOffset: [1x1 param.Continuous]
        OutputOffset: []
      Regularization: [1x1 struct]
        SearchMethod: 'auto'
       SearchOptions: [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, stored as a string or a cell array of character vectors. The property stores whichever data type you provide. For instance, if `sys1` and `sys2` are dynamic system models, you can set their `Notes` properties as follows:

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

Default: `[0×1 string]`

`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

`idtf` | `procest` | `idss` | `tfest` | `ssest` | `pem`

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

idSaturation

Create a saturation nonlinearity estimator object

Syntax

```
NL = idSaturation
NL = idSaturation('LinearInterval',[a,b])
```

Description

`NL = idSaturation` 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 = idSaturation('LinearInterval',[a,b])` creates a saturation nonlinearity estimator object initialized with linear interval, `[a,b]`.

Alternatively, use `NL = idSaturation([a,b])`.

Object Description

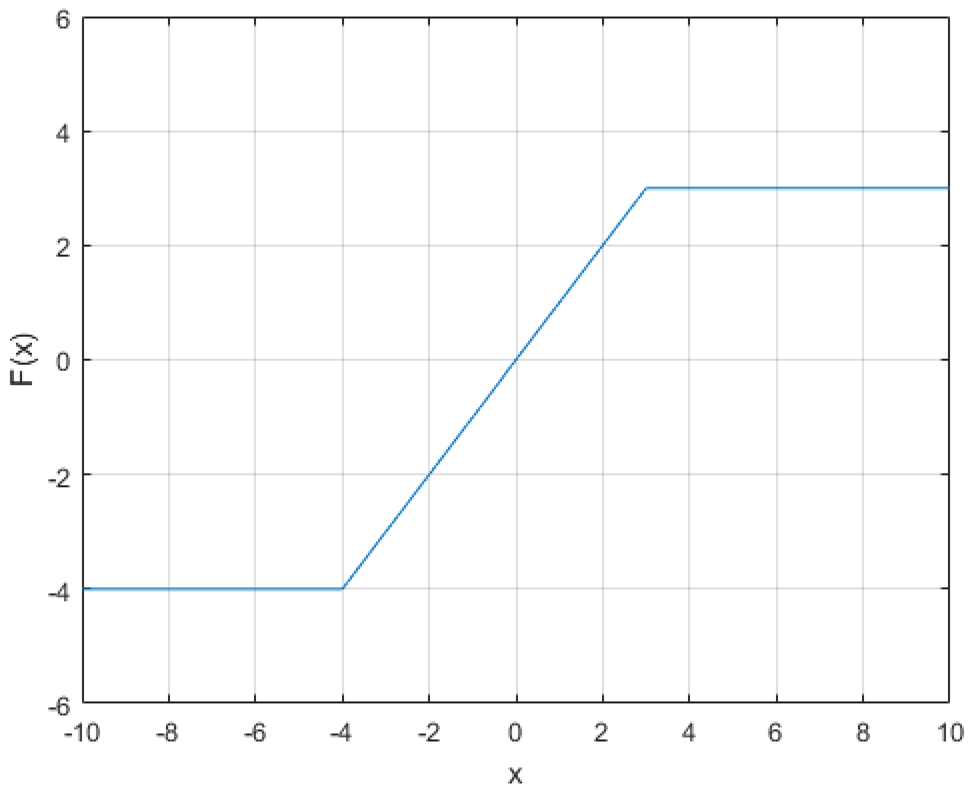
`idSaturation` is an object that stores the saturation nonlinearity estimator for estimating Hammerstein-Wiener models.

Use `idSaturation` 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:

$a \le x < b$	$F(x) = x$
$a > x$	$F(x) = a$
$b \le x$	$F(x) = b$

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 `idSaturation` object.

For `idSaturation` object properties, see “Properties” on page 1-705.

Examples

Create a Default Saturation Nonlinearity Estimator

```
NL = idSaturation;
```

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 = idSaturation('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 = idSaturation;
```

Specify different nonlinearity estimators for each output channel.

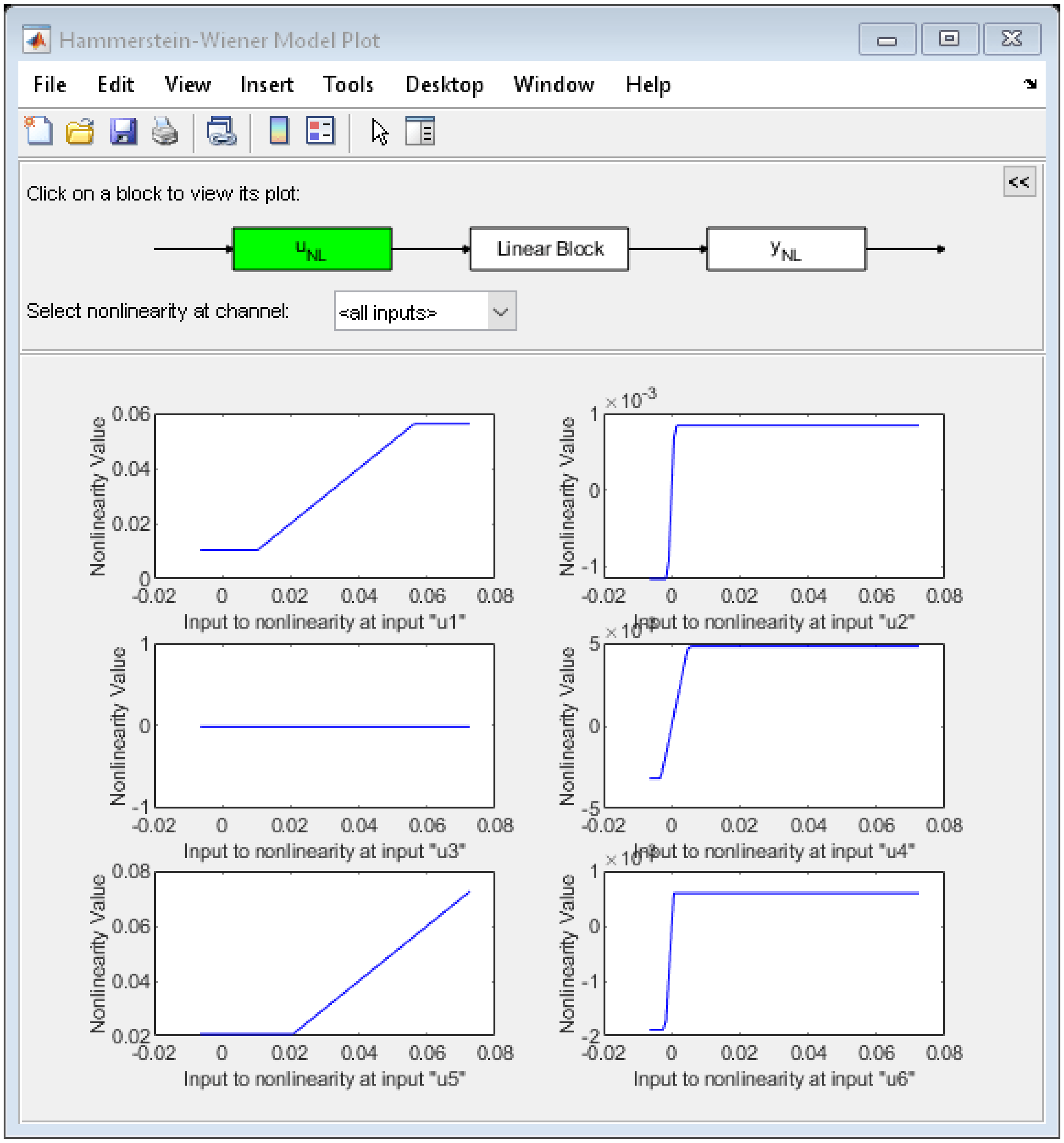
```
 OutputNL = [idDeadZone,idWaveletNetwork];
```

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"

Click on the input and output nonlinearity blocks on the top of the plot to see the nonlinearities.

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]

Free

Option to fix or free the parameters of LinearInterval, specified as a 2–element logical row vector. When you set an element of Free to false, the corresponding value in LinearInterval remains fixed during estimation to the initial value that you specify.

Default: [true true]

Output Arguments

NL — Saturation nonlinearity estimator object
idSaturation object

Saturation nonlinearity estimator object, returned as an idSaturation object.

Compatibility Considerations

Use of previous idnlarx and idnlhw mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the idnlarx and idnlhw objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
wavenet	idWaveletNetwork
sigmoidnet	idSigmoidNetwork
treepartition	idTreePartition
customnet	idCustomNetwork
saturation	idSaturation
deadzone	idDeadZone
pwlinear	idPiecewiseLinear
poly1d	idPolynomial1D

Pre-R2021b Name	R2021b Name
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

See Also
`nlhw`

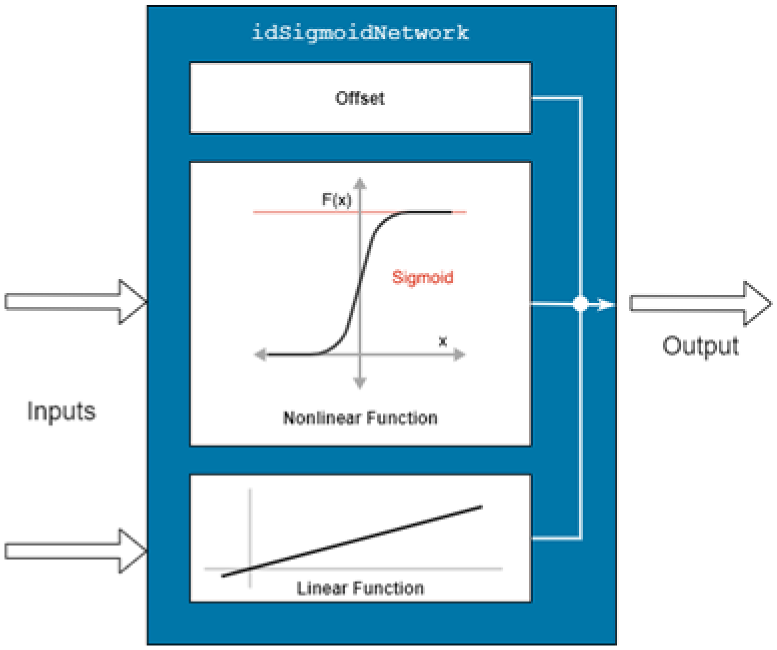
Introduced in R2007a

idSigmoidNetwork

Sigmoid network function for nonlinear ARX and Hammerstein-Wiener models

Description

An `idSigmoidNetwork` object implements a sigmoid network function, and is a nonlinear mapping function for estimating nonlinear ARX and Nonlinear Hammerstein-Wiener models. The mapping function, which is also referred to as a nonlinearity, uses a combination of linear weights, an offset and a nonlinear function to compute its output. The nonlinear function contains sigmoid unit functions that operate on a ridge combination (weighted linear sum) of inputs.



Mathematically, `idSigmoidNetwork` is a function that maps m inputs $X(t) = [x(t_1),x_2(t),\ldots,x_m(t)]^T$ to a scalar output $y(t)$ using the following relationship:

$$y(t) = y_0 + (X(t) - \bar{X})^T PL + S(X(t))$$

Here:

- $X(t)$ is an m-by-1 vector of inputs, or regressors, with mean $\bar{X}$.
- y_0 is the output offset, a scalar.
- P is an m-by-p projection matrix, where m is the number of regressors and is p is the number of linear weights. m must be greater than or equal to p.
- L is a p-by-1 vector of weights.
- $S(X)$ is a sum of dilated and translated sigmoid functions. The total number of sigmoid functions is referred to as the number of units n of the network.

For the definition of the sigmoid function term $S(X)$, see “More About” on page 1-713.

Use `idSigmoidNetwork` as the value of the `OutputFcn` property of an `idnlarx` model or the `InputNonlinearity` and `OutputLinearity` properties of an `idnlhw` object. For example, specify `idSigmoidNetwork` when you estimate an `idnlarx` model with the following command.

```
sys = nlarx(data,regressors,idSigmoidNetwork)
```

When `nlarx` estimates the model, it essentially estimates the parameters of the `idSigmoidNetwork` function.

You can configure the `idSigmoidNetwork` function to disable components and fix parameters. To omit the linear component, set `LinearFcn.Use` to `false`. To omit the offset, set `Offset.Use` to `false`. To specify known values for the linear function and the offset, set their `Value` attributes directly and set the corresponding `Free` attributes to `False`. Use `evaluate` to compute the output of the function for a given vector of inputs.

Creation

Syntax

```
S = idSigmoidNetwork
S = idSigmoidNetwork(numUnits)
S = idSigmoidNetwork(numUnits,UseLinearFcn)
S = idSigmoidNetwork(numUnits,UseLinearFcn,UseOffset)
```

Description

`S = idSigmoidNetwork` creates a `idSigmoidNetwork` object `S` that uses 10 units. The number of inputs is determined during model estimation and the number of outputs is 1.

`S = idSigmoidNetwork(numUnits)` specifies the number of sigmoid functions `numUnits`.

`S = idSigmoidNetwork(numUnits,UseLinearFcn)` specifies whether the function uses a linear function as a subcomponent.

`S = idSigmoidNetwork(numUnits,UseLinearFcn,UseOffset)` specifies whether the function uses an offset term y_0 parameter.

Input Arguments

`numUnits` — Number of units
`10` (default) | positive integer

Number of units, specified as a positive integer. `numUnits` determines the number of sigmoid functions.

This argument sets the `S.NonlinearFcn.NumberOfUnits` property.

`UseLinearFcn` — Option to use linear function
`true` (default) | `false`

Option to use the linear function subcomponent, specified as `true` or `false`. This argument sets the value of the `S.LinearFcn.Use` property.

`UseOffset` — Option to use offset term
`true` (default) | `false`

Option to use an offset term, specified as `true` or `false`. This argument sets the value of the `S.Offset.Use` property.

Properties

`Input` — Input signal information
signal property values

Input signal information for signals used for estimation, specified as vectors of *m* property-specific values, where *m* is the number of input signals. The `Input` properties for each input signal are as follows:

- `Name` — Names of the input signals, specified as a 1-by-*m* string or character array, where *m* is the number of inputs
- `Mean` — Mean of the input signals, specified as a numeric scalar
- `Range` — Ranges of the input signals, specified as a 2-by-*m* numeric array that contains the minimum and maximum values

`Output` — Output signal information
signal property values

Output signal information, specified as property-specific values. The `Output` properties are as follows:

- `Name` — Name of the output signal, specified as a string or a character array
- `Mean` — Mean of the output signal, specified as a numeric scalar
- `Range` — Range of the output signal, specified as a 2-by-1 numeric array that contains the minimum and maximum values

`LinearFcn` — Parameters of linear function
linear function property values (default)

Parameters of the linear function, specified as follows:

- `Use` — Option to use the linear function in the sigmoid network, specified as a scalar logical. The default value is `true`.
- `Value` — Linear weights that compose *L*', specified as a 1-by-*p* vector.
- `InputProjection` — Input projection matrix *P*, specified as an *m*-by-*p* matrix, that transforms the detrended input vector of length *m* into a vector of length *p*. For Hammerstein-Wiener models, `InputProjection` is equal to `1`.
- `Free` — Option to update entries of `Value` during estimation, specified as a 1-by-*p* logical vector. The software honors the `Free` specification only if the starting value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a 1-by-*p* vector. If `Minimum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that minimum bound during model estimation.

- `Maximum` — Maximum bound on `Value`, specified as a 1-by-p vector. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation.
- `SelectedInputIndex` — Indices of `idSigmoidNetwork` inputs (see `Input.Name`) that are used as inputs to the linear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. For nonlinear ARX models, the `RegressorUsage` property determines these indices. For Hammerstein-Wiener models, `SelectedInputIndex` is always `1`.

`Offset` — Parameters of offset term
offset property values

Parameters of the offset term, specified as follows:

- `Use` — Option to use the offset in the sigmoid network, specified as a scalar logical. The default value is `true`.
- `Value` — Offset value, specified as a scalar.
- `Free` — Option to update `Value` during estimation, specified as a scalar logical. The software honors the `Free` specification of `false` only if the value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a numeric scalar or `–Inf`. If `Minimum` is specified with a finite value and the value of `Value` is finite, then the software enforces that minimum bound during model estimation. The default value is `-Inf`.
- `Maximum` — Maximum bound on `Value`, specified as a numeric scalar or `Inf`. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation. The default value is `Inf`.

`NonlinearFcn` — Parameters of nonlinear function
nonlinear function property values

Parameters of the nonlinear function, specified as follows:

- `NumberOfUnits` — Number of units, specified as a positive integer. `NumberOfUnits` determines the number of sigmoid functions.
- `Parameters` — Parameters of `idSigmoidNetwork`, specified as in the following table:

Field Name	Description	Default
`InputProjection`	Projection matrix Q, specified as an m-by-q matrix. Q transforms the detrended input vector $(X - \bar{X})$ of length m into a vector of length q. Typically, Q has the same dimensions as the linear projection matrix P. In this case, q is equal to p, which is the number of linear weights. For Hammerstein-Wiener models, `InputProjection` is equal to `1`.	`[]`
`OutputCoefficient`	Sigmoid function output coefficients s_i, specified as an n-by-1 vector.	`[]`
`Translation`	Translation matrix, specified as an n-by-q matrix of translation row vectors c_i.	`[]`
`Dilation`	Dilation coefficients b_i, specified as an n-by-1 vector.	`[]`

- `Free` — Option to estimate parameters, specified as a logical scalar. If all the parameters have finite values, such as when the `idSigmoidNetwork` object corresponds to a previously estimated model, then setting `Free` to `false` causes the parameters of the nonlinear function $S(X)$ to remain unchanged during estimation. The default value is `true`.
- `SelectedInputIndex` — Indices of `idSigmoidNetwork` inputs (see `Input.Name`) that are used as inputs to the nonlinear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. For nonlinear ARX models, the `RegressorUsage` property determines these indices. For Hammerstein-Wiener models, `SelectedInputIndex` is always `1`.

Examples

Estimate Nonlinear ARX Model with `idSigmoidNetwork` as Output Function

Load the data `z7` and create a subset to use as estimation data.

```
load iddata7 z7
ze = z7(1:300);
```

Create and configure an `idSigmoidNetwork` mapping object. Fix the offset to 0.2 and the number of units to 15.

```
S = idSigmoidNetwork;
S.Offset.Value = 0.2;
S.Offset.Free = false;
S.NonlinearFcn.NumberOfUnits = 15;
```

Create linear and polynomial model regressors. Use the input and output variable names from `z7` as the variable names for the regressors.

```
var_names = [z7.OutputName;z7.InputName]
```

```
var_names = 3x1 cell
    {'y1'}
    {'u1'}
    {'u2'}
```

```
Reg1 = linearRegressor(var_names,{1:4,0:4,1});
Reg2 = polynomialRegressor(var_names,{1:2,0:2,0},2);
```

Set the estimation options.

```
opt = nlarxOptions('SearchMethod','fmincon');
opt.SearchOptions.MaxIterations = 40;
```

Estimate the nonlinear ARX model.

```
sys = nlarx(ze,[Reg1;Reg2],S,opt)
```

```
sys =
Nonlinear ARX model with 1 output and 2 inputs
  Inputs: u1, u2
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1, u2
  2. Order 2 regressors in variables y1, u1, u2
  List of all regressors

Output function: Sigmoid network with 15 units
Sample time: 1 seconds

Status:
Estimated using NLARX on time domain data "ze".
Fit to estimation data: 73.95% (prediction focus)
FPE: 6.262, MSE: 0.671
```

Estimate Hammerstein-Wiener Model that Uses `idSigmoidNetwork`

Estimate a Hammerstein-Wiener model that uses `idSigmoidNetwork` as the output nonlinearity.

Load the data

```
load throttledata
```

Create an `idSigmoidNetwork` mapping object that has 15 units and that has no input nonlinearity or offset.

```
S = idSigmoidNetwork(15,false,false)
```

```
S = 
Sigmoid Network

 Nonlinear Function: Sigmoid network with 15 units
 Linear Function: not in use
 Output Offset: not in use

           Input: 'Function inputs'
          Output: 'Function output'
       LinearFcn: 'Linear function parameters'
    NonlinearFcn: 'Sigmoid units and their parameters'
          Offset: 'Offset parameters'
```

Estimate a Hammerstein-Wiener model.

```
sys = nlhw(ThrottleData,[4 4 0],[],S)
```

```
sys =
Hammerstein-Wiener model with 1 output and 1 input
 Linear transfer function corresponding to the orders nb = 4, nf = 4, nk = 0
 Input nonlinearity: absent
 Output nonlinearity: Sigmoid network with 15 units
Sample time: 0.01 seconds

Status:
Estimated using NLHW on time domain data "ThrottleData".
Fit to estimation data: 85.94%
FPE: 30.96, MSE: 21.81
```

More About

Sigmoid Nonlinear Function *S*(*X*)

The sigmoid nonlinear function is a sum of the dilated and translated sigmoid functions, and is described by the following equation:

$$S(X) = \sum_{i=1}^{n} s_i f((X - \bar{X})^T Q b_i + c_i)$$

Here:

- Q is an m-by-q projection matrix, where $m \geq q$.
- $s_1, s_2, \ldots, s_n$ are scalar weights called output coefficients.
- $b_1, b_2, \ldots, b_n$ are q-by-1 vectors called dilation coefficients .
- $c_1, c_2, \ldots, c_n$ are scalars called translations.
- $f(z) = \dfrac{1}{e^{-z} + 1}$. is the sigmoid function, also called a unit function of the sigmoid network. Here, z is a scalar of the form $b_i(X - \bar{X})^T Q + c_i$.

Algorithms

`idSigmoidNetwork` uses an iterative search technique for estimating parameters.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

Use of previous nonlinearity estimator properties is not recommended
Not recommended starting in R2021a

Starting in R2021a, the properties of the mapping objects, previously known as nonlinearity estimators, have been reorganized. These objects are `wavenet` (W), `sigmoidnet` (S), `treepartition` (T), `customnet` (C), and `linear` (L). The property changes do not apply to `neuralnet`. The use of the pre-R2021a properties in the following table is discouraged. However, the software still accepts commands that set these properties. There are no plans to exclude such commands at this time.

Pre-R2021a Property	R2021a Property	Applicable Mapping Objects
`NumberOfUnits`	`NonlinearFcn.NumberOfUnits`	W,S,T,C
`LinearTerm`	`LinearFcn.Use`, `Offset.Use`	W,S,C
`Parameters`	Split into three pieces: • `LinearFcn.Value` • `Offset.Value` • `NonlinearFcn.Parameters`	W,S,T,C,L `linear` (L) excludes `NonlinearFcn.Parameters`.
`Options`	`NonlinearFcn.Structure`	W,T

See Also
`nlhw` | `nlarx` | `idLinear` | `idPolynomial1D` | `idTreePartition` | `idWaveletNetwork` | `idSaturation` | `idPiecewiseLinear` | `idUnitGain` | `idDeadZone` | `idFeedforwardNetwork` | `idCustomNetwork` | `idnlhw` | `idnlarx` | `evaluate`

Introduced in R2007a

idss

State-space model with identifiable parameters

Description

Use `idss` to create a continuous-time or discrete-time state-space model with identifiable (estimable) coefficients, or to convert "Dynamic System Models" to state-space form.

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 following 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.

Creation

You can obtain an `idss` model object in one of three ways.

- Estimate the `idss` model based on the input-output measurements of a system by 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 on the handling of initial state values and the options used in estimation. For example:

  ```
  sys = ssest(data,nx);
  A = sys.A;
  B = sys.B;
  sys.Report
  ```

  For more examples of estimating an `idss` model, see `ssest` or `n4sid`.
- Create an `idss` model using the `idss` command. For example:

  ```
  sys = idss(A,B,C,D)
  ```

  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 instance, 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 (`ssest` or `n4sid`) to estimate parameter values with those constraints. For examples, see “Create State-Space Model with Identifiable Parameters” on page 1-725 and “Configure Identifiable Parameters of State-Space Model” on page 1-726.

- Convert an existing dynamic system model to an `idss` model using the `idss` command. For example:

  ```
  sys_ss = idss(sys_tf);
  ```

For information on functions you can use to extract information from or transform `idss` model objects, see “Object Functions” on page 1-724.

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

Create State-Space Model

`sys = idss(A,B,C,D)` creates a state-space model with specified state-space matrices `A,B,C,D`. By default, `sys` is a discrete-time model with an unspecified sample time and no state disturbance element. Use this syntax especially when you want to configure an initial parameterization as an input to a state-space estimation function such as `n4sid` or `ssest`.

`sys = idss(A,B,C,D,K)` specifies a disturbance matrix `K`.

`sys = idss(A,B,C,D,K,x0)` initializes the state values with the vector `x0`.

`sys = idss(A,B,C,D,K,x0,Ts)` specifies the sample time property `Ts`. Use `Ts = 0` to create a continuous-time model.

`sys = idss( ___ ,Name,Value)` sets additional properties using one or more name-value pair arguments. Specify name-value pair arguments after any of the input argument combinations in the previous syntaxes.

Convert Dynamic System Model to State-Space Model

`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 (nonidentified) `tf`, `zpk`, or `ss` model object. Also, `sys0` must have at least as many inputs as outputs.

Input Arguments

`x0` — Initial state values

column vector of zeros (default) | vector

Initial state values, specified as a column vector of N_x values.

`sys0` — Dynamic system
dynamic system model

Dynamic system, specified as a dynamic system model 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 (nonidentified) model, the state-space data of `sys0` defines 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 (nonidentified) `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 nonzero feedthrough term (the subsystem must be biproper).

Properties

`A,B,C,D` — Values of state-space matrices
matrices

Values of the state-space matrices, specified as matrices that correspond to each of the A, B, C, and D matrices.

For a system with N_y outputs, N_u inputs, and N_x states, the state-space matrices have the following dimensions:

- `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

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.

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.

For an `idss` model `sys`, each property `sys.A`, `sys.B`, `sys.C`, and `sys.D` is an alias of the corresponding `Value` entry in the `Structure` property of `sys`. For example, `sys.A` is an alias of the value of the property `sys.Structure.A.Value`.

`K` — Value of state disturbance matrix
zero matrix (default) | matrix

Value of the state disturbance matrix K, specified as an N_x-by-N_y matrix, where N_x is the number of states and N_y is the number of outputs.

If you obtain an `idss` model `sys` by identification using `ssest` or `n4sid`, then `sys.K` contains the estimated values of the matrix elements.

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.

For an `idss` model `sys`, `sys.K` is an alias to the value of the property `sys.Structure.K.Value`.

`StateName` — State names
`''` (default) | character vector | cell array

State names, specified as a character vector or cell array.

- First-order model — Character vector
- Model with two or more states — Cell array of character vectors
- Unnamed states — `''`

Example: `'velocity'` names the only state in a first-order model

`StateUnit` — State units
`''` (default) | character vector | cell array

State units, specified as a character vector or cell array.

- First-order model — Character vector
- Model with two or more states — Cell array of character vectors
- States without specified units — `''`

Use `StateUnit` to keep track of the units each state is expressed in. `StateUnit` has no effect on system behavior.

Example: `'rad'` corresponds to the units of the only state in a first-order model

`Structure` — Information about estimable parameters
structure property values

Information about the estimable parameters of the `idss` model, specified as property-specific values. `Structure.A`, `Structure.B`, `Structure.C`, `Structure.D`, and `Structure.K` contain information about the *A*, *B*, *C*, *D*, and *K* matrices, respectively. Each parameter in `Structure` contains the following fields.

Field	Description	Examples
Value	Parameter Values — Each property `sys.A`, `sys.B`, `sys.C`, and `sys.D` is an alias of the corresponding `Value` entry in the `Structure` property of `sys.NaN` represents unknown parameter values.	`sys.Structure.A.Value` contains the initial or estimated values of the *A* matrix.`sys.A` is an alias of the value of the property `sys.Structure.A.Value`.
Minimum	Minimum value that the parameter can assume during estimation	`sys.Structure.K.Minimum = 0` constrains all entries in the *K* matrix to be greater than or equal to zero.

Field	Description	Examples
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`.	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 value of the parameter. The estimation algorithm does not use `Scale`.	
Info	Structure array that contains the fields `Label` and `Unit` for storing parameter labels and units. Specify parameter labels and units as character vectors.	`'Time'`

For an example of configuring model parameters using the `Structure` property, see “Configure Identifiable Parameters of State-Space Model” on page 1-726.

`NoiseVariance` — Variance of model innovations

scalar | matrix

Variance (covariance matrix) of the model innovations *e*, specified as a scalar or matrix.

- SISO model — Scalar
- MIMO model with N_y outputs — N_y-by-N_y matrix

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.

`Report` — Summary report

report field values

This property is read-only.

Summary report that contains information about the estimation options and results for a state-space model obtained using estimation commands, such as `ssest`, `ssregest`, and `n4sid`. Use `Report` to find estimation information for the identified model, including the:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit and other quality metrics

If you create the model by construction, the contents of `Report` are irrelevant.

```
A = [-0.1 0.4; -0.4 -0.1];
B = [1; 0];
C = [1 0];
D = 0;
m = idss(A,B,C,D);
sys.Report.OptionsUsed
```

```
ans =

     []
```

If you obtain the 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'
EstimateCovariance: 1
           Display: 'off'
       InputOffset: []
      OutputOffset: []
      OutputWeight: []
      SearchMethod: 'auto'
     SearchOptions: [1x1 idoptions.search.identsolver]
    Regularization: [1x1 struct]
          Advanced: [1x1 struct]
```

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

`0` (default) | scalar | vector

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, setting `InputDelay` to `3` specifies a delay of three sample times.

For a system with N_u inputs, set `InputDelay` to an N_u-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.

`OutputDelay` — Output delay for each output channel

`0` (default)

For identified systems such as `idss`, `OutputDelay` is fixed to zero.

`Ts` — Sample Time

`-1` (default) | `0` | positive scalar

Sample time, specified as one of the following.

- Continuous-time model — `0`
- Discrete-time model with a specified sampling time — a positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model
- Discrete-time model with unspecified sample time — `-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.

`TimeUnit` — Units for time variable

`'seconds'` (default) | `'nanoseconds'` | `'microseconds'` | `'milliseconds'` | `'minutes'` | `'hours'` | `'days'` | `'weeks'` | `'months'` | `'years'`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as a scalar.

Changing this property does not resample or convert the data. Modifying the property changes only the interpretation of the existing data. Use `chgTimeUnit` to convert data to different time units

`InputName` — Input channel names

`''` (default) | character vector | cell array

Input channel names, specified as a character vector or cell array.

- Single-input model — Character vector. For example, `'controls'`.
- Multi-input model — Cell array of character vectors.

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`.

You can use input channel names in several ways, including:

- To identify channels on model display and plots
- To extract subsystems of MIMO systems
- To specify connection points when interconnecting models

`InputUnit` — Input channel units

`''` (default) | character vector | cell array

Input channel units, specified as a character vector or cell array:

- Single-input model — Character vector
- Multi-input Model — Cell array of character vectors

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

`InputGroup` — Input channel groups
`struct` with no fields (default) | `struct`

Input channel groups, specified as a structure. The `InputGroup` property lets you divide the input channels of MIMO systems into groups so that you can refer to each group by name. In the `InputGroup` structure, set field names to the group names, and field values to the input channels belonging to each group.

For example, create input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively.

```
sys.InputGroup.controls = [1 2];
sys.InputGroup.noise = [3 5];
```

You can then extract the subsystem from the `controls` inputs to all outputs using the following syntax:

```
sys(:,'controls')
```

`OutputName` — Output channel names
`''` (default) | character vector | cell array

Output channel names, specified as a character vector or cell array.

- Single-input model — Character vector. For example, `'measurements'`.
- Multi-input model — Cell array of character vectors.

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`.

You can use output channel names in several ways, including:

- To identify channels on model display and plots
- To extract subsystems of MIMO systems
- To specify connection points when interconnecting models

`OutputUnit` — Output channel units
`''` (default) | character vector | cell array

Output channel units, specified as a character vector or cell array.

- Single-input model — Character vector. For example, `'seconds'`.

- Multi-input Model — Cell array of character vectors.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

`OutputGroup` — Output channel groups
`struct` with no fields (default) | `struct`

Output channel groups, specified as a structure. The `OutputGroup` property lets you divide the output channels of MIMO systems into groups and refer to each group by name. In the `OutputGroup` structure, set field names to the group names, and field values to the output channels belonging to each group.

For example, create output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively.

```
sys.OutputGroup.temperature = [1];
sys.OutputGroup.measurement = [3 5];
```

You can then extract the subsystem from all inputs to the `measurement` outputs using the following syntax:

```
sys('measurement',:)
```

`Name` — System Name
`''` (default) | character vector

System name, specified as a character vector. For example, `'system_1'`.

`Notes` — Notes on system
`0`-by-`1` string (default) | string | character vector

Any text that you want to associate with the system, specified as a string or a cell array of character vectors. The property stores whichever data type you provide. For instance, if `sys1` and `sys2` are dynamic system models, you can set their `Notes` properties as follows.

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

`UserData` — Data to associate with system
`[]` (default) | any MATLAB data type

Data to associate with the system, specified as any MATLAB data type.

SamplingGrid — Sampling grid
[] (default) | struct

Sampling grid for model arrays, specified as a structure.

For arrays of identified linear (IDLTI) models that you derive 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 must be numeric and scalar valued, and all arrays of sampled values must match the dimensions of the model array.

For example, suppose that 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)
```

Here, sys is an array containing three identified models obtained at 1000, 5000, and 10000 rpm, respectively.

For model arrays that you generate 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.

Object Functions

In general, any function applicable to “Dynamic System Models” is applicable to an idss model object. These functions are of four general types.

- Functions that operate and return idss model objects enable you to transform and manipulate idss models. For instance:
  - Use canon to transform an idss model into canonical form
  - Use merge to merge estimated idss models.
  - Use c2d to convert an idss from continuous to discrete time. Use d2c to convert an idss from discrete to continuous time.
- Functions that perform analytical and simulation functions on idss objects, such as bode and sim
- Functions that retrieve or interpret model information, such as advice and getpar
- Functions that convert idss objects into a different model type, such as idpoly or idtf for time domain or idfrd for continuous domain

The following lists contain a representative subset of the functions that you can use with idss models.

Transformation and Manipulation

canon	Canonical state-space realization

ss2ss	State coordinate transformation for state-space model
balred	Model order reduction
translatecov	Translate parameter covariance across model transformation operations
setpar	Set attributes such as values and bounds of linear model parameters
chgTimeUnit	Change time units of dynamic system
d2d	Resample discrete-time model
d2c	Convert model from discrete to continuous time
c2d	Convert model from continuous to discrete time
merge	Merge estimated models

Analysis and Simulation

sim	Simulate response of identified model
predict	Predict state and state estimation error covariance at next time step using extended or unscented Kalman filter, or particle filter
compare	Compare identified model output and measured output
impulse	Impulse response plot of dynamic system; impulse response data
step	Step response plot of dynamic system; step response data
bode	Bode plot of frequency response, or magnitude and phase data
data2state	Map past data to states of state-space and nonlinear ARX models
findstates	Estimate initial states of model

Information Extraction and Interpretation

idssdata	State-space data of identified system
get	Access model property values
getpar	Obtain attributes such as values and bounds of linear model parameters
getcov	Parameter covariance of identified model
advice	Analysis and recommendations for data or estimated linear models

Conversion to Other Model Structures

idpoly	Polynomial model with identifiable parameters
idtf	Transfer function model with identifiable parameters
idfrd	Frequency response data or model

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.

```
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 parameterization 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 as the model time unit.

You can use name-value pair arguments to specify additional model properties during 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, change the output name.

```
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 nonzero 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 to 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 nonzero, and false everywhere else. This setting fixes the values of the zero entries in sys.A.

The remaining lines fix all the values in sys.B and sys.K to the values that you specified during model creation.

Convert Transfer Function into State-Space Model

Model a dynamic system using a transfer function. Then use idss to convert the transfer-function model into state-space form.

Using idtf, construct a continuous-time, single-input, single-output (SISO) transfer function described by:

$$G(s) = \frac{s + 4}{s^2 + 20s + 5}$$

```
num = [1 4];
den = [1 20 5];
G = idtf(num,den)
```

```
G =
 
      s + 4
  --------------
  s^2 + 20 s + 5
 
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:
Created by direct construction or transformation. Not estimated.
```

Convert the transfer function into state-space form.

```
sys0 = idss(G)
```

```
sys0 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)
 
  A = 
          x1    x2
   x1    -20  -2.5
   x2      2     0
 
  B = 
       u1
   x1   2
   x2   0

  C = 
        x1   x2
   y1  0.5    1

  D = 
       u1
   y1   0

  K = 
       y1
   x1   0
   x2   0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Array of State-Space Models

Create an array of state-space models.

You can create an array of state-space models in one of several ways:

- 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.

See Also

`idssdata` | `ssest` | `ssestOptions` | `n4sid` | `pem` | `idgrey` | `idpoly` | `idproc` | `idtf` | `translatecov`

Topics

"Dynamic System Models"
"What Are State-Space Models?"
"Canonical State-Space Realizations"
"Estimate State-Space Models with Structured Parameterization"

Introduced in 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
`ssdata` | `idss` | `tfdata` | `zpkdata` | `polydata`

Introduced in R2012a

idtf

Transfer function model with identifiable parameters

Description

An `idtf` model represents a system as a continuous-time or discrete-time transfer function with identifiable (estimable) coefficients. Use `idtf` to create a transfer function model, or to convert “Dynamic System Models” to transfer function form.

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.

Unlike `idss` and `idpoly`, `idtf` fixes the noise parameter to 1 rather than parameterizing it. So, in $y = Gu + He$, $H = 1$.

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.

Creation

You can obtain an `idtf` model object in one of three ways.

- 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. When you reference numerator and denominator properties, you can use the shortcuts `num` and `den`. The `Report` property of the resulting model stores information about the estimation, such as handling of initial conditions and options used in estimation. For example, you can use the following commands to estimate and get information about a transfer function.

  ```
  sys = tfest(data,nx);
  num = sys.Numerator;
  ```

```
den = sys.den;
sys.Report
```

For more examples of estimating an `idtf` model, see `tfest`.

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. For example, create an `idtf` model with the numerator and denominator that you specify.

  ```
  sys = idtf(num,den)
  ```

  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. For examples, see “Create Continuous-Time Transfer Function Model” on page 1-745 and “Create Discrete-Time Transfer Function” on page 1-746.
- Convert an existing dynamic system model to an `idtf` model using the `idtf` command. For example, convert the state-space model `sys_ss` to a transfer function.

  ```
  sys_tf = idtf(sys_ss);
  ```

  For a more detailed example, see “Convert Identifiable State-Space Model to Identifiable Transfer Function” on page 1-748

For information on functions you can use to extract information from or transform `idtf` model objects, see “Object Functions” on page 1-744.

Syntax

```
sys = idtf(numerator,denominator)
sys = idtf(numerator,denominator,Ts)
sys = idtf( ___ ,Name,Value)

sys = idtf(sys0)
```

Description

Create Transfer Function Model

`sys = idtf(numerator,denominator)` creates a continuous-time transfer function model with identifiable parameters. `numerator` specifies the current values of the transfer function numerator coefficients. `denominator` specifies the current values of the transfer function denominator coefficients.

`sys = idtf(numerator,denominator,Ts)` creates a discrete-time transfer function model with sample time `Ts`.

`sys = idtf( ___ ,Name,Value)` creates a transfer function with the properties on page 1-736 specified by one or more `Name,Value` pair arguments. Specify name-value pair arguments after any of the input argument combinations in the previous syntaxes.

Convert Dynamic System Model to Transfer Function Model

`sys = idtf(sys0)` converts any dynamic system model `sys0` to `idtf` model form.

Input Arguments

`sys0` — Dynamic system
dynamic system model

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`.

Properties

`Numerator` — Values of transfer function numerator coefficients
vector | cell array

Values of transfer function numerator coefficients, specified as a row vector or a cell array.

For SISO transfer functions, the values of the numerator coefficients are stored as a row vector in the following order:

- 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. For an example of a MIMO transfer function, see “Create MIMO Discrete-Time Transfer Function” on page 1-747.

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 `numerator` 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`.

`Denominator` — Values of transfer function denominator coefficients
vector | cell array

Values of transfer function denominator coefficients, specified as a row vector or a cell array.

For SISO transfer functions, the values of the denominator coefficients are stored as a row vector in the following order:

- 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 an *Ny*-by-*Nu* cell array of denominator coefficients for each input-output pair. For an example of a MIMO transfer function, see "Create MIMO Discrete-Time Transfer Function" on page 1-747.

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 `denominator` 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`.

`Variable` — Transfer function display variable
`'s'` (default) | `'p'` | `'z^-1'` | `'q^-1'`

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 `Numerator` and `Denominator` coefficient vectors for discrete-time models. When `Variable` is set to `'z^-1'` or `'q^-1'`, the coefficient vectors are ordered as ascending powers of the variable.

For an example of using the `Variable` property, see "Specify Transfer Function Display Variable" on page 1-747.

`IODelay` — Transport delays
`0` (default) | numeric array

Transport delays, specified as a numeric array containing a separate transport delay for each input-output pair.

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` as an *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.

If you create an `idtf` model `sys` using the `idtf` command, then `sys.IODelay` contains the initial values of the transport delay that you specify with a name-value pair argument.

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`.

`Structure` — Information about estimable parameters
structure property values | array of structure property values

Property-specific information about the estimable parameters of the `idtf` model, specified as a single structure or an array of structures.

- SISO system — Single structure.
- MIMO system with *Ny* outputs and *Nu* inputs — *Ny*-by-*Nu* array. The element `Structure(i,j)` contains information corresponding to the transfer function for the `(i,j)` input-output pair.

`Structure.Numerator`, `Structure.Denominator`, and `Structure.IODelay` contain information about the numerator coefficients, denominator coefficients, and transport delay, respectively. Each parameter in `Structure` contains the following fields.

Field	Description	Examples
Value	Parameter values. Each property is an alias of the corresponding `Value` entry in the `Structure` property of `sys`.NaN represents unknown parameter values.	`sys.Structure.Numerator.Value` contains the initial or estimated values of SISO numerator coefficients. `sys.Numerator` is an alias of the value of this property. `sys.Numerator{i,j}` is the alias of the MIMO property `sys.Structure(i,j).Numerator.Value`.
Minimum	Minimum value that the parameter can assume during estimation.	`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.	`sys.Structure.IODelay.Maximum = 0.5` constrains the transport delay to values less than or equal to 0.5. `sys.Structure.IODelay.Maximum` must be greater than or equal to zero.

Field	Description	Examples
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` value to `false`. 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).	`sys.Structure.Denominator.Free = false` fixes all of the denominator coefficients in `sys` to the values specified in `sys.Structure.Denominator.Value`.
Scale	Scale of the value of the parameter. The estimation algorithm does not use `Scale`.	
Info	Structure array that contains the fields `Label` and `Unit` for storing parameter labels and units. Specify parameter labels and units as character vectors.	`'Time'`

`NoiseVariance` — Variance of model innovations
scalar | matrix

Variance (covariance matrix) of the model innovations *e*, specified as a scalar or matrix.

- SISO model — Scalar
- MIMO model with N_y outputs — N_y-by-N_y matrix

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.

`Report` — Summary report
report field values

This property is read-only.

Summary report that contains information about the estimation options and results for a transfer function model obtained using estimation commands, such as `tfest` and `impulseest`. Use `Report` to find estimation information for the identified model, including:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit and other quality metrics

If you create the model by construction, the contents of `Report` are irrelevant.

```
sys = idtf([1 4],[1 20 5]);
sys.Report.OptionsUsed

ans =

     []
```

If you obtain the model using estimation commands, the fields of `Report` contain information on the estimation data, options, and results.

```
load iddata2 z2;
sys = tfest(z2,3);
sys.Report.OptionsUsed

 InitializeMethod: 'iv'
     InitializeOptions: [1×1 struct]
      InitialCondition: 'auto'
               Display: 'off'
           InputOffset: []
          OutputOffset: []
    EstimateCovariance: 1
        Regularization: [1×1 struct]
          SearchMethod: 'auto'
         SearchOptions: [1×1 idoptions.search.identsolver]
       WeightingFilter: []
      EnforceStability: 0
          OutputWeight: []
              Advanced: [1×1 struct]
```

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
`0` (default) | scalar | vector

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, setting `InputDelay` to `3` specifies a delay of three sample times.

For a system with N_u inputs, set `InputDelay` to an N_u-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.

Estimation treats `InputDelay` as a fixed constant of the model. Estimation uses the `IODelay` property for estimating time delays. To specify initial values and constrains for estimation of time delays, use `sys.Structure.IODelay`.

`OutputDelay` — Output delay for each output channel
`0` (default)

For identified systems such as `idtf`, `OutputDelay` is fixed to zero.

`Ts` — Sample Time
`0` (default) | `-1` | positive scalar

Sample time, specified as one of the following.

- Continuous-time model — `0`
- Discrete-time model with a specified sampling time — Positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model
- Discrete-time model with unspecified sample time — `-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.

`TimeUnit` — Units for time variable
`'seconds'` (default) | `'nanoseconds'` | `'microseconds'` | `'milliseconds'` | `'minutes'` | `'hours'` | `'days'` | `'weeks'` | `'months'` | `'years'`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as a scalar.

Changing this property does not resample or convert the data. Modifying the property changes only the interpretation of the existing data. Use `chgTimeUnit` to convert data to different time units.

`InputName` — Input channel names
`''` (default) | character vector | cell array

Input channel names, specified as a character vector or cell array.

- Single-input model — Character vector, for example, `'controls'`
- Multi-input model — Cell array of character vectors

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`.

You can use input channel names in several ways, including:

- To identify channels on model display and plots
- To extract subsystems of MIMO systems
- To specify connection points when interconnecting models

`InputUnit` — Input channel units
`''` (default) | character vector | cell array

Input channel units, specified as a character vector or cell array.

- Single-input model — Character vector

- Multi-input Model — Cell array of character vectors

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

`InputGroup` — Input channel groups
`struct` with no fields (default) | `struct`

Input channel groups, specified as a structure. The `InputGroup` property lets you divide the input channels of MIMO systems into groups so that you can refer to each group by name. In the `InputGroup` structure, set field names to the group names, and field values to the input channels belonging to each group.

For example, create input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively.

```
sys.InputGroup.controls = [1 2];
sys.InputGroup.noise = [3 5];
```

You can then extract the subsystem from the `controls` inputs to all outputs using the following syntax:

```
sys(:,'controls')
```

`OutputName` — Output channel names
`''` (default) | character vector | cell array

Output channel names, specified as a character vector or cell array.

- Single-input model — Character vector, for example, `'measurements'`
- Multi-input model — Cell array of character vectors

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`.

You can use output channel names in several ways, including:

- To identify channels on model display and plots
- To extract subsystems of MIMO systems
- To specify connection points when interconnecting models

`OutputUnit` — Output channel units
`''` (default) | character vector | cell array

Output channel units, specified as a character vector or cell array.

- Single-input model — Character vector, for example, `'seconds'`
- Multi-input Model — Cell array of character vectors

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

`OutputGroup` — Output channel groups

`struct` with no fields (default) | `struct`

Output channel groups, specified as a structure. The `OutputGroup` property lets you divide the output channels of MIMO systems into groups and refer to each group by name. In the `OutputGroup` structure, set field names to the group names, and field values to the output channels belonging to each group.

For example, create output groups named `temperature` and `measurement` that include output channels 1 and 3, 5, respectively.

```
sys.OutputGroup.temperature = [1];
sys.OutputGroup.measurement = [3 5];
```

You can then extract the subsystem from all inputs to the `measurement` outputs using the following syntax:

```
sys('measurement',:)
```

`Name` — System Name

`''` (default) | character vector

System name, specified as a character vector, for example, `'system_1'`.

`Notes` — Notes on system

`0`-by-1 string (default) | string | character vector

Any text that you want to associate with the system, specified as a string or a cell array of character vectors. The property stores whichever data type you provide. For instance, if `sys1` and `sys2` are dynamic system models, you can set their `Notes` properties as follows.

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

`UserData` — Data to associate with system

`[]` (default) | any MATLAB data type

Data to associate with the system, specified as any MATLAB data type.

SamplingGrid — Sampling grid
[] (default) | struct

Sampling grid for model arrays, specified as a structure.

For arrays of identified linear (IDLTI) models that you derive 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 must be numeric and scalar valued, and all arrays of sampled values must match the dimensions of the model array.

For example, suppose that 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)
```

Here, `sys` is an array containing three identified models obtained at 1000, 5000, and 10000 rpm, respectively.

For model arrays that you generate 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.

Object Functions

In general, any function applicable to “Dynamic System Models” is applicable to an `idtf` model object. These functions are of four general types.

- Functions that operate and return `idtf` model objects enable you to transform and manipulate `idtf` models. For instance:
  - Use `merge` to merge estimated `idtf` models.
  - Use `c2d` to convert an `idtf` from continuous to discrete time. Use `d2c` to convert an `idtf` from discrete to continuous time.
- Functions that perform analytical and simulation functions on `idtf` objects, such as `bode` and `sim`
- Functions that retrieve or interpret model information, such as `advice` and `getpar`
- Functions that convert `idtf` objects into a different model type, such as `idpoly` for time domain or `idfrd` for frequency domain

The following lists contain a representative subset of the functions that you can use with `idtf` models.

Transformation and Manipulation

translatecov	Translate parameter covariance across model transformation operations
setpar	Set attributes such as values and bounds of linear model parameters

chgTimeUnit	Change time units of dynamic system
d2d	Resample discrete-time model
d2c	Convert model from discrete to continuous time
c2d	Convert model from continuous to discrete time
merge	Merge estimated models

Analysis and Simulation

sim	Simulate response of identified model
predict	Predict state and state estimation error covariance at next time step using extended or unscented Kalman filter, or particle filter
compare	Compare identified model output and measured output
impulse	Impulse response plot of dynamic system; impulse response data
step	Step response plot of dynamic system; step response data
bode	Bode plot of frequency response, or magnitude and phase data

Information Extraction and Interpretation

tfdata	Access transfer function data
get	Access model property values
getpar	Obtain attributes such as values and bounds of linear model parameters
getcov	Parameter covariance of identified model
advice	Analysis and recommendations for data or estimated linear models

Conversion to Other Model Structures

idpoly	Polynomial model with identifiable parameters
idss	State-space model with identifiable parameters
idfrd	Frequency response data or model

Examples

Create 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 given by the following equation:

$$G(s) = \frac{s + 4}{s^2 + 20s + 5}$$

```
num = [1 4];
den = [1 20 5];
G = idtf(num,den)
```

```
G =
 
      s + 4
  --------------
  s^2 + 20 s + 5
 
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:
Created by direct construction or transformation. Not estimated.
```

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 with initial values 1 and 4 are estimable parameters. The denominator coefficients with 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 parameterization 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 the following equation:

$$G(s) = e^{-5.8s}\frac{5}{s+5}$$

Label the input of the transfer function with the name `'Voltage'` and specify the input units as `volt`.

Use name-value pair arguments 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 parameterization 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 given by the following equation:

$$H(z) = \frac{z-0.1}{z+0.8}$$

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 given by the following equation:

$$H(z) = \begin{bmatrix} \frac{1}{z+0.2} & \frac{z}{z+0.7} \\ \frac{-z+2}{z-0.3} & \frac{3}{z+0.3} \end{bmatrix}$$

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 given by the following 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 the time-domain system-response data `z1`.

```
load iddata1 z1;
```

Set the number of poles `np` to `2` and estimate a transfer function.

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

`sys` is an `idtf` model containing the estimated two-pole transfer function.

View the numerator and denominator coefficients of the resulting estimated model `sys`.

```
sys.Numerator
```

```
ans = 1×2

    2.4554  176.9856
```

```
sys.Denominator
```

```
ans = 1×3

    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 $a = k/10$.

See Also

`tfdata` | `getcov` | `getpar` | `idpoly` | `idss` | `idproc` | `idfrd` | `oe` | `tfest` | `translatecov`

Topics

"Dynamic System Models"
"What are Transfer Function Models?"
"Estimate Transfer Function Models With Prior Knowledge of Model Structure and Constraints"

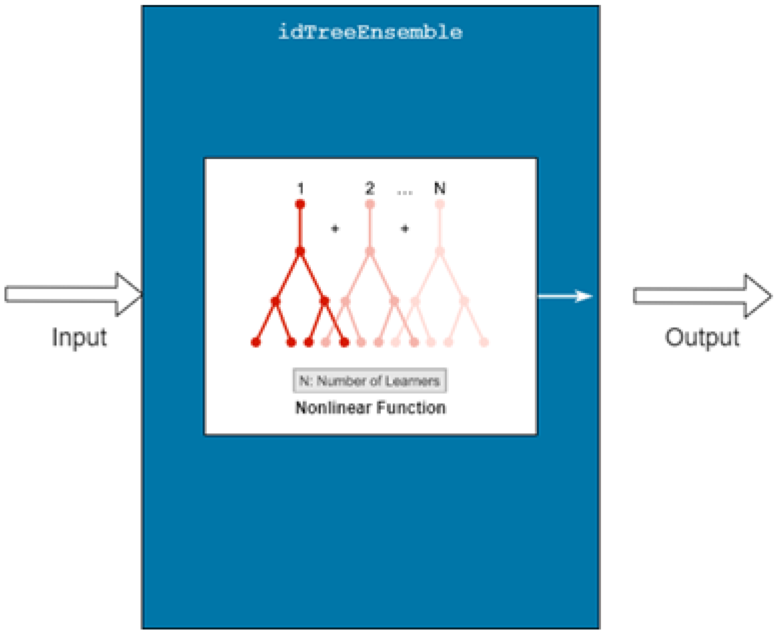
Introduced in R2012a

idTreeEnsemble

Decision tree ensemble mapping function for nonlinear ARX models (requires Statistics and Machine Learning Toolbox)

Description

An `idTreeEnsemble` object implements a decision tree ensemble model, and is a nonlinear mapping function for estimating nonlinear ARX models. This mapping object lets you use `RegressionBaggedEnsemble`, `RegressionEnsemble`, and `CompactRegressionEnsemble` objects that are created using Statistics and Machine Learning Toolbox. Unlike most other mapping objects for `idnlarx` models, which typically contain offset, linear, and nonlinear components, the `idTreeEnsemble` model contains only a nonlinear component.



Mathematically, the `idTreeEnsemble` object maps *m* inputs $x(t) = [x_1(t),x_2(t),\ldots,x_m(t)]^T$ to a scalar output $y(t)$ using a decision tree regression ensemble model.

Here:

- $x(t)$ is an *m*-by-1 vector of inputs, or regressors.
- $y(t)$ is the scalar output.

For more information about creating regression tree ensembles, see `fitrensemble`.

Use `idTreeEnsemble` as the value of the `OutputFcn` property of an `idnlarx` model. For example, specify `idTreeEnsemble` when you estimate an `idnlarx` model with the following command.

```
sys = nlarx(data,regressors,idTreeEnsemble)
```

When `nlarx` estimates the model, it essentially estimates the parameters of the `idTreeEnsemble` object.

You can configure the `idTreeEnsemble` function to set options and fix parameters. To modify the estimation options, set the option property in `E.EstimationOptions`, where `E` is the `idTreeEnsemble` object. For example, to change the fit method to `'lsboost-resampled'`, use `E.EstimationOptions.FitMethod = 'lsboost-resampled'`. To fix the values of an existing estimated `idTreeEnsemble` during subsequent `nlarx` estimations, set the `Free` property to `False`. Use `evaluate` to compute the output of the function for a given vector of regressor inputs.

Creation

Syntax

```
E = idTreeEnsemble
E = idTreeEnsemble(fitmethod)
```

Description

`E = idTreeEnsemble` creates an empty `idTreeEnsemble` object `E` with the default estimation fit method of `'bag'`. The number of regressor inputs is determined during model estimation and the number of `idTreeEnsemble` outputs is 1.

`E = idTreeEnsemble(fitmethod)` sets the ensemble estimation method to the value in `fitmethod`.

Input Arguments

`fitmethod` — Ensemble estimation method
`'bag'` (default) | `'lsboost-reweighted'` | `'lsboost-resampled'`

Method to use for estimating the parameters of the `idTreeEnsemble` model, specified as `'bag'`, `'lsboost-reweighted'`, or `'lsboost-resampled'`.

This argument sets the property `E.EstimationOptions.FitMethod`. For more information, see `Estimation Options`.

Properties

`RegressionEnsembleModel` — Regression tree ensemble model
`RegressionBaggedEnsemble` | `RegressionEnsemble` | `CompactRegressionEnsemble`

Regression tree ensemble, specified as a `RegressionBaggedEnsemble`, `RegressionEnsemble`, or `CompactRegressionEnsemble` object. This property is empty when you first create an `idTreeEnsemble` object. The estimation process determines the ensemble type based on the values of the `FitMethod` and `Shrink` properties, as shown in the following table.

FitMethod	Shrink	Model Type
'bag'	false	RegressionBaggedEnsemble
'lsboost-reweighted'	false	RegressionEnsemble

FitMethod	Shrink	Model Type
'lsboost-resampled'	false	RegressionBaggedEnsemble
'bag' or 'lsboost-reweighted' or 'lsboost-resampled'	true	CompactRegressionEnsemble

free — Option to update parameters
true (default) | false

Option to update the parameters of RegressionEnsembleModel during nonlinear ARX model estimation, specified as true or false. When free is true, the estimation process updates the ensemble model when it estimates the idnlarx model that contains it. When free is false, the ensemble model is fixed during estimation. Setting free to false is useful when you are using a previously estimated ensemble model as a mapping function for nlarx.

Input — Input signal information
signal property values

Input signal information for signals used for estimation, specified as vectors of *m* property-specific values, where *m* is the number of input signals. The Input properties for each input signal are as follows:

- Name — Names of the input signals, specified as a 1-by-*m* string or character array, where *m* is the number of inputs
- Mean — Mean of the input signals, specified as a numeric scalar
- Range — Ranges of the input signals, specified as a 2-by-*m* numeric array that contains the minimum and maximum values

Output — Output signal information
signal property values

Output signal information, specified as property-specific values. The Output properties are as follows:

- Name — Name of the output signal, specified as a string or a character array
- Mean — Mean of the output signal, specified as a numeric scalar
- Range — Range of the output signal, specified as a 2-by-1 numeric array that contains the minimum and maximum values

Estimation Options — Estimation options
estimation option property values

Estimation options for the idTreeEnsemble model, specified as follows. For more information on any of these options, see the corresponding name-value argument in fitrensemble.

- FitMethod — Method to use for estimating the parameters of the idTreeEnsemble model, specified as one of the items in the following table.

Option	Description
'bag'	Bagging (bootstrap aggregation) (default)

Option	Description
'lsboost-reweighted'	Least-squares boosting with reweighting
'lsboost-resampled'	Least-squares boosting with resampling

- NumLearningCycles — Number of ensemble learning cycles, specified as a positive integer. The default value is 100.
- ResampleData — Option to resample the data, specified as 'on' (default) or 'off'.
  - If FitMethod is set to 'bag', then ResampleData must be set to 'on'.
  - If FitMethod is set to 'lsboost-reweighted', then ResampleData has no effect.
- ResampleFraction — Fraction of training set to resample, specified as a positive scalar in (0,1].
  - If FitMethod is set to 'lsboost-reweighted', then ResampleFraction has no effect.
- ReplaceData — Option to sample with replacement, specified as 'off' (default) or 'on'. This property has an effect only if either FitMethod is set to 'bag' or Resample is set to true and FitMethod is set to 'lsboost-resampled' .
- Regularize — Option to find optimal weights for learners, specified as true (default) or false.
- Shrink — Option to prune ensemble and return a compact version, specified as true (default) or false.

Examples

Estimate Nonlinear ARX Model with idTreeEnsemble as output function

Load the data mrdamper. This data contains damping force (F) and velocity (V) information for a fluid damper, with a sample time of Ts.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','mrdamper'))
```

Create an iddata object data that uses F as the output and V as the input. Divide data into estimation and validation data sets ze and zv.

```
data = iddata(F,V,Ts);
ze = data(1:3000);
zv = data(3001:end);
```

Create an empty idTreeEnsemble mapping object E.

```
E = idTreeEnsemble;
```

Estimate a nonlinear ARX model sys that uses E for the output function.

```
sys = nlarx(ze,[16 16 0],E);
```

```
Warning: Reached maximum number of iterations. 'Lambda' = 247.7731.
```

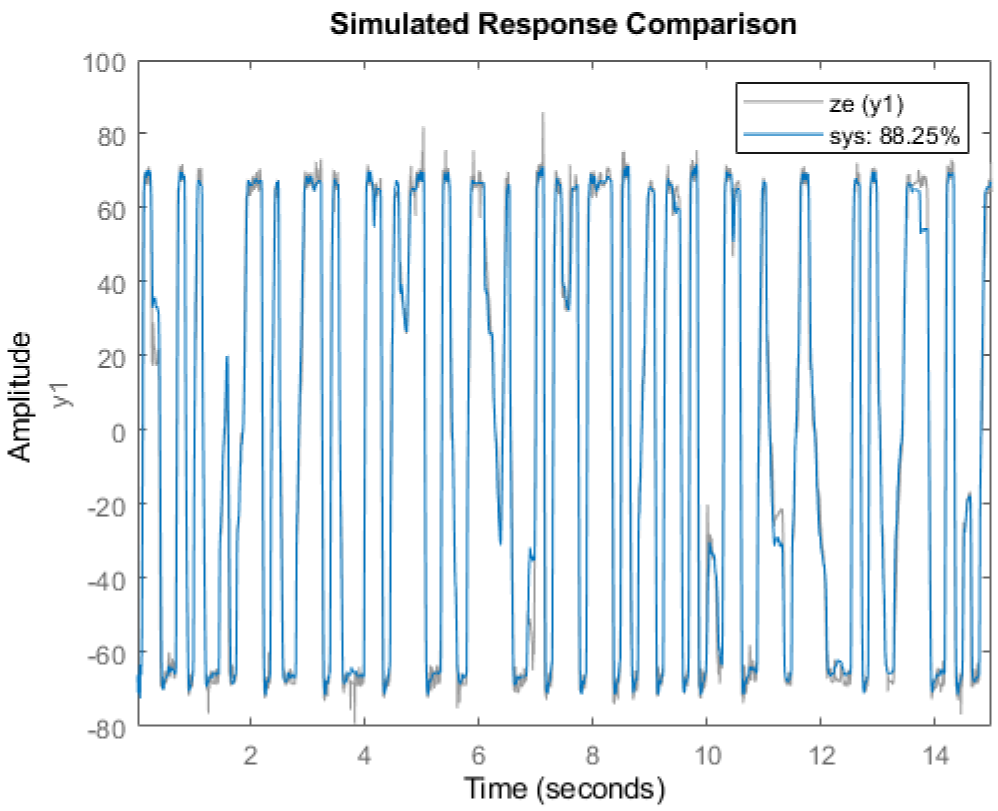
The model stores the estimated mapping object in the property sys.OutputFcn.

```
sys.OutputFcn
```

```
ans =
Regression Tree Ensemble
Inputs: y1(t-1), y1(t-2), y1(t-3), y1(t-4), y1(t-5), y1(t-6), y1(t-7), y1(t-8), y1(t-9), y1(t-10
Output: y1

 Nonlinear Function: Regression Tree Ensemble

                      Input: 'Function inputs'
                     Output: 'Function output'
    RegressionEnsembleModel: [1×1 classreg.learning.regr.RegressionBaggedEnsemble]
                       Free: 1
          EstimationOptions: 'Estimation option set'
```

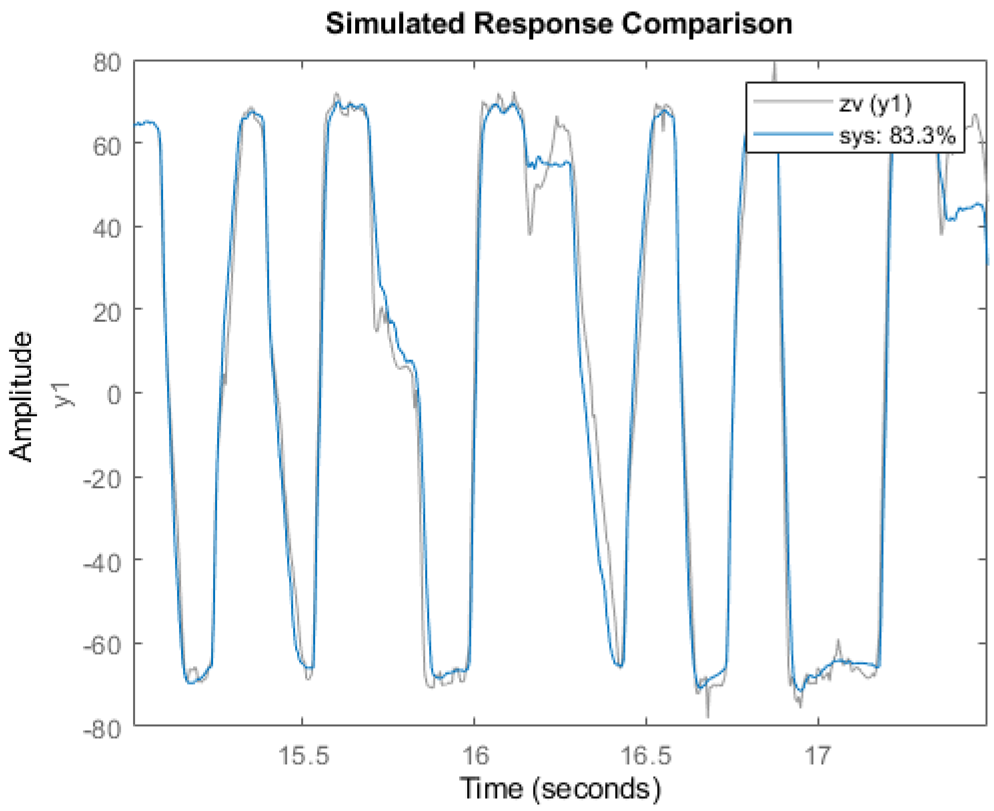
Compare the model simulated output to the estimation data output.

```
compare(ze,sys)
```



Compare the model simulated output to the validation data output.

```
compare(zv,sys)
```



sys shows a good fit to both the estimation data and the validation data.

See Also

nlarx | RegressionBaggedEnsemble | RegressionEnsemble | CompactRegressionEnsemble | fitrensemble | idLinear | idTreePartition | idWaveletNetwork | idSigmoidNetwork | idFeedforwardNetwork | idCustomNetwork | idGaussianProcess | idnlarx | evaluate

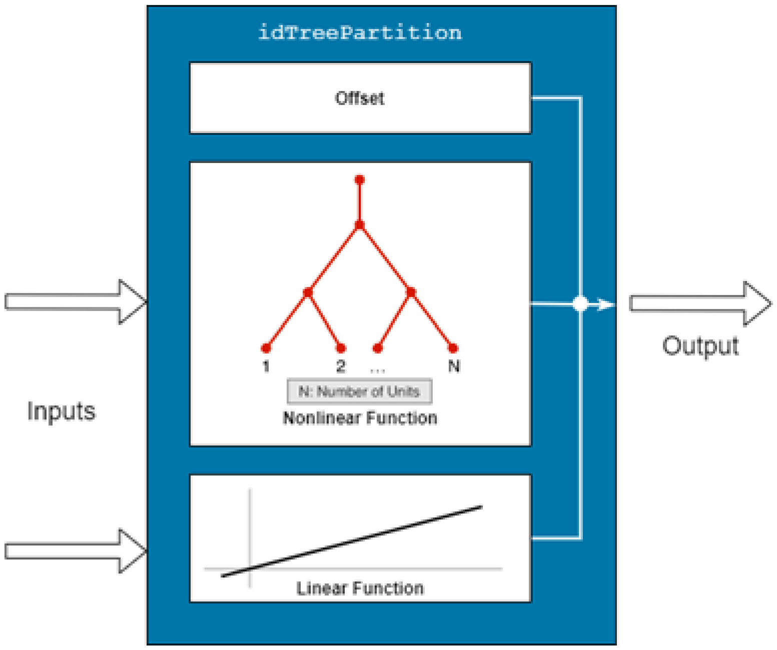
Introduced in R2021b

idTreePartition

Tree-partitioned nonlinear function for nonlinear ARX models

Description

An `idTreePartition` object implements a tree-partitioned nonlinear function, and is a nonlinear mapping function for estimating nonlinear ARX models. The mapping function, which is also referred to as a nonlinearity, uses a combination of linear weights, an offset and a nonlinear function to compute its output. The nonlinear function contains `idTreePartition` unit functions that operate on a radial combination of inputs.



Mathematically, `idTreePartition` is a nonlinear function $y = F(x)$ that maps m inputs $X(t) = [x(t_1),x_2(t),\ldots,x_m(t)]^T$ to a scalar output $y(t)$. F is a piecewise-linear (affine) function of x:

$$F(x) = xL + [1, x]C_k + d$$

Here, x belongs to the partition P_k. L is a 1-by-m vector, C_k is a 1-*by*-m+1 vector, and P_k is a partition of the x-space.

For more information about the mapping function $F(x)$ see “Algorithms” on page 1-763.

Use `idTreePartition` as the value of the `OutputFcn` property of an `idnlarx` model. For example, specify `idTreePartition` when you estimate an `idnlarx` model with the following command.

```
sys = nlarx(data,regressors,idTreePartition)
```

When `nlarx` estimates the model, it essentially estimates the parameters of the `idTreePartition` function.

You can configure the `idTreePartition` function to fix parameters. To omit the linear component, set `LinearFcn.Use` to `false`. Use `evaluate` to compute the output of the function for a given vector of inputs.

Creation

Syntax

```
T = idTreePartition
T = idTreePartition(numUnits)
```

Description

`T = idTreePartition` creates a `idTreePartition` object `t` that is a binary tree nonlinear mapping object. The function computes the number of tree nodes *J*, represented by the property `NumberOfUnits`, automatically during estimation. The tree has the number of leaves equal to `2^(J+1)-1`.

`T = idTreePartition(numUnits)` specifies the number of `idTreePartition` nodes `numUnits`.

Input Arguments

`numUnits` — Number of units
`'auto'` (default) | positive integer

Number of units, specified as `'auto'` or a positive integer. `numUnits` determines the number of tree nodes.

This argument sets the `T.NonlinearFcn.NumberOfUnits` property.

Properties

`Input` — Input signal information
signal property values

Input signal information for signals used for estimation, specified as vectors of *m* property-specific values, where *m* is the number of input signals. The `Input` properties for each input signal are as follows:

- `Name` — Names of the input signals, specified as a 1-by-*m* string or character array, where *m* is the number of inputs
- `Mean` — Mean of the input signals, specified as a numeric scalar
- `Range` — Ranges of the input signals, specified as a 2-by-*m* numeric array that contains the minimum and maximum values

`Output` — Output signal information
signal property values

Output signal information, specified as property-specific values. The `Output` properties are as follows:

- `Name` — Name of the output signal, specified as a string or a character array
- `Mean` — Mean of the output signal, specified as a numeric scalar
- `Range` — Range of the output signal, specified as a 2-by-1 numeric array that contains the minimum and maximum values

`LinearFcn` — Parameters of linear function
linear function property values (default)

Parameters of the linear function, specified as follows:

- `Value` — Value of *L*, specified as a 1-by-*m* vector.
- `Free` — Option to update entries of `Value` during estimation. specified as a logical scalar. The software honors the `Free` specification only if the starting value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a 1-by-*p* vector. If `Minimum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that minimum bound during model estimation.
- `Maximum` — Maximum bound on `Value`, specified as a 1-by-*p* vector. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation.
- `SelectedInputIndex` — Indices of `idTreePartition` inputs (see `Input.Name`) that are used as inputs to the linear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. The `RegressorUsage` property of the `idnlarx` model determines these indices.

The software computes the value of `LinearFcn` as `(X-X0)'*L`, where `X0` is the regressor mean.

`Offset` — Parameters of offset term
offset property values

Parameters of the offset term, specified as follows:

- `Value` — Offset value, specified as a scalar.
- `Free` — Option to update `Value` during estimation, specified as a scalar logical. The software honors the `Free` specification of `false` only if the value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a numeric scalar or `–Inf`. If `Minimum` is specified with a finite value and the value of `Value` is finite, then the software enforces that minimum bound during model estimation. The default value is `-Inf`.
- `Maximum` — Maximum bound on `Value`, specified as a numeric scalar or `Inf`. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation. The default value is `Inf`.

`NonlinearFcn` — Parameters of nonlinear function
nonlinear function property values

Parameters of the nonlinear function, specified as follows:

- `NumberOfUnits` — Number of units, specified as `'auto'` or a positive integer. `NumberOfUnits` determines the number of nodes *N* in the tree. When *N* is set to:

  - `'auto'`, the software selects *N* by pruning.

  - a positive integer before estimation, then the software sets N to the largest value of the form `2^(J+1)-1` less than this integer.
- `Parameters` — estimated parameter values.`idTreePartition`, specified as in the following table:

Field Name	Description
`SampleLength`	Length of the estimation data
`NoiseVariance`	Estimated variance of the noise in the estimation data
`Tree`	Structure that contains the tree parameters, as described in the following list: • `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.

- `Free` — Option to estimate parameters, specified as a logical scalar. If all the parameters have finite values, such as when the `idTreePartition` object corresponds to a previously estimated model, then setting `Free` to `false` causes the parameters of the nonlinear component of the function *F*(*X*) to remain unchanged during estimation. The default value is `true`.
- `SelectedInputIndex` — Indices of `idTreePartition` inputs (see `Input.Name`) that are used as inputs to the nonlinear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. The `RegressorUsage` property determines these indices.
- `Structure` — Advanced options that affect the initial model.

Property	Description	Default
`FinestCell`	Integer or character vector specifying the minimum number of data points in the smallest partition.	`'auto'`
`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	`1.0`

Property	Description	Default
`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.	`1e-6`

Examples

Estimate Nonlinear ARX Model with `idTreePartition` as Output Function

Load the data

```
load iddata1 z1
```

Create an `idTreePartition` object with default settings.

```
T = idTreePartition
```

```
T = 
Tree Partition

 Nonlinear Function: Tree Partition with number of units chosen automatically
 Linear Function: uninitialized
 Output Offset: uninitialized

               Input: 'Function inputs'
              Output: 'Function output'
           LinearFcn: 'Linear function parameters'
        NonlinearFcn: 'Tree structure'
              Offset: 'Offset parameters'
   EstimationOptions: 'Estimation options'
```

Estimate a nonlinear ARX model using `T`.

```
sys = nlarx(z1,[2 2 1],T);
```

View the output function of `sys`.

```
disp(sys.OutputFcn)
```

```
Tree Partition
Inputs: y1(t-1), y1(t-2), u1(t-1), u1(t-2)
Output: y1

 Nonlinear Function: Tree Partition with 31 units
 Linear Function: initialized to [1.19 -0.419 0.873 0.844]
 Output Offset: initialized to 0.249

               Input: '<Function inputs>'
              Output: '<Function output>'
```

```
            LinearFcn: '<Linear function parameters>'
         NonlinearFcn: '<Tree structure>'
               Offset: '<Offset parameters>'
    EstimationOptions: '<Estimation options>'
```

T now has 31 nodes.

Specify idTreePartition Parameters

Load the data

```
load iddata7 z7
ze = z7(1:300);
```

Create an idTreePartition object and use dot notation to set parameters.

```
T = idTreePartition;
T.Offset.Value = 0.2;
T.Offset.Free = false;
T.NonlinearFcn.NumberOfUnits = 30;
```

Specify model regressors.

```
Reg1 = linearRegressor({'y1','u1'},{1:4, 0:4});
Reg2 = polynomialRegressor({'y1','u1'},{1:2, 0:2},2);
```

Estimate a nonlinear ARX model.

```
sys = nlarx(ze, [Reg1;Reg2], T);
```

View postestimation OutputFcn properties.

```
sys.OutputFcn
```

```
ans = 
Tree Partition
Inputs: y1(t-1), y1(t-2), y1(t-3), y1(t-4), u1(t), u1(t-1), u1(t-2), u1(t-3), u1(t-4), y1(t-1)^2
Output: y1

 Nonlinear Function: Tree Partition with 15 units
 Linear Function: initialized to [0.0725 0.895 -0.0727 -0.475 0.0725 -0.106 0.0304 1.02 1.43 0.0
 Output Offset: fixed to 0.2

                Input: 'Function inputs'
               Output: 'Function output'
            LinearFcn: 'Linear function parameters'
         NonlinearFcn: 'Tree structure'
               Offset: 'Offset parameters'
    EstimationOptions: 'Estimation options'
```

```
sys.OutputFcn.Input
```

```
ans = 
Function inputs
```

```
     Name: {1x14 cell}
     Mean: [0.2033 0.2035 0.2149 0.2159 0.0405 0.0338 0.0338 0.0338 ... ]
    Range: [2x14 double]
```

```
disp(sys.OutputFcn.Offset)
```

```
Output Offset: fixed to 0.2
    Value: 0.2000
     Free: 0
```

```
sys.OutputFcn.NonlinearFcn
```

```
ans = 
Tree structure

         NumberOfUnits: 15
            Parameters: 'Tree Partition parameters'
                  Free: 1
    SelectedInputIndex: [1 2 3 4 5 6 7 8 9 10 11 12 13 14]
```

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:

$$F(x) = xL + [1, x]C_k + d$$

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 `idTreePartition` object, is computed at x, an adaptive algorithm selects the *active node k* of the tree on the branch of partitions that contain x.

When the `Focus` option in `nlarxOptions` is `'prediction'`, `idTreePartition` uses a noniterative technique for estimating parameters. Iterative refinements are not possible for models containing this nonlinearity estimator.

You cannot use `idTreePartition` when `Focus` is `'simulation'` because this nonlinear mapping object is not differentiable. Minimization of simulation error requires differentiable nonlinear functions.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.

Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

Use of previous nonlinearity estimator properties is not recommended

Not recommended starting in R2021a

Starting in R2021a, the properties of the mapping objects, previously known as nonlinearity estimators, have been reorganized. These objects are `idWaveletNetwork` (W), `idSigmoidNetwork` (S), `idTreePartition` (T), `customnet` (C), and `linear` (L). The property changes do not apply to `neuralnet`. The use of the pre-R2021a properties in the following table is discouraged. However, the software still accepts commands that set these properties. There are no plans to exclude such commands at this time.

Pre-R2021a Property	R2021a Property	Applicable Mapping Objects
`NumberOfUnits`	`NonlinearFcn.NumberOfUnits`	W,S,T,C
`LinearTerm`	`LinearFcn.Use`, `Offset.Use`	W,S,C

Pre-R2021a Property	R2021a Property	Applicable Mapping Objects
`Parameters`	Split into three pieces: • `LinearFcn.Value` • `Offset.Value` • `NonlinearFcn.Parameters`	W,S,T,C,L `linear` (L) excludes `NonlinearFcn.Parameters`.
`Options`	`NonlinearFcn.Structure`	W,T

See Also

`nlarx` | `idLinear` | `idSigmoidNetwork` | `idWaveletNetwork` | `idFeedforwardNetwork` | `idCustomNetwork` | `idnlarx` | `evaluate` | `linearRegressor` | `polynomialRegressor`

Introduced in R2007a

idUnitGain

Specify absence of nonlinearities for specific input or output channels in Hammerstein-Wiener models

Syntax

```
unit=idUnitGain
```

Description

`unit=idUnitGain` 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 `idUnitGain` 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,['idSaturation''idUnitGain'],'idDeadZone')
```

In this case, the first input saturates and the output has an associated `deadzone` nonlinearity.

idUnitGain Properties

`idUnitGain` 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,'idSaturation','idUnitGain')
```

In this case, the input has a saturation nonlinearity.

If nonlinearities are absent in input or output channels, you can replace `idUnitGain` 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,[],'idSigmoidNetwork');
```

Tips

Use the `idUnitGain` object to exclude specific input and output channels from being affected by a nonlinearity in Hammerstein-Wiener models. `idUnitGain` is a linear function $y = F(x)$, where *F(x)=x*.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

See Also
`idDeadZone` | `nlhw` | `idSaturation` | `idSigmoidNetwork`

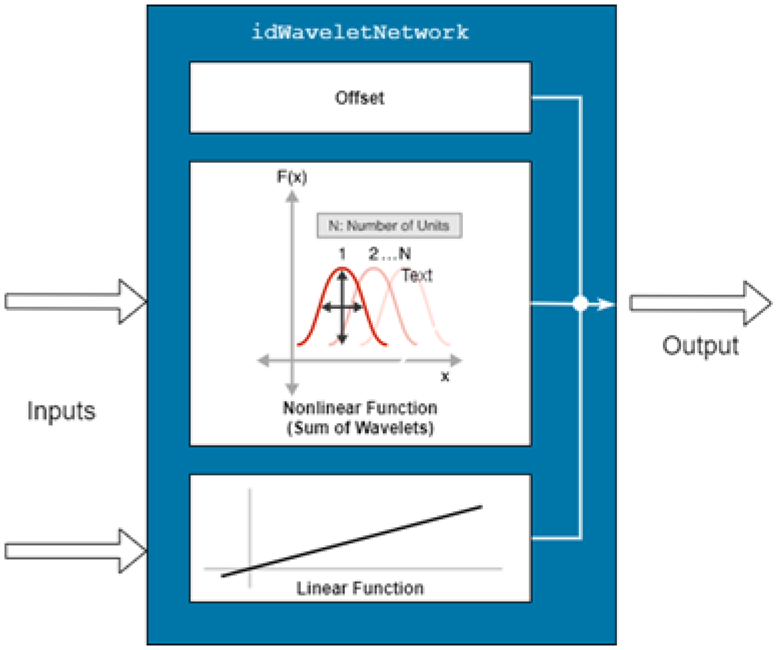
Introduced in R2007a

idWaveletNetwork

Wavelet network function for nonlinear ARX and Hammerstein-Wiener models

Description

An `idWaveletNetwork` object implements a wavelet network function, and is a nonlinear mapping function for estimating nonlinear ARX and Nonlinear Hammerstein-Wiener models. The mapping function, which is also referred to as a nonlinearity, uses a combination of linear weights, an offset and a nonlinear function to compute its output. The nonlinear function contains wavelet unit functions that operate on a radial combination of inputs.



Mathematically, `idWaveletNetwork` is a function that maps m inputs $X(t) = [x(t_1),x_2(t),\ldots,x_m(t)]^T$ to a scalar output $y(t)$ using the following relationship:

$$y(t) = y_0 + (X(t) - \bar{X})^T PL + W(X(t)) + S(X(t))$$

Here:

- $X(t)$ is an m-by-1 vector of inputs, or regressors, with mean $\bar{X}$.
- y_0 is the output offset, a scalar.
- P is an m-by-p projection matrix, where m is the number of regressors and is p is the number of linear weights. m must be greater than or equal to p.
- L is a p-by-1 vector of weights.
- $W(X)$ and $S(X)$ together constitute the nonlinear function of the wavelet network. $W(X)$ is a sum of dilated and translated wavelets while $S(X)$ is a sum of dilated and translated scaling functions

(also known as scalelets). The total number of wavelet d_w and scaling functions d_s is referred to as the number of units of the network.

For definitions of the wavelet function term $W(X)$ and the scaling function term $S(X)$, see “More About” on page 1-778.

Use `idWaveletNetwork` as the value of the `OutputFcn` property of an `idnlarx` model or the `InputNonlinearity` and `OutputLinearity` properties of an `idnlhw` object. For example, specify `idWaveletNetwork` when you estimate an `idnlarx` model with the following command.

```
sys = nlarx(data,regressors,idWaveletNetwork)
```

When `nlarx` estimates the model, it essentially estimates the parameters of the `idWaveletNetwork` function.

You can configure the `idWaveletNetwork` function to disable components and fix parameters. To omit the linear component, set `LinearFcn.Use` to `false`. To omit the offset, set `Offset.Use` to `false`. To specify known values for the linear function and the offset, set their `Value` attributes directly and set the corresponding `Free` attributes to `False`. Use `evaluate` to compute the output of the function for a given vector of inputs.

Creation

Syntax

```
W = idWaveletNetwork
W = idWaveletNetwork(numUnits)
W = idWaveletNetwork(numUnits,UseLinearFcn)
W = idWaveletNetwork(numUnits,UseLinearFcn,UseOffset)
```

Description

`W = idWaveletNetwork` creates a `idWaveletNetwork` object `W`, for which the function computes the number of units automatically during model estimation.

`W = idWaveletNetwork(numUnits)` specifies the number of units `numUnits`. This syntax includes an option that allows you to interactively assess the relationship between the number of units and unexplained variance.

`W = idWaveletNetwork(numUnits,UseLinearFcn)` specifies whether the function uses a linear function as a subcomponent.

`W = idWaveletNetwork(numUnits,UseLinearFcn,UseOffset)` specifies whether the function uses an offset term.

Input Arguments

`numUnits` — Number of units
`'auto'` (default) | `'interactive'` | positive integer

Number of units, specified as the string or character vector that represents `'auto'` or `'interactive'`, or as a positive integer. `numUnits` determines the number of wavelets or scaling functions, or, if both elements are present, the combined number of wavelets and scaling functions.

Typically, the wavelet network contains either wavelets or scaling functions, but not both. Specify `numUnits` as one of the following values:

- `'auto'` — The software determines the number of units automatically during model estimation.
- `'interactive'` — During model estimation, the software displays an interactive bar plot that relates unexplained variance to the number of units. Click on a bar to view the achievable fit to the estimation data for the selected number of units. A blue bar indicates the optimal choice, based on the generalized cross-validation (GCV) criterion. A general rule for the selection of the number of units is to use the smallest number of units that capture most of the variance.
- Positive integer — The software uses the specified value directly.

This argument sets the `W.NonlinearFcn.NumberOfUnits` property.

`UseLinearFcn` — Option to use linear function
`true` (default) | `false`

Option to use the linear function subcomponent, specified as `true` or `false`. This argument sets the value of the `W.LinearFcn.Use` property.

`UseOffset` — Option to use offset term
`true` (default) | `false`

Option to use an offset term, specified as `true` or `false`. This argument sets the value of the `W.Offset.Use` property.

Properties

`Input` — Input signal information
signal property values

Input signal information for signals used for estimation, specified as vectors of *m* property-specific values, where *m* is the number of input signals. The `Input` properties for each input signal are as follows:

- `Name` — Names of the input signals, specified as a 1-by-*m* string or character array, where *m* is the number of inputs
- `Mean` — Mean of the input signals, specified as a numeric scalar
- `Range` — Ranges of the input signals, specified as a 2-by-*m* numeric array that contains the minimum and maximum values

`Output` — Output signal information
signal property values

Output signal information, specified as property-specific values. The `Output` properties are as follows:

- `Name` — Name of the output signal, specified as a string or a character array
- `Mean` — Mean of the output signal, specified as a numeric scalar
- `Range` — Range of the output signal, specified as a 2-by-1 numeric array that contains the minimum and maximum values

`LinearFcn` — Parameters of linear function
linear function property values (default)

Parameters of the linear function, specified as follows:

- `Use` — Option to use the linear function in the wavelet network, specified as a scalar logical. The default value is `true`. For an example of setting this option, see “Exclude Linear Term from Wavelet Network Mapping Object” on page 1-773.
- `Value` — Linear weights that compose *L*', specified as a 1-by-*p* vector.
- `InputProjection` — Input projection matrix *P*, specified as an *m*-by-*p* matrix, that transforms the input vector of length *m* into a vector of length *p*. For Hammerstein-Wiener models, `InputProjection` is equal to `1`.
- `Free` — Option to update entries of `Value` during estimation, specified as a 1-by-*p* logical vector. The software honors the `Free` specification only if the starting value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a 1-*p* vector. If `Minimum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that minimum bound during model estimation.
- `Maximum` — Maximum bound on `Value`, specified as a 1-*p* vector. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation.
- `SelectedInputIndex` — Indices of `idWaveletNetwork` inputs (see `Input.Name`) that are used as inputs to the linear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. For nonlinear ARX models, the `RegressorUsage` property determines these indices. For Hammerstein-Wiener models, `SelectedInputIndex` is always `1`.

`Offset` — Parameters of offset term

offset property values

Parameters of the offset term, specified as follows:

- `Use` — Option to use the offset in the wavelet network, specified as a scalar logical. The default value is `true`.
- `Value` — Offset value, specified as a scalar.
- `Free` — Option to update `Value` during estimation, specified as a scalar logical. The software honors the `Free` specification of `false` only if the value of `Value` is finite. The default value is `true`.
- `Minimum` — Minimum bound on `Value`, specified as a numeric scalar or –`Inf`. If `Minimum` is specified with a finite value and the value of `Value` is finite, then the software enforces that minimum bound during model estimation. The default value is `-Inf`.
- `Maximum` — Maximum bound on `Value`, specified as a numeric scalar or `Inf`. If `Maximum` is specified with a finite value and the starting value of `Value` is finite, then the software enforces that maximum bound during model estimation. The default value is `Inf`.

`NonlinearFcn` — Parameters of nonlinear function

nonlinear function property values

Parameters of the nonlinear function, specified as follows:

- `NumberOfUnits` — Number of units, specified as `'auto'`, `'interactive'`, or a positive integer. `NumberOfUnits` determines the number of wavelets or scaling functions, or, if both elements are present, the combined number of wavelets and scaling functions. Typically, the wavelet network contains either wavelets or scaling functions, but not both. The options for `NumberOfUnits` are as follows:

  - `'auto'` — The software determines the number of units automatically during model estimation.
  - `'interactive'` — During model estimation, the software displays an interactive bar plot that relates unexplained variance to the number of units. Click on a bar to view the achievable fit to the estimation data for the selected number of units. A blue bar indicates the optimal choice, based on the generalized cross-validation (GCV) criterion. A general rule for the selection of the number of units is to use the smallest number of units that capture most of the variance.
  - Positive integer — The software uses the specified value directly.
- `Structure` — Advanced options that control the structure of the wavelet and scaling functions, specified as in the following table.

Property	Description	Default
`FinestCell`	Minimum number of data points in the smallest cell, specified as `'auto'` or a positive integer. A cell is the area covered by the part of a wavelet that is significantly nonzero. The default setting of `'auto'` specifies that the software determines this value during estimation.	`'auto'`
`MinimumCells`	Minimum number of cells in the partition, specified as a positive integer.	`16`
`MaximumCells`	Maximum number of cells in the partition, specified as a positive integer.	`16`
`MaximumLevels`	Maximum number of wavelet levels, specified as a positive integer.	`10`
`DilationStep`	Dilation step size, specified as a positive integer.	`2`
`TranslationStep`	Translation step size, specified as a positive integer.	`1`

- `Parameters` — Parameters of `idWaveletNetwork`, specified as in the following table.

Field Name	Description	Default
`InputProjection`	Projection matrix Q, specified as an m-by-q matrix. Q transforms the detrended input vector $(X - \bar{X})$ of length m into a vector of length q. Typically, Q has the same dimensions as the linear projection matrix P. In this case, q is equal to p, which is the number of linear weights. For Hammerstein-Wiener models, `InputProjection` is equal to `1`.	`[]`

Field Name	Description	Default
ScalingCoefficient	Scaling function coefficients s_i, specified as a *ds*-by-1 vector.	[]
ScalingTranslation	Scaling translation matrix, specified as a *ds*-by-*q* matrix of scaling translation row vectors e_i.	[]
ScalingDilation	Scaling function dilation coefficients d_i, specified as an *ds*-by-1 vector.	[]
WaveletCoefficient	Wavelet function coefficients w_i, specified as a *dw*-by-1 vector.	[]
WaveletTranslation	Wavelet translation matrix, e_i, specified as a *dw*-by-*q* matrix of wavelet translation row vectors c_i.	[]
WaveletDilation	Wavelet dilation coefficients b_i, specified as an *dw*-by-1 vector.	[]

- `Free` — Option to estimate parameters, specified as a logical scalar. If all the parameters have finite values, such as when the `idWaveletNetwork` object corresponds to a previously estimated model, then setting `Free` to `false` causes the parameters of the nonlinear functions $W(X)$ and $S(X)$ to remain unchanged during estimation. The default value is `true`.
- `SelectedInputIndex` — Indices of `idWaveletNetwork` inputs (see `Input.Name`) that are used as inputs to the nonlinear function, specified as an 1-by-n_r integer vector, where n_r is the number of inputs. For nonlinear ARX models, the `RegressorUsage` property determines these indices. For Hammerstein-Wiener models, `SelectedInputIndex` is always `1`.

Examples

Create Wavelet Network Mapping Object

```
MO = idWaveletNetwork;
```

View the `idWaveletNetwork` object.

```
disp(MO)
```

```
Wavelet Network

 Nonlinear Function: Wavelet network with number of units chosen automatically
 Linear Function: uninitialized
 Output Offset: uninitialized

               Input: '<Function inputs>'
              Output: '<Function output>'
           LinearFcn: '<Linear function parameters>'
        NonlinearFcn: '<Wavelet and scaling function units and their parameters>'
              Offset: '<Offset parameters>'
   EstimationOptions: '<Estimation options>'
```

Exclude Linear Term from Wavelet Network Mapping Object

Create the `idWaveletNetwork` mapping object `MO`.

```
MO = idWaveletNetwork;
```

Exclude the linear term from MO.

```
MO.LinearFcn.Use = false;
```

View the idWaveletNetwork object.

```
disp(MO)
```

```
Wavelet Network

 Nonlinear Function: Wavelet network with number of units chosen automatically
 Linear Function: not in use
 Output Offset: uninitialized

                Input: '<Function inputs>'
               Output: '<Function output>'
            LinearFcn: '<Linear function parameters>'
         NonlinearFcn: '<Wavelet and scaling function units and their parameters>'
               Offset: '<Offset parameters>'
    EstimationOptions: '<Estimation options>'
```

The linear function is not in use.

Estimate Nonlinear ARX Model with Specific Mapping Function

Load the estimation data.

```
load twotankdata y u;
```

Create an iddata object from the estimation data.

```
z = iddata(y,u,0.2);
```

Create a wavelet network mapping object with five units.

```
MO = idWaveletNetwork(5);
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,[4 4 1],MO)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Wavelet network with 5 units
Sample time: 0.2 seconds

Status:
```

```
Estimated using NLARX on time domain data "z".
Fit to estimation data: 96.8% (prediction focus)
FPE: 3.553e-05, MSE: 3.515e-05
```

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 six inputs and two 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 = idWaveletNetwork;
```

Specify different nonlinearity estimators for each output channel.

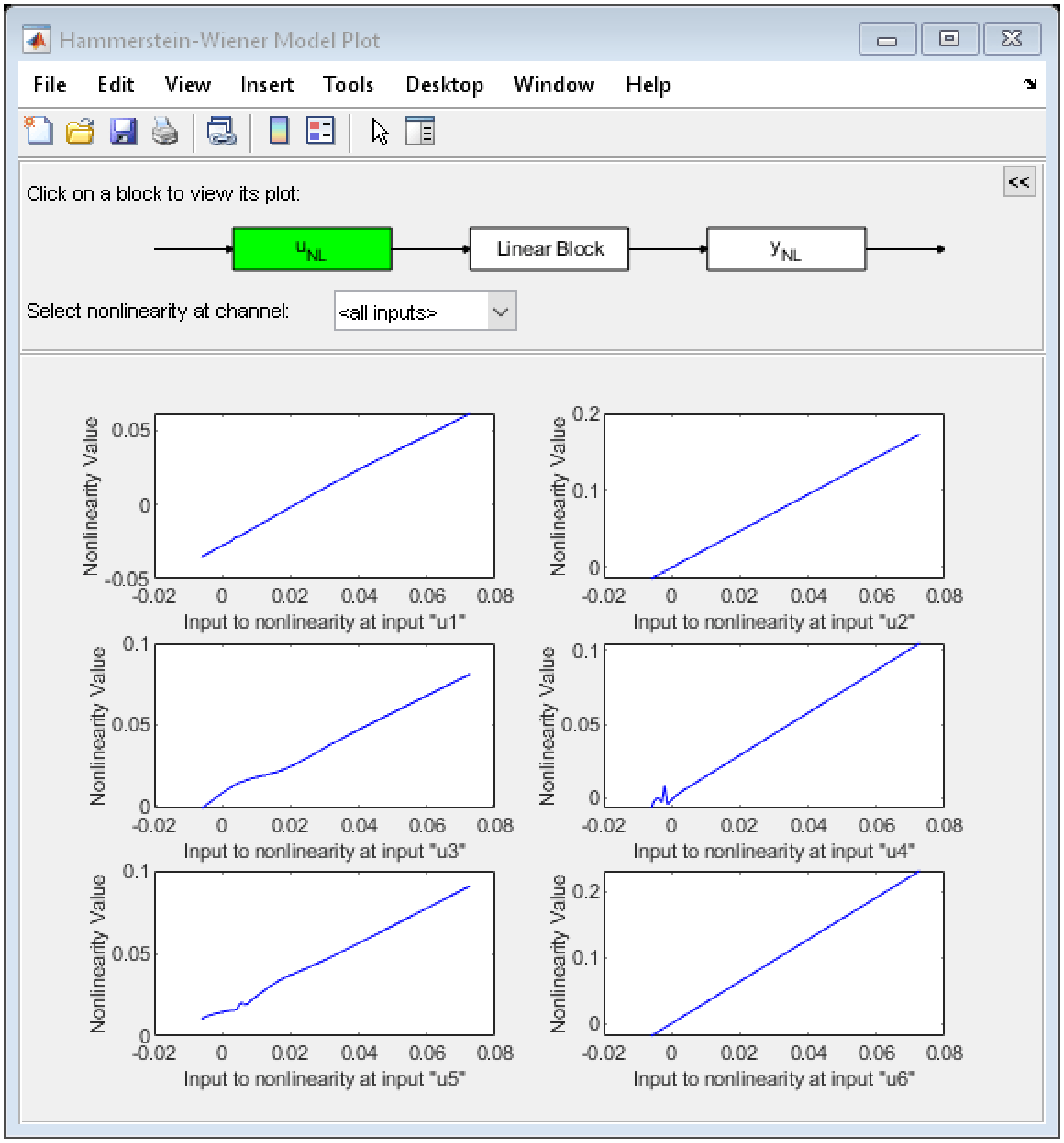
```
 OutputNL = [idDeadZone,idWaveletNetwork];
```

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"

Click on the input and output nonlinearity blocks on the top of the plot to see the nonlinearities.

Apply `idWaveletNetwork` Constraints when Estimating Nonlinear ARX Model

Load the estimation data.

```
load iddata7 z7;
```

Specify `idWaveletNetwork` for the model output function. Configure the `idWaveletNetwork` object to have a fixed offset value of 1 and to use five units in the nonlinear component.

```
w = idWaveletNetwork;
w.Offset.Value = 1;
w.Offset.Free = false;
w.NonlinearFcn.NumberOfUnits = 5;
```

Specify a linear model regressor set using a lag array that produces four consecutive output regressors and five consecutive input regressors.

```
reg = linearRegressor({'y1','u1'},{1:4,0:4});
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z7,reg,w);
```

Examine the postestimation properties of the output function.

```
disp(sys.OutputFcn)
```

```
Wavelet Network
Inputs: y1(t-1), y1(t-2), y1(t-3), y1(t-4), u1(t), u1(t-1), u1(t-2), u1(t-3), u1(t-4)
Output: y1

 Nonlinear Function: Wavelet network with 5 units
 Linear Function: initialized to [-1.12 0.469 1.25 0.556 -0.81 -0.261 -0.074 0.711 1.15]
 Output Offset: fixed to 1

                Input: '<Function inputs>'
               Output: '<Function output>'
            LinearFcn: '<Linear function parameters>'
         NonlinearFcn: '<Wavelet and scaling function units and their parameters>'
               Offset: '<Offset parameters>'
    EstimationOptions: '<Estimation options>'
```

```
disp(sys.OutputFcn.Input)
```

```
Function inputs

     Name: {1x9 cell}
     Mean: [0.0795 0.0848 0.1082 0.1134 0.0253 0.0253 0.0202 0.0202 0.0202]
    Range: [2x9 double]
```

```
disp(sys.OutputFcn.NonlinearFcn)
```

```
Wavelet and scaling function units and their parameters

         NumberOfUnits: 5
            Parameters: '<Wavelet parameters>'
                  Free: 1
    SelectedInputIndex: [1 2 3 4 5 6 7 8 9]
```

Apply `idWaveletNetwork` Constraints when Estimating Hammerstein-Wiener Model

Load the estimation data.

```
load throttledata
```

Specify `idWaveletNetwork` for the model output nonlinearity and set the arguments for `NumUnits` to `5` and `UseLinearFcn` and `UseOffset` to `false`.

```
w = idWaveletNetwork(5,false,false);
```

An alternative method for removing the linear function and offset is to use dot notation after creating the nonlinearity.

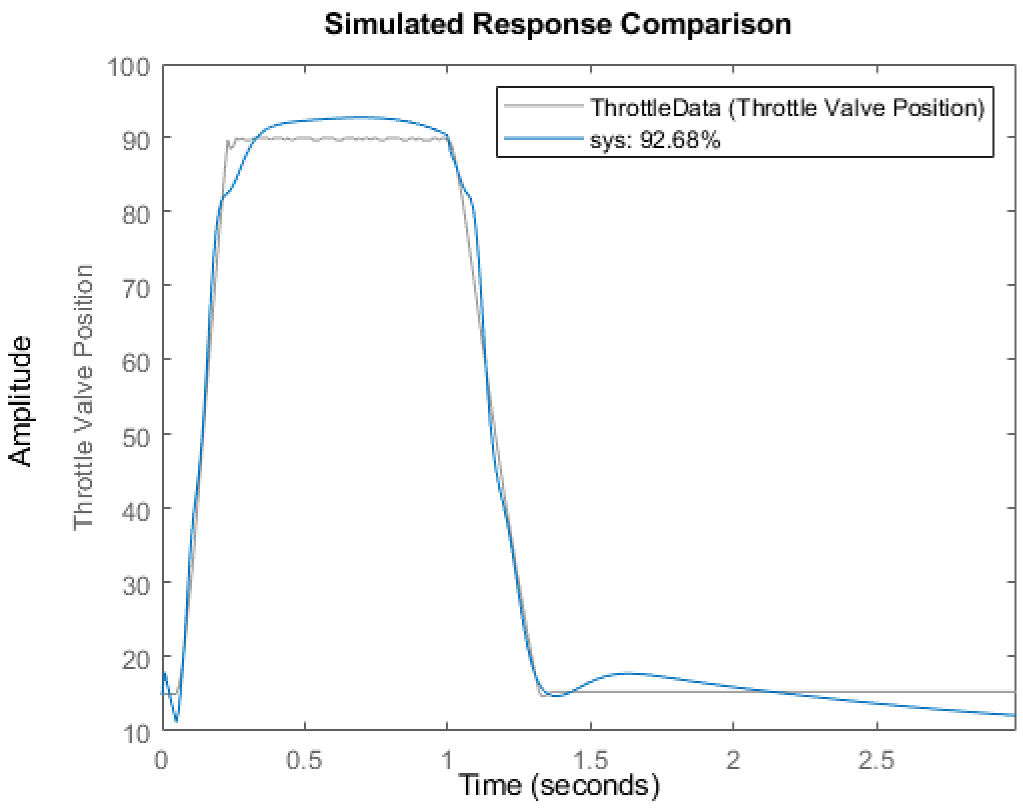
```
w.LinearFcn.Use = false;
w.Offset.Use = false;
```

Estimate the model, using an `order` specification of `[4 4 1]`.

```
sys = nlhw(ThrottleData,[4 4 1],[],w);
```

Compare the simulated model response with the estimation data.

```
compare(ThrottleData,sys);
```



More About

Wavelet Nonlinear Function *W*(*X*)

The wavelet nonlinear function is a sum of the dilated and translated wavelets, and is described by the following equation:

$$W(X) = \sum_{i=1}^{d_W} w_i f_W(b_i(X - \bar{X})^T Q - c_i)$$

Here:

- Q is an m-by-q projection matrix, where $m \geq q$
- w_1, w_2, ..., w_{dw} are scalar coefficients called wavelet coefficients.
- b_1, b_2, ..., b_{dw} are scalars called wavelet dilations that multiply the detrended input matrix $(X - \bar{X})$.
- c_1, c_2, ..., c_{dw} are 1-by-q row vectors called wavelet translations.
- $f_W(x) = e^{-xx^T/2}$ is a radial function that depends only upon the squared magnitude of the vector x. x is a row vector that is composed of a linear combination of inputs with an offset c_i.

Scaling Nonlinear Function *S*(*X*)

The scaling nonlinear function is a sum of the dilated and translated scaling functions, and is described by the following equation:

$$S(X) = \sum_{i=1}^{d_S} s_i f_S(b_i(X - \bar{X})^T Q - e_i)$$

Here:

- Q is an m-by-q projection matrix, where $m \geq q$.
- s_1, s_2, ..., s_{ds} are scalar coefficients called scaling coefficients.
- d_1, d_2, ..., d_{ds} are scalars called scaling dilations that multiply the detrended input matrix $(X - \bar{X})$.
- e_1, e_2, ..., e_{ds} are 1-by-q row vectors called scaling translations.
- $f_S(x) = (\dim(x) - xx^T)e^{-xx^T/2}$ is a radial function that depends only upon the squared magnitude of the vector x. x is a row vector that is composed of a linear combination of inputs with an offset e_i.

Algorithms

You can use `idWaveletNetwork` in both nonlinear ARX and Hammerstein-Wiener models. The algorithms for estimating `idWaveletNetwork` parameters depend on which model you are estimating.

- In a nonlinear ARX model, `idWaveletNetwork` uses either a noniterative or an iterative technique for predicting the parameters, depending on option settings in `nlarxOptions`.
  - If the `Focus` option is set to `'prediction'`, then `idWaveletNetwork` uses a fast noniterative technique to estimate parameters [1]. Successive refinements after the first estimation use an iterative algorithm.
  - If the `Focus` option is set to `'simulation'`, then `idWaveletNetwork` uses an iterative technique to estimate parameters.
  - To always use either an iterative or a noniterative algorithm, specify the `IterativeWavenet` property of `nlarxOptions` as `'on'` or `'off'`, respectively.
- In a Hammerstein-Wiener model, `idWaveletNetwork` uses iterative minimization to determine the parameters.

Compatibility Considerations

Use of previous idnlarx and idnlhw mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

Use of previous nonlinearity estimator properties is not recommended
Not recommended starting in R2021a

Starting in R2021a, the properties of the mapping objects, previously known as nonlinearity estimators, have been reorganized. These objects are `wavenet` (W), `sigmoidnet` (S), `treepartition` (T), `customnet` (C), and `linear` (L). The property changes do not apply to `neuralnet`. The use of the pre-R2021a properties in the following table is discouraged. However, the software still accepts commands that set these properties. There are no plans to exclude such commands at this time.

Pre-R2021a Property	R2021a Property	Applicable Mapping Objects
`NumberOfUnits`	`NonlinearFcn.NumberOfUnits`	W,S,T,C
`LinearTerm`	`LinearFcn.Use`, `Offset.Use`	W,S,C
`Parameters`	Split into three pieces: • `LinearFcn.Value` • `Offset.Value` • `NonlinearFcn.Parameters`	W,S,T,C,L `linear` (L) excludes `NonlinearFcn.Parameters`.

Pre-R2021a Property	R2021a Property	Applicable Mapping Objects
`Options`	`NonlinearFcn.Structure`	W,T

References

[1] Qinghua Zhang. “Using Wavelet Network in Nonparametric Estimation.” *IEEE Transactions on Neural Networks* 8, no. 2 (March 1997): 227–36. https://doi.org/10.1109/72.557660.

See Also

`nlhw` | `nlarx` | `idLinear` | `idPolynomial1D` | `idTreePartition` | `idSigmoidNetwork` | `idSaturation` | `idPiecewiseLinear` | `idUnitGain` | `idDeadZone` | `idFeedforwardNetwork` | `idCustomNetwork` | `idnlhw` | `idnlarx` | `evaluate`

Introduced in R2007a

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 1-786). 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

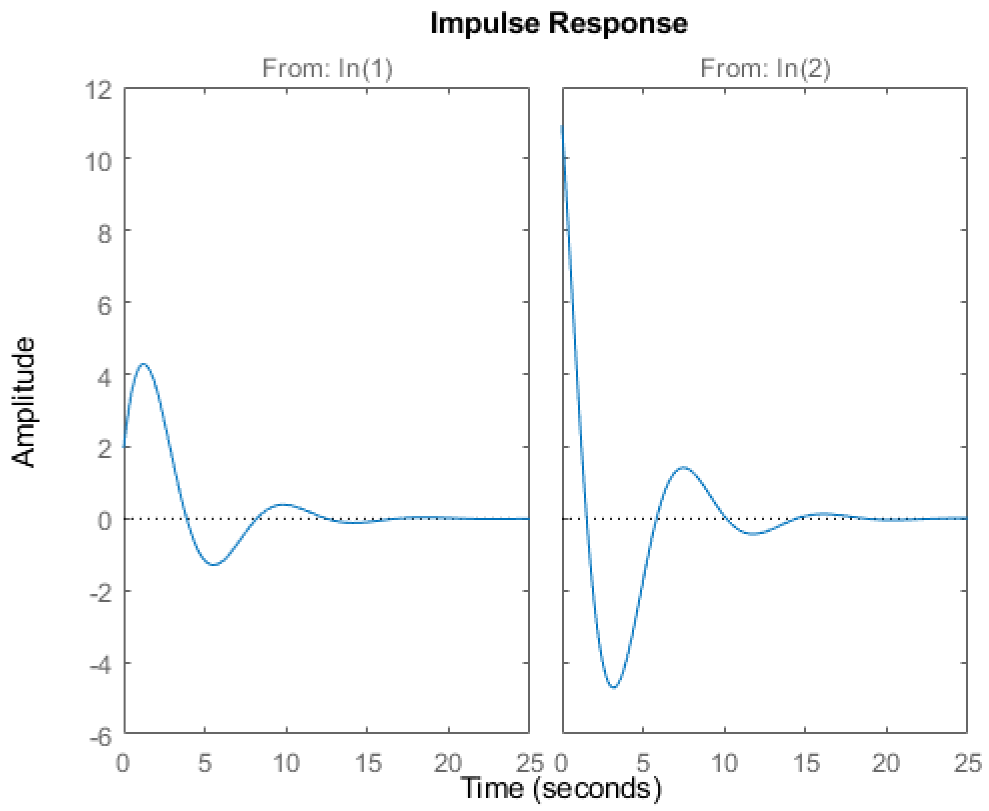
Impulse Response Plot of Second-Order State-Space Model

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 = 1×3

   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 = 1×2

   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 $t = 0$.

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” (Control System Toolbox).

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.

$$\begin{aligned}\dot{x} &= Ax,\ x(0) = b \\ y &= Cx\end{aligned}$$

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

Linear System Analyzer | `step` | `lsim` | `impulseest`

Introduced before R2006a

impulseest

Nonparametric impulse response estimation

Syntax

```
sys = impulseest(data)
sys = impulseest(data,n)
sys = impulseest(data,n,nk)
sys = impulseest( ___ ,opt)
```

Description

`sys = impulseest(data)` estimates an impulse response model `sys`, also known as a finite impulse response (FIR) model, using time-domain or frequency-domain data `data`. The function uses persistence-of-excitation analysis on the input data to select the model order (number of nonzero impulse response coefficients.

Use nonparametric impulse response estimation to analyze input/output data for feedback effects, delays, and significant time constants.

`sys = impulseest(data,n)` estimates an `n`th-order impulse response model that corresponds 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( ___ ,opt)` specifies estimation options using the options set `opt`. You can use this syntax with any of the previous input argument combinations.

Examples

Identify Nonparametric Impulse Response Model from Data

Estimate 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.

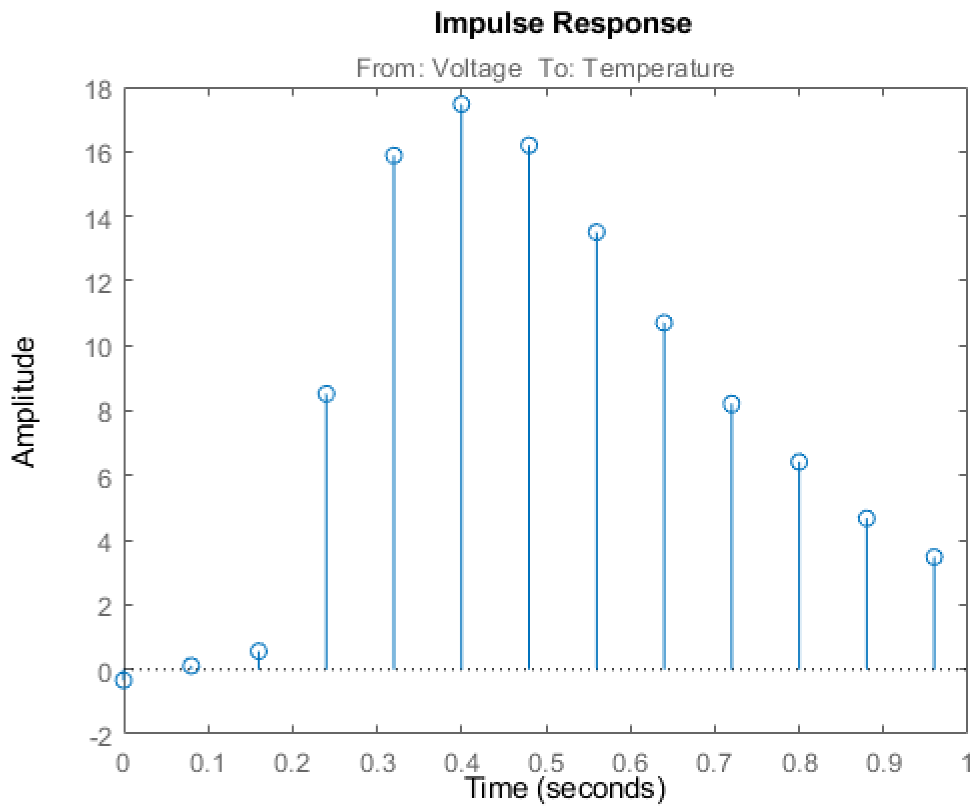
Load the data and use the first 500 samples to estimate the model.

```
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.

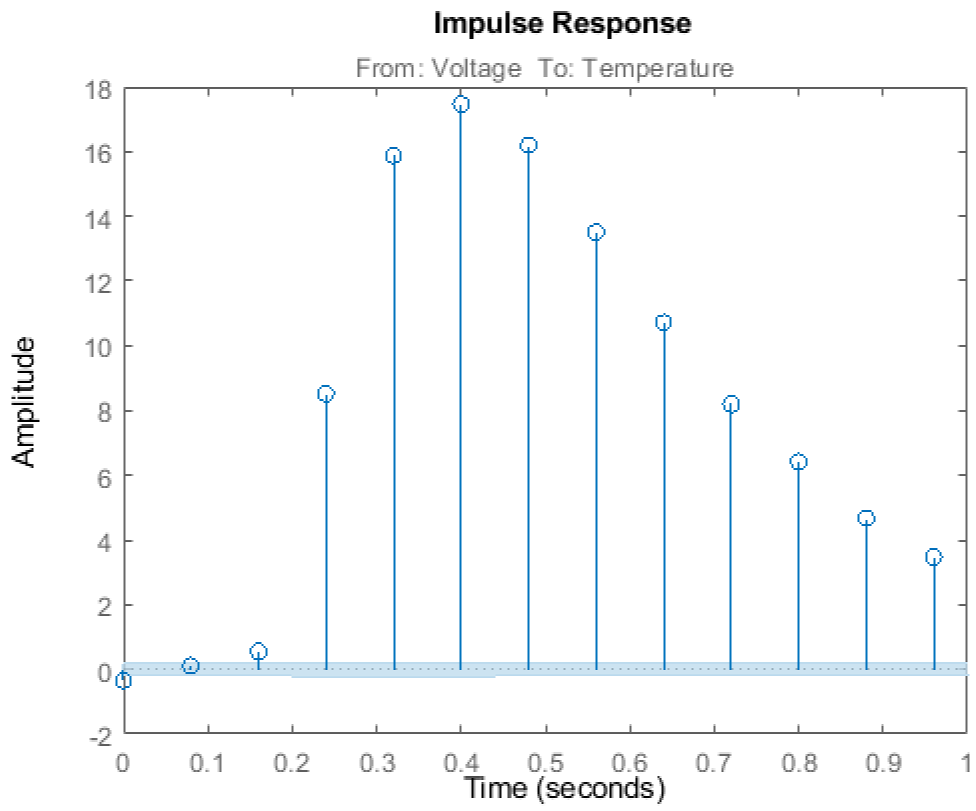
```
h = impulseplot(sys,1);
```



Determine the point at which a significant response to the impulse begins. First, display the region that bounds amplitudes that are not significantly different from zero. To do so, right-click the plot and select **Characteristics > Confidence Region**. For impulse response plots, by default, this selection displays a confidence region with a width of one standard deviation that is centered at zero, instead of one centered at the response values. You can modify these defaults by right-clicking the plot and selecting **Properties > Options**.

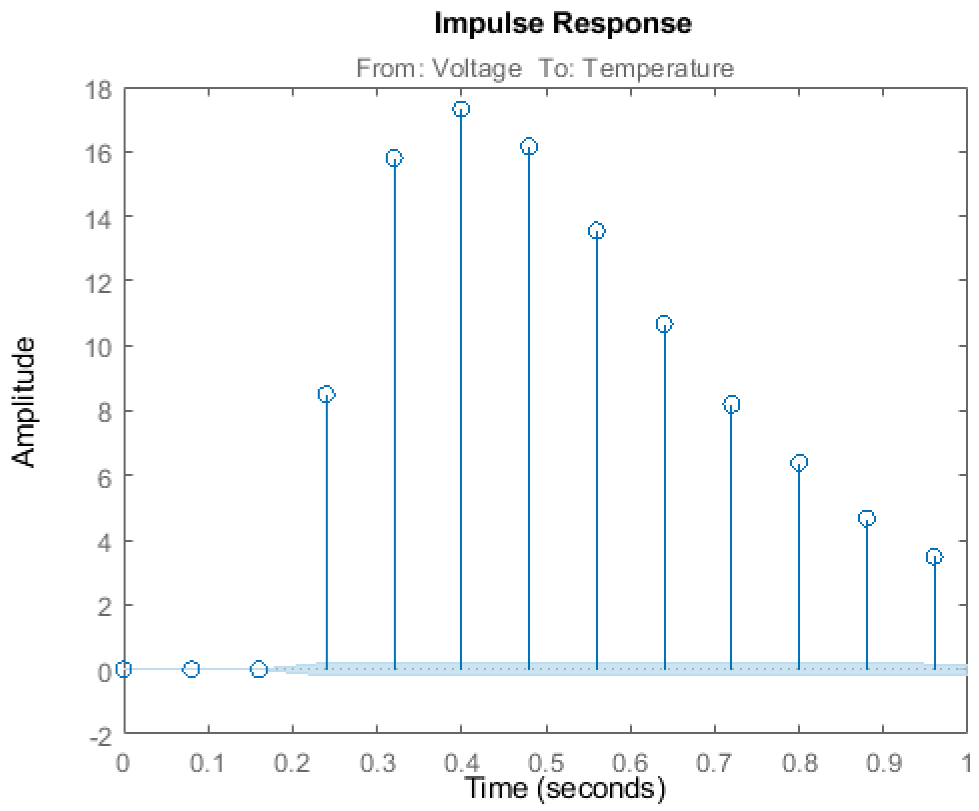
Alternatively, you can use the `showConfidence` command.

```
showConfidence(h);
```



The first response value that is significantly different than zero occurs at 0.24 seconds, or the third sample. This implies that the transport delay is three samples. To generate a model that imposes the three-sample delay, set the transport delay, which is the third argument, to 3. You must also set the second argument, the order `n`, to its default value of `[]` as a placeholder.

```
sys1 = impulseest(ze,[],3);
h1 = impulseplot(sys1,1);
showConfidence(h1);
```



The response is identically zero until 0.24 seconds.

Specify Order of FIR Model

Load the estimation data.

```
load iddata3 z3;
```

Estimate a 35th-order FIR model.

```
n = 35;
sys = impulseest(z3,n);
```

You can confirm the model order of `sys` by displaying the number of terms.

```
nsys = size(sys.num)
```

```
nsys = 1×2

     1    35
```

Set `n` to `[]` so that the function automatically determines `n`. Display the model order.

```
n = [];
sys1 = impulseest(z3,n);
nsys1 = size(sys1.Numerator)
```

```
nsys1 = 1×2

     1    70
```

The model order is 70. The default value for the order is `[]`, so setting the order to `[]` is equivalent to omitting the specification.

Specify Transport Delay in FIR Model

Estimate an impulse response model with a transport delay of 3 samples.

If you know about the presence of delay in the input/output data in advance, use the delay value as a transport delay for impulse response estimation.

Generate data that contains a 3-sample input-to-output lag.

Create a random input signal. Construct an `idpoly` model that includes three sample delays, which you implement by using three leading zeros in the B polynomial.

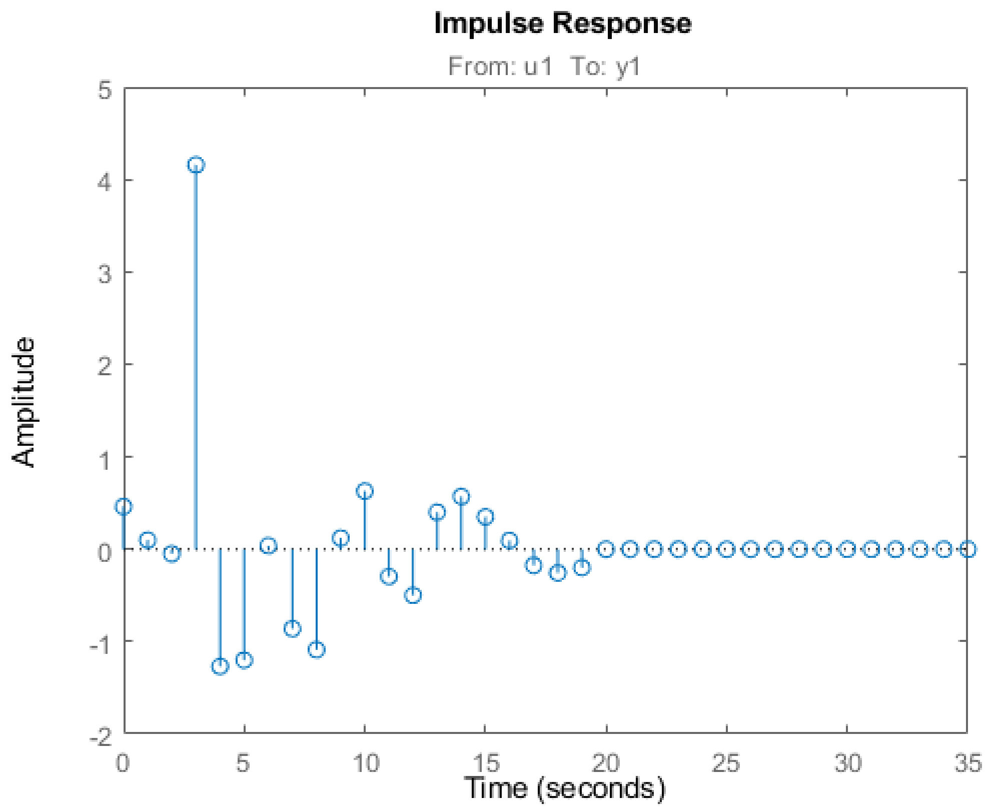
```
u = rand(100,1);
A = [1 .1 .4];
B = [0 0 0 4 -2];
C = [1 1 .1];
sys = idpoly(A,B,C);
```

Simulate the model response `y` to the noise signal, using the `AddNoise` option and a sample time of 1 second. Encapsulate `y` in an `iddata` object.

```
opt = simOptions('AddNoise',true);
y = sim(sys,u,opt);
data = iddata(y,u,1);
```

Estimate and plot a 20th order model with no transport delay.

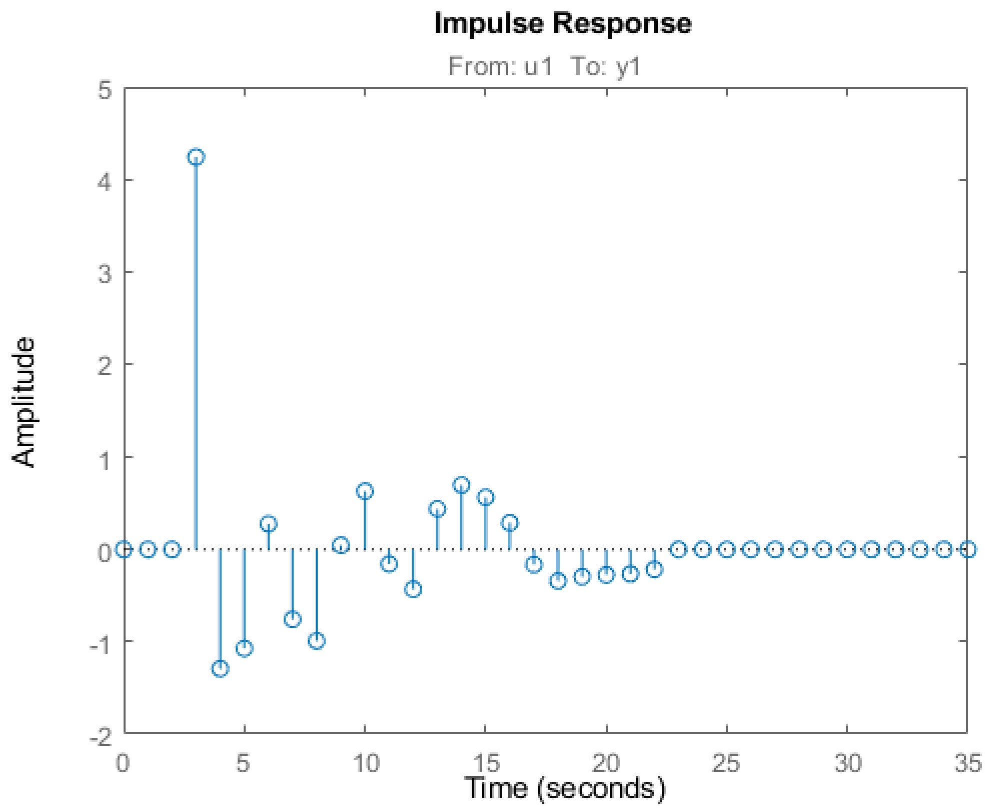
```
n = 20;
model1 = impulseest(data,n);
impulseplot(model1);
```



The plot shows that the impulse response includes nonzero samples during the 3-second delay period.

Estimate a model with a 3-sample transport delay.

```
nk = 3;
model2 = impulseest(data,n,nk);
impulseplot(model2)
```



The first three samples are identically zero.

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. `impulseest` performs regularized estimates by default, using the tuned and correlated kernel (`'TC'`).

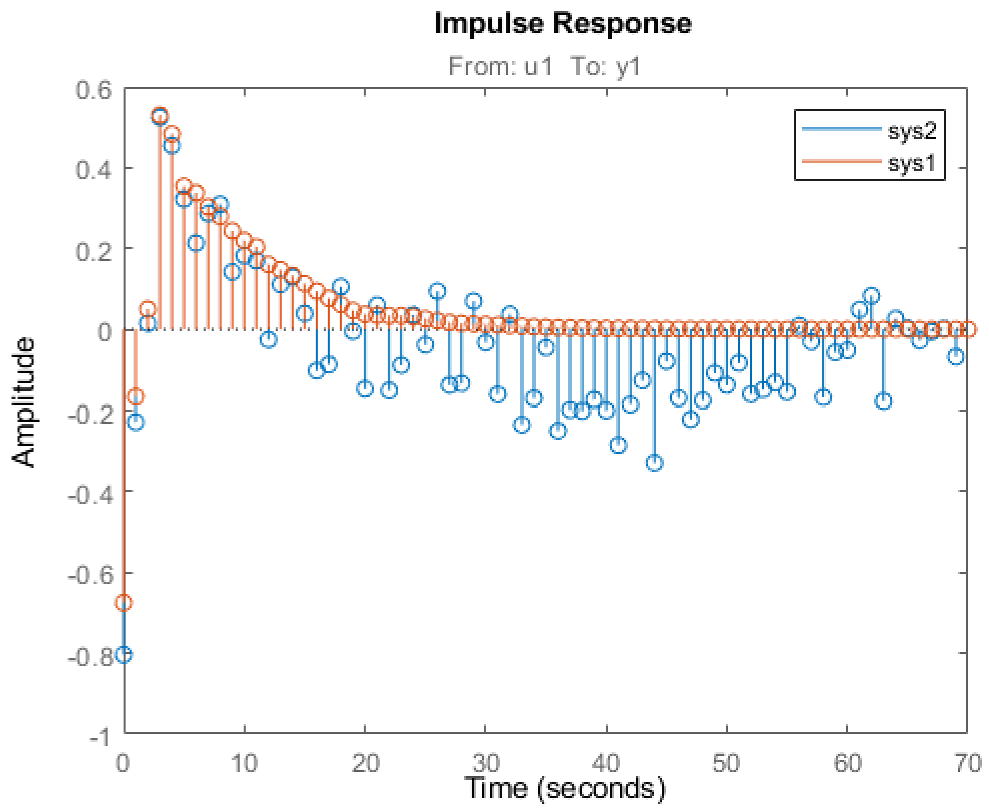
```
load iddata3 z3;
sys1 = impulseest(z3);
```

Estimate a model with no regularization.

```
opt = impulseestOptions('RegularizationKernel','none');
sys2 = impulseest(z3,opt);
```

Compare the impulse responses of both models.

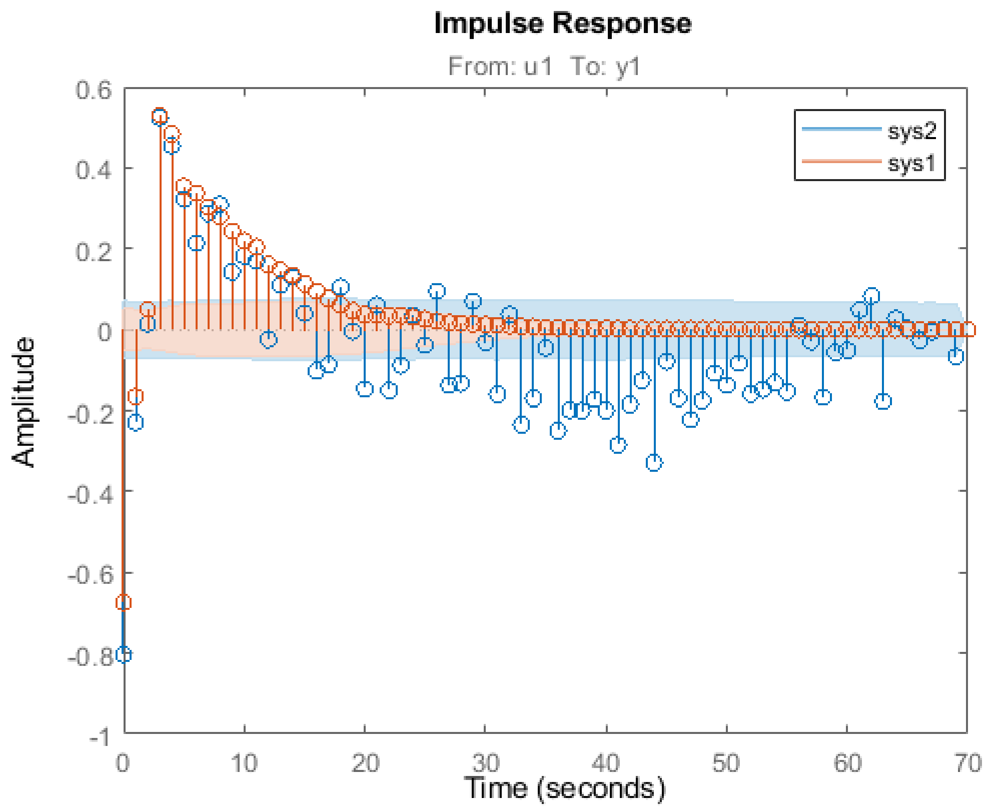
```
h = impulseplot(sys2,sys1,70);
legend('sys2','sys1')
```



As the plot shows, using regularization makes the response smoother.

Plot the confidence intervals.

```
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 `'TC'` regularization is such that the variance error dominates the overall error.

Use Regularized Impulse Response Model to Estimate State-Space Model

Load the estimation data.

```
load regularizationExampleData eData;
```

Recreate the transfer function model that was used for generating the estimation data (true system).

```
num = [0.02008 0.04017 0.02008];
den = [1 -1.561 0.6414];
Ts = 1;
trueSys = idtf(num,den,Ts);
```

Obtain a regularized impulse response (FIR) model with an order of 70.

```
opt = impulseestOptions('RegularizationKernel','DC');
m0 = impulseest(eData,70,opt);
```

Convert the model into a state-space model and reduce the model order.

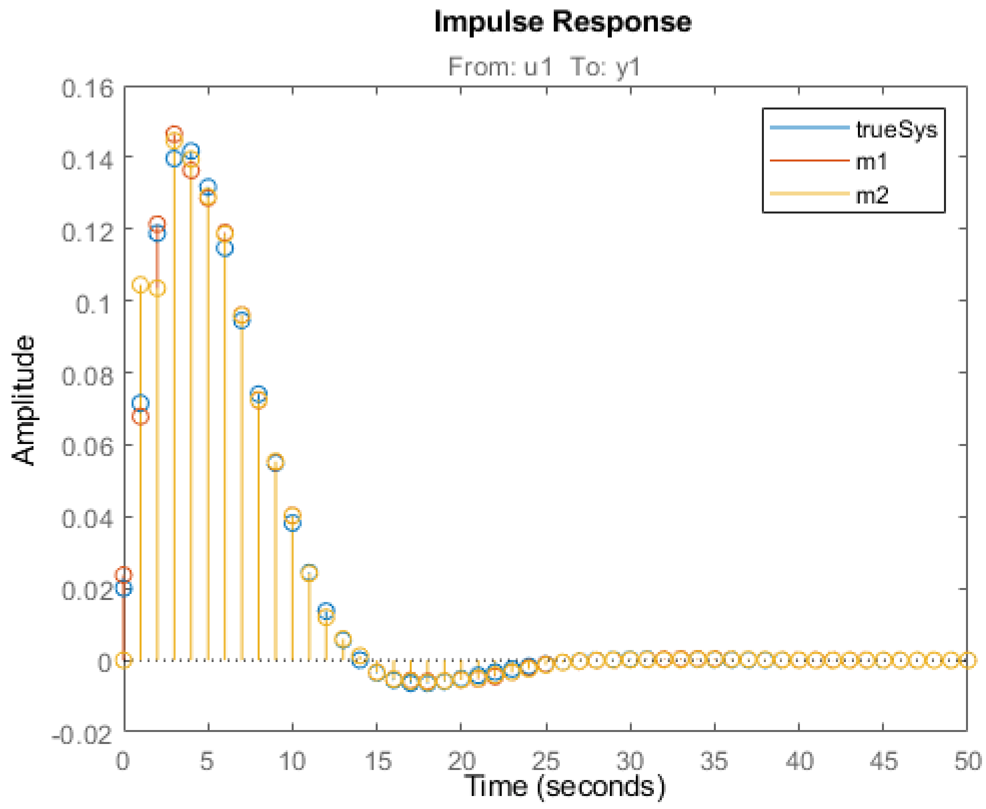
```
m1 = idss(m0);
m1 = balred(m1,15);
```

Estimate a second state-space model directly from `eData` by 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');
```



The three model responses are similar.

Test Measured Data for Feedback Effects

Use the empirical impulse response to measured data to determine whether the data includes feedback effects. Feedback effects can be present when the impulse response includes statistically significant response values for negative time values.

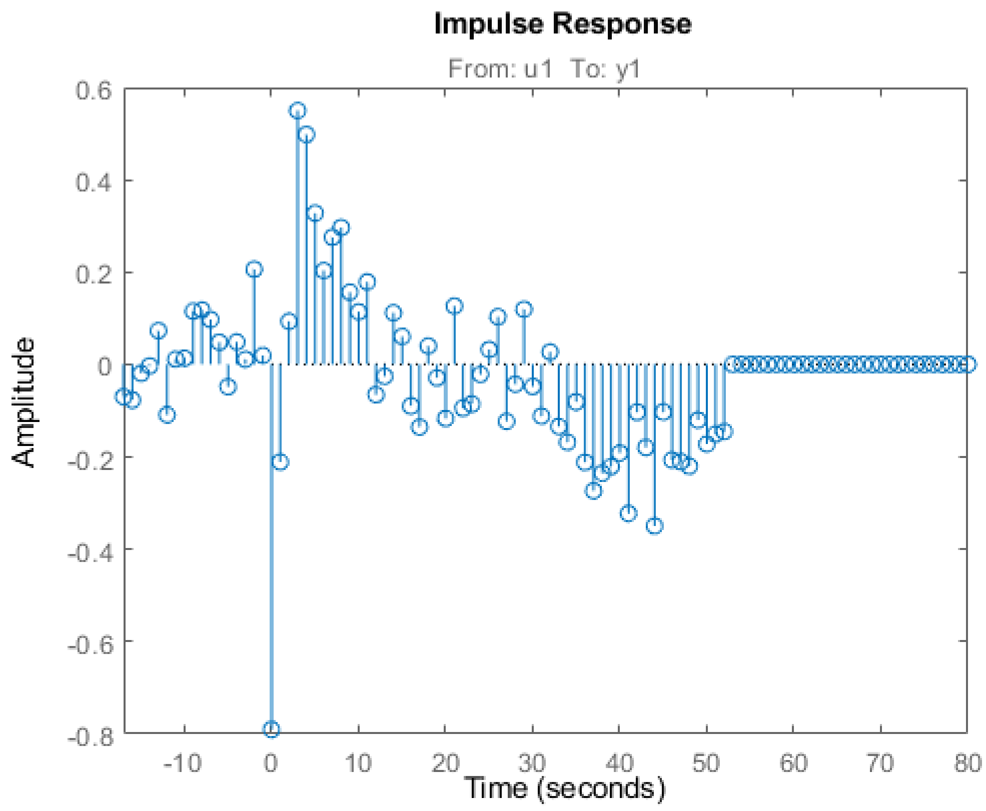
Compute the noncausal impulse response using a fourth-order prewhitening filter and no regularization, automatic order selection, and negative lag.

```
load iddata3 z3;
opt = impulseestOptions('pw',4,'RegularizationKernel','none');
sys = impulseest(z3,[],'negative',opt);
```

`sys` is a noncausal model containing response values for negative time.

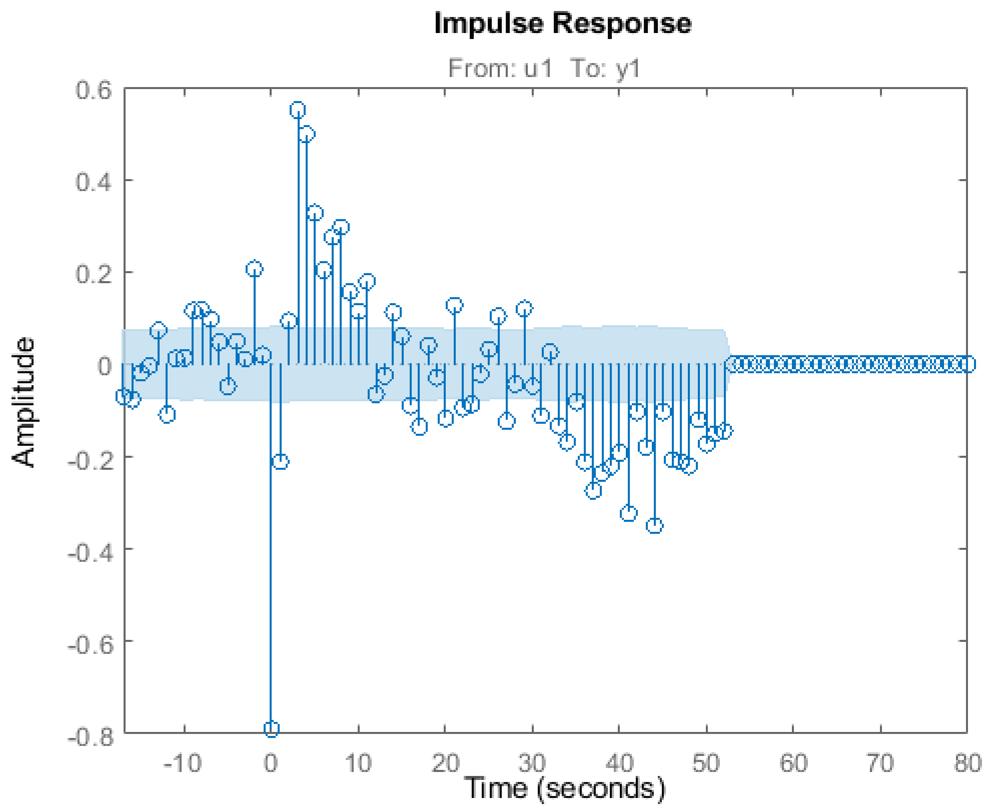
Analyze the impulse response of the identified model.

```
h = impulseplot(sys);
```



View the zero-response region at one standard deviation 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. The large response value can also indicate direct feedback, such as proportional control without some delay so that `y(t)` partly determines `u(t)`.

Other indications of feedback in the data are the significant response values such as those at -7 seconds and -9 seconds.

Compute Impulse Response on Frequency Response Data

Compute an impulse response model for frequency response data.

Load the frequency response data, which contains measured amplitude AMP and phase PHA for the frequency vector W.

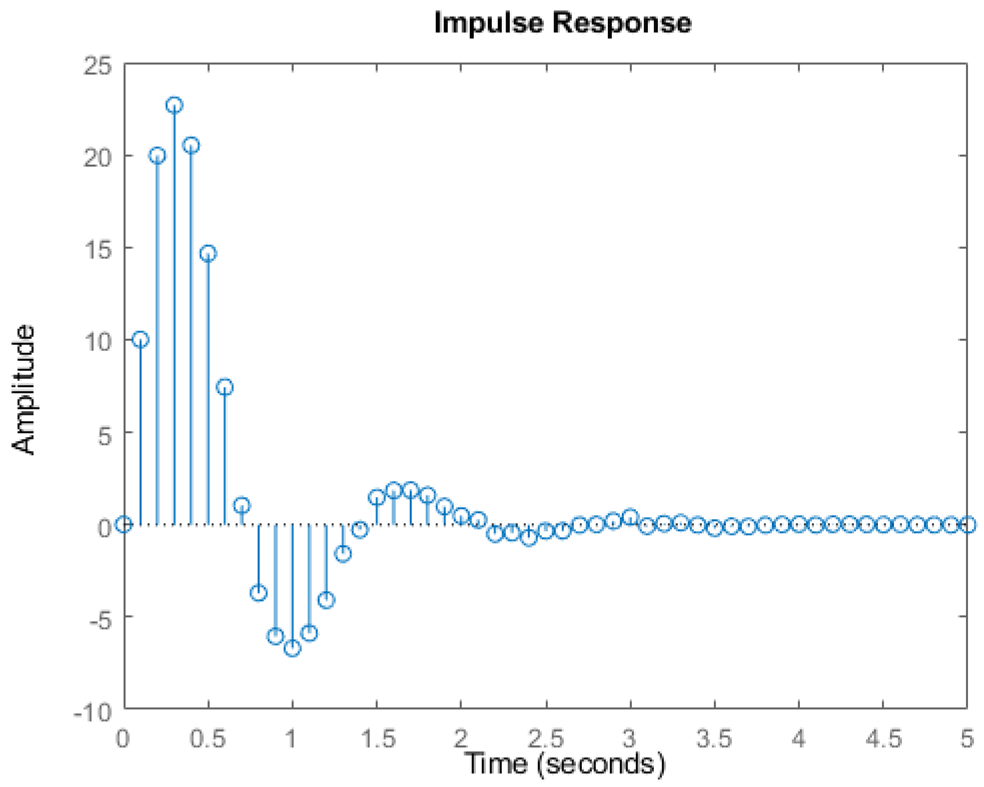
```
load demofr;
```

Create the complex frequency response `zfr` and encapsulate it in an `idfrd` object that has a sample time of 0.1 seconds. Plot the data.

```
zfr = AMP.*exp(1i*PHA*pi/180);
Ts = 0.1;
data = idfrd(zfr,W,Ts);
```

Estimate an impulse response model from `data` and plot the response.

```
sys = impulseest(data);
impulseplot(sys)
```



Compare Identified Nonparametric and Parametric Models

Identify parametric and nonparametric models for a data set, and compare their step responses.

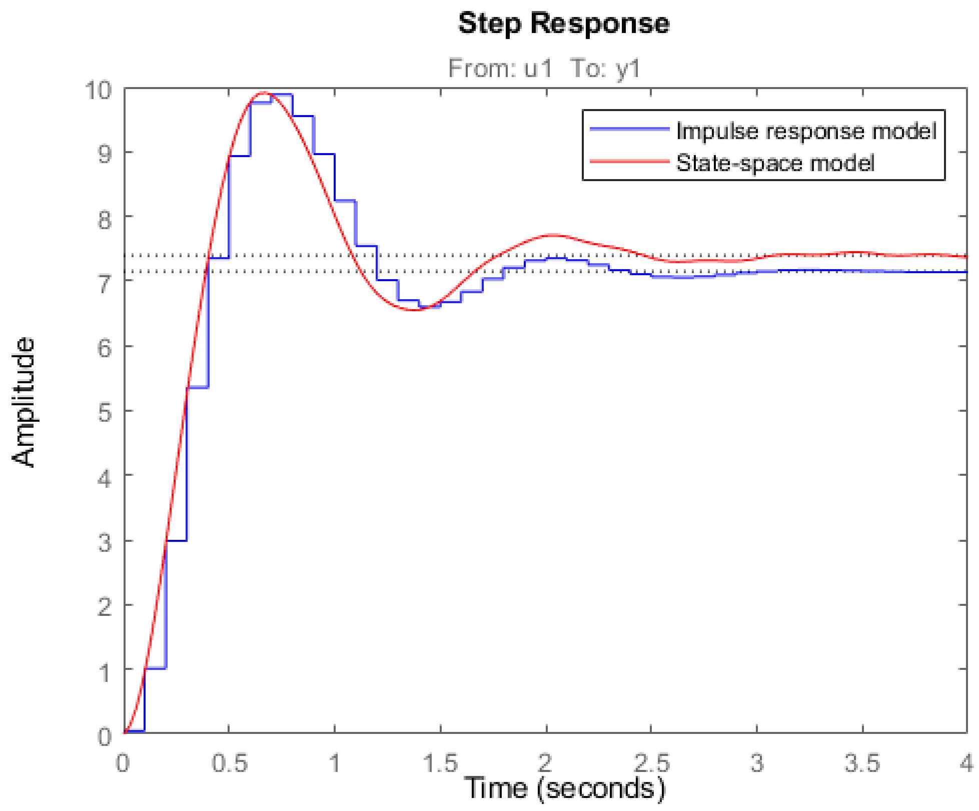
Estimate the impulse response model `sys1` (nonparametric) and state-space model `sys2` (parametric) using the estimation data set `z1`.

```
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 responses for `sys1` and `sys2`.

```
step(sys1,'b',sys2,'r');
legend('Impulse response model','State-space model');
```



Input Arguments

data — Estimation data

`iddata` object | `idfrd` object | `frd` object

Estimation data, specified as an `iddata` object, an `idfrd` object, or an `frd` object, with at least one input signal and a nonzero sample time.

For time-domain estimation, specify `data` as an `iddata` object containing the input and output signal values.

For frequency-domain estimation, specify `data` as one of the following:

- Frequency response data (`idfrd` object or `frd` object)
- `iddata` object with properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — `'Frequency'`

n — Order of FIR model

`[]` (default) | positive integer | matrix

Order of the FIR model, specified as a positive integer, `[]`, or a matrix.

- If `data` contains a single input channel and output channel, or if you want to apply the same model order to all input/output pairs, specify `n` as a positive integer.
- If `data` contains N_u input channels and N_y output channels, and you want to specify individual model orders for the input/output pairs, specify `n` as an N_y-by-N_u matrix of positive integers, such that $N(i,j)$ represents the length of the impulse response from input j to output i.
- If you want the function to determine the order automatically, specify `n` as `[]`. The software uses persistence-of-excitation analysis on the input data to select the order.

Example: `sys = impulseest(data,70)` estimates an impulse response model of order 70.

nk — Transport delay
zero matrix (default) | `'negative'` | `0` | `1` | scalar integer | matrix

Transport delay in the estimated impulse response, specified as a scalar integer, `'negative'`, or an N_y-by-N_u matrix, where N_y is the number of outputs and N_u is the number of inputs. 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`.

- If you know the value of the transport delay, specify `nk` as a scalar integer or a matrix of scalar integers.
- If you do not know the delay value, specify `nk` as `0`. Once you estimate the impulse response, you can determine the true delay from the nonsignificant impulse response values in the beginning portion of the response. For an example of finding a true delay, see “Identify Nonparametric Impulse Response Model from Data” on page 1-787.
- 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 specify `nk` as `'negative'` to automatically pick negative lags for all input/output channels of the model. For an example of using negative time values, see “Test Measured Data for Feedback Effects” on page 1-796.
- To create a system whose leading numerator coefficient is zero, specify `nk` as `1`.

The function stores positive values of `nk` greater than 1 in the `IODelay` property of `sys` (`sys.IODelay` = `max(nk-1,0)`), and negative values in the `InputDelay` property.

opt — Estimation options
`impulseestOptions` option set

Estimation options, specified as an `impulseestOptions` option set, that specify the following:

- Prefilter order
- Regularization algorithm
- Input and output data offsets
- Advanced options such as structure

Use `impulseestOptions` to create the options set.

Output Arguments

sys — Estimated impulse response model
`idtf` object

Estimated impulse response model, returned as an `idtf` model that 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.

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.
`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:

Field	Description
`FitPercent`	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
`LossFcn`	Value of the loss function when the estimation completes.
`MSE`	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
`FPE`	Final prediction error for the model.
`AIC`	Raw Akaike Information Criteria (AIC) measure of model quality.
`AICc`	Small-sample-size corrected AIC.
`nAIC`	Normalized AIC.
`BIC`	Bayesian Information Criteria (BIC).

`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this field is a set of default options. See `impulseestOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
DataUsed	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** — **Description** `Name` — Name of the data set. `Type` — Data type. `Length` — Number of data samples. `Ts` — Sample time. `InterSample` — Input intersample behavior, returned as one of the following values: • `'zoh'` — 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. `InputOffset` — Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`. `OutputOffset` — Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.

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

Tips

- To view the impulse or step response of `sys`, use either `impulseplot` or `stepplot`, respectively.
- A response value that corresponds to a negative time value and that is significantly different from zero in the impulse response of `sys` indicates the presence of feedback in the data.
- To view the region of responses that are not significantly different from zero (the zero-response region) 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 confidence in significance depends on the number of standard deviations specified in `showConfidence` or options in the property editor. The default value is 1 standard deviation, which gives 68% confidence. A common choice is 3 standard deviations, which gives 99.7% confidence.

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 output when the input is an impulse signal. The output response to a general input, $u(t)$, is the convolution with the impulse response. In continuous time:

$$y(t) = \int_{-\infty}^{t} g(\tau)u(t-\tau)d\tau$$

In discrete time:

$$y(t) = \sum_{k=1}^{\infty} g(k)u(t-k)$$

The values of $g(k)$ are the discrete-time impulse response coefficients.

You can estimate the values from observed input/output data in several different ways. `impulseest` estimates the first n coefficients using the least-squares method to obtain a finite impulse response (FIR) model of order n.

`impulseest` provides several important options for the estimation:

- Regularization — Regularize the least-squares estimate. With regularization, the algorithm forms an estimate of the prior decay and mutual correlation among $g(k)$, and then merges this prior estimate with the current information about g from the observed data. This approach results in an estimate that has less variance but also some bias. You can choose one of several kernels to encode the prior estimate.

  This option is essential because the model order `n` can often be quite large. In cases without regularization, `n` can be automatically decreased to secure a reasonable variance.

  Specify the regularizing kernel using the `RegularizationKernel` name-value argument of `impulseestOptions`.
- Prewhitening — Prewhiten the input by applying an input-whitening filter of order `PW` to the data. Use prewhitening when you are performing unregularized estimation. Using a prewhitening filter minimizes the effect of the neglected tail—`k > n`—of the impulse response. To achieve prewhitening, the algorithm:

  1. Defines a filter `A` of order `PW` that whitens the input signal `u`:

     `1/A = A(u)e`, where `A` is a polynomial and `e` is white noise.
  2. Filters the inputs and outputs with `A`:

     `uf = Au`, `yf = Ay`
  3. Uses the filtered signals `uf` and `yf` for estimation.

  Specify prewhitening using the `PW` name-value pair argument of `impulseestOptions`.
- Autoregressive Parameters — Complement the basic underlying FIR model by NA autoregressive parameters, making it an ARX model.

$$y(t) = \sum_{k=1}^{n} g(k)u(t-k) - \sum_{k=1}^{NA} a_k y(t-k)$$

This option both gives better results for small n values and allows unbiased estimates when data are generated in closed loop. `impulseest` sets NA to `5` when t > 0 and sets NA to `0` (no autoregressive component) when t < 0.

- Noncausal effects — Include response to negative lags. Use this option if the estimation data includes output feedback:

  $$u(t) = \sum_{k=0}^{\infty} h(k)y(t-k) + r(t)$$

  where $h(k)$ is the impulse response of the regulator and r is a setpoint or disturbance term. The algorithm handles the existence and character of such feedback h, and estimates h in the same way as g by simply trading places between y and u in the estimation call. Using `impulseest` with an indication of negative delays, `mi = impulseest(data,nk,nb)`, where `nk` < 0, returns a model `mi` with an impulse response

  $$[h(-nk), h(-nk-1), \ldots, h(0), g(1), g(2), \ldots, g(nb+nk)]$$

  that has an alignment that corresponds to the lags $[nk, nk+1, .., 0, 1, 2, \ldots, nb+nk]$. The algorithm achieves this alignment because the input delay (`InputDelay`) of model `mi` is `nk`.

For a multi-input multi-output system, the impulse response $g(k)$ is an ny-by-nu matrix, where ny is the number of outputs and nu is the number of inputs. The i–j element of the matrix $g(k)$ describes the behavior of the ith output after an impulse in the jth input.

See Also

`impulseestOptions` | `impulse` | `impulseplot` | `idtf` | `step` | `cra` | `spa`

Topics

"What Is Time-Domain Correlation Analysis?"

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`RegularizationKernel`

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 `RegularizationKernel` is `'none'`; otherwise, 0.

- Nonnegative integer

Use a nonzero value of prewhitening only for unregularized estimation (`RegularizationKernel` 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. 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 `RegularizationKernel` is not `'none'`.

  Default: `'fmincon'`
- `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('RegularizationKernel','HF','PW',5);
```

Alternatively, use dot notation to specify these options.

```
options = impulseestOptions;
options.RegularizationKernel = 'HF';
options.PW = 5;
```

Tips

- A linear model cannot describe arbitrary input-output offsets. Therefore, before using the data, you must either detrend it or remove the levels using InputOffset and OutputOffset. You can reintroduce the removed data during simulations by using the InputOffset and OutputOffset simulation options. For more information, see simOptions.
- Estimating the impulse response by specifying either InputOffset, OutputOffset or both is equivalent to detrending the data using getTrend and detrend. For example:

  ```
  opt = impulseestOptions('InputOffset',in_off,'OutputOffset',out_off);
  impulseest(data,opt);
  ```

  is the same as:

  ```
  Tr = getTrend(data),
  Tr.InputOffset = in_off
  TR.OutputOffset = out_off
  ```

```
dataT = detrend(data,Tr)
impulseest(dataT)
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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 with additional plot customization options

Syntax

```
h = impulseplot(sys)
h = impulseplot(sys1,sys2,...,sysN)
h = impulseplot(sys1,LineSpec1,...,sysN,LineSpecN)
h = impulseplot( ___ ,tFinal)
h = impulseplot( ___ ,t)
h = impulseplot(AX, ___ )
h = impulseplot( ___ ,plotoptions)
```

Description

`impulseplot` lets you plot dynamic system impulse responses with a broader range of plot customization options than `impulse`. You can use `impulseplot` to obtain the plot handle and use it to customize the plot, such as modify the axes labels, limits and units. You can also use `impulseplot` to draw an impulse response plot on an existing set of axes represented by an axes handle. To customize an existing impulse plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox). To create impulse plots with default options or to extract impulse response data, use `impulse`.

`h = impulseplot(sys)` plots the impulse response of the dynamic system model `sys` and returns the plot handle `h` to the plot. You can use this handle `h` to customize the plot with the `getoptions` and `setoptions` commands.

`h = impulseplot(sys1,sys2,...,sysN)` plots the impulse response of multiple dynamic systems `sys1,sys2,…,sysN` on the same plot. All systems must have the same number of inputs and outputs to use this syntax.

`h = impulseplot(sys1,LineSpec1,...,sysN,LineSpecN)` sets the line style, marker type, and color for the impulse response of each system. All systems must have the same number of inputs and outputs to use this syntax.

`h = impulseplot( ___ ,tFinal)` simulates the impulse response from `t = 0` to the final time `t = tFinal`. Specify `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.

`h = impulseplot( ___ ,t)` simulates the impulse response using the time vector `t`. Specify `t` in the system time units, specified in the `TimeUnit` property of `sys`.

h = impulseplot(AX, ___) plots the impulse response on the Axes object in the current figure with the handle AX.

h = impulseplot(___ ,plotoptions) plots the impulse response with the options set specified in plotoptions. You can use these options to customize the impulse plot appearance using the command line. Settings you specify in plotoptions overrides the preference settings in the MATLAB session in which you run impulseplot. Therefore, this syntax is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

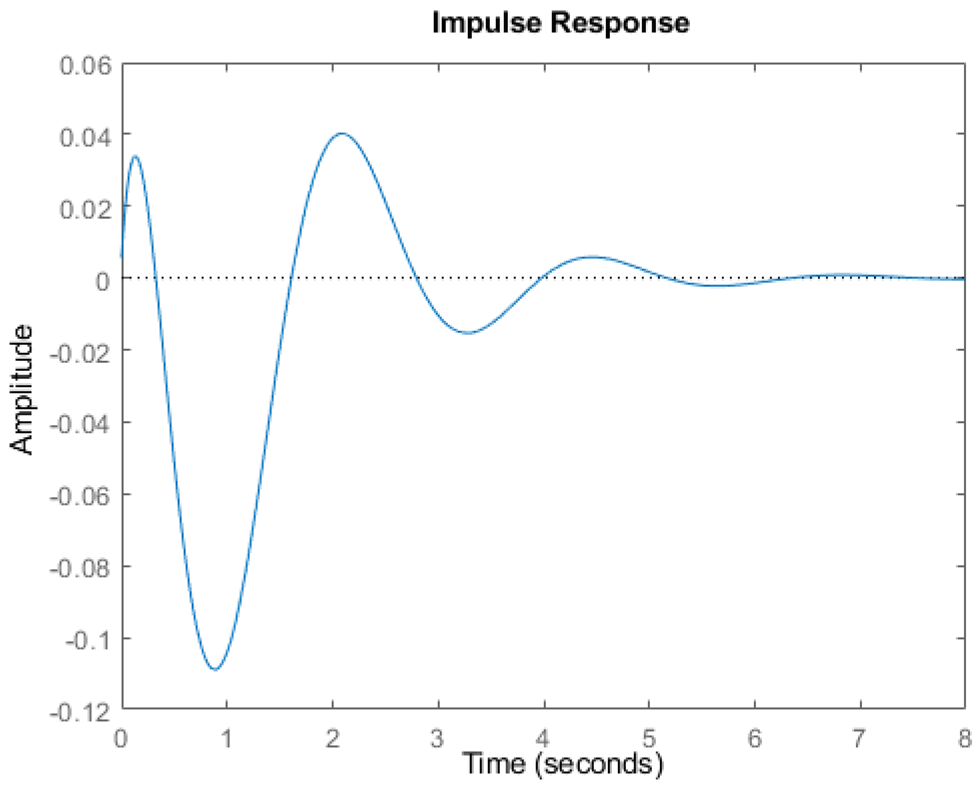
Examples

Customize Impulse Plot using Plot Handle

For this example, use the plot handle to change the time units to minutes and turn on the grid.

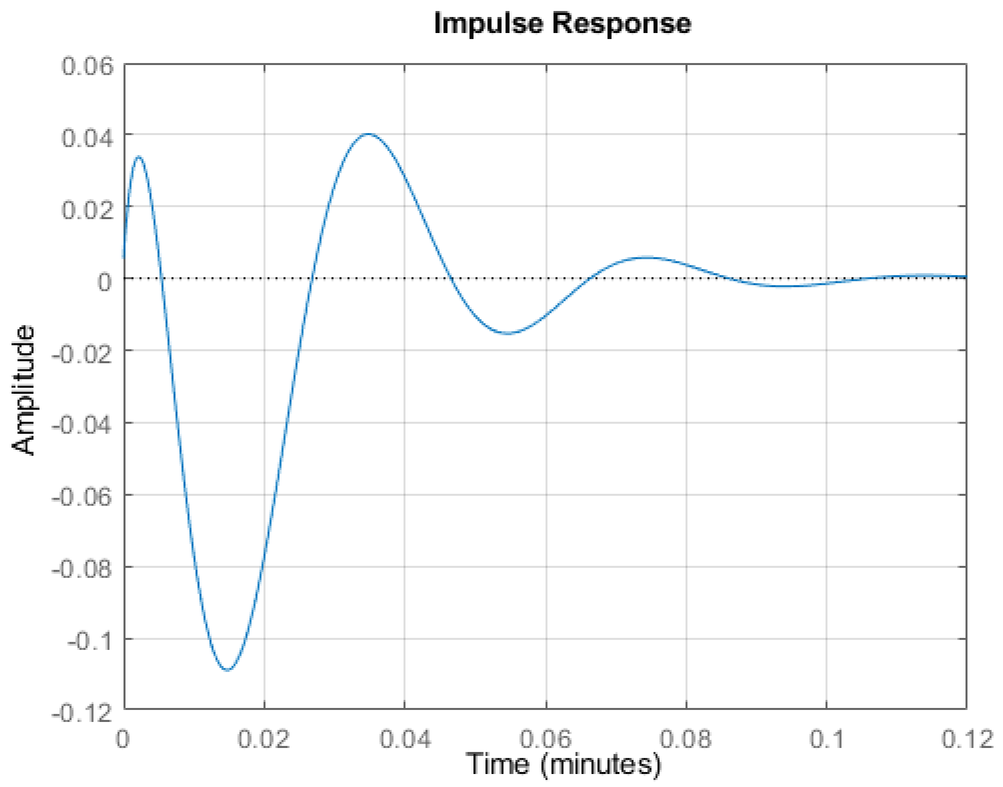
Generate a random state-space model with 5 states and create the impulse response plot with plot handle h.

```
rng("default")
sys = rss(5);
h = impulseplot(sys);
```



Change the time units to minutes and turn on the grid. To do so, edit properties of the plot handle, h using setoptions.

```
setoptions(h,'TimeUnits','minutes','Grid','on');
```



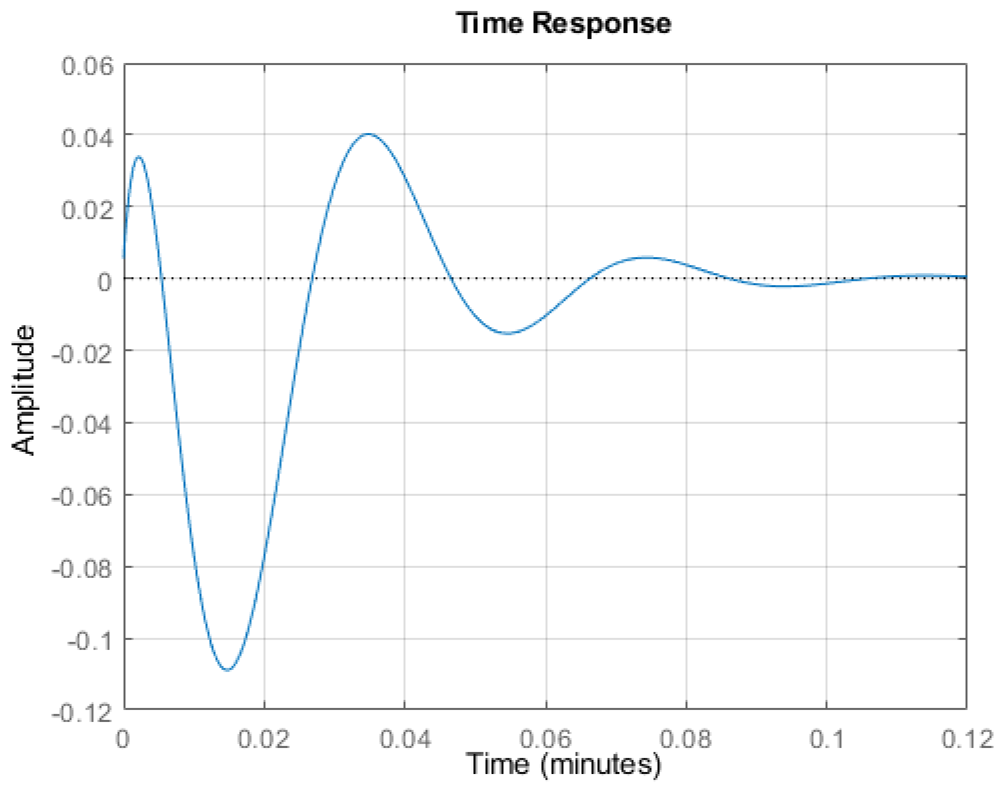
The impulse plot automatically updates when you call `setoptions`.

Alternatively, you can also use the `timeoptions` command to specify the required plot options. First, create an options set based on the toolbox preferences.

```
plotoptions = timeoptions('cstprefs');
```

Change properties of the options set by setting the time units to minutes and enabling the grid.

```
plotoptions.TimeUnits = 'minutes';
plotoptions.Grid = 'on';
impulseplot(sys,plotoptions);
```



You can use the same option set to create multiple impulse plots with the same customization. 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 `TimeUnits` and `Grid`, override the toolbox preferences.

Impulse Plot with Specified Grid Color

For this example, consider a MIMO state-space model with 3 inputs, 3 outputs and 3 states. Create a impulse plot with red colored grid lines.

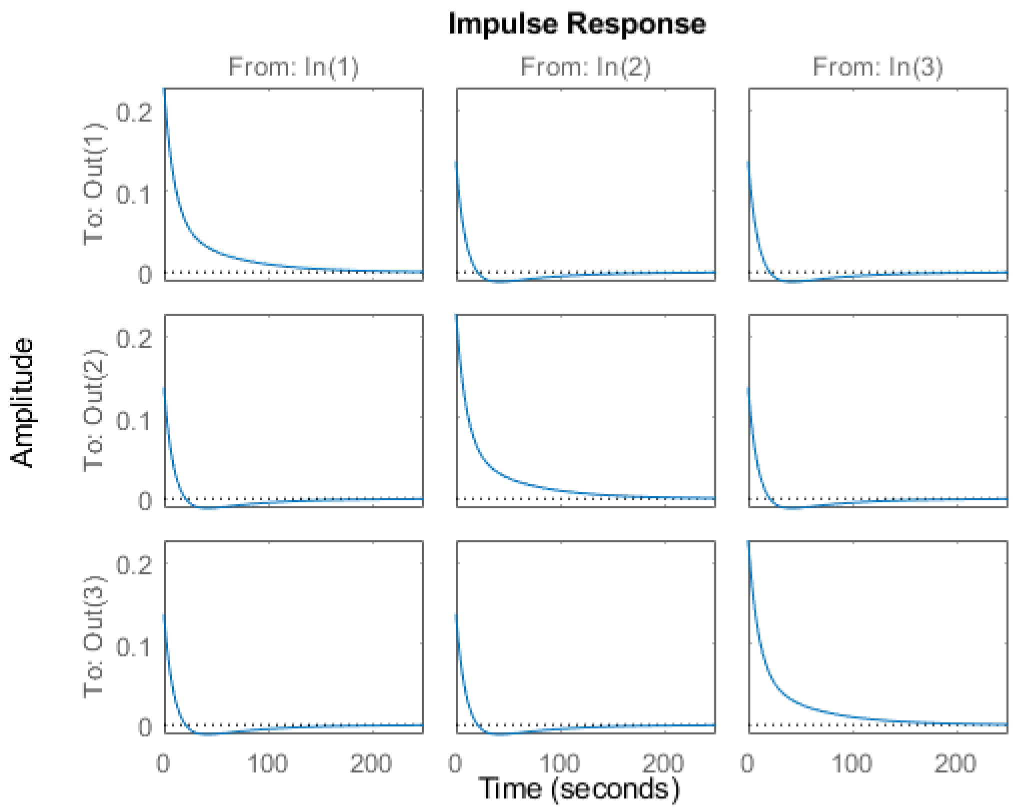
Create the MIMO state-space model `sys_mimo`.

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
size(sys_mimo)
```

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

Create an impulse plot with plot handle `h` and use `getoptions` for a list of the options available.

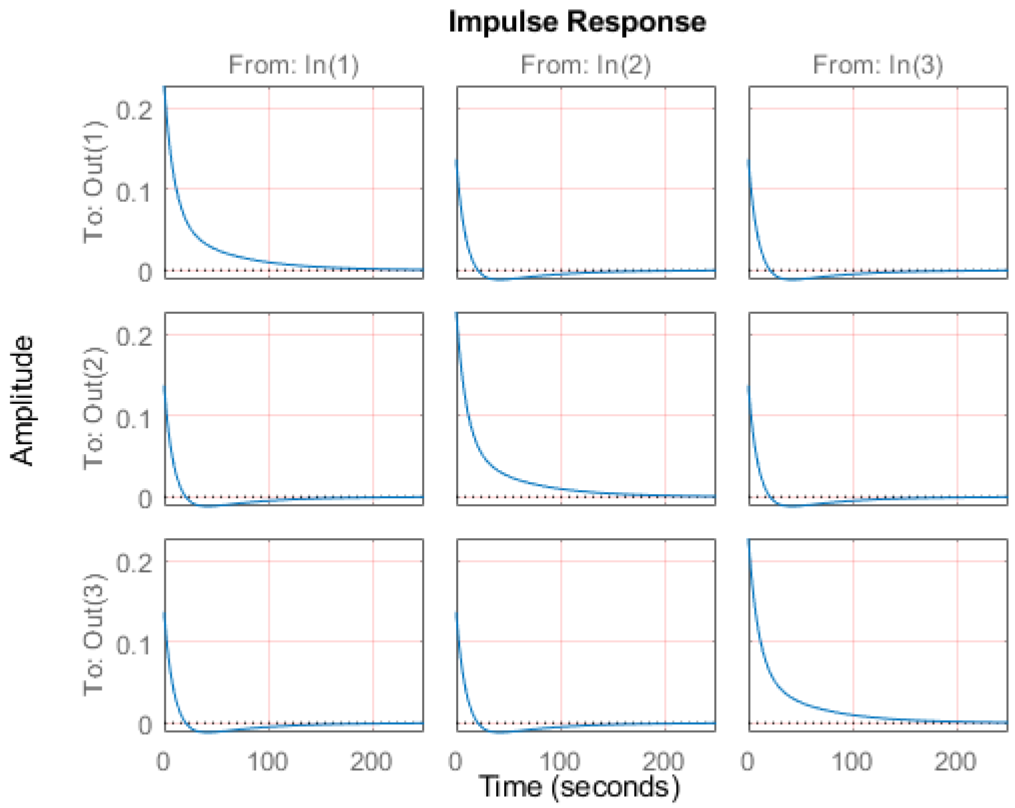
```
h = impulseplot(sys_mimo)
```



```
h =

    resppack.timeplot

p = getoptions(h)

p =

                 Normalize: 'off'
        SettleTimeThreshold: 0.0200
             RiseTimeLimits: [0.1000 0.9000]
                  TimeUnits: 'seconds'
    ConfidenceRegionNumberSD: 1
                 IOGrouping: 'none'
                InputLabels: [1x1 struct]
               OutputLabels: [1x1 struct]
               InputVisible: {3x1 cell}
              OutputVisible: {3x1 cell}
                      Title: [1x1 struct]
                     XLabel: [1x1 struct]
                     YLabel: [1x1 struct]
                  TickLabel: [1x1 struct]
                       Grid: 'off'
                  GridColor: [0.1500 0.1500 0.1500]
                       XLim: {3x1 cell}
                       YLim: {3x1 cell}
                   XLimMode: {3x1 cell}
                   YLimMode: {3x1 cell}
```

Use `setoptions` to update the plot with the requires customization.

```
setoptions(h,'Grid','on','GridColor',[1 0 0]);
```



The impulse plot automatically updates when you call `setoptions`. For MIMO models, `impulseplot` produces a grid of plots, each plot displaying the impulse response of one I/O pair.

Plot Impulse Responses of Identified Models with Confidence Region

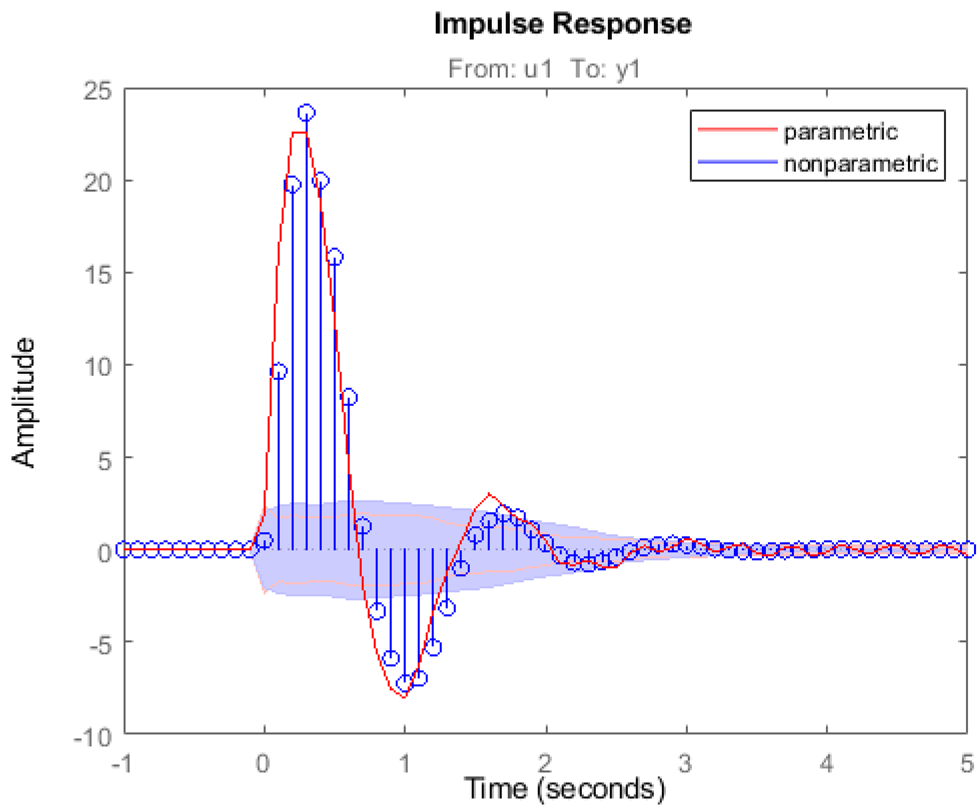
Compare the impulse response of a parametric identified model to a nonparametric (empirical) model, and view their 3-σ confidence regions. (Identified models require System Identification Toolbox™ software.)

Identify a parametric and a nonparametric model from sample data.

```
load iddata1 z1
sys1 = ssest(z1,4);
sys2 = impulseest(z1);
```

Plot the impulse responses of both identified models. Use the plot handle to display the 3-σ confidence regions.

```
t = -1:0.1:5;
h = impulseplot(sys1,'r',sys2,'b',t);
showConfidence(h,3)
legend('parametric','nonparametric')
```



The nonparametric model `sys2` shows higher uncertainty.

Customized Impulse Response Plot at Specified Time

For this example, examine the impulse response of the following zero-pole-gain model and limit the impulse plot to `tFinal` = 15 s. Use 15-point blue text for the title. This plot should look the same, regardless of the preferences of the MATLAB session in which it is generated.

```
sys = zpk(-1,[-0.2+3j,-0.2-3j],1)*tf([1 1],[1 0.05]);
tFinal = 15;
```

First, create a default options set using `timeoptions`.

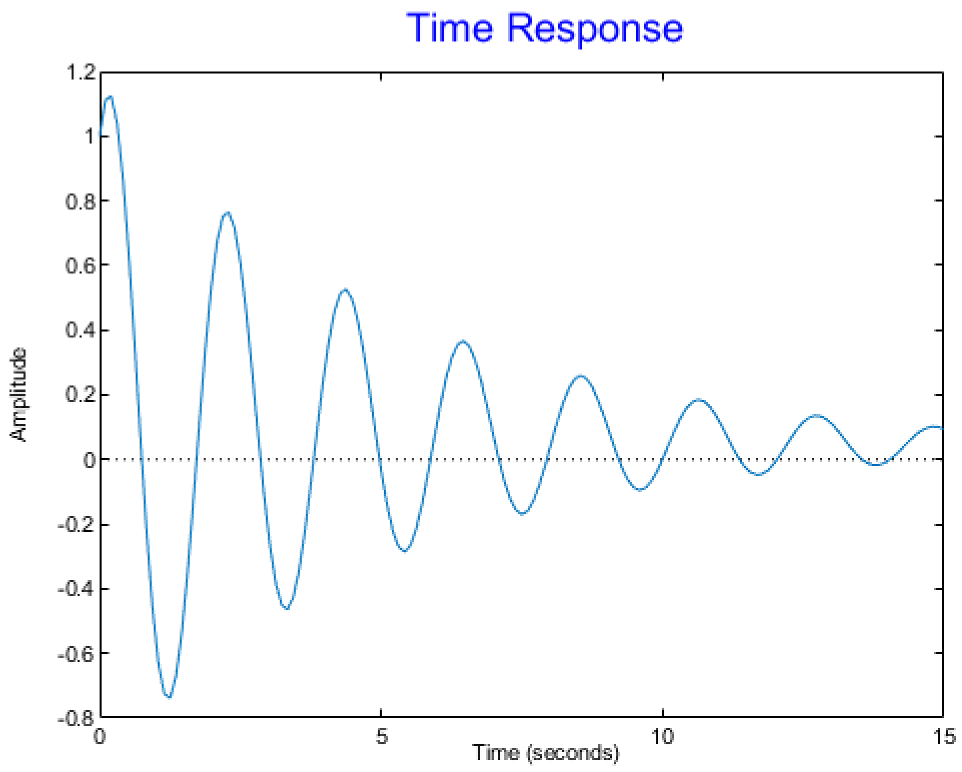
```
plotoptions = timeoptions;
```

Next change the required properties of the options set `plotoptions`.

```
plotoptions.Title.FontSize = 15;
plotoptions.Title.Color = [0 0 1];
```

Now, create the impulse response plot using the options set `plotoptions`.

```
h = impulseplot(sys,tFinal,plotoptions);
```



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

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Sparse state-space models, such as `sparss` or `mechss` models. Final time `tFinal` must be specified when using sparse models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value to plot the impulse response data.

  - For uncertain control design blocks, the function plots the nominal value and random samples of the model.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models.

If `sys` is an array of models, the function plots the impulse response of all models in the array on the same axes.

LineSpec — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'_'`	Horizontal line
`'|'`	Vertical line
`'s'`	Square
`'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'p'`	Pentagram
`'h'`	Hexagram

Color	Description
`y`	yellow
`m`	magenta
`c`	cyan

Color	Description
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

`tFinal` — Final time for impulse response computation
scalar

Final time for impulse response computation, specified as a scalar. Specify `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.

`t` — Time for impulse response simulation
vector

Time for impulse response simulation, specified as a vector. Specify the time vector `t` in the system time units, specified in the `TimeUnit` property of `sys`. The time vector must be real, finite, and must contain monotonically increasing and evenly spaced time samples.

The time vector `t` is:

- $t = T_{initial}:T_{sample}:T_{final}$, for discrete-time systems.
- $t = T_{initial}:dt:T_{final}$, for continuous-time systems. Here, dt is the sample time of a discrete approximation of the continuous-time system.

`AX` — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes and if the current axes are Cartesian axes, then `impulseplot` plots on the current axes. Use `AX` to plot into specific axes when creating a impulse plot.

`plotoptions` — Impulse plot options set
`TimePlotOptions` object

Impulse plot options set, specified as a `TimePlotOptions` object. You can use this option set to customize the impulse plot appearance. Use `timeoptions` to create the option set. Settings you specify in `plotoptions` overrides the preference settings in the MATLAB session in which you run `impulseplot`. Therefore, `plotoptions` is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

For the list of available options, see `timeoptions`.

Output Arguments

`h` — Plot handle
`handle` object

Plot handle, returned as a `handle` object. Use the handle `h` to get and set the properties of the impulse plot using `getoptions` and `setoptions`. For the list of available options, see the *Properties* *and Values Reference* section in “Customizing Response Plots from the Command Line” (Control System Toolbox).

See Also

`getoptions` | `impulse` | `setoptions` | `showConfidence`

Topics

“Customizing Response Plots from the Command Line” (Control System Toolbox)

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

initialCondition

Initial condition representation for linear time-invariant systems

Description

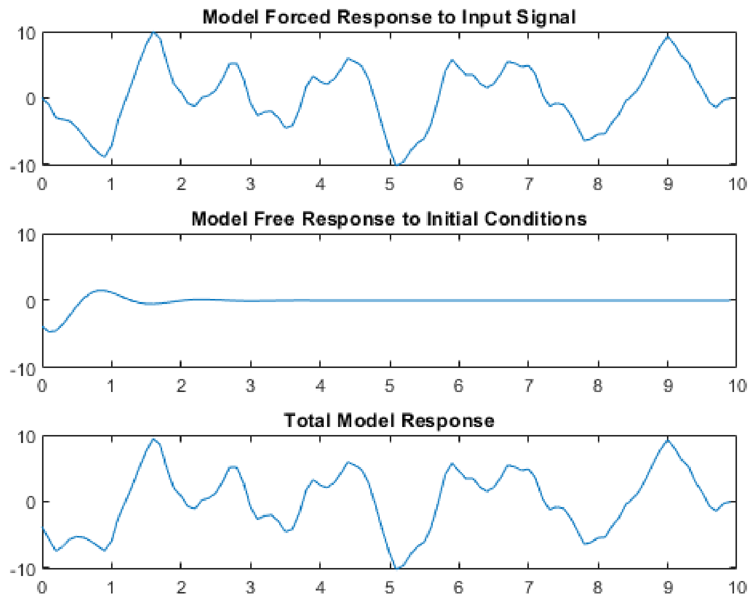
An `initialCondition` object encapsulates the initial-condition information for a linear time-invariant (LTI) model. The object generalizes the numeric vector representation of the initial states of a state-space model so that the information applies to linear models of any form—transfer functions, polynomial models, or state-space models.

You can estimate and retrieve initial conditions when you identify a linear model using commands such as `tfest` or compare model response to measured input/output data using `compare`. The software estimates the initial condition value by minimizing the simulation or prediction error against the measured output data. You can then apply those initial conditions in a subsequent simulation, using commands such as `sim` or `predict`, to confirm model performance with respect to the same measurement data. Use the `initialCondition` command to create an `initialCondition` object from a state-space model specification or from any LTI model of a free response.

The `initialCondition` object can also be seen as a representation of the free response of a linear model. The simulation functions use this information to compute the model response in the following manner:

1. Compute the forced response of the model to the input signal. The forced response is the standard simulation output when there are no specified initial conditions.
2. Compute the impulse response of the model and scale the result to generate the free response of the model to the specified initial conditions.
3. Add the forced response and the free response together to form the total system response.

The figure illustrates this process.



For continuous systems (*Ts* = 0), the free response *G*(*s*) for the initial state vector x_0 is

$$G(s) = C(sI - A)^{-1}x_0$$

Here, *C* is equivalent to the state-space measurement matrix *C* and *A* is equivalent to the state-space state matrix *A*.

For discrete systems (*Ts* >0), the free response *G*(*z*) is

$$G(z) = zC(zI - A)^{-1}x_0$$

The `initialCondition` object represents the free response in state-space form. The object is a structure with properties containing the state-space *A* and *C* matrices and the initial state vector x_0. For `idtf` and `idpoly` models, using an `initialCondition` object is the only way to represent and use initial conditions for simulation. For `idss` models, you can use either an `initialCondition` object or a numeric initial state vector. When you obtain initial conditions `ic` for multiexperiment data, `ic` is an object array, with one `initialCondition` object for each experiment.

Creation

You can obtain an `initialCondition` object in one of four ways.

- Model estimation — Specify that the estimation function return the estimated initial condition that corresponds to the estimation input/output data. For example, you can use the following command to obtain the estimated initial condition `ic` for a transfer function model `sys` that is estimated with input/output data `z`.

  ```
  [sys,ic] = tfest(z,2,1)
  ```

  For an example, see “Obtain Estimated Initial Conditions” on page 1-827.

- Model-to-data comparison using any input/output data — Specify that `compare` return the estimated initial condition that the function estimates internally to support the `fit` assessment. For example, you can use the following command to obtain the initial condition `ic` for the linear model `sys` when determining the fit against input/output data `z`. `yp` is the simulated or predicted model output.

  ```
  [yp,fit,ic] = compare(z,sys)
  ```

  For an example, see “Obtain Initial Conditions for New Data” on page 1-831.
- Direct construction — Use the `initialCondition` command to encapsulate the state-space form of a free-response model in an `initialCondition` object.

  ```
  ic = initialCondition(A,X0,C,Ts)
  ```

  For an example, see “Construct initialCondition Object from State-Space Model” on page 1-837.
- Free-response model conversion — Use the `initialCondition` command to convert an LTI free-response model into an `initialCondition` object.

  ```
  ic = initialCondition(G)
  ```

  For an example, see “Convert Free-Response Model to initialCondition Object” on page 1-840.

For information on functions you can use to extract information from or transform `initialCondition` objects, see “Object Functions” on page 1-825.

Syntax

```
ic = initialCondition(A,X0,C,Ts)
ic = initialCondition(G)
```

Description

`ic = initialCondition(A,X0,C,Ts)` creates an `initialCondition` object that represents the free response to an initial condition, expressed in state-space form, of an LTI model.

$$dx = Ax$$
$$y = Cx$$
$$x(0) = x_0$$

`ic` stores this model in the form of properties on page 1-825. `A` and `C` correspond to a state-space realization of the model, `X0` to the initial state vector x_0, and `Ts` to the sample time. You can use `ic` to specify initial conditions when simulating any type of LTI system.

`ic = initialCondition(G)` creates an `initialCondition` object corresponding to a linear model `G` of the free response.

Input Arguments

`G` — Free-response model
LTI model

Free-response model, specified as an LTI model with no inputs. In the continuous-time case, `G` must be strictly proper. In the discrete-time case, `G` must be biproper. For an example of using a free-response model to obtain an `initialCondition` object, see “Convert Free-Response Model to initialCondition Object” on page 1-840.

Properties

`A` — *A* matrix of state-space realization of LTI free response
numeric matrix

A matrix of the state-space realization of the LTI free response, specified as an N_x-by-N_x numeric matrix, where N_x is the number of states. For an example of using this property, see “Obtain Estimated Initial Conditions” on page 1-827.

`X0` — Initial states of state-space realization of LTI free response
numeric vector

Initial states of the state-space realization of the LTI free response, specified as a numeric vector of length N_x. For an example of using this property, see “Obtain Estimated Initial Conditions” on page 1-827.

`C` — *C* matrix of state-space realization of LTI free response
numeric matrix

C matrix of the state-space realization of the LTI free response, specified as an N_y-by-N_x numeric matrix, where N_y is the number of outputs. For an example of using this property, see “Obtain Estimated Initial Conditions” on page 1-827.

`Ts` — Sample time
0 | –1 | positive scalar

Sample time of the LTI free response, specified as one of the following:

- Continuous-time model — 0
- Discrete-time model with a specified sampling time — Positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model
- Discrete-time model with unspecified sample time — –1

The sample time of an `initialCondition` object is the same as for the dynamic system model that the object corresponds to.

Object Functions

Functions applicable to `initialCondition` objects are those that can return, use, or convert the objects.

Initial Condition (IC) Role	LTI Function Type	Syntax Example	Example Links
Return estimated IC objects	All estimation functions, `compare`	`[sys,ic] = tfest(data,2,1)`	“Obtain Estimated Initial Conditions” on page 1-827 “Obtain Initial Conditions for Multiexperiment Data” on page 1-834 “Obtain Initial Conditions for New Data” on page 1-831
Use IC objects for model response	Option sets for model response functions	`opt = simOptions('InitialCondition',ic)`	“Apply Initial Conditions in Simulation” on page 1-829
Convert IC objects into Dynamic System Models (DSMs)	DSM object functions	`g = idtf(ic)`	“Visualize Free Response to Initial Condition” on page 1-828
Analyze models converted from IC objects	DSM analysis functions	`y_g = impulse(g)`	“Visualize Free Response to Initial Condition” on page 1-828

Estimate and Return Initial Conditions

tfest	Estimate transfer function
procest	Estimate process model using time or frequency data
arx	Estimate parameters of ARX, ARIX, AR, or ARI model
armax	Estimate parameters of ARMAX, ARIMAX, ARMA, or ARIMA model using time-domain data
bj	Estimate Box-Jenkins polynomial model using time domain data
oe	Estimate output-error polynomial model using time-domain or frequency-domain data
polyest	Estimate polynomial model using time- or frequency-domain data
compare	Compare identified model output and measured output

Use Initial Conditions

sim	Simulate response of identified model
simOptions	Option set for sim
predict	Predict state and state estimation error covariance at next time step using extended or unscented Kalman filter, or particle filter
predictOptions	Option set for predict
pe	Prediction error for identified model
peOptions	Option set for pe
resid	Compute and test residuals
residOptions	Option set for resid
compare	Compare identified model output and measured output
compareOptions	Option set for compare

Convert Initial Conditions to Dynamic System Model

idss	State-space model with identifiable parameters
idpoly	Polynomial model with identifiable parameters
idtf	Transfer function model with identifiable parameters

Analyze Free Model Response using `idss` Form

impulse	Impulse response plot of dynamic system; impulse response data
freqresp	Frequency response over grid

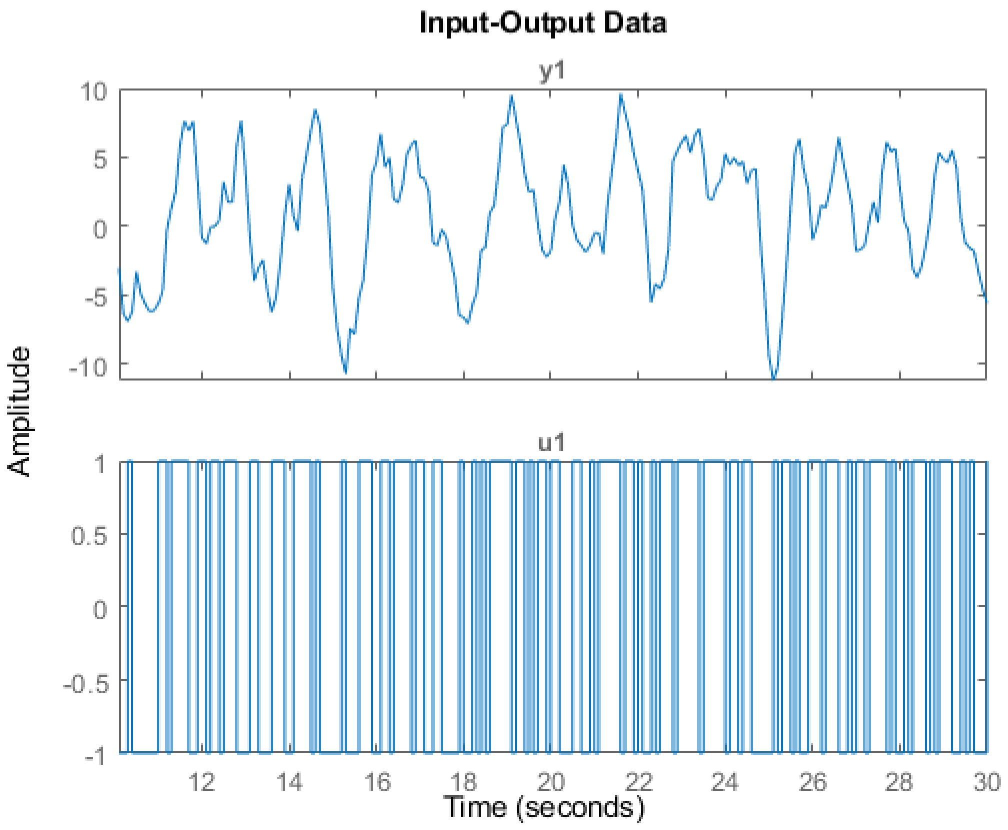
Examples

Obtain Estimated Initial Conditions

Estimate a transfer function model and obtain estimated initial conditions.

Load and plot the data.

```
load iddata1ic.mat z1i
plot(z1i)
```



The output data does not start at 0.

Estimate a second-order transfer function `sys_tf`. Specify that the function return the initial conditions `ic`.

```
[sys_tf,ic] = tfest(z1i,2,1);
```

Examine the contents of ic. ic includes, in state-space form, the free response model defined by matrices A and C, the initial state vector X0, and the sample time Ts.

```
A = ic.A
```

```
A = 2×2

   -2.9841   -5.5848
    4.0000         0
```

```
C = ic.C
```

```
C = 1×2

    0.2957    5.2441
```

```
x0 = ic.X0
```

```
x0 = 2×1

   -0.9019
   -0.6161
```

```
Ts = ic.Ts
```

```
Ts = 0
```

ic is specific to the estimation data z1i. You can use ic to establish initial conditions when you simulate any LTI model using the input signal from z1i and compare the response with the z1i output signal.

Visualize Free Response to Initial Condition

Visualize the free response encapsulated in an initialCondition object by generating an impulse response.

Estimate a transfer function and return the initial condition ic_tf.

```
load iddata1ic z1i
[sys_tf,ic_tf] = tfest(z1i,2,1);
ic_tf
```

```
ic_tf = 
  initialCondition with properties:

      A: [2x2 double]
     X0: [2x1 double]
      C: [0.2957 5.2441]
     Ts: 0
```

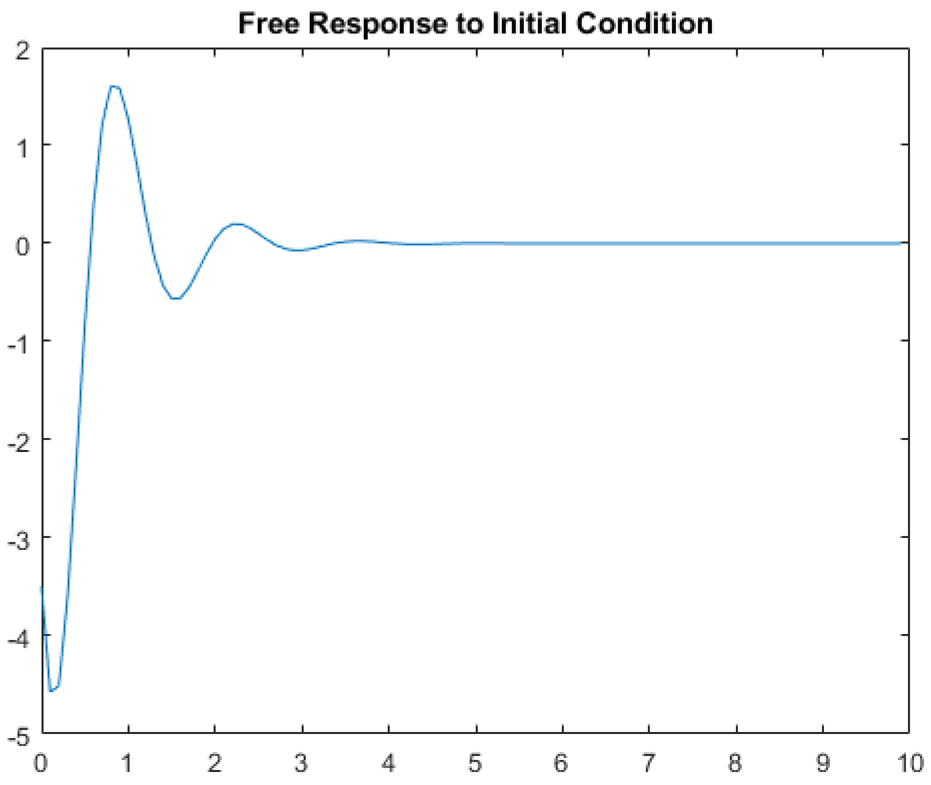
ic_tf contains the information necessary to compute the free response to an initial condition.

Convert `ic_tf` into an `idss` object that can be passed to the `impulse` function.

```
ic_tfss = idss(ic_tf);
```

Create a time vector `t` that spans the data set. Compute the impulse response.

```
t = 0:0.1:9.9;
t = t';
yimp = impulse(ic_tfss,t);
plot(t,yimp)
title('Free Response to Initial Condition')
```



The free response is a transient that lasts for about four seconds.

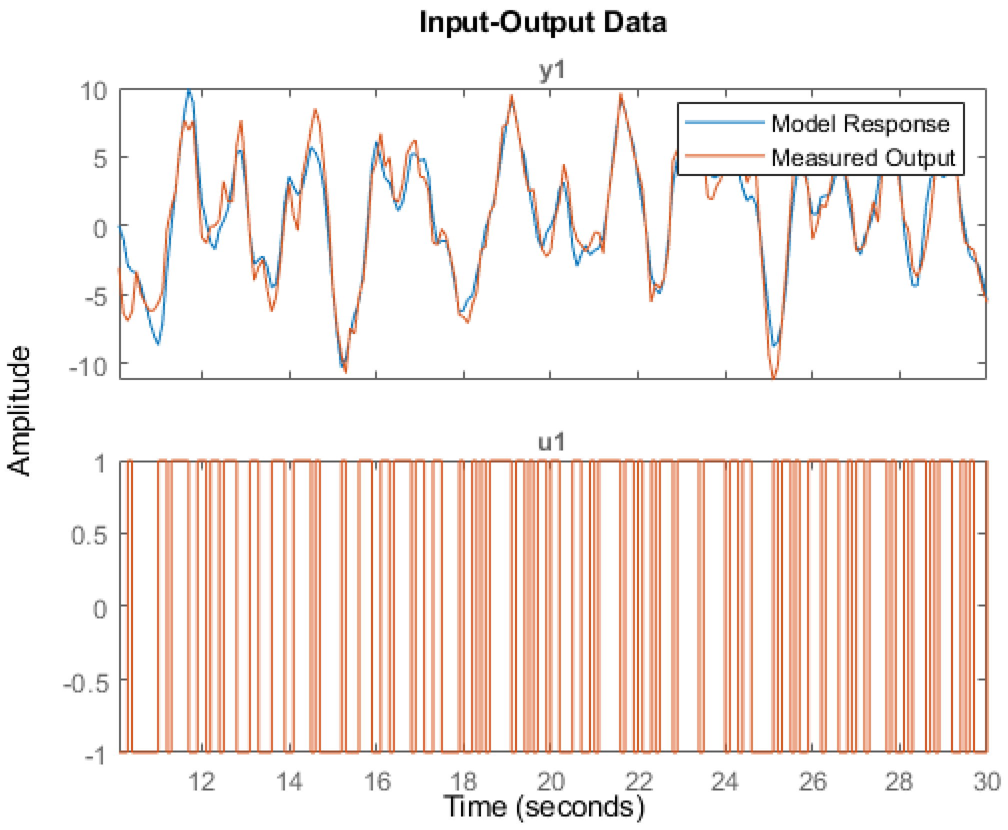
Apply Initial Conditions in Simulation

Load the data and estimate a second-order transfer function `sys`. Return initial conditions in `ic`.

```
load iddata1ic z1i
[sys,ic] = tfest(z1i,2,1);
```

Simulate `sys` using the estimation data, but without incorporating the initial condition. Plot the simulated output with the measured output.

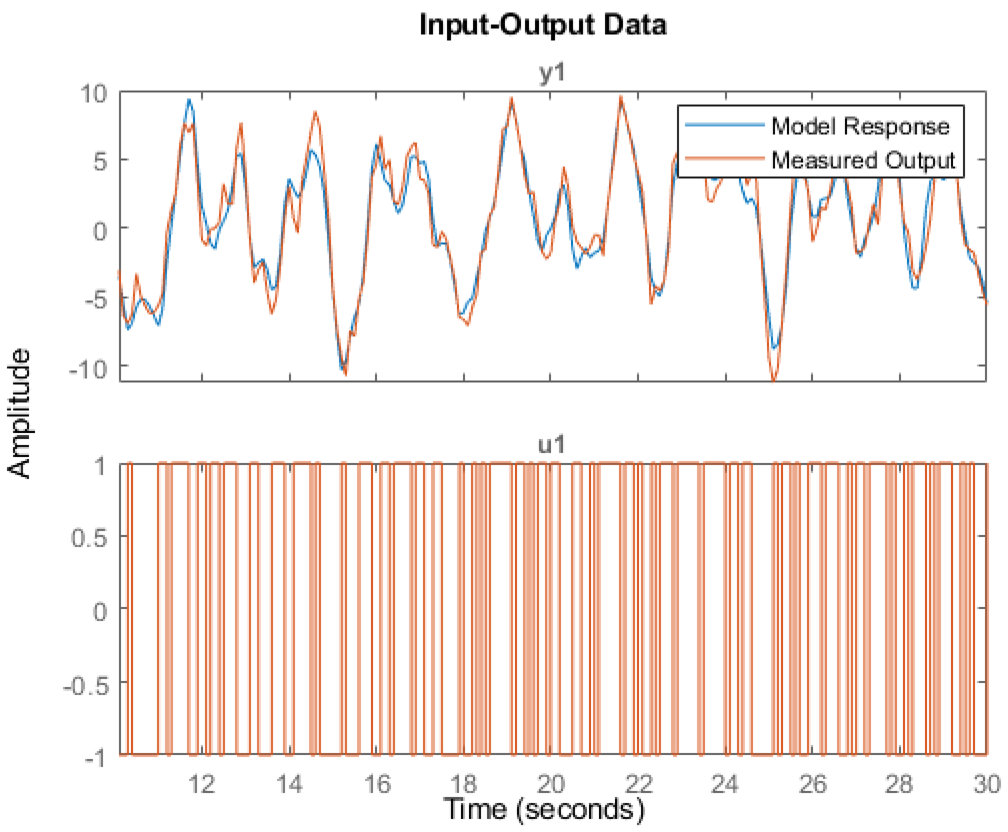
```
y_no_ic = sim(sys,z1i);
plot(y_no_ic,z1i)
legend('Model Response','Measured Output')
```



The measured and simulated outputs do not agree at the beginning of the simulation.

Incorporate `ic` into the `simOptions` option set `opt`. Simulate and plot the model response using `opt`.

```
opt = simOptions('InitialCondition',ic);
y_ic = sim(sys,z1i,opt);
plot(y_ic,z1i);
legend('Model Response','Measured Output')
```



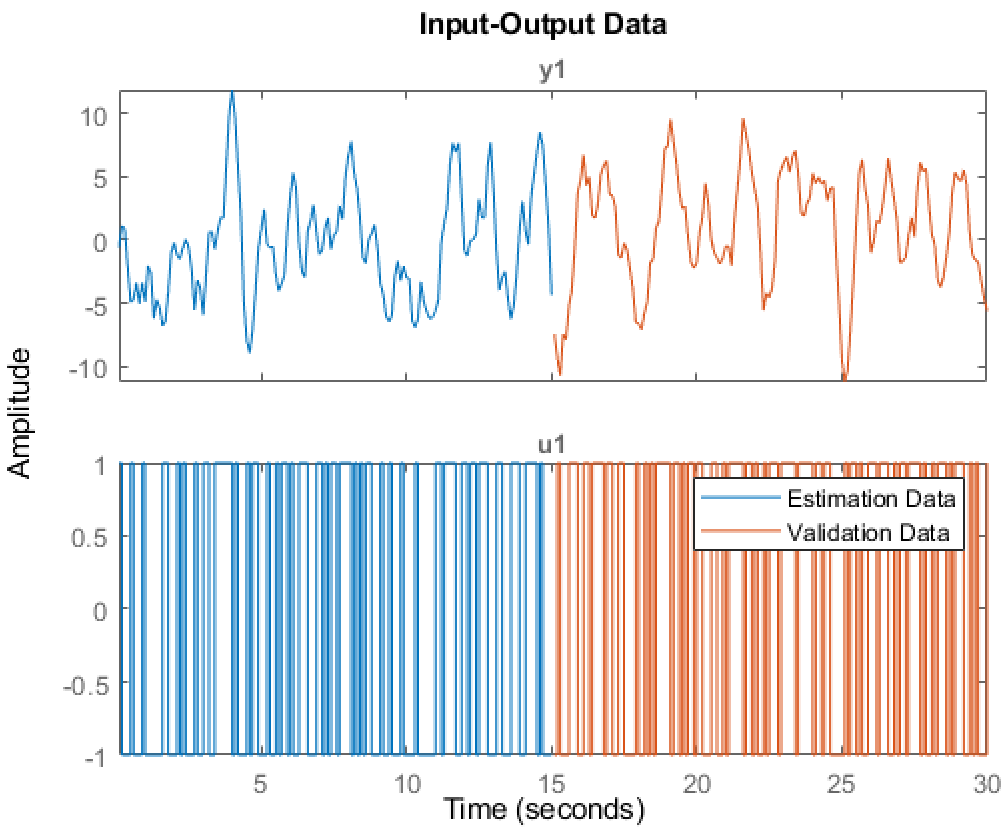
The simulation combines the model response to the input signal with the free response to the initial condition. The measured and simulated outputs now have better agreement at the beginning of the simulation.

Obtain Initial Conditions for New Data

An estimated `initialCondition` object is specific to the data from which you estimated it. If you want to simulate your model with new data, such as validation data, you need to estimate a new initial condition for that data. To do so, use the `compare` command.

Load data and split it into estimation and validation data sets.

```
load iddata1 z1
z1_est = z1(1:150);
z1_val = z1(151:300);
plot(z1_est,z1_val);
legend('Estimation Data','Validation Data')
```



Examine the start points of each output data set.

```
e0 = z1_est.y(1)
```

```
e0 = -0.5872
```

```
v0 = z1_val.y(1)
```

```
v0 = -7.4390
```

The two data sets have different starting conditions.

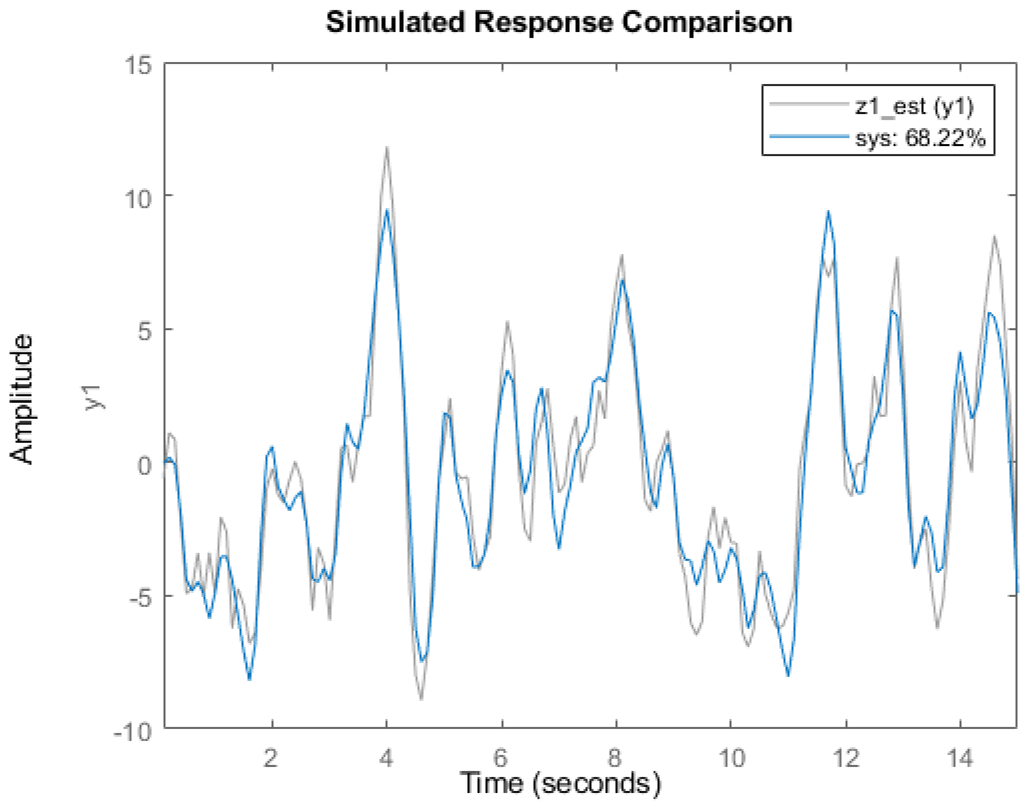
Estimate a second-order transfer function model using `z1_est`. Return the estimated initial conditions in `ic_est`. Display the `X0` property of `ic_est`. This property represents the estimated initial state vector that the free-response model defined by `ic_est.A` and `ic_est.C` responds to.

```
[sys,ic_est] = tfest(z1_est,2,1);
ic_est.X0
```

```
ans = 2×1

   -0.4082
    0.0095
```

You can use `ic_est` if you want to simulate `sys` using `z1_est`. Alternatively, you can use `compare`, which estimates the initial condition independently. Use `compare` twice, once to plot the data and once to return the results. Display the initial state vector `ic_estc.X0` that `compare` estimates.

```
compare(z1_est,sys)
```



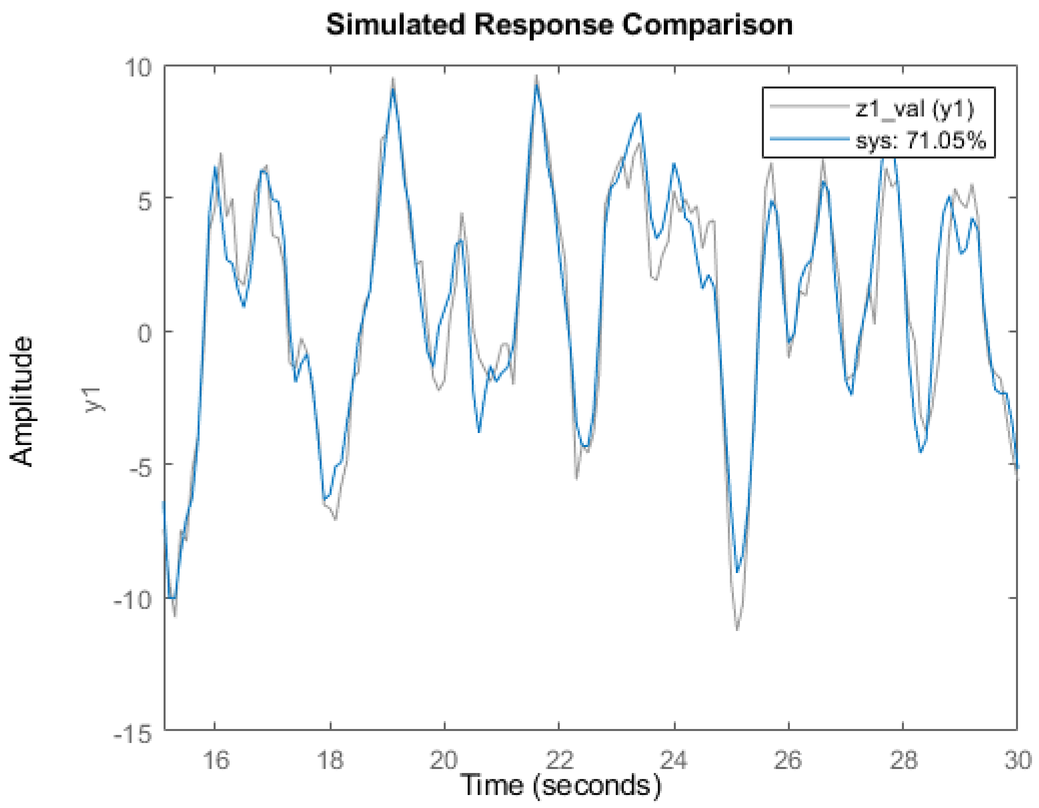
```
[yce,fit,ic_estc] = compare(z1_est,sys);
ic_estc.X0
```

```
ans = 2×1

   -0.4082
    0.0095
```

The initial state vector `ic_estc.X0` is identical to `ic_est`.

Now evaluate the model with the validation data set. Estimate the initial conditions with the validation data.

```
compare(z1_val,sys)
```



```
[ycv,fit,ic_valc] = compare(z1_val,sys);
ic_valc.X0
```

```
ans = 2×1

   -1.7536
   -0.9547
```

You can use `ic_val` when you simulate `sys` with the `z1_val` input signal and compare the model response to the `z1_val` output signal.

Obtain Initial Conditions for Multiexperiment Data

Estimate an `initialCondition` object array using multiexperiment data.

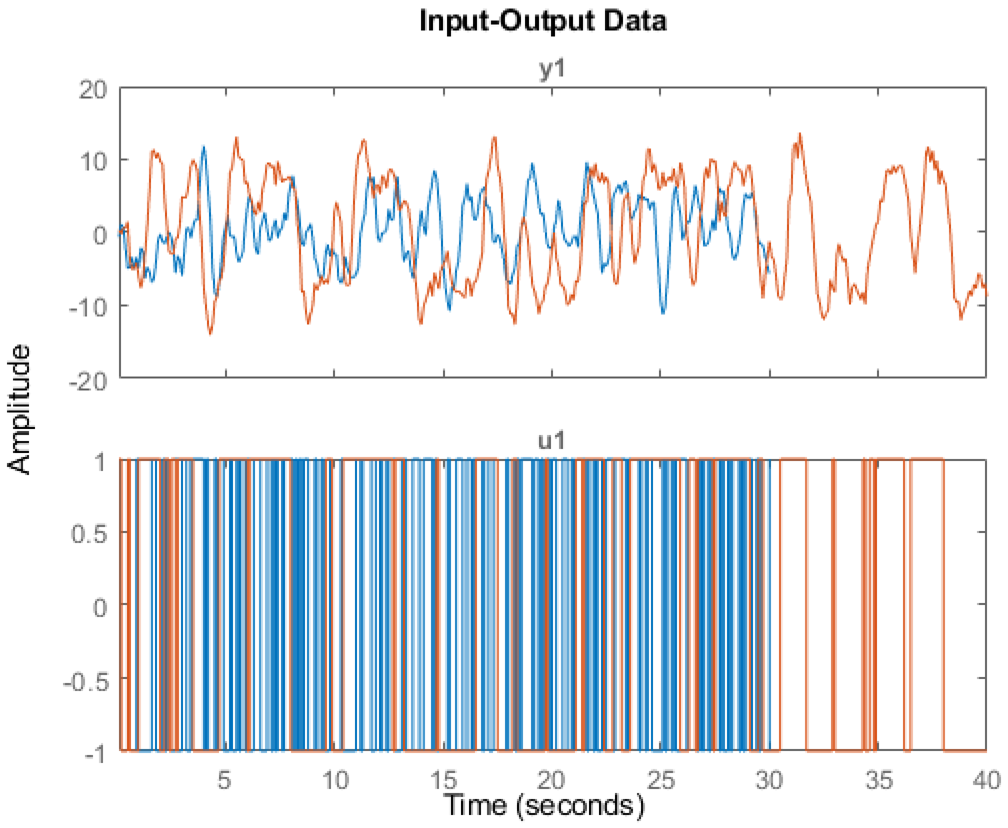
Load data from two experiments. Merge the two data sets into one multiexperiment data set.

```
load iddata1 z1
load iddata2 z2
z12 = merge(z1,z2)
```

```
z12 =
Time domain data set containing 2 experiments.
```

```
Experiment   Samples      Sample Time
   Exp1        300            0.1
   Exp2        400            0.1

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

```
plot(z12)
```



Estimate the second-order transfer function sys and return the initial conditions in ic.

```
np = 2;
nz = 1;
[sys,ic] = tfest(z12,np,nz);
ic
```

```
ic=1×2 object
  1x2 initialCondition array with properties:

    A
    X0
    C
    Ts
```

`ic` is an object array. Display the contents of each object.

```
ic(1,1)
```

```
ans = 
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [-0.7814 5.2530]
    Ts: 0
```

```
ic(1,2)
```

```
ans = 
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [-0.7814 5.2530]
    Ts: 0
```

Compare the `A`, `X0`, and `C` properties for each object.

```
A1 = ic(1,1).A
```

```
A1 = 2×2

   -3.4824   -5.5785
    4.0000         0
```

```
A2 = ic(1,2).A
```

```
A2 = 2×2

   -3.4824   -5.5785
    4.0000         0
```

```
C1 = ic(1,1).C
```

```
C1 = 1×2

   -0.7814    5.2530
```

```
C2 = ic(1,2).C
```

```
C2 = 1×2

   -0.7814    5.2530
```

```
X01 =ic(1,1).X0
```

```
X01 = 2×1
```

```
   -0.6528
   -0.0067
```

```
X02 =ic(1,2).X0
```

```
X02 = 2×1

    0.3076
   -0.0715
```

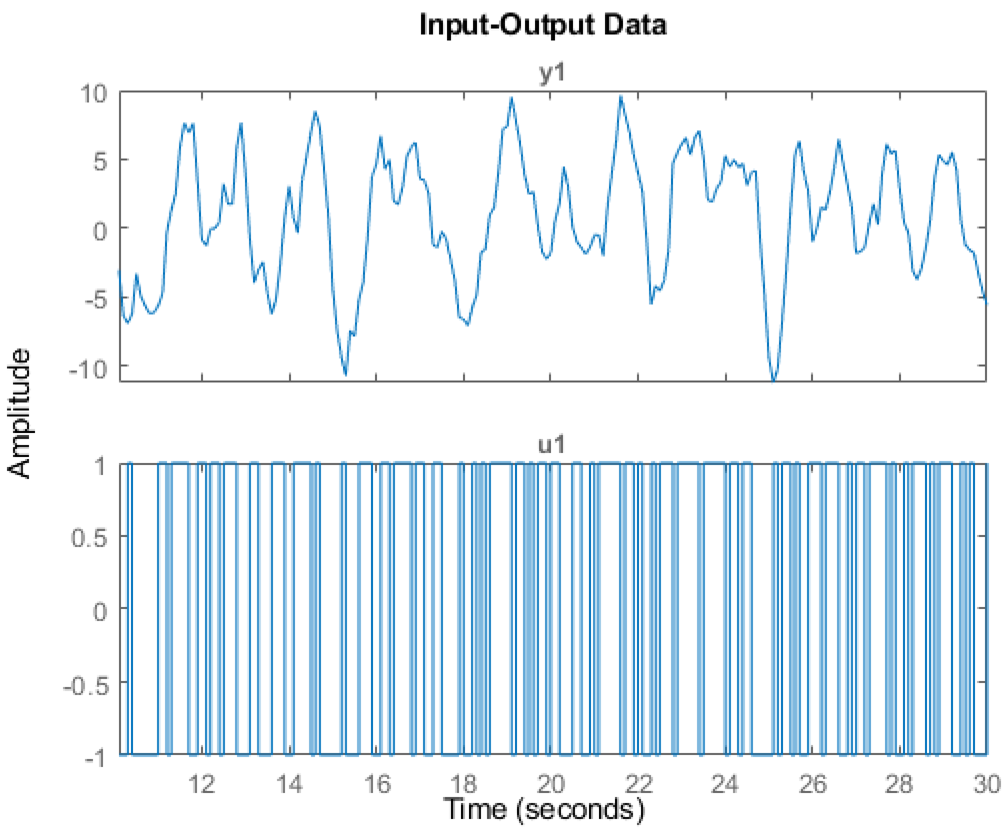
The A and C matrices are identical. These matrices represent the state-space form of sys. The X0 vectors are different. This difference results from the different initial conditions for the two experiments.

Construct initialCondition Object from State-Space Model

Estimate a state-space model and return the initial states. From the model and the initial state vector, construct an initialCondition object that can be used with any linear model.

Load and plot the data.

```
load iddata1ic z1i
plot(z1i)
```



Estimate a state-space model and obtain the estimated initial conditions.

First, set the 'InitialState' name-value pair argument in ssestOptions to 'estimate', which overrides the default setting of 'auto'. The 'estimate' setting always estimates the initial states. The 'auto' setting uses the 'zero' setting if the effect of the initial states on the overall model estimation error is relatively small, and can therefore result in an initial-state vector containing only zeros.

```
opt = ssestOptions;
opt = ssestOptions('InitialState','estimate');
```

Estimate a second-order state-space model sys_ss. Specify the output argument x0 to return the initial state vector. Specify the input argument opt to use your 'InitialState' setting. After estimating, examine x0.

```
[sys_ss,x0] = ssest(z1i,2,opt);
x0
```

```
x0 = 2×1

    0.0631
    0.0329
```

x0 is a nonzero initial state vector.

Simulate the model using x0 and compare the output with the original output data.

To use x0 as the initial condition, specify the 'InitialCondition' name-value pair argument in simOptions as x0.

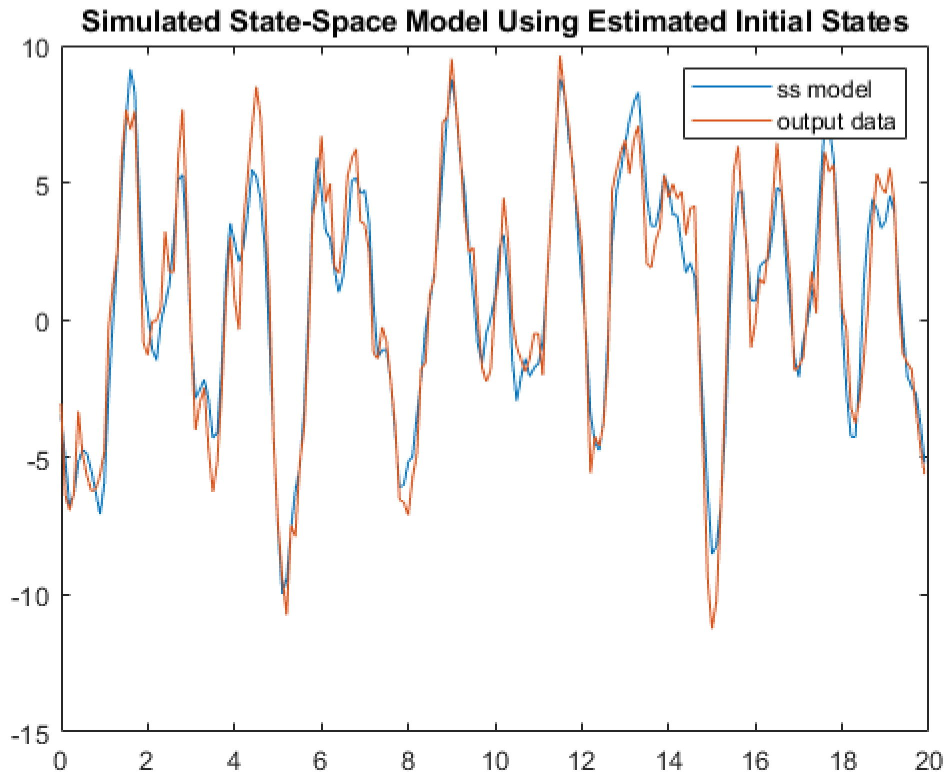
```
opt = simOptions;
opt = simOptions('InitialCondition',x0);
```

Simulate the model using opt and store the response in xss.

```
xss = sim(sys_ss,z1i,opt);
```

Plot the model response with the original output data.

```
t = 0:0.1:19.9;
plot(t',[xss.y z1i.y])
legend('ss model','output data')
title('Simulated State-Space Model Using Estimated Initial States')
```



The simulation starts at a point close to the starting point of the data.

With the `A` and `C` matrices, `x0`, and the sample time `Ts` from `sys_ss`, construct an `initialCondition` object `ic` that you can use with a transfer function model.

```
A = sys_ss.A;
C = sys_ss.C;
Ts = sys_ss.Ts;
ic = initialCondition(A,x0,C,Ts)
```

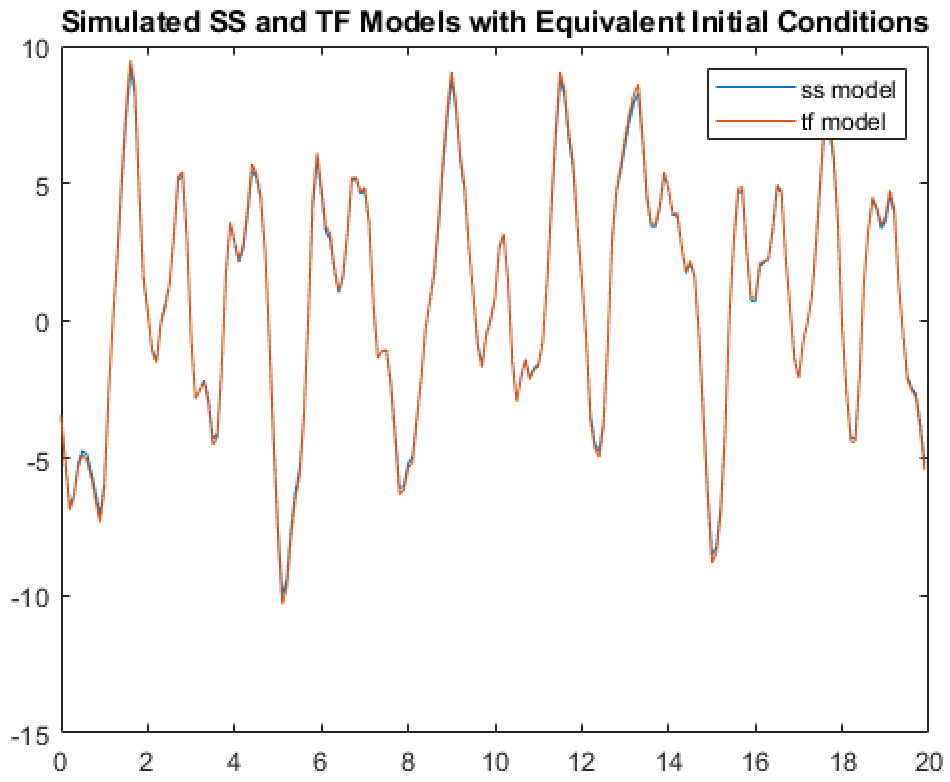
```
ic = 
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [-61.3674 13.4811]
    Ts: 0
```

Estimate a transfer function model and simulate the model using `ic` as the initial condition. Store the response in `xtf`.

```
sys_tf = tfest(z1i,2,1);
opt = simOptions('InitialCondition',ic);
xtf = sim(sys_tf,z1i,opt);
```

Plot the model responses `xss` and `xtf` together.

```
plot(t',[xss.y xtf.y])
legend('ss model','tf model')
title('Simulated SS and TF Models with Equivalent Initial Conditions')
```



The models track each other closely throughout the simulation.

Convert Free-Response Model to `initialCondition` Object

Obtain the initial conditions when estimating a transfer function model. Convert the `initialCondition` into a free-response model, and the free-response model back into an `initialCondition` object.

Load the data and estimate a transfer function model `sys`. Obtain the estimated initial conditions `ic`.

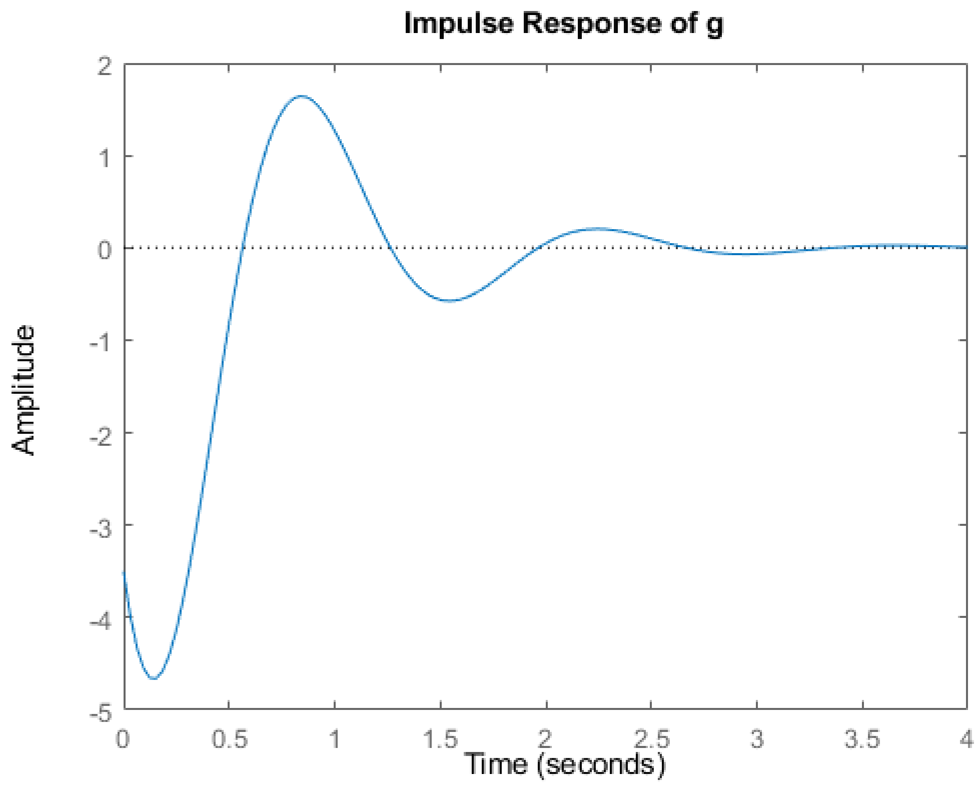
```
load iddata1ic.mat z1i
[sys,ic] = tfest(z1i,2,1);
```

Convert `ic` into the `idtf` free-response model `g`.

```
g = idtf(ic);
```

Plot the impulse response of `g`.

```
impulse(g)
title('Impulse Response of g')
```

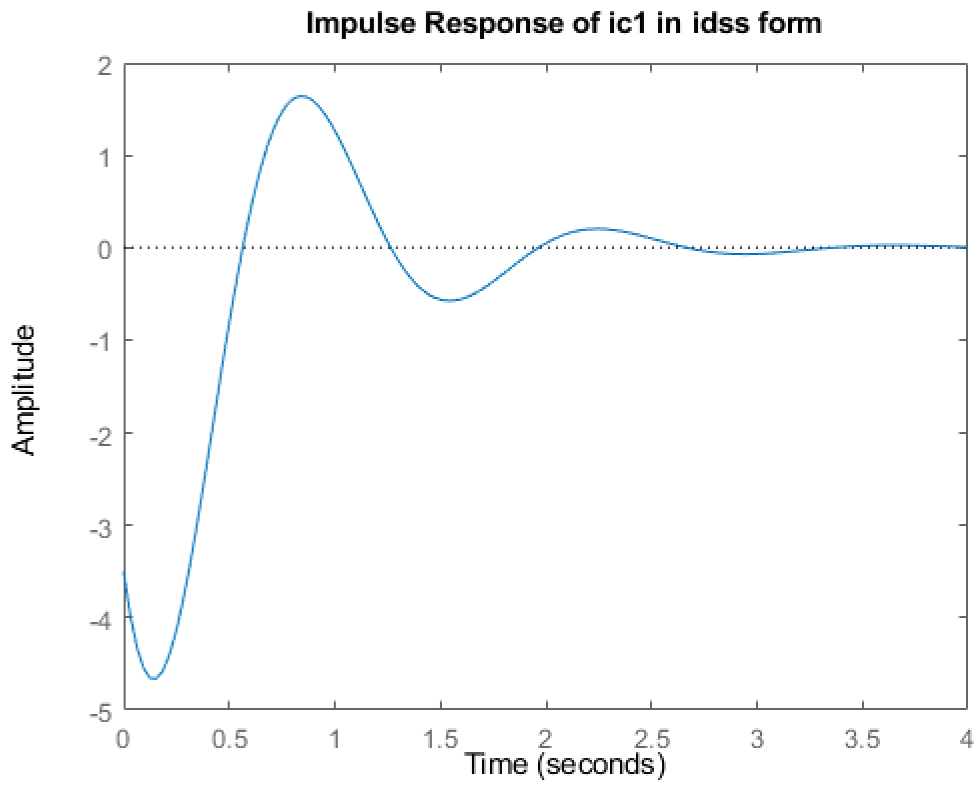


Convert `g` back into the `initialCondition` object `ic1`.

```
ic1 = initialCondition(g);
```

Plot the impulse response of `ic1` by converting `ic1` into an `idss` model.

```
impulse(idss(ic1))
title('Impulse Response of ic1 in idss form')
```



The impulse responses appear identical.

Compare `ic` and `ic1`.

```
ic.A
```

```
ans = 2×2

   -2.9841   -5.5848
    4.0000         0
```

```
ic1.A
```

```
ans = 2×2

   -2.9841   -5.5848
    4.0000         0
```

```
ic.X0
```

```
ans = 2×1

   -0.9019
   -0.6161
```

```
ic1.X0
```

```
ans = 2×1

     4
     0
```

```
ic.C
```

```
ans = 1×2

    0.2957    5.2441
```

```
ic1.C
```

```
ans = 1×2

   -0.8745   -1.7215
```

The `A` matrices of `ic` and `ic1` are identical. The `C` matrix and the `X0` vector are different. There are infinitely many state-space representations possible for a given linear model. The two objects are equivalent, as illustrated by the impulse responses.

See Also

`ssest` | `tfest` | `polyest` | `compare` | `predictOptions` | `predict` | `simOptions` | `sim` | `impulse`

Topics

"Apply Initial Conditions when Simulating Identified Linear Models"
"Estimate Initial Conditions for Simulating Identified Models"

Introduced in R2020b

initialize

Initialize the state of the particle filter

Syntax

```
initialize(pf,numParticles,mean,covariance)
initialize(pf,numParticles,stateBounds)
initialize( ___ ,Name,Value)
```

Description

`initialize(pf,numParticles,mean,covariance)` initializes a particle filter object with a specified number of particles. The initial states of the particles in the state space are determined by sampling from the multivariate normal distribution with the specified `mean` and `covariance`. The number of state variables (`NumStateVariables`) is retrieved automatically based on the length of the `mean` vector.

`initialize(pf,numParticles,stateBounds)` determines the initial location of `numParticles` particles by sampling from the multivariate uniform distribution with the given `stateBounds`.

`initialize( ___ ,Name,Value)` initializes the particles with additional options specified by one or more `Name,Value` pair arguments.

Examples

Initialize Particle Filter Object for Online State Estimation

To create a particle filter object for estimating the states of your system, create appropriate state transition function and measurement function for the system.

In this example, the functions `vdpParticleFilterStateFcn and vdpMeasurementLikelihoodFcn` describe a discrete-approximation to van der Pol oscillator with nonlinearity parameter, mu, equal to 1.

Create the particle filter object. Use function handles to provide the state transition and measurement likelihood functions to the object.

```
myPF = particleFilter(@vdpParticleFilterStateFcn,@vdpMeasurementLikelihoodFcn);
```

Initialize the particle filter at state `[2; 0]` with unit covariance, and use `1000` particles.

```
initialize(myPF, 1000, [2;0], eye(2));
myPF
```

```
myPF = 
  particleFilter with properties:

           NumStateVariables: 2
                NumParticles: 1000
          StateTransitionFcn: @vdpParticleFilterStateFcn
```

```
    MeasurementLikelihoodFcn: @vdpMeasurementLikelihoodFcn
     IsStateVariableCircular: [0 0]
            ResamplingPolicy: [1x1 particleResamplingPolicy]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'column'
                   Particles: [2x1000 double]
                     Weights: [1.0000e-03 1.0000e-03 1.0000e-03 ... ]
                       State: 'Use the getStateEstimate function to see the value.'
             StateCovariance: 'Use the getStateEstimate function to see the value.'
```

To estimate the states and state estimation error covariance from the constructed object, use the `predict` and `correct` commands.

Input Arguments

`pf` — Particle filter
`particleFilter` object

Particle filter, specified as a object. See `particleFilter` for more information.

`numParticles` — Number of particles used in the filter
scalar

Number of particles used in the filter, specified as a scalar.

Unless performance is an issue, do not use fewer than 1000 particles. A higher number of particles can improve the estimate but sacrifices performance speed, because the algorithm has to process more particles. Tuning the number of particles is the best way to improve the tracking of your particle filter.

`mean` — Mean of particle distribution
vector

Mean of particle distribution, specified as a vector. The `NumStateVariables` property of `pf` is set based on the length of this vector.

`covariance` — Covariance of particle distribution
N-by-*N* matrix

Covariance of particle distribution, specified as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`.

`stateBounds` — Bounds of state variables
n-by-2 matrix

Bounds of state variables, specified as an *n*-by-2 matrix. The `NumStateVariables` property of `pf` is set based on the value of *n*. Each row corresponds to the lower and upper limit of the corresponding state variable. The number of state variables (`NumStateVariables`) is retrieved automatically based on the number of rows of the `stateBounds` array.

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: `...'StateOrientation','row'`

`CircularVariables` — Circular variables
logical vector

Circular variables, the comma-separated pair consisting of `CircularVariables` and specified as a logical vector. Each state variable that uses circular or angular coordinates is indicated with a 1. The length of the vector is equal to the `NumStateVariables` property of `particleFilter`.

`StateOrientation` — Orientation of states
'column' (default) | 'row'

Orientation of states, specified as the comma-separated pair consisting of `StateOrientation` as one of these values: 'column' or 'row'. If it is 'column', `State` property and `getStateEstimate` method of the object `pf` returns the states as a column vector, and the `Particles` property has dimensions `NumStateVariables`-by-`NumParticles`. If it is 'row', the states have the row orientation and `Particles` has dimensions `NumParticles`-by-`NumStateVariables`.

See Also
`predict` | `correct` | `particleFilter` | `unscentedKalmanFilter` | `extendedKalmanFilter` | `clone`

Topics
"Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter"
"What Is Online Estimation?"
"Generate Code for Online State Estimation in MATLAB"

Introduced in R2017b

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

`freqresp` | `frd` | `idfrd`

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

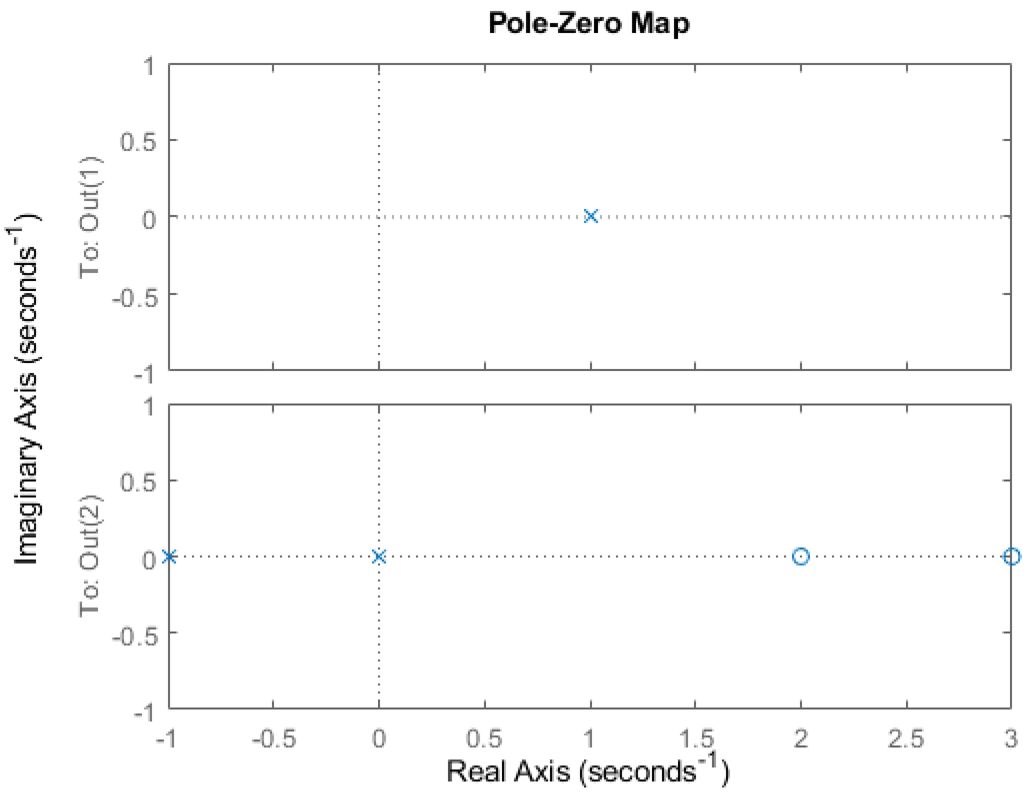
Pole-Zero Map for MIMO System

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)
```

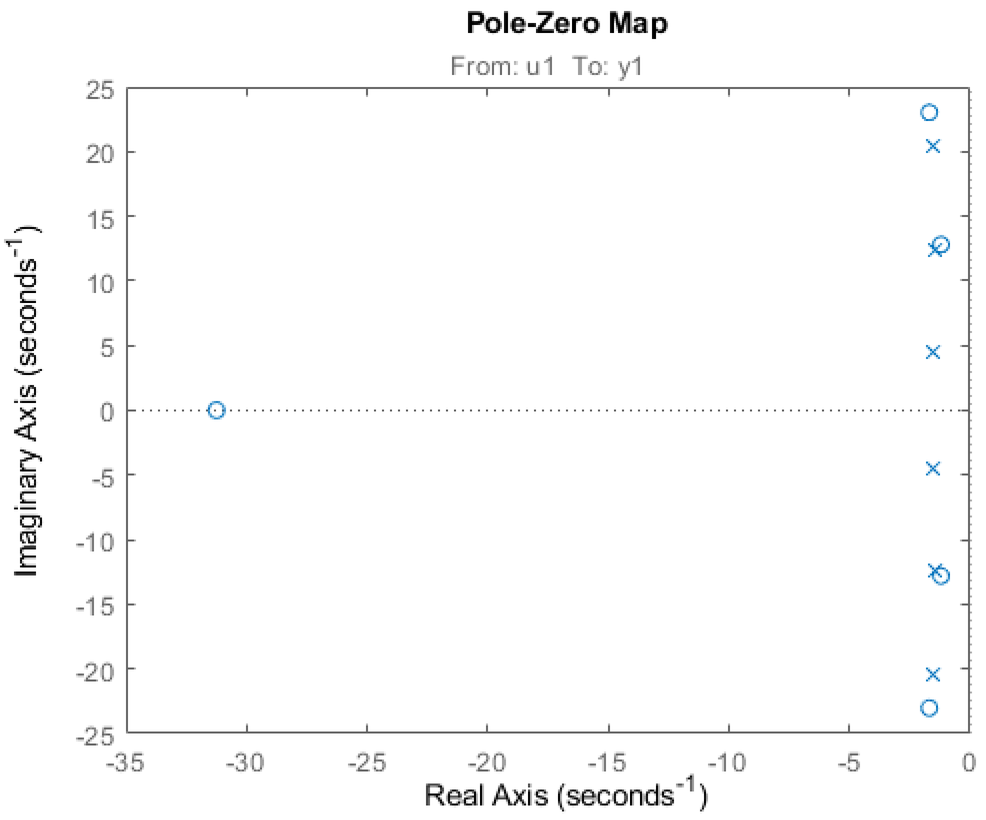


`iopzmap` generates a separate map for each I/O pair in the system.

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.

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 with additional plot customization options

Syntax

```
h = iopzplot(sys)
h = iopzplot(sys1,sys2,...,sysN)
h = iopzplot(sys1,LineSpec1,...,sysN,LineSpecN)
h = iopzplot(ax,...)
h = iopzplot(...,plotoptions)
```

Description

`iopzplot` lets you plot pole-zero maps for input/output pairs with a broader range of plot customization options than `iopzmap`. You can use `iopzplot` to obtain the plot handle and use it to customize the plot, such as modify the axes labels, limits and units. You can also use `iopzplot` to draw a pole-zero plot on an existing set of axes represented by an axes handle. To customize an existing plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox). To create pole-zero maps with default options or to extract pole-zero data, use `iopzmap`.

`h = iopzplot(sys)` plots the poles and zeros of each input/output pair of the dynamic system model `sys` and returns the plot handle `h` to the plot. `x` and `o` indicates poles and zeros respectively.

`h = iopzplot(sys1,sys2,...,sysN)` displays the poles and transmission zeros of multiple models on a single plot. You can specify distinct colors for each model individually.

`h = iopzplot(sys1,LineSpec1,...,sysN,LineSpecN)` sets the line style, marker type, and color for the plot of each system. All systems must have the same number of inputs and outputs to use this syntax.

`h = iopzplot(ax,...)` plots into the axes specified by `ax` instead of the current axis `gca`.

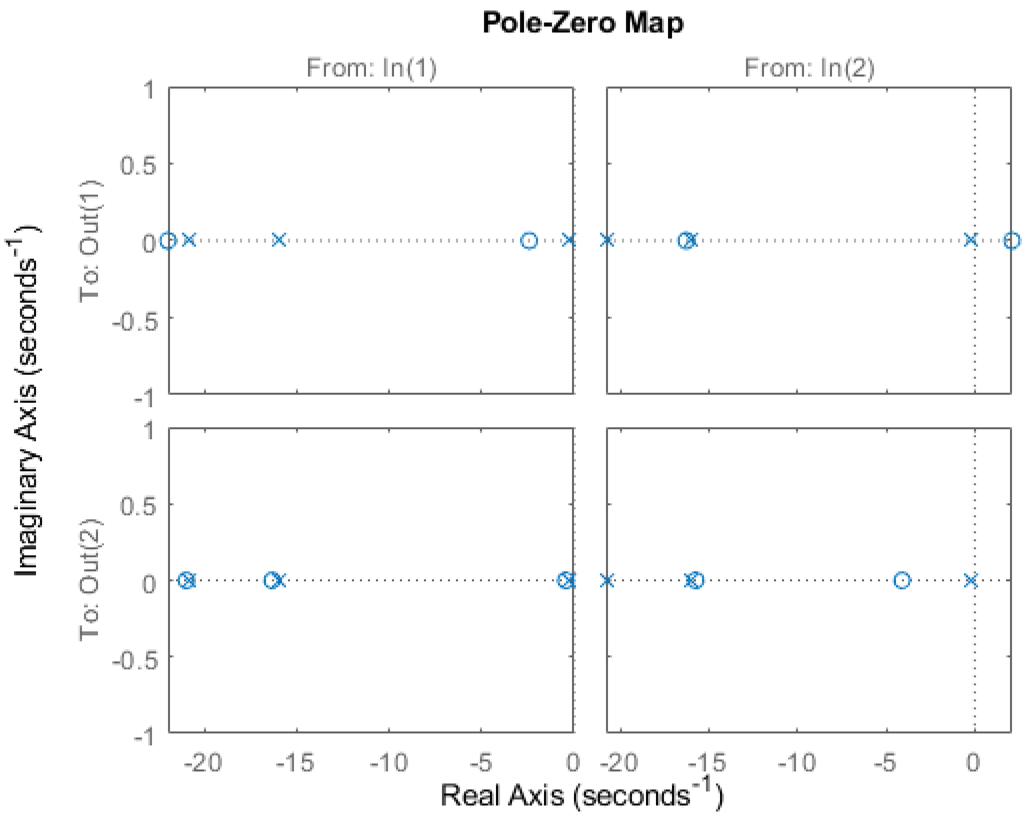
`h = iopzplot(...,plotoptions)` plots the poles and transmission zeros with the options specified in `plotoptions`. For more information on the ways to change properties of your plots, see “Ways to Customize Plots” (Control System Toolbox).

Examples

Change I/O Grouping on Pole/Zero Map

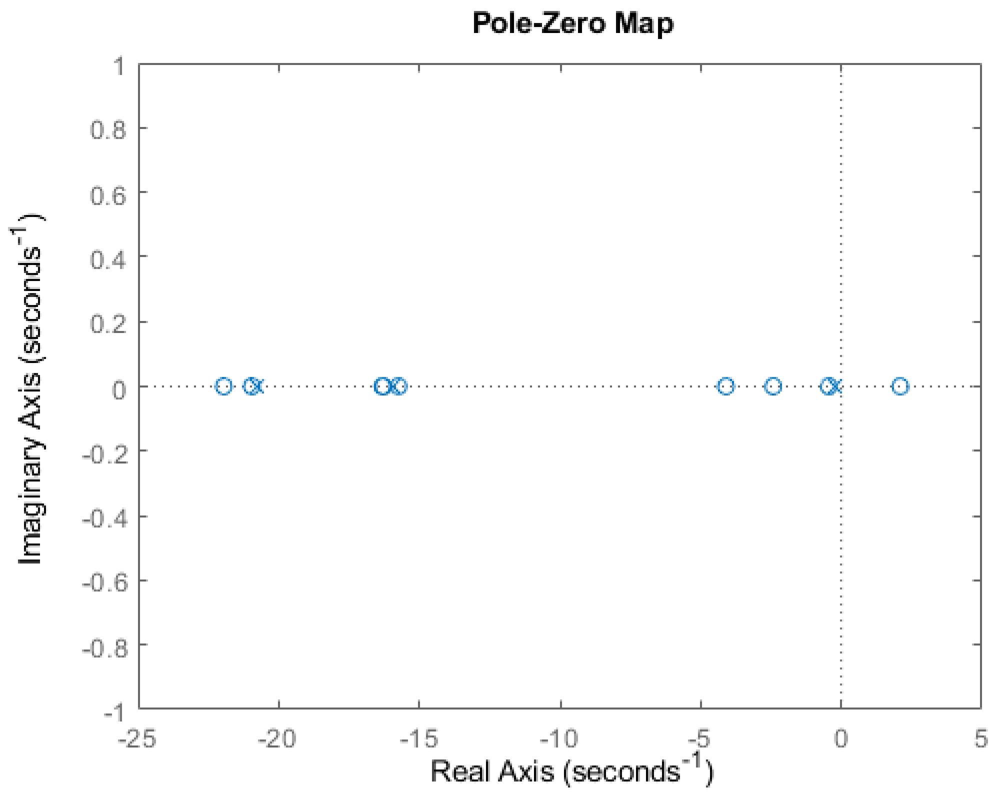
Create a pole/zero map of a two-input, two-output dynamic system.

```
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.

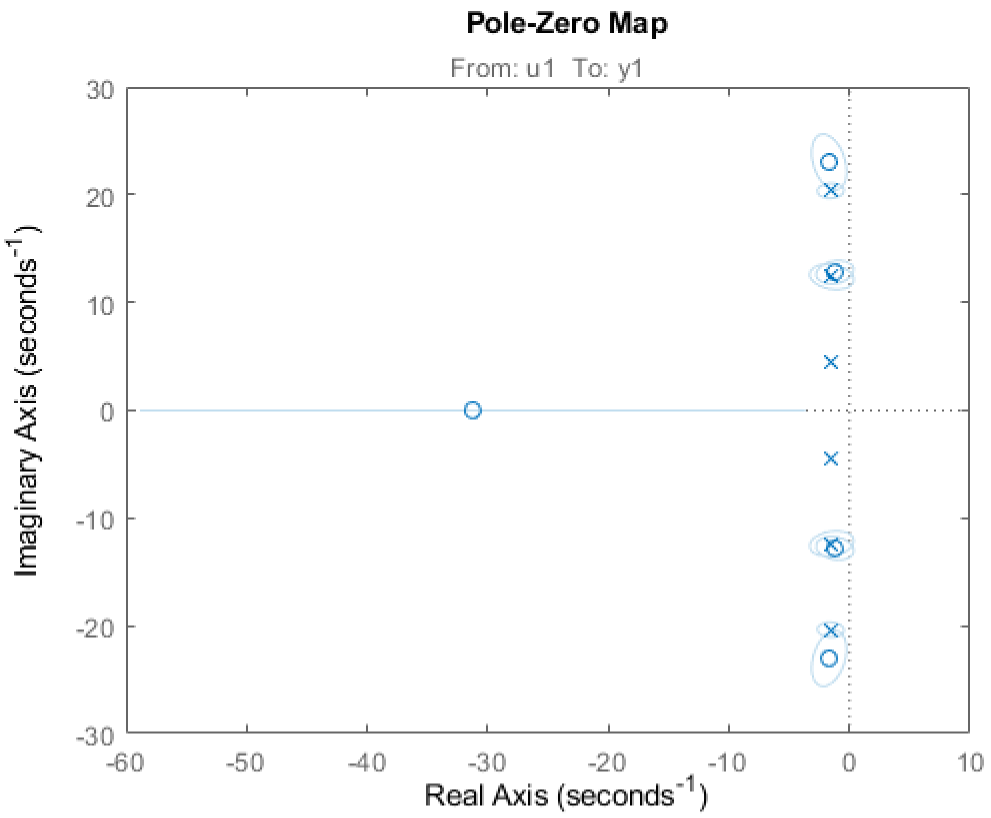
```
setoptions(h,'IOGrouping','all')
```



Use Pole-Zero Map to Examine Identified Model

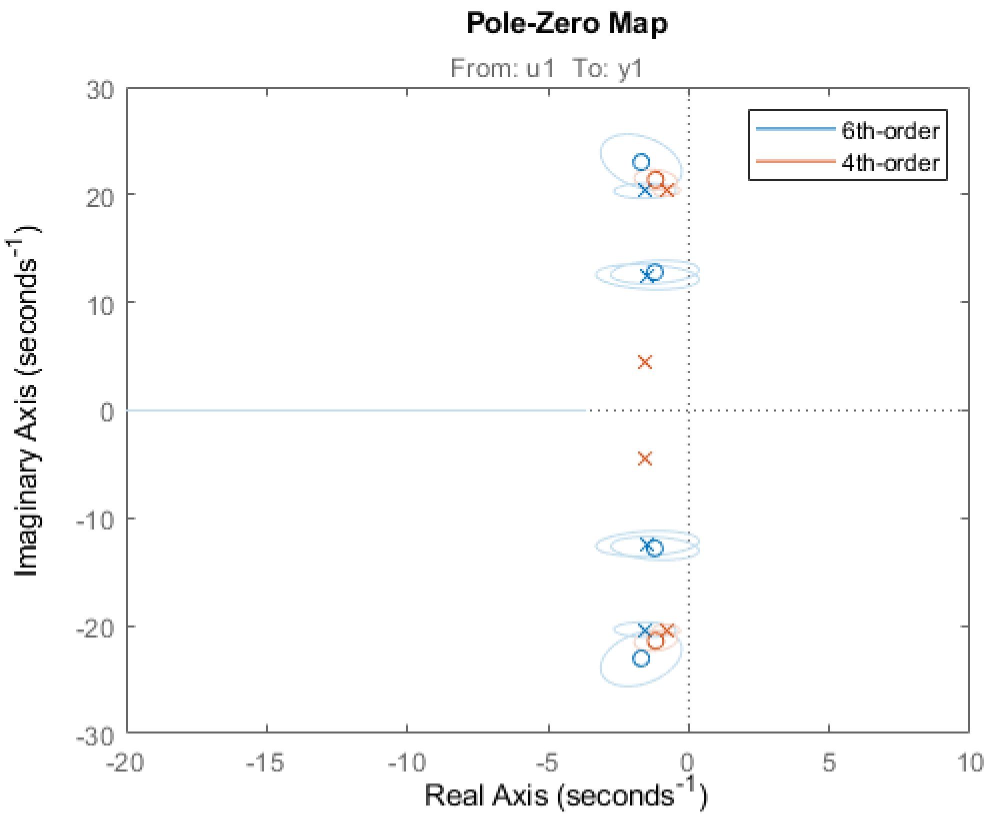
View the poles and zeros of a sixth-order state-space model estimated from input-output data. Use the plot handle to display the confidence intervals of the identified model's pole and zero locations.

```
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.

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model, or an array of SISO or MIMO dynamic system models. Dynamic systems that you can use include continuous-time or discrete-time numeric LTI models such as `tf`, `zpk`, or `ss` models.

If `sys` is a generalized state-space model `genss` or an uncertain state-space model `uss`, `pzplot` returns the poles and transmission of the current or nominal value of `sys`. If `sys` is an array of models, `pzplot` plots the poles and zeros of each model in the array on the same diagram.

LineSpec — Line style, marker, and color

character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'_'`	Horizontal line
`'|'`	Vertical line
`'s'`	Square
`'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'p'`	Pentagram
`'h'`	Hexagram

Color	Description
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

ax — Axes handle
axes object

Axes handle, specified as an axes object. If you do not specify the axes object, then `pzplot` uses the current axes `gca` to plot the poles and zeros of the system.

`plotoptions` — Pole-zero plot options
options object

Pole-zero plot options, specified as an options object. See `pzoptions` for a list of available plot options.

Output Arguments

`h` — Pole-zero plot options handle
scalar

Pole-zero plot options handle, returned as a scalar. Use `h` to query and modify properties of your pole-zero plot. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands.

Tips

- Use `sgrid` or `zgrid` to plot lines of constant damping ratio and natural frequency in the *s*- or *z*-plane.

See Also

`getoptions` | `iopzmap` | `setoptions` | `showConfidence`

Topics
"Ways to Customize Plots" (Control System Toolbox)

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

Determine Whether Dynamic Model Is Empty

Create a continuous-time state-space model with 1 input and no outputs. In this example, specify the `A` and `B` matrices as 1 and 2, respectively.

```
sys1 = ss(1,2,[],[]);
```

Determine whether `sys1` is empty.

```
isempty(sys1)
```

```
ans = logical
   1
```

The `isempty` command returns `1` because the system does not have any outputs.

Similarly, `isempty` returns `1` for an empty transfer function.

```
isempty(tf)
```

```
ans = logical
   1
```

Now create a state-space model with 1 input and 1 output. In this example, specify the `A`, `B`, `C`, and `D` matrices as 1, 2, 3, and 4, respectively.

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

Determine whether `sys2` is empty.

```
isempty(sys2)
```

```
ans = logical
   0
```

The command returns `0` because the system has inputs and outputs.

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.

See Also

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

Topics
"What Is Online Estimation?"

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);
```

The large detection ratio indicates that the test was robust and a significant nonlinearity was detected. Additionally, the estimated discrepancy 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);
```

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','signaltransmissiondata'))
```

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));
```

Detect nonlinearities in the data, and search for the best nonlinear regressor combination.

```
% isnlarx(z,orders,'NonlinearRegressors','search');
```

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 | numeric matrix

Time-domain estimation data, specified as an `iddata` object or a numeric matrix.

- If `data` is an `iddata` object, then `data` can have one or more output channels and zero or more input channels.
- If `data` is a numeric matrix, then the number of columns of data must match the sum of the number of inputs (n_u) and the number of outputs (n_y.

`data` must be uniformly sampled and cannot contain missing (`NaN`) samples.

`orders` — ARX model orders
`nlarx` orders `[na nb nk]`

ARX model order matrix `[na nb nk]`. `na` denotes the number of delayed outputs, `nb` denotes the number of delayed inputs, and `nk` denotes the minimum input delay. The minimum output delay is fixed to `1`. For more information on how to construct the `orders` matrix, see `arx`.

When you specify `orders`, the software converts the order information into linear regressor form in the `idnlarx` `Regressors` property. For an example, see “Create Nonlinear ARX Model Using ARX Model Orders” on page 1-614

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 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.

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 $E(t)$.

See Also

`nlarx` | `idnlarx` | `getreg` | `treepartition`

Topics
“Structure of Nonlinear ARX Models”

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

Create a SISO continuous-time transfer function, $H_1 = s$

```
H1 = tf([1 0],1);
```

Check whether `H1` is proper.

```
B1 = isproper(H1)
```

```
B1 = logical
   0
```

SISO transfer functions are proper if the degree of their numerator is less than or equal to the degree of their denominator That is, if the transfer function has at least as many poles as zeroes. Since `H1` has one zero and no poles, the `isproper` command returns `0`.

Now create a transfer function with one pole and one zero, $H_2 = s/(s + 1)$

```
H2 = tf([1 0],[1 1]);
```

Check whether H2 is proper.

```
B2 = isproper(H2)
```

```
B2 = logical
   1
```

Since H2 has equal number of poles and zeros, `isproper` returns 1.

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.
```

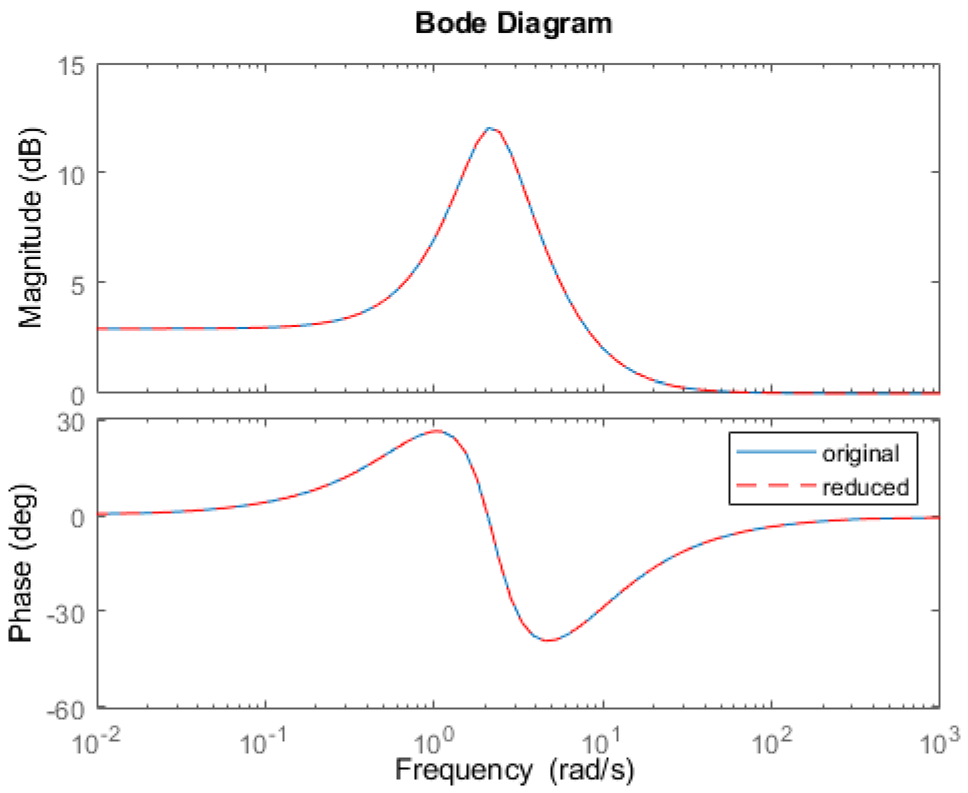
H is proper and reducible. `isproper` returns the reduced model.

```
[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')
```



References

[1] Varga, Andràs. "Computation of irreducible generalized state-space realizations." *Kybernetika* 26.2 (1990): 89-106.

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 if dynamic system model 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 (Control System Toolbox) `sys` has stable dynamics, and a logical value of `0` (`false`) otherwise. If `sys` is a model array, then the function returns `1` only if all the models in `sys` are stable.

`isstable` returns a logical value of `1` (`true`) for stability of a dynamic system if:

- In continuous-time systems, all the poles lie in the open left half of the complex plane.
- In discrete-time systems, all the poles lie inside the open unit disk.

`isstable` is supported only for analytical models with a finite number of poles.

`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.

Examples

Determine Stability of Discrete-Time Transfer Function Model

Determine the stability of this discrete-time SISO transfer function model with a sample time of `0.1` seconds.

$$\mathrm{sys}(z) = \frac{2z}{4z^3 + 3z - 1}$$

Create the discrete-time transfer function model.

```
sys = tf([2,0],[4,0,3,-1],0.1);
```

Examine the poles of the system.

```
P = abs(pole(sys))
```

```
P = 3×1

    0.9159
    0.9159
    0.2980
```

All the poles of the transfer function model have a magnitude less than 1, so all the poles lie within the open unit disk and the system is stable.

Confirm the stability of the model using isstable.

```
B = isstable(sys)
```

```
B = logical
   1
```

The system sys is stable.

Determine Stability of Continuous-Time Zero-Pole-Gain Model

Determine the stability of this continuous-time zero-pole-gain model.

$$\mathrm{sys}(s) = \frac{2}{(s+2+3j)(s+2-3j)(s-0.5)}$$

Create the model as a zpk model object by specifying the zeros, poles, and gain.

```
sys = zpk([],[-2-3*j,-2+3*j,0.5],2);
```

Because one pole of the model lies in the right half of the complex plane, the system is unstable.

Confirm the instability of the model using isstable.

```
B = isstable(sys)
```

```
B = logical
   0
```

The system sys is unstable.

Determine Stability of Models in Model Array

Determine the stability of an array of SISO transfer function models with poles varying from -2 to 2.

$$\left[\frac{1}{s+2}, \frac{1}{s+1}, \frac{1}{s}, \frac{1}{s-1}, \frac{1}{s-2}\right]$$

To create the array, first initialize an array of dimension [length(a),1] with zero-valued SISO transfer functions.

```
a = [-2:2];
sys = tf(zeros(1,1,length(a)));
```

Populate the array with transfer functions of the form 1/(s-a).

```
for j = 1:length(a)
    sys(1,1,j) = tf(1,[1 -a(j)]);
end
```

`isstable` can tell you whether all the models in model array are stable or each individual model is stable.

Examine the stability of the model array.

```
B_all = isstable(sys)
```

```
B_all = logical
   0
```

By default, `isstable` returns a single logical 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 the stability of each model in the array by using `'elem'` flag.

```
B_elem = isstable(sys,'elem')
```

```
B_elem = 5x1 logical array

   1
   1
   0
   0
   0
```

The function returns an array of logical values that indicate the stability of the corresponding entry in the model array. For example, `B_elem(2)` is `1`, which indicates that the second model in the array, `sys(1,1,2)` is stable. This is because `sys(1,1,2)` has a pole at `-1`.

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or an array of SISO or MIMO dynamic system models. Dynamic systems that you can use include continuous-time or discrete-time numeric LTI models such as `tf`, `zpk`, or `ss` models.

If `sys` is a generalized state-space model `genss` or an uncertain state-space model `uss`, `isstable` checks the stability of the current or nominal value of `sys`.

If `sys` is an array of models, `isstable` checks the stability of every model in the array.

- If you use `B = isstable(sys)`, the output is `1` (`true`) only if all the models in the array are stable.
- If you use `B = isstable(sys,'elem')`, the output is a logical array, the entries of which indicate the stability of the corresponding entry in the model array.

For more information on model arrays, see “Model Arrays” (Control System Toolbox).

Output Arguments

`B` — True or false result
`1` | `0` | logical array

True or false result, returned as `1` for a stable model or `0` for an unstable model.

The `'elem'` flag causes `isstable` to return an array of logical values with same dimensions as the model array. The values in the array indicate the stability of the corresponding entry in the model array.

See Also
`isproper` | `issiso` | `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` | `step` | `spectrum`

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-Hall 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);
```

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

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

See Also

`arxstruc` | `iv4` | `selstruc` | `struc`

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:

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.
`InitialCondition`	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'`.

Report Field	Description
`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: **Field** — **Description** `FitPercent` — Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE). `LossFcn` — Value of the loss function when the estimation completes. `MSE` — Mean squared error (MSE) measure of how well the response of the model fits the estimation data. `FPE` — Final prediction error for the model. `AIC` — Raw Akaike Information Criteria (AIC) measure of model quality. `AICc` — Small-sample-size corrected AIC. `nAIC` — Normalized AIC. `BIC` — Bayesian Information Criteria (BIC).
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `arxOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
DataUsed	Attributes of the data used for estimation, returned as a structure with the following fields.

Field	Description
Name	Name of the data set.
Type	Data type.
Length	Number of data samples.
Ts	Sample time.
InterSample	Input intersample behavior, returned as one of the following values: • 'zoh' — 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.
InputOffset	Offset removed from time-domain input data during estimation. For nonlinear models, it is [].
OutputOffset	Offset removed from time-domain output data during estimation. For nonlinear models, it is [].

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

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-Hall 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 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:

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.
`InitialCondition`	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. This field is especially useful to view how the initial conditions were handled when the `InitialCondition` option in the estimation option set is `'auto'`.

Report Field	Description
`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: **Field** — **Description** `FitPercent` — Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE). `LossFcn` — Value of the loss function when the estimation completes. `MSE` — Mean squared error (MSE) measure of how well the response of the model fits the estimation data. `FPE` — Final prediction error for the model. `AIC` — Raw Akaike Information Criteria (AIC) measure of model quality. `AICc` — Small-sample-size corrected AIC. `nAIC` — Normalized AIC. `BIC` — Bayesian Information Criteria (BIC).
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `iv4Options` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields.

Field	Description
`Name`	Name of the data set.
`Type`	Data type.
`Length`	Number of data samples.
`Ts`	Sample time.
`InterSample`	Input intersample behavior, returned as one of the following values: • `'zoh'` — 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.
`InputOffset`	Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`.
`OutputOffset`	Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.

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]);
```

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-Hall PTR, 1999.

See Also

`iv4Options` | `arx` | `armax` | `bj` | `idpoly` | `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 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

EstimateCovariance — Control whether to generate parameter covariance data
true (default) | false

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

If EstimateCovariance 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

See Also

iv4

Topics
“Loss Function and Model Quality Metrics”

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`.

See Also

`idnlarx` | `idnlhw` | `idnlarx/findop` | `idnlhw/findop` | `idnlhw/linearize` | `idnlarx/linearize`

Topics

"Linear Approximation of Nonlinear Black-Box Models"

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 1-623.

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.

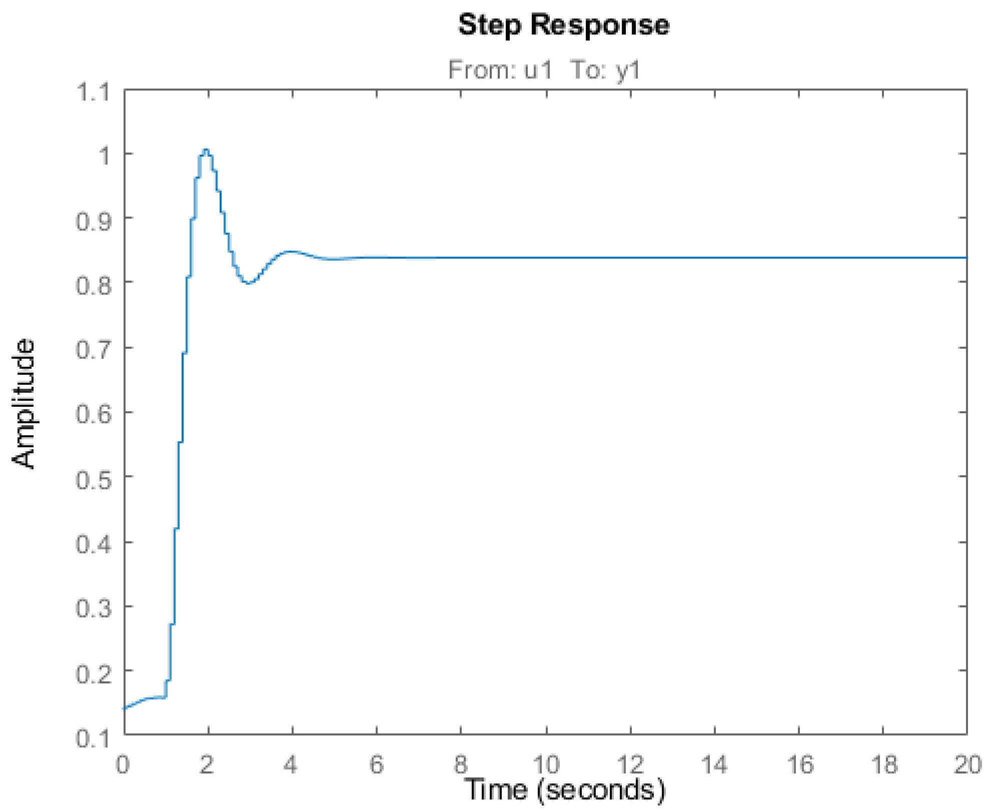
```
load iddata2
```

Estimate nonlinear ARX model from sample data.

```
nlsys = nlarx(z2,[4 3 10],idTreePartition,'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.

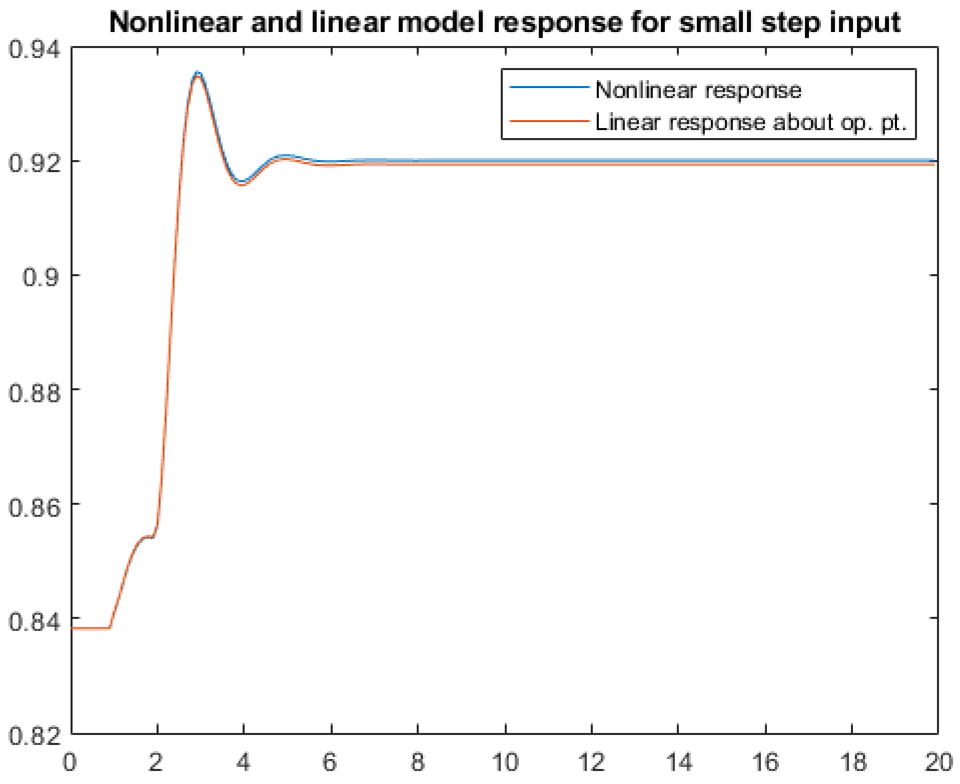
```
u_nl = 1 + delta_u;
y_nl = sim(nlsys,u_nl,x);
```

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')
```



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:

$$\Delta X(t+1) = (A + B_1 f_X)\Delta X(t) + (B_1 f_u + B_2)\Delta u(t)$$
$$\Delta y(t) = f_X \Delta X(t) + f_u \Delta u(t)$$

where

- $\Delta X(t) = X(t) - X^*(t)$
- $\Delta u(t) = u(t) - u^*(t)$
- $\Delta y(t) = y(t) - y^*(t)$
- $B\widetilde{U} = [B_1, B_2]\begin{bmatrix} Y \\ U \end{bmatrix} = B_1 Y + B_2 U$
- $f_X = \frac{\partial}{\partial X} f(X, U)\Big|_{X^*, U^*}$
- $f_U = \frac{\partial}{\partial U} f(X, U)\Big|_{X^*, U^*}$

Note For linear approximations over larger input ranges, use `linapp`.

See Also

`idnlarx/findop` | `getDelayInfo` | `idnlarx` | `linapp`

Topics

“Linear Approximation of Nonlinear Black-Box Models”

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 1-654.

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.

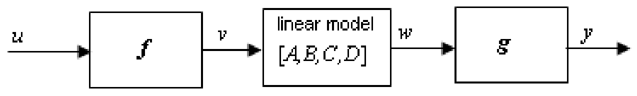
Output Arguments

- `SYS` is an `idss` model.

  When the Control System Toolbox product is installed, `SYS` is an LTI object.

Algorithms

The `idnlhw` model structure represents a nonlinear system using a linear system connected in series with one or two static nonlinear systems. For example, you can use a static nonlinearity to simulate saturation or dead-zone behavior. The following figure shows the nonlinear system as a linear system that is modified by static input and output nonlinearities, where function ***f*** represents the input nonlinearity, ***g*** represents the output nonlinearity, and [*A*,*B*,*C*,*D*] represents a state-space parameterization of the linear model.



The following equations govern the dynamics of an `idnlhw` model:

$v(t) = f(u(t))$

$X(t+1) = AX(t)+Bv(t)$

$w(t) = CX(t)+Dv(t)$

$y(t) = g(w(t))$

where

- u is the input signal
- v and w are intermediate signals (outputs of the input nonlinearity and linear model respectively)
- y is the model output

The linear approximation of the Hammerstein-Wiener model around an operating point (X^*, u^*) is as follows:

$$\Delta X(t+1) = A\Delta X(t) + Bf_u\Delta u(t)$$
$$\Delta y(t) \approx g_w C\Delta X(t) + g_w D f_u \Delta u(t)$$

where

- $\Delta X(t) = X(t) - X^*(t)$
- $\Delta u(t) = u(t) - u^*(t)$
- $\Delta y(t) = y(t) - y^*(t)$
- $f_u = \frac{\partial}{\partial u} f(u)\Big|_{u = u^*}$
- $g_w = \frac{\partial}{\partial w} g(w)\Big|_{w = w^*}$

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.

See Also

`idnlhw/findop` | `idnlhw` | `linapp`

Topics

"Linear Approximation of Nonlinear Black-Box Models"

Introduced in R2014b

linearRegressor

Specify linear regressor for nonlinear ARX model

Description

A linear regressor is a lagged output or input variable, such as *y*(*t*-1) or *u*(*t*-2). Here, the *y* term has a lag of 1 sample and the *u* term has a lag of 2 samples. A `linearRegressor` object encapsulates a set of linear regressors. Use `linearRegressor` objects when you create nonlinear ARX models using `idnlarx` or `nlarx`. `linearRegressor` generalizes the concept of orders in ARX models, or in other words, the `[na nb nk]` matrix, to allow absolute values and noncontiguous lags. Using `linearRegressor` objects also lets you combine linear regressors with polynomial and custom regressors in a single regressor set.

Creation

Syntax

```
lReg = linearRegressor(Variables,Lags)
lreg = linearRegressor(Variables,Lags,useAbsolute)
```

Description

`lReg = linearRegressor(Variables,Lags)` creates a `linearRegressor` object that contains output and input names in `Variables` and the corresponding lags in `Lags`.

`lreg = linearRegressor(Variables,Lags,useAbsolute)` specifies in `UseAbsolute` whether to use the absolute values of the variables to create the regressors.

Properties

`Variables` — Output and input variable names
cell array of strings | `iddata` object properties

Output and input variable names, specified as a cell array of strings or a cell array that references the `OutputName` and `InputName` properties of an `iddata` object. Each entry must be a string with no special characters other than white space. For an example of using this property, see "Estimate Nonlinear ARX Model Using Linear Regressor Set" on page 1-907.

Example: `{'y1','u1'}`

Example: `[z.OutputName; z.InputName]'`

Lags — Lags in each variable
cell array of non-negative integers

Lags in each variable, specified as a 1-by-n_v cell array of non-negative integer row vectors, where n_v is the total number of regressor variables. Each row vector contains n_r integers that specify the n_r

regressor lags for the corresponding variable. For instance, suppose that you want the following regressors:

- Output variable y_1: $y_1(t–1)$ and $y_1(t–2)$
- Input variable u_1: $u_1(t–3)$

To obtain these lags, set `Lags` to `{[1 2],3}`.

If a lag corresponds to an output variable of an `idnlarx` model, the minimum lag must be greater than or equal to 1.

For an example of using this property, see "Estimate Nonlinear ARX Model Using Linear Regressor Set" on page 1-907.

Example: `{1 1}`

Example: `{[1 2],[1,3,4]}`

`UseAbsolute` — Absolute value indicator
`false` (default) | logical vector

Absolute value indicator that determines whether to use the absolute value of a regressor variable instead of the signed value, specified as a logical vector with a length equal to the number of variables. For an example of setting this property, see "Use Absolute Value in Linear Regressor Set" on page 1-909.

Example: `[true,false]`

`TimeVariable` — Name of time variable
`'t'` (default) | character array | string

Name of the time variable, specified as a valid MATLAB variable name that is distinct from values in `Variables`.

Example: `'ClockTime'`

Examples

Estimate Nonlinear ARX Model Using Linear Regressor Set

Specify a linear regressor that is equivalent to an ARX model order matrix of `[4 4 1]`.

An order matrix of `[4 4 1]` specifies that both input and output regressor sets contain four regressors with lags ranging from 1 to 4. For example, $u_1(t-2)$ represents the second input regressor.

Specify the output and input names.

```
output_name = 'y1';
input_name = 'u1';
names = {output_name,input_name};
```

Specify the output and input lags.

```
output_lag = [1 2 3 4];
input_lag = [1 2 3 4];
lags = {output_lag,input_lag};
```

Create the linear regressor object.

```
lreg = linearRegressor(names,lags)
```

```
lreg = 
Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1 2 3 4]  [1 2 3 4]}
     UseAbsolute: [0 0]
    TimeVariable: 't'

  Regressors described by this set
```

Load the estimation data and create an iddata object.

```
load twotankdata
z = iddata(y,u,0.2);
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,lreg)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Wavelet network with 11 units
Sample time: 0.2 seconds

Status:
Estimated using NLARX on time domain data "z".
Fit to estimation data: 96.84% (prediction focus)
FPE: 3.482e-05, MSE: 3.431e-05
```

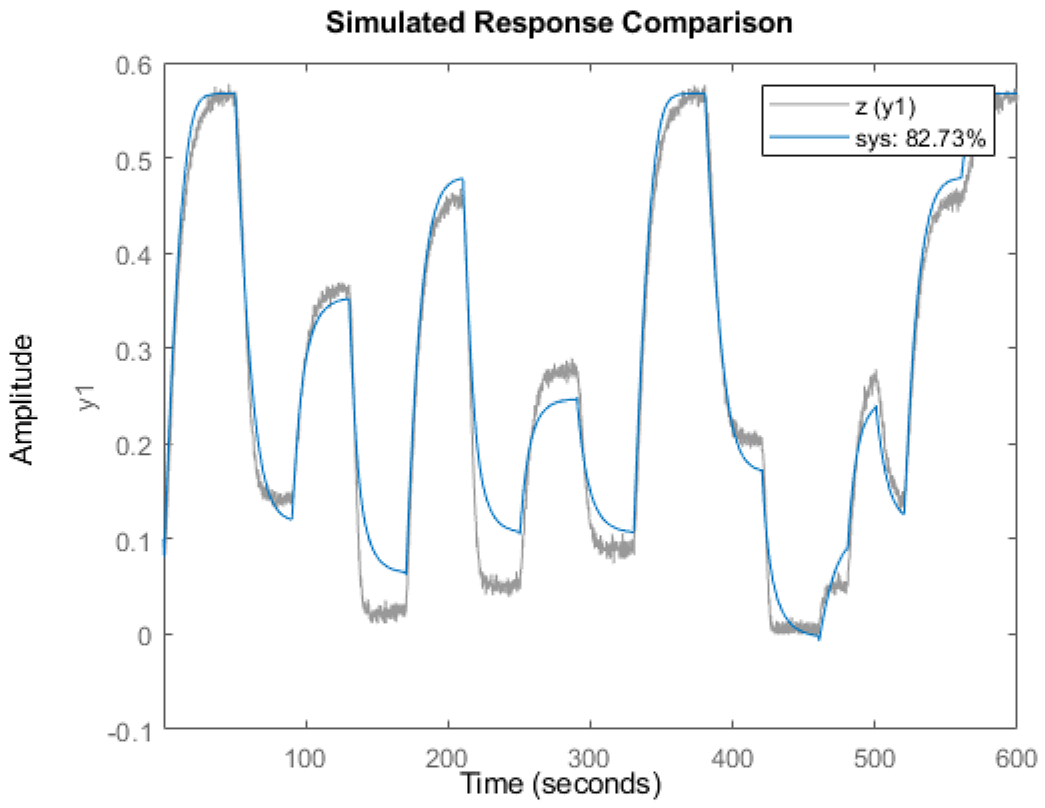
View the regressors

```
getreg(sys)
```

```
ans = 8x1 cell
    {'y1(t-1)'}
    {'y1(t-2)'}
    {'y1(t-3)'}
    {'y1(t-4)'}
    {'u1(t-1)'}
    {'u1(t-2)'}
    {'u1(t-3)'}
    {'u1(t-4)'}
```

Compare the model output to the estimation data.

```
compare(z,sys)
```



Use Absolute Value in Linear Regressor Set

Create a linear regressor set that uses lags of 3, 10, and 100 in variable `y1` and lags of 0 and 4 in variable `u1`.

```
vars = {'y1','u1'};
lags = {[3 10 100],[0,4]};
```

Specify that the `y1` regressor use the absolute value of `y1`.

```
UseAbs = [true,false];
```

Create the linear regressor.

```
reg = linearRegressor(vars,lags,UseAbs)
```

```
reg = 
Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[3 10 100]  [0 4]}
     UseAbsolute: [1 0]
    TimeVariable: 't'

  Regressors described by this set
```

Specify Linear, Polynomial, and Custom Regressors

Load the estimation data z1, which has one input and one output, and obtain the output and input names.

```
load iddata1 z1;
names = [z1.OutputName z1.InputName]
```

```
names = 1x2 cell
    {'y1'}    {'u1'}
```

Specify L as the set of linear regressors that represents $y_1(t-1)$, $u_1(t-2)$, and $u_1(t-5)$.

```
L = linearRegressor(names,{1,[2 5]});
```

Specify P as the polynomial regressor $y_1(t-1)^2$.

```
P = polynomialRegressor(names(1),1,2);
```

Specify C as the custom regressor $y_1(t-2)u_1(t-3)$. Use an anonymous function handle to define this function.

```
C = customRegressor(names,{2 3},@(x,y)x.*y)
```

```
C = 
Custom regressor: y1(t-2).*u1(t-3)
    VariablesToRegressorFcn: @(x,y)x.*y
                  Variables: {'y1'  'u1'}
                       Lags: {[2]  [3]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

Combine the regressors in the column vector R.

```
R = [L;P;C]
```

```
R = 
[3 1] array of linearRegressor, polynomialRegressor, customRegressor objects.
------------------------------------
1. Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1]  [2 5]}
     UseAbsolute: [0 0]
    TimeVariable: 't'

------------------------------------
2. Order 2 regressors in variables y1
               Order: 2
           Variables: {'y1'}
                Lags: {[1]}
         UseAbsolute: 0
    AllowVariableMix: 0
```

```
         AllowLagMix: 0
        TimeVariable: 't'

------------------------------------
3. Custom regressor: y1(t-2).*u1(t-3)
    VariablesToRegressorFcn: @(x,y)x.*y
                  Variables: {'y1'  'u1'}
                       Lags: {[2]  [3]}
                 Vectorized: 1
               TimeVariable: 't'


Regressors described by this set
```

Estimate a nonlinear ARX model with R.

```
sys = nlarx(z1,R)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables y1
  3. Custom regressor: y1(t-2).*u1(t-3)
  List of all regressors

Output function: Wavelet network with 1 units
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 59.73% (prediction focus)
FPE: 3.356, MSE: 3.147
```

View the full regressor set.

```
getreg(sys)
```

```
ans = 5x1 cell
    {'y1(t-1)'         }
    {'u1(t-2)'         }
    {'u1(t-5)'         }
    {'y1(t-1)^2'       }
    {'y1(t-2).*u1(t-3)'}
```

See Also
idnlarx | nlarx | getreg | polynomialRegressor | customRegressor

Introduced in R2021a

lsim

Plot simulated time response of dynamic system to arbitrary inputs; simulated response data

Syntax

```
lsim(sys,u,t)
lsim(sys,u,t,x0)
lsim(sys,u,t,x0,method)
lsim(sys1,sys2,...,sysN,u,t, ___ )
lsim(sys1,LineSpec1,...,sysN,LineSpecN, ___ )

y = lsim(sys,u,t)
y = lsim(sys,u,t,x0)
y = lsim(sys,u,t,x0,method)
[y,tOut,x] = lsim( ___ )

lsim(sys)
```

Description

Response Plots

`lsim(sys,u,t)` plots the simulated time response of the dynamic system model `sys` to the input history (`t`,`u`). The vector `t` specifies the time samples for the simulation. For single-input systems, the input signal `u` is a vector of the same length as `t`. For multi-input systems, `u` is an array with as many rows as there are time samples (`length(t)`) and as many columns as there are inputs to `sys`.

`lsim(sys,u,t,x0)` further specifies a vector `x0` of initial state values, when `sys` is a state-space model.

`lsim(sys,u,t,x0,method)` specifies how `lsim` interpolates the input values between samples, when `sys` is a continuous-time model.

`lsim(sys1,sys2,...,sysN,u,t, ___ )` simulates the responses of several dynamic system models to the same input history and plots these responses on a single figure. All systems must have the same number of inputs and outputs. You can also use the `x0` and `method` input arguments when computing the responses of multiple models.

`lsim(sys1,LineSpec1,...,sysN,LineSpecN, ___ )` specifies a color, line style, and marker for each system in the plot. When you need additional plot customization options, use `lsimplot` instead.

Response Data

`y = lsim(sys,u,t)` returns the system response `y`, sampled at the same times `t` as the input. For single-output systems, `y` is a vector of the same length as `t`. For multi-output systems, `y` is an array having as many rows as there are time samples (`length(t)`) and as many columns as there are outputs in `sys`. This syntax does not generate a plot.

`y = lsim(sys,u,t,x0)` further specifies a vector `x0` of initial state values, when `sys` is a state-space model.

`y = lsim(sys,u,t,x0,method)` specifies how `lsim` interpolates the input values between samples, when `sys` is a continuous-time model.

`[y,tOut,x] = lsim( ___ )` returns the state trajectories `x`, when `sys` is a state-space model. `x` is an array with as many rows as there are time samples and as many columns as there are states in `sys`. This syntax also returns the time samples used for the simulation in `tOut`.

Linear Simulation Tool

`lsim(sys)` opens the Linear Simulation Tool. For more information about using this tool for linear analysis, see Working with the Linear Simulation Tool (Control System Toolbox).

Examples

Simulated Response to Arbitrary Input Signal

Consider the following transfer function.

```
sys = tf(3,[1 2 3])
```

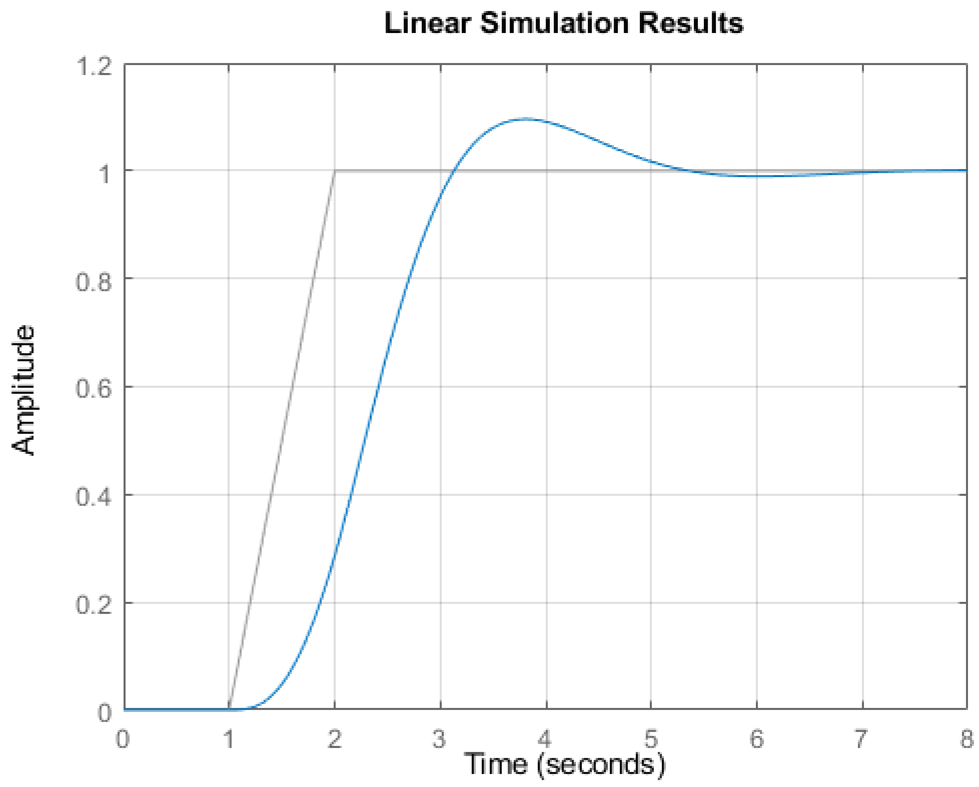
```
sys =

        3
  -------------
  s^2 + 2 s + 3

Continuous-time transfer function.
```

To compute the response of this system to an arbitrary input signal, provide `lsim` with a vector of the times `t` at which you want to compute the response and a vector `u` containing the corresponding signal values. For instance, plot the system response to a ramping step signal that starts at 0 at time `t = 0`, ramps from 0 at `t = 1` to 1 at `t = 2`, and then holds steady at 1. Define `t` and compute the values of `u`.

```
t = 0:0.04:8;  % 201 points
u = max(0,min(t-1,1));
```

Use `lsim` without an output argument to plot the system response to the signal.

```
lsim(sys,u,t)
grid on
```



The plot shows the applied input `(u,t)` in gray and the system response in blue.

Use `lsim` with an output argument to obtain the simulated response data.

```
y = lsim(sys,u,t);
size(y)
```

```
ans = 1×2

   201     1
```

The vector `y` contains the simulated response at the corresponding times in `t`.

Response to Periodic Signal

Use `gensig` (Control System Toolbox) to create periodic input signals such as sine waves and square waves for use with `lsim`. Simulate the response to a square wave of the following SISO state-space model.

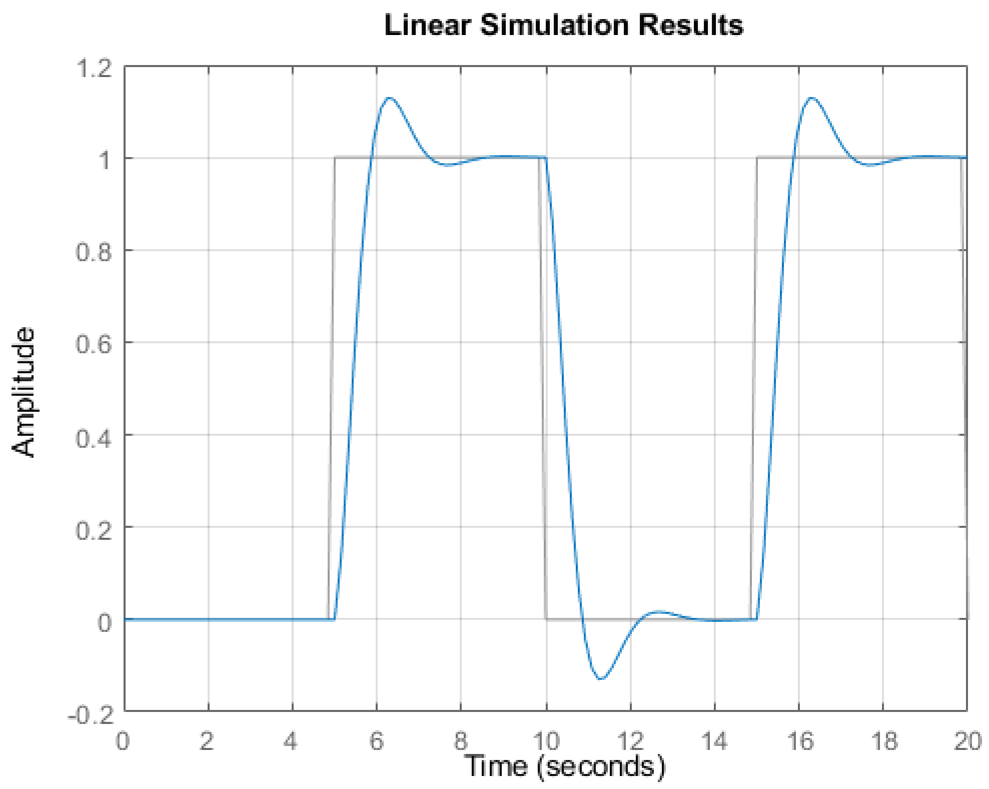
```
A = [-3 -1.5; 5 0];
B = [1; 0];
C = [0.5 1.5];
D = 0;
sys = ss(A,B,C,D);
```

For this example, create a square wave with a period of 10 s and a duration of 20 s.

```
[u,t] = gensig("square",10,20);
```

`gensig` returns the vector `t` of time steps and the vector `u` containing the corresponding values of the input signal. (If you do not specify a sample time for `t`, then `gensig` generates 64 samples per period.) Use these with `lsim` and plot the system response.

```
lsim(sys,u,t)
grid on
```



The plot shows the applied square wave in gray and the system response in blue. Call `lsim` with an output argument to obtain the response values at each point in `t`.

```
[y,~] = lsim(sys,u,t);
```

Response of Discrete-Time System

When you simulate the response of a discrete-time system, the time vector `t` must be of the form `Ti:dT:Tf`, where `dT` is the sample time of the model. Simulate the response of the following discrete-time transfer function to a ramp step input.

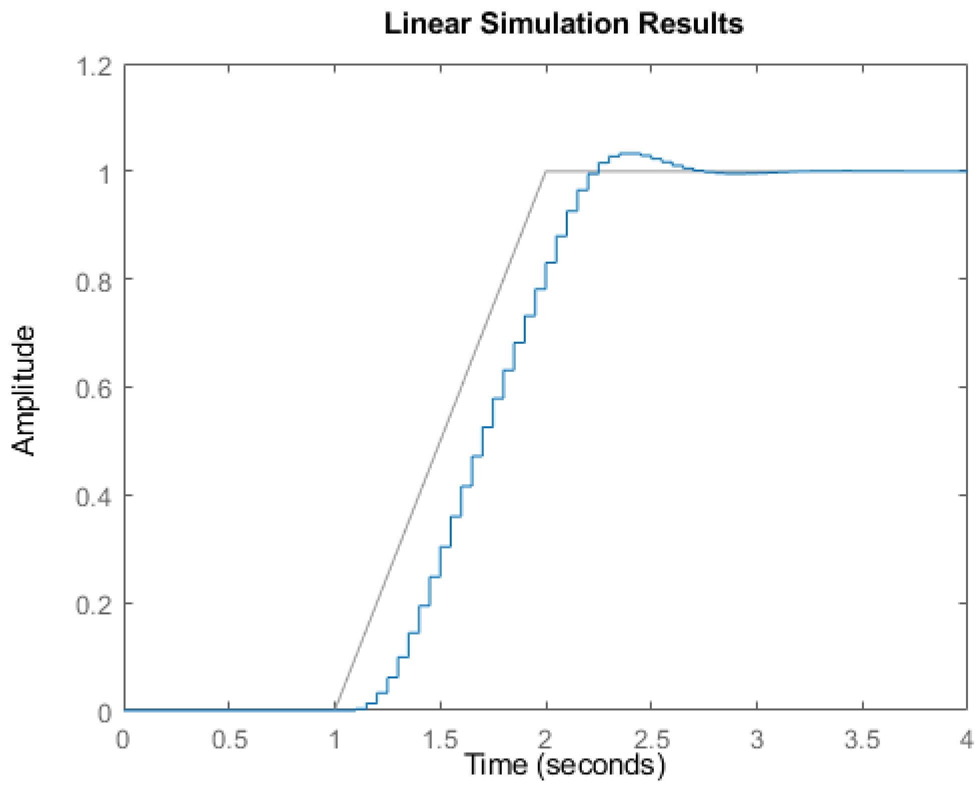
```
sys = tf([0.06 0.05],[1 -1.56 0.67],0.05);
```

This transfer function has a sample time of 0.05 s. Use the same sample time to generate the time vector `t` and a ramped step signal `u`.

```
t = 0:0.05:4;
u = max(0,min(t-1,1));
```

Plot the system response.

```
lsim(sys,u,t)
```

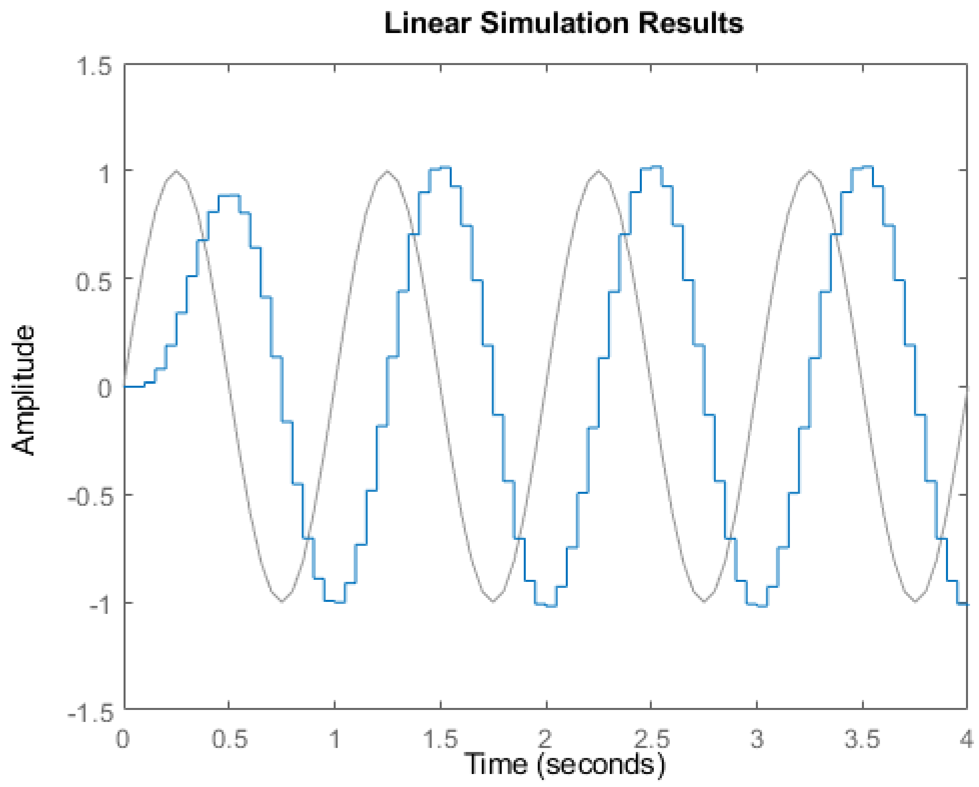


To simulate the response of a discrete-time system to a periodic input signal, use the same sample time with `gensig` to generate the input. For instance, simulate the system response to a sine wave with period of 1 s and a duration of 4 s.

```
[u,t] = gensig("sine",1,4,0.05);
```

Plot the system response.

```
lsim(sys,u,t)
```



Plot Response of Multiple Systems to Same Input

`lsim` allows you to plot the simulated responses of multiple dynamic systems on the same axis. For instance, compare the closed-loop response of a system with a PI controller and a PID controller. Create a transfer function of the system and tune the controllers.

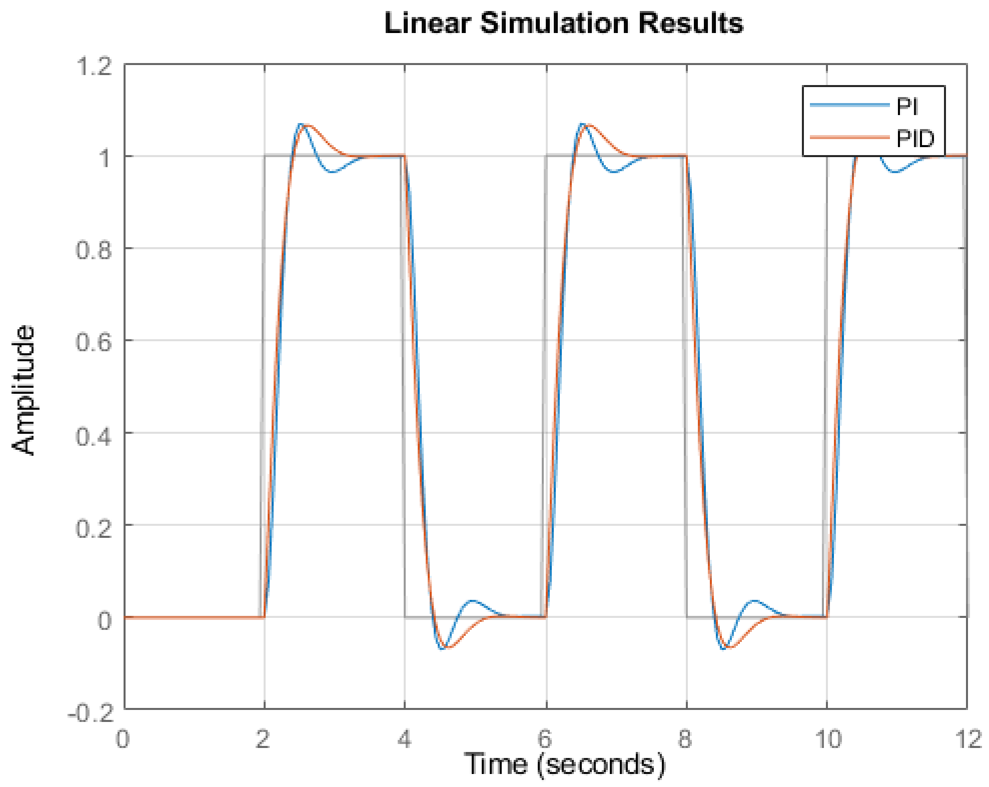
```
H = tf(4,[1 10 25]);
C1 = pidtune(H,'PI');
C2 = pidtune(H,'PID');
```

Form the closed-loop systems.

```
sys1 = feedback(H*C1,1);
sys2 = feedback(H*C2,1);
```

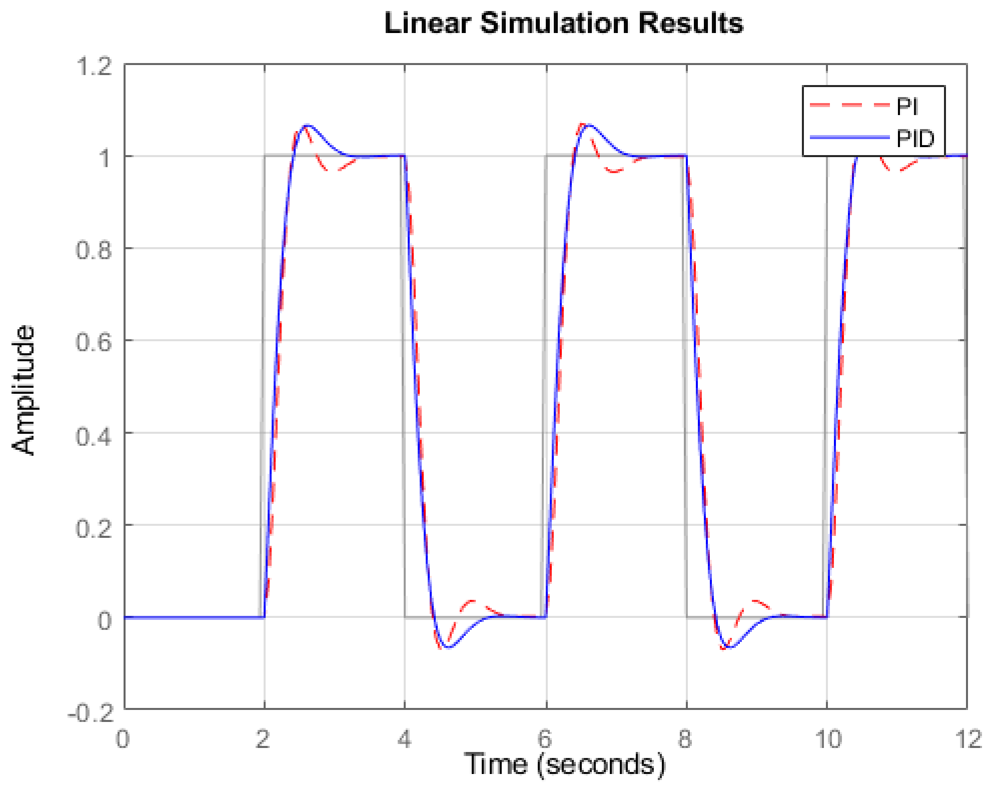
Plot the responses of both systems to a square wave with a period of 4 s.

```
[u,t] = gensig("square",4,12);
lsim(sys1,sys2,u,t)
grid on
legend("PI","PID")
```



By default, `lsim` chooses distinct colors for each system that you plot. You can specify colors and line styles using the `LineSpec` input argument.

```
lsim(sys1,"r--",sys2,"b",u,t)
grid on
legend("PI","PID")
```



The first `LineSpec` `"r--"` specifies a dashed red line for the response with the PI controller. The second `LineSpec` `"b"` specifies a solid blue line for the response with the PID controller. The legend reflects the specified colors and line styles. For more plot customization options, use `lsimplot`.

Plot Simulated Response of MIMO System

In a MIMO system, at each time step `t`, the input `u(t)` is a vector whose length is the number of inputs. To use `lsim`, you specify `u` as a matrix with dimensions `Nt`-by-`Nu`, where `Nu` is the number of system inputs and `Nt` is the length of `t`. In other words, each column of `u` is the input signal applied to the corresponding system input. For instance, to simulate a system with four inputs for 201 time steps, provide `u` as a matrix of four columns and 201 rows, where each row `u(i,:)` is the vector of input values at the `i`th time step; each column `u(:,j)` is the signal applied at the `j`th input.

Similarly, the output `y(t)` computed by `lsim` is a matrix whose columns represent the signal at each system output. When you use `lsim` to plot the simulated response, `lsim` provides separate axes for each output, representing the system response in each output channel to the input `u(t)` applied at all inputs.

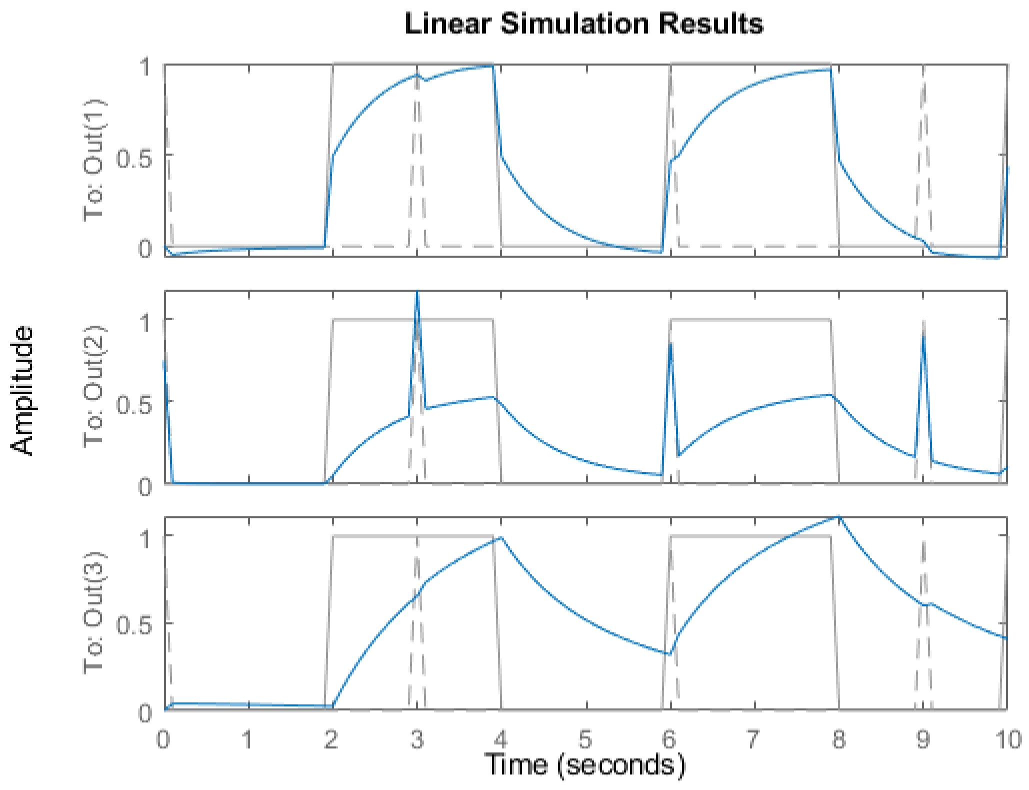
Consider the two-input, three-output state-space model with the following state-space matrices.

```
A = [-1.5  -0.2   1.0;
     -0.2  -1.7   0.6;
      1.0   0.6  -1.4];
```

```
B = [ 1.5  0.6;
     -1.8  1.0;
      0    0  ];

C = [ 0    -0.5 -0.1;
      0.35 -0.1 -0.15
      0.65  0    0.6];

D = [ 0.5  0;
      0.05 0.75
      0    0];

sys = ss(A,B,C,D);
```

Plot the response of `sys` to a square wave of period 4 s, applied to the first input `sys` and a pulse applied to the second input every 3 s. To do so, create column vectors representing the square wave and the pulsed signal using `gensig`. Then stack the columns into an input matrix. To ensure the two signals have the same number of samples, specify the same end time and sample time.

```
Tf = 10;
Ts = 0.1;
[uSq,t] = gensig("square",4,Tf,Ts);
[uP,~] = gensig("pulse",3,Tf,Ts);
u = [uSq uP];
lsim(sys,u,t)
```



Each axis shows the response of one of the three system outputs to the signals u applied at all inputs. Each plot also shows all input signals in gray.

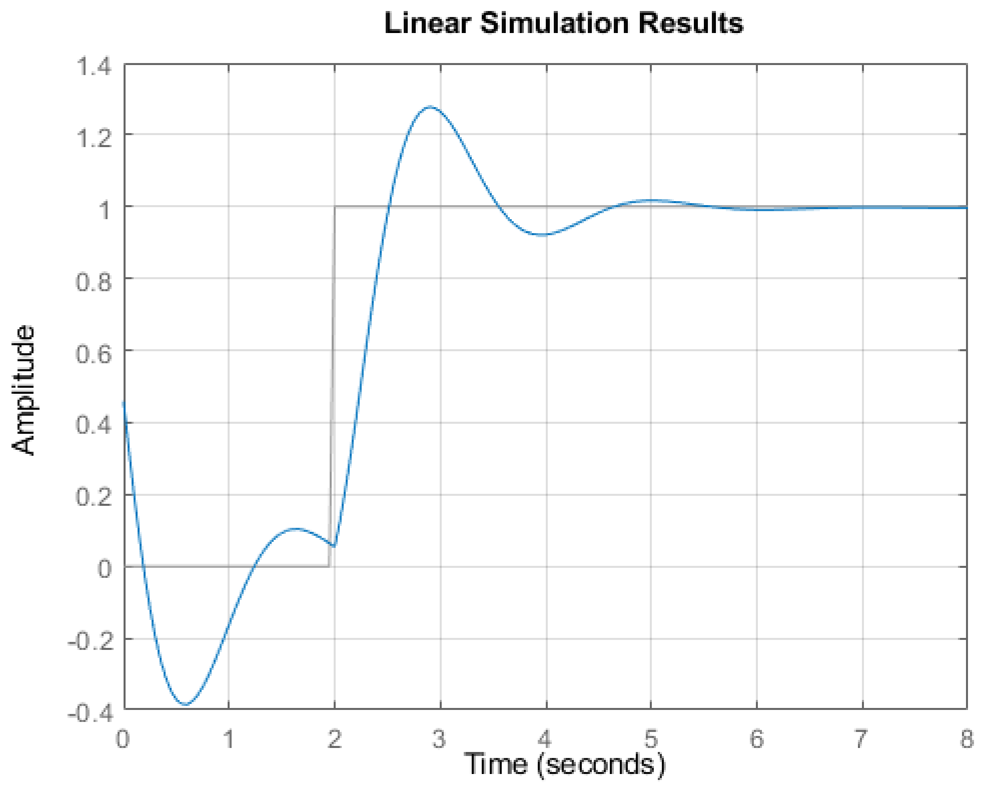
Plot System Evolution from Initial Condition

By default, lsim simulates the model assuming all states are zero at the start of the simulation. When simulating the response of a state-space model, use the optional x0 input argument to specify nonzero initial state values. Consider the following two-state SISO state-space model.

```
A = [-1.5 -3;
      3   -1];
B = [1.3; 0];
C = [1.15 2.3];
D = 0;

sys = ss(A,B,C,D);
```

Suppose that you want to allow the system to evolve from a known set of initial states with no input for 2 s, and then apply a unit step change. Specify the vector x0 of initial state values, and create the input vector.

```
x0 = [-0.2 0.3];
t = 0:0.05:8;
u = zeros(length(t),1);
u(t>=2) = 1;
lsim(sys,u,t,x0)
grid on
```



The first half of the plot shows the free evolution of the system from the initial state values `[-0.2 0.3]`. At `t = 2` there is a step change to the input, and the plot shows the system response to this new signal beginning from the state values at that time.

Extract Simulated Response Data

When you use `lsim` with output arguments, it returns the simulated response data in an array. For a SISO system, the response data is returned as a column vector of the same length as `t`. For instance, extract the response of a SISO system to a square wave. Create the square wave using `gensig`.

```
sys = tf([2 5 1],[1 2 3]);
[u,t] = gensig("square",4,10,0.05);
[y,t] = lsim(sys,u,t);
size(y)
```

```
ans = 1×2

   201     1
```

The vector `y` contains the simulated response at each time step in `t`. (`lsim` returns the time vector `t` as a convenience.)

For a MIMO system, the response data is returned in an array of dimensions *N*-by-*Ny*-by-*Nu*, where *Ny* and *Nu* are the number of outputs and inputs of the dynamic system. For instance, consider the following state-space model, representing a three-state system with two inputs and three outputs.

```
A = [-1.5  -0.2   1.0;
     -0.2  -1.7   0.6;
      1.0   0.6  -1.4];

B = [ 1.5  0.6;
     -1.8  1.0;
      0    0  ];

C = [ 0   -0.1 -0.2;
      0.7 -0.2 -0.3
     -0.65 0   -0.6];

D = [ 0.1  0;
      0.1  1.5
      0    0];

sys = ss(A,B,C,D);
```

Extract the responses of the three output channels to the square wave applied at both inputs.

```
uM = [u u];
[y,t] = lsim(sys,uM,t);
size(y)
```

```
ans = 1×2

   201     3
```

`y(:,j)` is a column vector containing response at the *j*th output to the square wave applied to both inputs. That is, `y(i,:)` is a vector of three values, the output values at the *i*th time step.

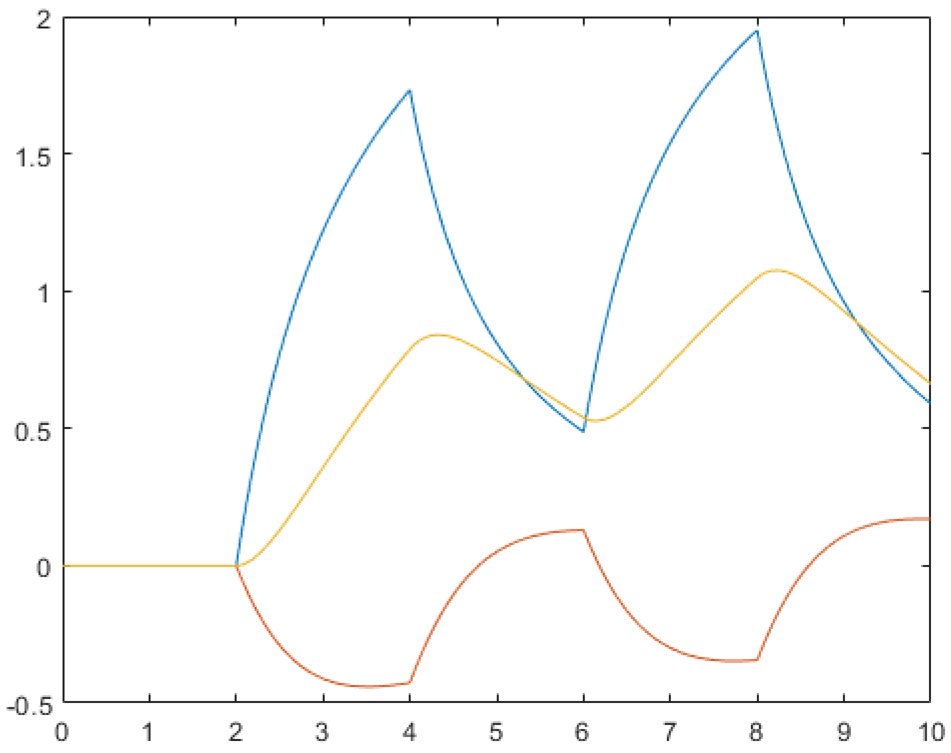
Because `sys` is a state-space model, you can extract the time evolution of the state values in response to the input signal.

```
[y,t,x] = lsim(sys,uM,t);
size(x)
```

```
ans = 1×2

   201     3
```

Each row of `x` contains the state values `[x1,x2,x3]` at the corresponding time in `t`. In other words, `x(i,:)` is the state vector at the *i*th time step. Plot the state values.

```
plot(t,x)
```



Response of Systems in Model Array

The example Plot Response of Multiple Systems to Same Input shows how to plot responses of several individual systems on a single axis. When you have multiple dynamic systems arranged in a model array, `lsim` plots all their responses at once.

Create a model array. For this example, use a one-dimensional array of second-order transfer functions having different natural frequencies. First, preallocate memory for the model array. The following command creates a 1-by-5 row of zero-gain SISO transfer functions. The first two dimensions represent the model outputs and inputs. The remaining dimensions are the array dimensions. (For more information about model arrays and how to create them, see “Model Arrays” (Control System Toolbox).)

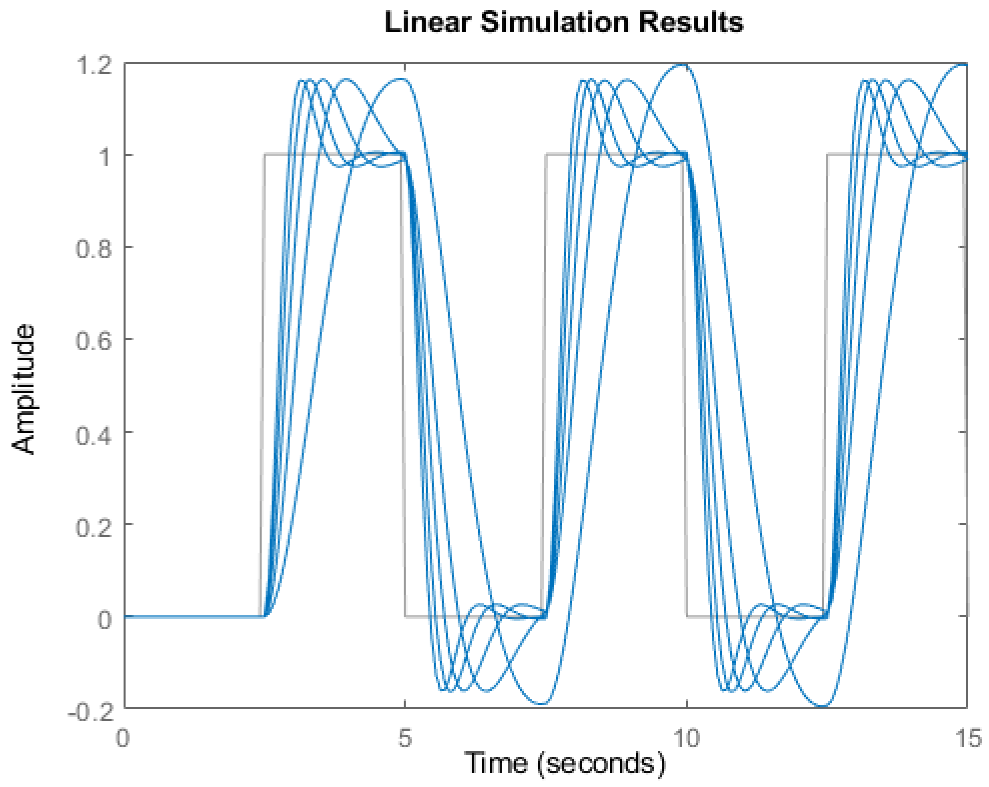
```
sys = tf(zeros(1,1,1,5));
```

Populate the array.

```
w0 = 1.5:1:5.5;    % natural frequencies
zeta = 0.5;        % damping constant
for i = 1:length(w0)
   sys(:,:,1,i) = tf(w0(i)^2,[1 2*zeta*w0(i) w0(i)^2]);
end
```

Plot the responses of all models in the array to a square wave input.

```
[u,t] = gensig("square",5,15);
lsim(sys,u,t)
```



`lsim` uses the same line style for the responses of all entries in the array. One way to distinguish among entries is to use the `SamplingGrid` property of dynamic system models to associate each entry in the array with the corresponding `w0` value.

```
sys.SamplingGrid = struct('frequency',w0);
```

Now, when you plot the responses in a MATLAB figure window, you can click a trace to see which frequency value it corresponds to.

Simulate Response of Identified Model

Load estimation data to estimate a model.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
z = iddata(y,u,0.1,'Name','DC-motor');
```

`z` is an `iddata` object that stores the one-input two-output estimation data with a sample time of 0.1 s.

Estimate a state-space model of order 4 using estimation data `z`.

```
[sys,x0] = n4sid(z,4);
```

`sys` is the estimated model and `x0` is the estimated initial states.

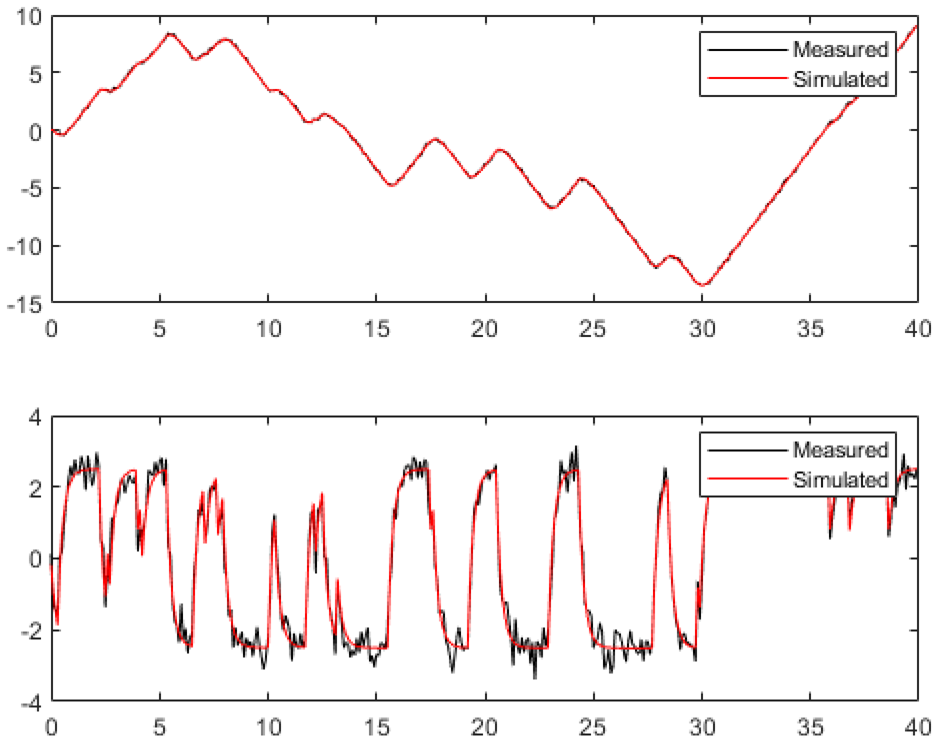
Simulate the response of `sys` using the same input data as the one used for estimation and the initial states returned by the estimation command.

```
[y,t,x] = lsim(sys,z.InputData,[],x0);
```

Here, `y` is the system response, `t` is the time vector used for simulation, and `x` is the state trajectory.

Compare the simulated response `y` to the measured response `z.OutputData` for both outputs.

```
subplot(211), plot(t,z.OutputData(:,1),'k',t,y(:,1),'r')
legend('Measured','Simulated')
subplot(212), plot(t,z.OutputData(:,2),'k',t,y(:,2),'r')
legend('Measured','Simulated')
```



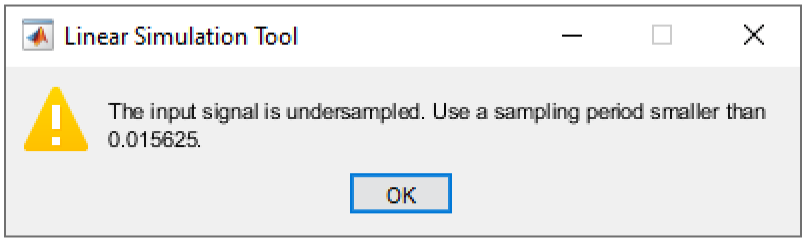
Effect of Sample Time on Simulation

The choice of sample time can drastically affect simulation results. To illustrate why, consider the following second-order model.

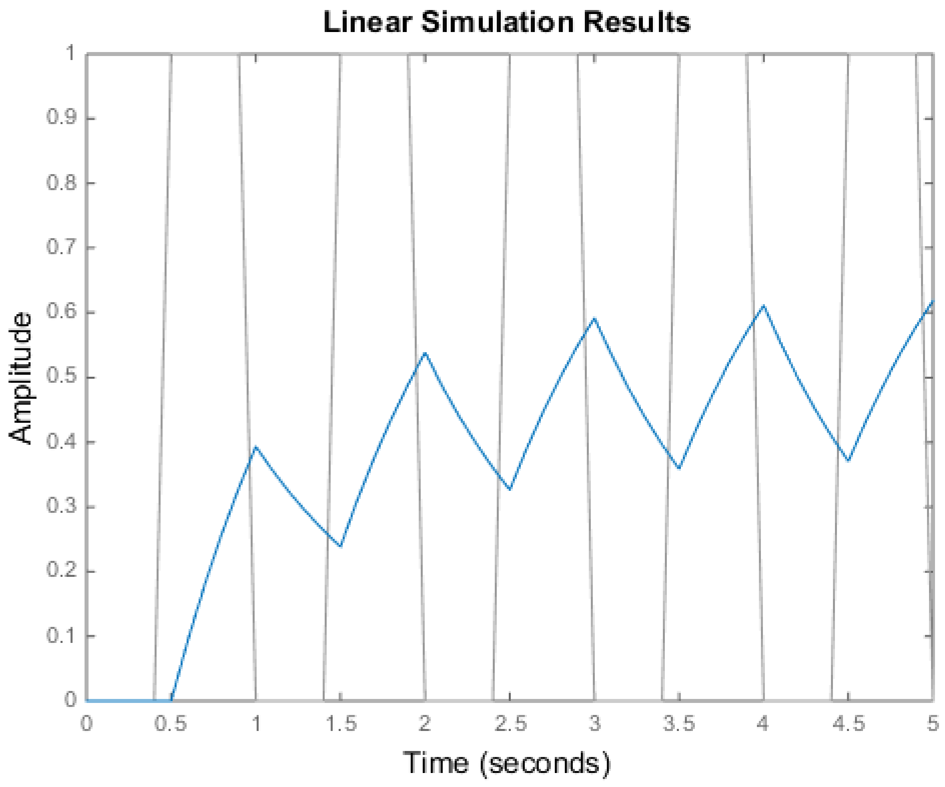
$$sys(s) = \frac{\omega^2}{s^2 + 2s + \omega^2}, \ \omega = 62.83\,.$$

Simulate the response of this model to a square wave with period 1 s, using a sample time of 0.1 s.

```
w2 = 62.83^2;
sys = tf(w2,[1 2 w2]);

tau = 1;
Tf = 5;
Ts = 0.1;
[u,t] = gensig("square",tau,Tf,Ts);
lsim(sys,u,t)
```

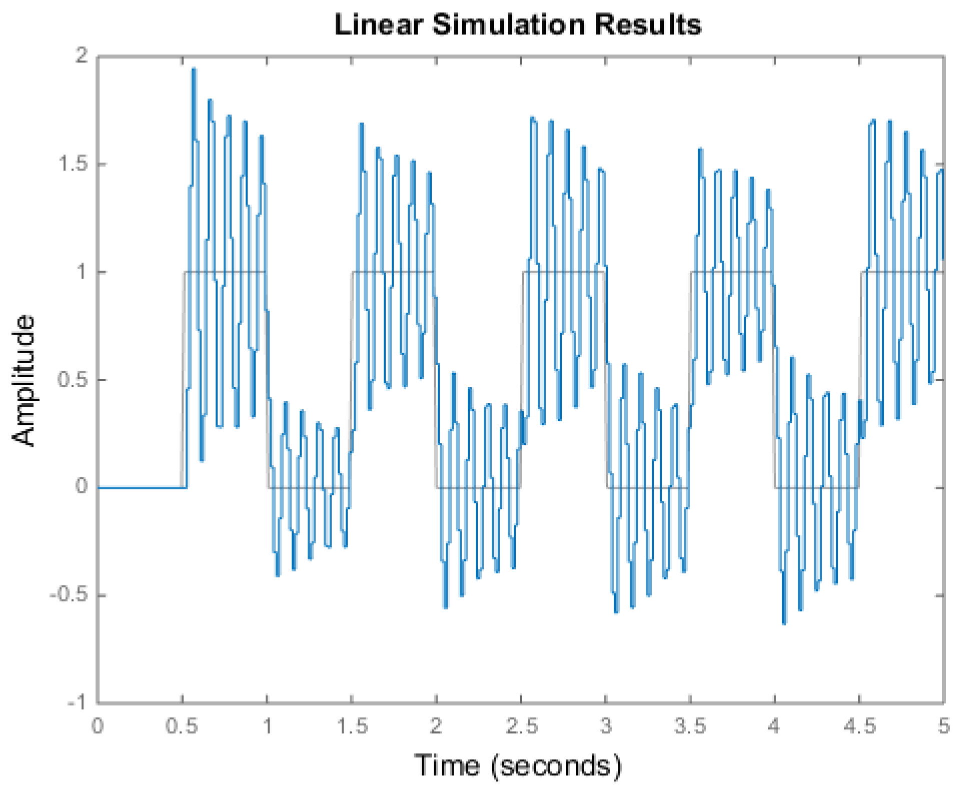

Linear Simulation Tool

The input signal is undersampled. Use a sampling period smaller than 0.015625.



`lsim` simulates the model using the specified input signal, but it issues a warning that the input signal is undersampled. `lsim` recommends a sample time that generates at least 64 samples per period of the input `u`. To see why this recommendation matters, simulate `sys` again using a sample time smaller than the recommended maximum.

```
figure
Ts2 = 0.01;
[u2,t2] = gensig("square",tau,Tf,Ts2);
lsim(sys,u2,t2)
```



This response exhibits strong oscillatory behavior that is hidden in the undersampled version.

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems whose responses you can simulate include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value for both plotting and returning response data.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model. When you use output arguments, the function returns response data for the nominal model only.
- Sparse state-space models such as `sparss` and `mechss` models.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models. For identified models, you can also use the `sim` command, which can compute the standard deviation of the simulated response and state trajectories. `sim` can also simulate all types of models with nonzero initial conditions, and can simulate nonlinear identified models.

`lsim` does not support frequency-response data models such as `frd`, `genfrd`, or `idfrd` models.

If `sys` is an array of models, the function plots the responses of all models in the array on the same axes. See “Response of Systems in Model Array” (Control System Toolbox).

u — Input signal
vector | array

Input signal for simulation, specified as a vector for single-input systems, and an array for multi-input systems.

- For single-input systems, `u` is a vector of the same length as `t`.
- For multi-input systems, `u` is an array with as many rows as there are time samples (`length(t)`) and as many columns as there are inputs to `sys`. In other words, each row `u(i,:)` represents the values applied at the inputs of `sys` at time `t(i)`. Each column `u(:,j)` is the signal applied to the `j`th input of `sys`.

t — Time samples
vector

Time samples at which to compute the response, specified as a vector of the form `0:dT:Tf`. The `lsim` command interprets `t` as having the units specified in the `TimeUnit` property of the model `sys`.

For continuous-time `sys`, the `lsim` command uses the time step `dT` to discretize the model. If `dT` is too large relative to the system dynamics (undersampling), `lsim` issues a warning recommending a faster sampling time. For further discussion of the impact of sampling time on simulation, see “Effect of Sample Time on Simulation” (Control System Toolbox).

For discrete-time `sys`, the time step `dT` must equal the sample time of `sys`. Alternatively, you can omit `t` or set it to `[]`. In that case, `lsim` sets `t` to a vector of the same length as `u` that begins at 0 with a time step equal to `sys.Ts`.

x0 — Initial state values
vector of zeros (default) | vector

Initial state values for simulating a state-space model, specified as a vector having one entry for each state in `sys`. If you omit this argument, then `lsim` sets all states to zero at `t = 0`.

method — Discretization method
`'zoh` | `'foh'`

Discretization method for sampling continuous-time models, specified as one of the following.

- `'zoh'` — Zero-order hold
- `'foh'` — First-order hold

When `sys` is a continuous-time model, `lsim` computes the time response by discretizing the model using a sample time equal to the time step `dT = t(2)-t(1)` of `t`. If you do not specify a discretization method, then `lsim` selects the method automatically based on the smoothness of the signal `u`. For more information about these two discretization methods, see “Continuous-Discrete Conversion Methods” (Control System Toolbox).

LineSpec — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a string or vector of one, two, or three characters. The characters can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line. For more information about configuring this argument, see the `LineSpec` input argument of the `plot` function.

Example: `'r--'` specifies a red dashed line

Example: `'*b'` specifies blue asterisk markers

Example: `'y'` specifies a yellow line

Output Arguments

y — Simulated response data
array

Simulated response data, returned as an array.

- For single-input systems, `y` is a column vector of the same length as `t`.
- For multi-output systems, `y` is an array with as many rows as there are time samples (`length(t)`) and as many columns as there are outputs in `sys`. Thus, the *j*th column of `y`, or `y(:,j)`, contains the response at the *j*th output to `u` applied at all inputs.

`tOut` — Time vector
vector

Time vector used for simulation, returned as a column vector. When you specify an input time vector `t` of the form `0:dT:Tf`, then `tOut = t`. If `t` is nearly equisampled, `lsim` adjusts the sample times for simulation and returns the result in `tOut`. For discrete-time `sys`, you can omit `t` or set it to `[]`. In that case, `lsim` sets `t` to a vector of the same length as `u` that begins at 0 with a time step equal to `sys.Ts`, and returns the result in `tOut`.

x — State trajectories
array

State trajectories, returned as an array. When `sys` is a state-space model, `x` contains the evolution of the states of `sys` in response to the input. `x` is an array with as many rows as there are time samples `(length(t))` and as many columns as there are states in sys.

Tips

- When you need additional plot customization options, use `lsimplot` instead.

Algorithms

For a discrete-time transfer function,

$$sys\left(z^{-1}\right) = \frac{a_0 + a_1 z^{-1} + \ldots + a_n z^{-n}}{1 + b_1 z^{-1} + \ldots + b_n z^{-n}},$$

`lsim` filters the input based on the recursion associated with this transfer function:

$$y[k] = a_0 u[k] + \ldots + a_n u[k-n] - b_1 y[k-1] - \ldots - b_n [k-n].$$

For discrete-time `zpk` models, `lsim` filters the input through a series of first-order or second-order sections. This approach avoids forming the numerator and denominator polynomials, which can cause numerical instability for higher-order models.

For discrete-time state-space models, `lsim` propagates the discrete-time state-space equations,

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n], \\ y[n] &= Cx[n] + Du[n]\,. \end{aligned}$$

For continuous-time systems, `lsim` first discretizes the system using `c2d`, and then propagates the resulting discrete-time state-space equations. Unless you specify otherwise with the `method` input argument, `lsim` uses the first-order-hold discretization method when the input signal is smooth, and zero-order hold when the input signal is discontinuous, such as for pulses or square waves. The sample time for discretization is the spacing `dT` between the time samples you supply in `t`.

See Also

Functions
`gensig` | `impulse` | `initial` | `step` | `sim` | `lsiminfo` | `lsimplot`

Apps
Linear System Analyzer

Introduced in R2012a

lsiminfo

Compute linear response characteristics

Syntax

```
S = lsiminfo(y,t)
S = lsiminfo(y,t,yfinal)
S = lsiminfo(y,t,yfinal,yinit)

S = lsiminfo( ___ ,'SettlingTimeThreshold',ST)
```

Description

`lsiminfo` lets you compute linear response characteristics from an array of response data `[y,t]`. For a linear response $y(t)$, `lsiminfo` computes characteristics relative to y_{init} and y_{final}, where y_{init} is the initial offset, that is, the value before the input is applied, and y_{final} is the steady-state value of the response.

`lsiminfo` uses $y_{init} = 0$ and y_{final} = last sample value of $y(t)$ unless you explicitly specify these values.

The function returns the characteristics in a structure containing the fields:

- `TransientTime` — The first time T such that the error $|y(t) - y_{final}| \leq SettlingTimeThreshold \times e_{max}$ for $t \geq T$, where e_{max} is the maximum error $|y(t) - y_{final}|$ for $t \geq 0$.

  By default, $SettlingTimeThreshold = 0.02$ (2% of the peak error). Transient time measures how quickly the transient dynamics die off.
- `SettlingTime` — The first time T such that $|y(t) - y_{final}| \leq SettlingTimeThreshold \times |y_{final} - y_{init}|$ for $t \geq T$.

  By default, settling time measures the time it takes for the error to stay below 2% of $|y_{final} - y_{init}|$.
- `Min` — Minimum value of $y(t)$.
- `MinTime` — Time the response takes to reach the minimum value.
- `Max` — Maximum value of $y(t)$.
- `MaxTime` — Time the response takes to reach the maximum value.

`S = lsiminfo(y,t)` computes linear response characteristics from an array of response data `y` and corresponding time vector `t`. This syntax uses $y_{init} = 0$ and the last value in `y` (or the last value in each channel's corresponding response data) as y_{final} to compute characteristics that depend on these values.

For SISO system responses, `y` is a vector with the same number of entries as `t`. For MIMO response data, `y` is an array containing the responses of each I/O channel.

`S = lsiminfo(y,t,yfinal)` computes linear response characteristics relative to the steady-state value `yfinal`. This syntax is useful when you know that the expected steady-state system response differs from the last value in y for reasons such as measurement noise. This syntax uses $y_{init} = 0$.

For SISO responses, `t` and `y` are vectors with the same length `NS`. For systems with `NY` outputs, you can specify `y` as an `NS`-by-`NY` array and `yfinal` as an `NY`-by-`1` array. `lsiminfo` then returns a `NY`-by-`1` structure array `S` of response characteristics corresponding to each output channel.

`S = lsiminfo(y,t,yfinal,yinit)` computes response characteristics relative to the response initial value `yinit`. This syntax is useful when your `y` data has an initial offset, that is, `y` is nonzero before the input is applied.

For SISO responses, `t` and `y` are vectors with the same length `NS`. For systems with `NY` outputs, you can specify `y` as an `NS`-by-`NY` array and `yfinal` and `yinit` as an `NY`-by-`1` arrays. `lsiminfo` then returns a `NY`-by-`1` structure array `S` of response characteristics corresponding to each output channel.

`S = lsiminfo( ___ ,'SettlingTimeThreshold',ST)` lets you specify the threshold `ST` used in definition of settling and transient times. The default value is `ST = 0.02` (2%). You can use this syntax with any of the previous input-argument combinations.

Examples

Compute Response Characteristics of Transfer Function

Create the following continuous-time transfer function:

$$H(s) = \frac{s-1}{s^3 + 2s^2 + 3s + 4}$$

```
sys = tf([1 -1],[1 2 3 4]);
```

Calculate the impulse response.

```
[y,t] = impulse(sys);
```

`impulse` returns the output response `y` and the time vector `t` used for simulation.

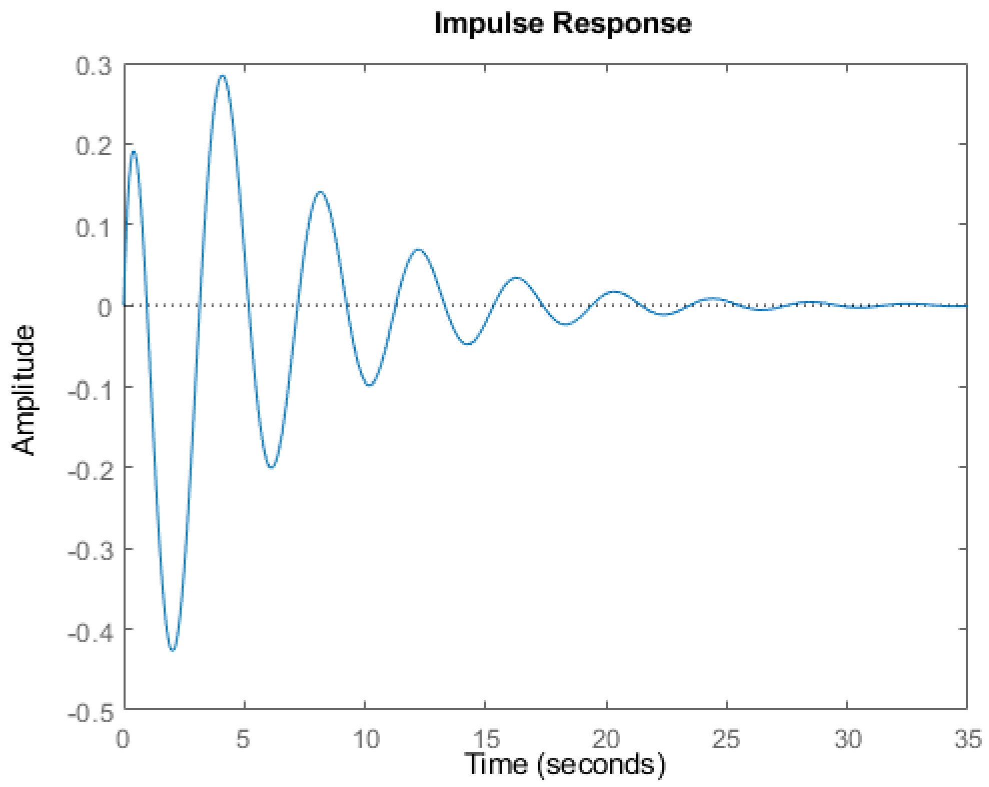
Compute the response characteristics using a final response value of `0`.

```
s = lsiminfo(y,t,0)
```

```
s = struct with fields:
    TransientTime: 22.8700
     SettlingTime: NaN
              Min: -0.4268
          MinTime: 2.0088
              Max: 0.2847
          MaxTime: 4.0733
```

You can plot the impulse response and verify these response characteristics. For example, the time at which the minimum response value (`MinTime`) is reached is approximately 2 seconds.

```
impulse(sys)
```



Input Arguments

y — Response data
vector | array

Response data, specified as one of the following:

- For SISO response data, a vector of length `Ns`, where `Ns` is the number of samples in the response data.
- For MIMO response data, an `Ns`-by-`Ny` array, where `Ny` is the number of system outputs.

t — Time vector
vector

Time vector corresponding to the response data in `y`, specified as a vector of length `Ns`.

`yfinal` — Steady-state value
scalar | array

Response steady-state value, specified as a scalar or an array.

- For SISO response data, specify a scalar value.
- For MIMO response data, specify an `Ny`-by-`1` array, where each entry provides the steady-state response value for the corresponding system channel.

If you do not provide `yfinal`, then `lsiminfo` uses the last value in the corresponding channel of `y` as the steady-state response value.

`yinit` — Response initial value
scalar | array

Value of `y` before the input is applied, specified as a scalar or an array.

- For SISO response data, specify a scalar value.
- For MIMO response data, specify an `Ny`-by-`1` array, where each entry provides the response initial value for the corresponding system channel.

If you do not provide `yinit`, then `lsiminfo` uses zero as the response initial value.

ST — Settling time threshold
0.02 (default) | scalar between 0 and 1

Threshold for defining settling and transient times, specified as a scalar value between 0 and 1. To change the default settling and transient time definitions (see “Description” on page 1-932), set `ST` to a different value. For instance, to measure when the error falls below 5%, set `ST` to 0.05.

Output Arguments

S — Response characteristics
structure

Linear response characteristics, returned as a structure containing the fields:

- `TransientTime`
- `SettlingTime`
- `Min`
- `MinTime`
- `Max`
- `MaxTime`

For more information on how `lsiminfo` defines these characteristics, see “Description” on page 1-932.

For MIMO models or responses data, `S` is a structure array in which each entry contains the step-response characteristics of the corresponding I/O channel. For instance, if you provide a 3-input, 3-output model or array of response data, then `S(2,3)` contains the characteristics of the response from the third input to the second output.

Compatibility Considerations

Settling time computation changed
Behavior changed in R2021b

The settling time calculation is now based on the time it takes for the error to stay below 2% of $|y_{final} - y_{init}|$. The following table summarizes the changes to the fields of the structure returned by `lsiminfo`.

Before R2021b	R2021b
`SettlingTime` — The first time T such that the error $\|y(t) - y_{final}\| \leq SettlingTimeThreshold \times e_{max}$ for $t \geq T$, where e_{max} is the maximum error $\|y(t) - y_{final}\|$ for $t \geq 0$. By default, *SettlingTimeThreshold* = 0.02 (2% of the peak error). `SettlingTime` measures the time for the error to fall below 2% of the peak value of the error.	`SettlingTime` — The first time T such that the error $\|y(t) - y_{final}\| \leq SettlingTimeThreshold \times \|y_{final} - y_{init}\|$ for $t \geq T$. By default, `SettlingTime` measures the time it takes for the error to stay below 2% of $\|y_{final} - y_{init}\|$.

Additionally, the output structure `S` now contains a `TransientTime` field. This characteristic contains the peak-error-based settling time calculation used in releases before R2021b. Transient time is used to measure how quickly the transient dynamics die off.

See Also

`impulse` | `lsim` | `stepinfo`

Introduced in R2012a

lsimplot

Plot simulated time response of dynamic system to arbitrary inputs with additional plot customization options

Syntax

```
h = lsimplot(sys)
h = lsimplot(sys,u,t)
h = lsimplot(sys1,sys2,...,sysN,u,t)
h = lsimplot(sys1,LineSpec1,...,sysN,LineSpecN,u,t)
h = lsimplot( ___ ,x0)
h = lsimplot( ___ ,method)
h = lsimplot(AX, ___ )
h = lsimplot( ___ ,plotoptions)
```

Description

`lsimplot` lets you plot simulated time response of dynamic system to arbitrary inputs with a broader range of plot customization options than `lsim`. You can use `lsimplot` to obtain the plot handle and use it to customize the plot, such as modify the axes labels, limits and units. You can also use `lsimplot` to draw a simulated time response plot on an existing set of axes represented by an axes handle. To customize an existing simulated time response plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox). To create simulated time response plots with default options or to extract simulated response data, use `lsim`.

`h = lsimplot(sys)` opens the Linear Simulation Tool for the dynamic system model `sys`, where you can interactively specify the driving input(s), the time vector, and initial state. It also returns the plot handle `h`. You can use this handle `h` to customize the plot with the `getoptions` and `setoptions` commands.

For more information about using the Linear Simulation Tool for linear analysis, see Working with the Linear Simulation Tool (Control System Toolbox).

`h = lsimplot(sys,u,t)` plots the simulated time response of the model `sys` to the input signal `u` and the corresponding time vector `t`. For MIMO systems, `u` is a matrix with as many columns as the number of inputs and whose `i`th row specifies the input value at time `t(i)`. For SISO systems, the input `u` can be specified either as a row or column vector.

`h = lsimplot(sys1,sys2,...,sysN,u,t)` plots the simulated response of multiple dynamic systems `sys1,sys2,…,sysN` using the input `u` and time vector `t` on the same plot. All systems must have the same number of inputs and outputs to use this syntax.

h = lsimplot(sys1,LineSpec1,...,sysN,LineSpecN,u,t) sets the line style, marker type, and color for the simulated time response of each system. All systems must have the same number of inputs and outputs to use this syntax.

h = lsimplot(___ ,x0) further specifies a vector x0 of initial state values, when sys is a state-space model.

h = lsimplot(___ ,method) specifies how lsimplot interpolates the input values between samples, when sys is a continuous-time model.

h = lsimplot(AX, ___) plots the simulated response on the Axes object in the current figure with the handle AX.

h = lsimplot(___ ,plotoptions) plots the simulated response with the options set specified in plotoptions. You can use these options to customize the plot appearance using the command line. Settings you specify in plotoptions overrides the preference settings in the MATLAB session in which you run lsimplot. Therefore, this syntax is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

Examples

Customized Plot of Simulated Response to Arbitrary Input Signal

For this example, change time units to minutes and turn the grid on for the simulated response plot. Consider the following transfer function.

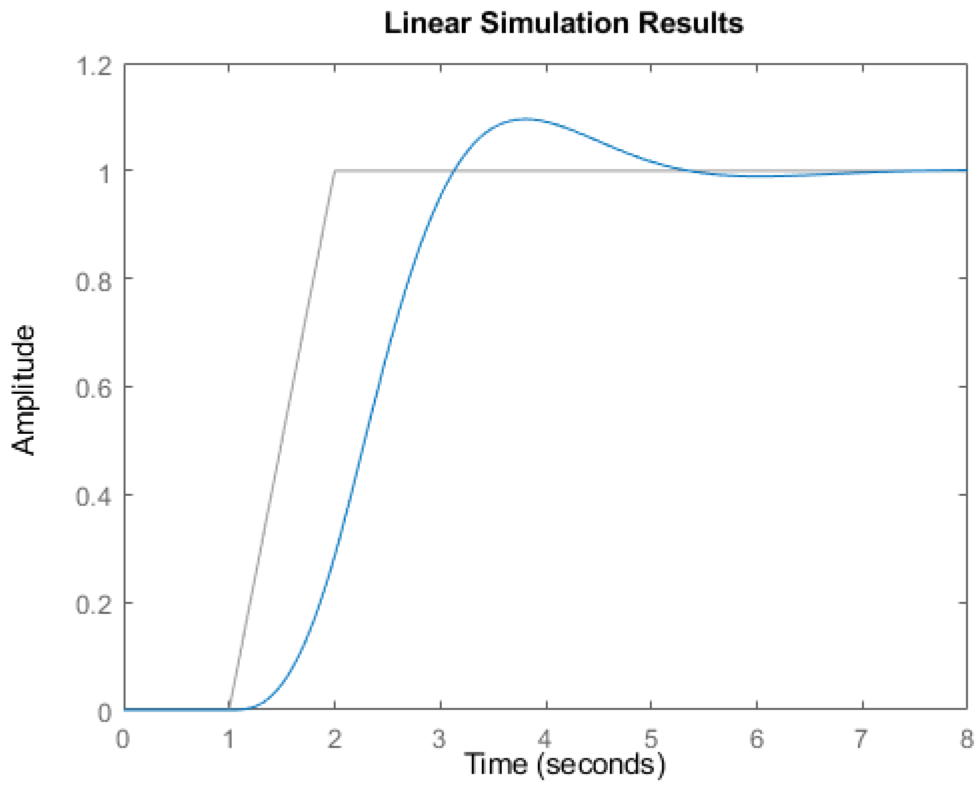
```
sys = tf(3,[1 2 3]);
```

To compute the response of this system to an arbitrary input signal, provide lsimplot with a vector of the times t at which you want to compute the response and a vector u containing the corresponding signal values. For instance, plot the system response to a ramping step signal that starts at 0 at time t = 0, ramps from 0 at t = 1 to 1 at t = 2, and then holds steady at 1. Define t and compute the values of u.

```
t = 0:0.04:8;
u = max(0,min(t-1,1));
```

Use lsimplot plot the system response to the signal with a plot handle h.

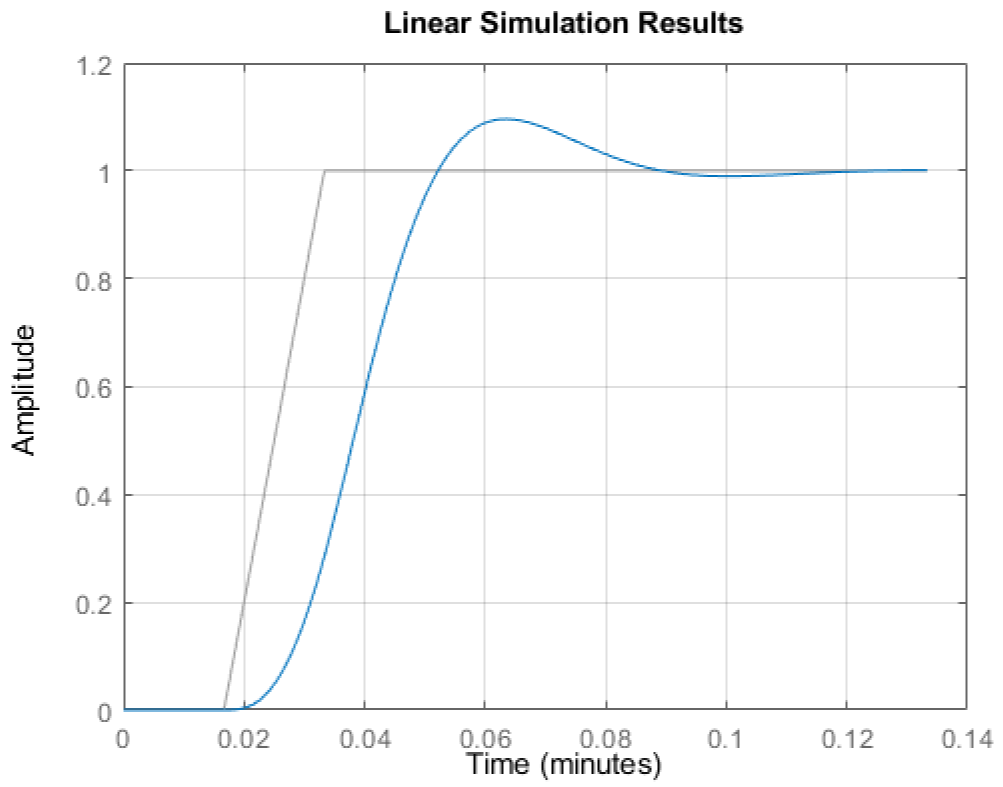
```
h = lsimplot(sys,u,t);
```



The plot shows the applied input `(u,t)` in gray and the system response in blue.

Use the plot handle to change the time units to minutes and to turn the grid on. To do so, edit properties of the plot handle, `h` using `setoptions`.

```
setoptions(h,'TimeUnits','minutes','Grid','on')
```



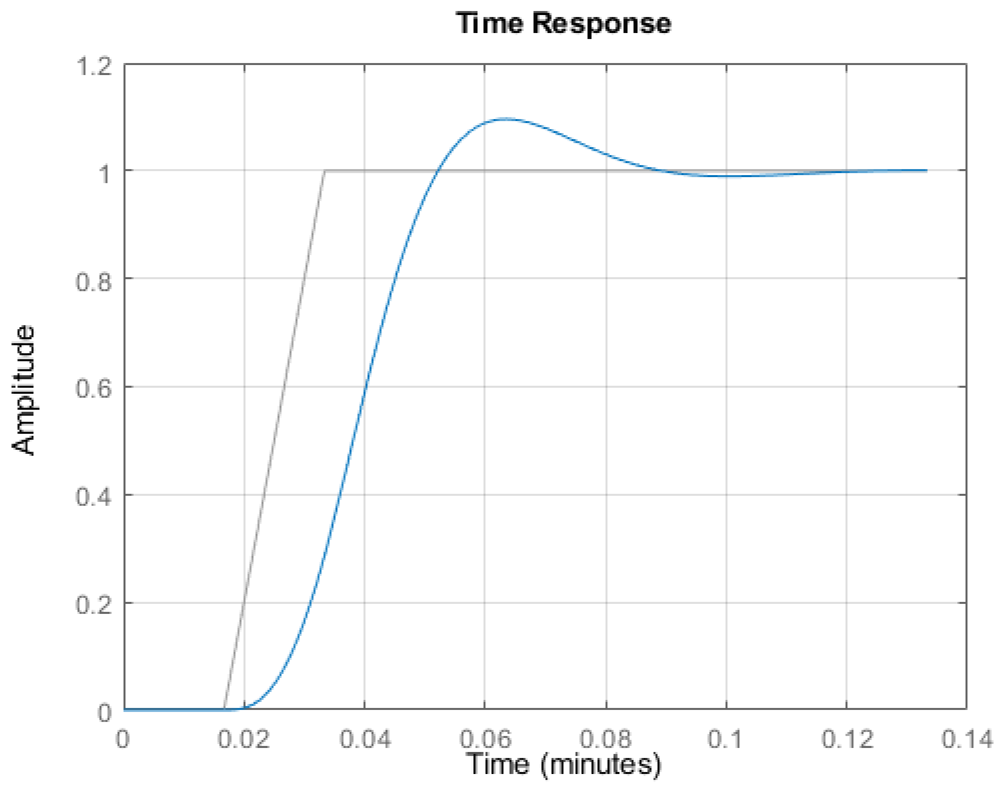
The plot automatically updates when you call `setoptions`.

Alternatively, you can also use the `timeoptions` command to specify the required plot options. First, create an options set based on the toolbox preferences.

```
plotoptions = timeoptions('cstprefs');
```

Change properties of the options set by setting the time units to minutes and enabling the grid.

```
plotoptions.TimeUnits = 'minutes';
plotoptions.Grid = 'on';
lsimplot(sys,u,t,plotoptions);
```



Customized Plot Response of Multiple Systems to Same Input

`lsimplot` allows you to plot the simulated responses of multiple dynamic systems on the same axis. For instance, compare the closed-loop response of a system with a PI controller and a PID controller. Then, customize the plot by enabling normalization and turning the grid on.

First, create a transfer function of the system and tune the controllers.

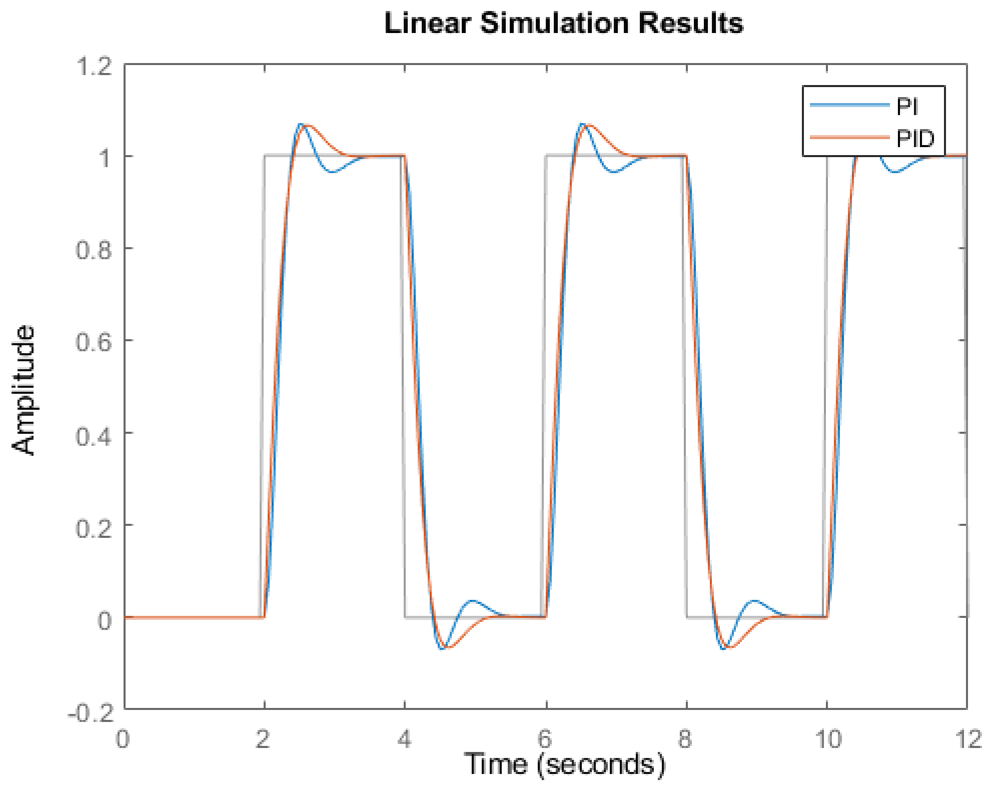
```
H = tf(4,[1 10 25]);
C1 = pidtune(H,'PI');
C2 = pidtune(H,'PID');
```

Form the closed-loop systems.

```
sys1 = feedback(H*C1,1);
sys2 = feedback(H*C2,1);
```

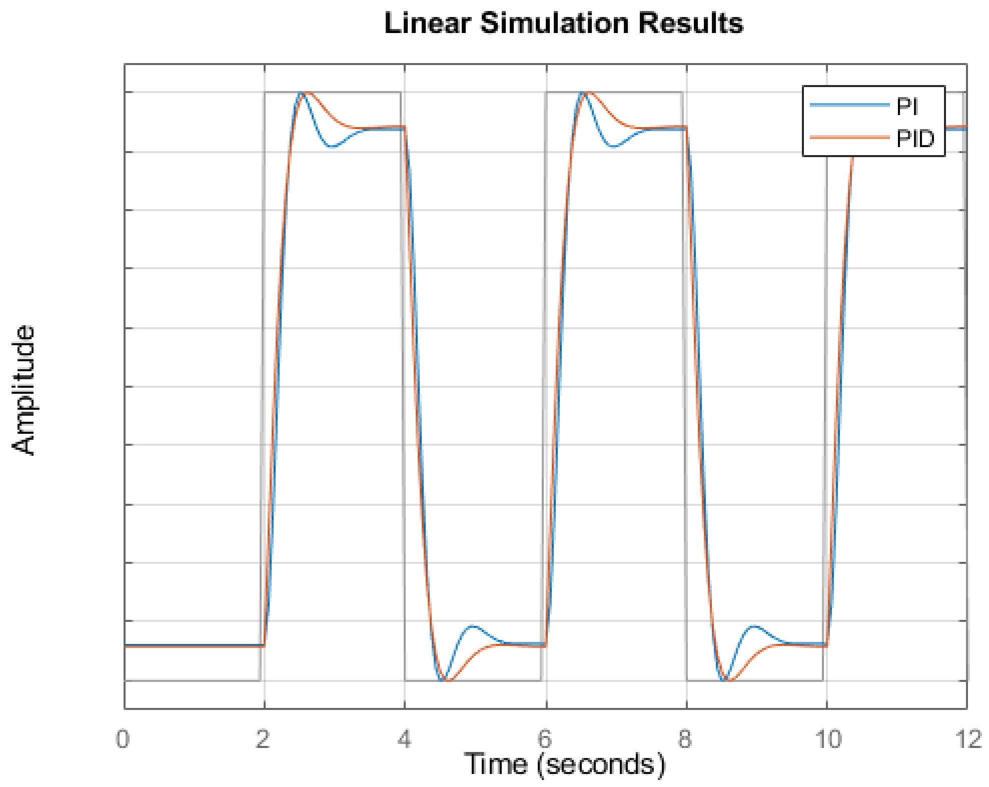
Plot the responses of both systems to a square wave with a period of 4 s.

```
[u,t] = gensig("square",4,12);
h1 = lsimplot(sys1,sys2,u,t);
legend("PI","PID")
```



Use `setoptions` to enable normalization and to turn on the grid.

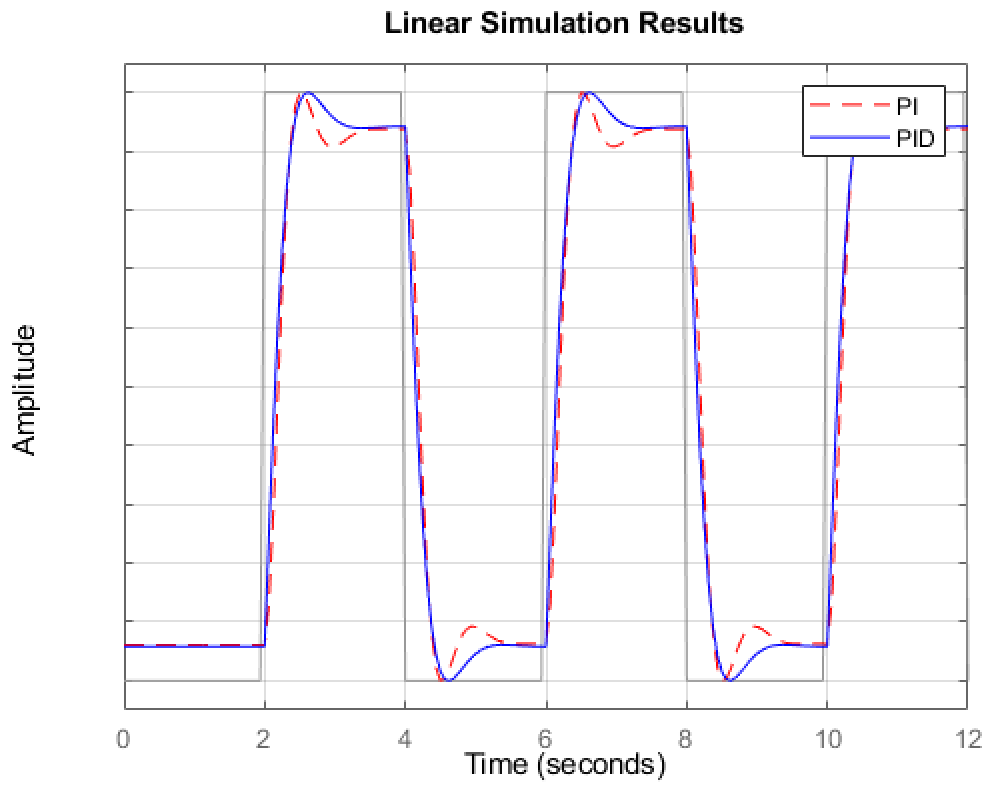
```
setoptions(h1,'Normalize','on','Grid','on')
```



The plot automatically updates when you call `setoptions`.

By default, `lsimplot` chooses distinct colors for each system that you plot. You can specify colors and line styles using the `LineSpec` input argument.

```
h2 = lsimplot(sys1,"r--",sys2,"b",u,t);
legend("PI","PID")
setoptions(h2,'Normalize','on','Grid','on')
```



The first LineSpec "r--" specifies a dashed red line for the response with the PI controller. The second LineSpec "b" specifies a solid blue line for the response with the PID controller. The legend reflects the specified colors and line styles.

Custom Plot of System Evolution from Initial Condition

By default, lsimplot simulates the model assuming all states are zero at the start of the simulation. When simulating the response of a state-space model, use the optional x0 input argument to specify nonzero initial state values. Consider the following two-state SISO state-space model.

```
A = [-1.5 -3;
      3   -1];
B = [1.3; 0];
C = [1.15 2.3];
D = 0;
sys = ss(A,B,C,D);
```

Suppose that you want to allow the system to evolve from a known set of initial states with no input for 2 s, and then apply a unit step change. Specify the vector x0 of initial state values, and create the input vector.

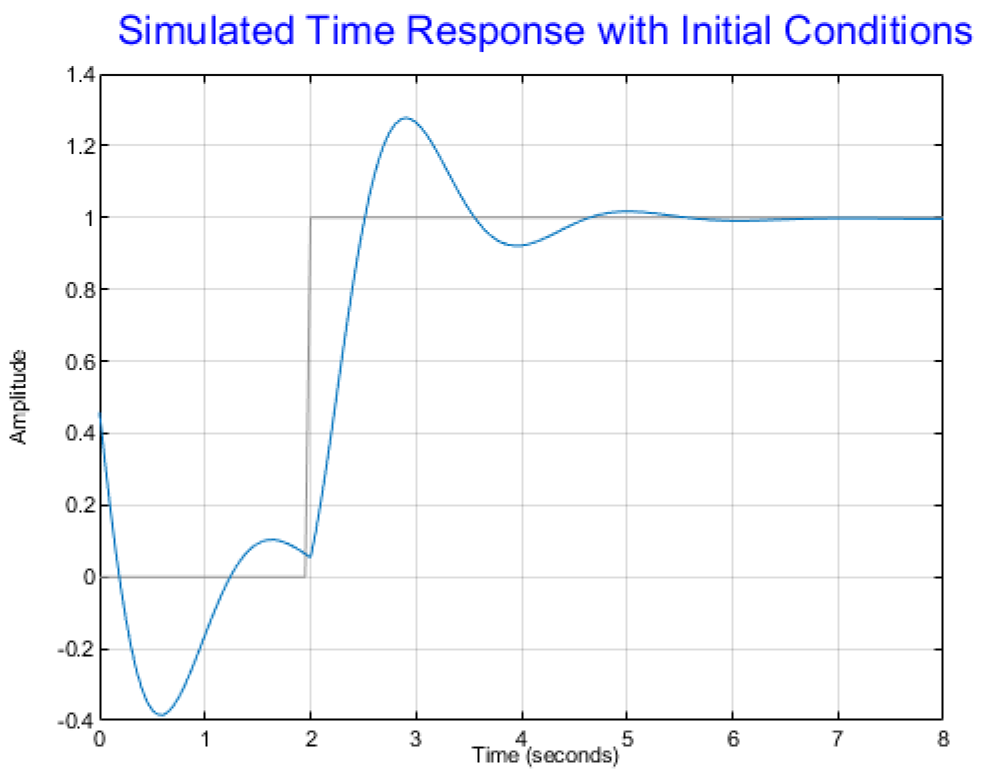
```
x0 = [-0.2 0.3];
t = 0:0.05:8;
u = zeros(length(t),1);
u(t>=2) = 1;
```

First, create a default options set using `timeoptions`.

```
plotoptions = timeoptions;
```

Next change the required properties of the options set `plotoptions` and plot the simulated response with the zero order hold option.

```
plotoptions.Title.FontSize = 15;
plotoptions.Title.Color = [0 0 1];
plotoptions.Grid = 'on';
h = lsimplot(sys,u,t,x0,plotoptions,'zoh');
hold on
title('Simulated Time Response with Initial Conditions')
```



The first half of the plot shows the free evolution of the system from the initial state values `[-0.2 0.3]`. At `t = 2` there is a step change to the input, and the plot shows the system response to this new signal beginning from the state values at that time. Because `plotoptions` begins with a fixed set of options, the plot result is independent of the toolbox preferences of the MATLAB session.

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Sparse state-space models, such as `sparss` or `mechss` models. Final time `tFinal` must be specified when using sparse models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value to plot the simulated response.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models. For identified models, you can also use the `sim` command, which can compute the standard deviation of the simulated response and state trajectories. `sim` can also simulate all types of models with nonzero initial conditions, and can simulate nonlinear identified models.

`lsimplot` does not support frequency-response data models such as `frd`, `genfrd`, or `idfrd` models.

If `sys` is an array of models, the function plots the responses of all models in the array on the same axes.

LineSpec — Line style, marker, and color

character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'_'`	Horizontal line
`'|'`	Vertical line
`'s'`	Square
`'d'`	Diamond

Marker	Description
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'p'`	Pentagram
`'h'`	Hexagram

Color	Description
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

u — Input signal for simulation
vector | array

Input signal for simulation, specified as a vector for single-input systems, and an array for multi-input systems.

- For single-input systems, `u` is a vector of the same length as `t`.
- For multi-input systems, `u` is an array with as many rows as there are time samples (`length(t)`) and as many columns as there are inputs to `sys`. In other words, each row `u(i,:)` represents the values applied at the inputs of `sys` at time `t(i)`. Each column `u(:,j)` is the signal applied to the `j`th input of `sys`.

t — Time samples
vector

Time samples at which to compute the response, specified as a vector of the form `0:dT:Tf`. The `lsimplot` command interprets `t` as having the units specified in the `TimeUnit` property of the model `sys`. The time vector must be real, finite, and must contain monotonically increasing and evenly spaced time samples.

For continuous-time systems, the `lsimplot` command uses the time step `dT` to discretize the model. If `dT` is too large relative to the system dynamics (undersampling), `lsimplot` issues a warning recommending a faster sampling time.

For discrete-time systems, the time step `dT` must equal the sample time of `sys`. Alternatively, you can omit `t` or set it to `[]`. In that case, `lsimplot` sets `t` to a vector of the same length as `u` that begins at 0 with a time step equal to `sys.Ts`.

`method` — Discretization method
`'zoh'` | `'foh'`

Discretization method for sampling continuous-time models, specified as one of the following.

- `'zoh'` — Zero-order hold
- `'foh'` — First-order hold

When `sys` is a continuous-time model, `lsimplot` computes the time response by discretizing the model using a sample time equal to the time step `dT = t(2)-t(1)` of `t`. If you do not specify a discretization method, then `lsimplot` selects the method automatically based on the smoothness of the signal `u`. For more information about these two discretization methods, see “Continuous-Discrete Conversion Methods” (Control System Toolbox).

`x0` — Initial state values
vector of zeros (default) | vector

Initial state values for simulating a state-space model, specified as a vector having one entry for each state in `sys`. If you omit this argument, then `lsim` sets all states to zero at `t = 0`.

`AX` — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes and if the current axes are Cartesian axes, then `stepplot` plots on the current axes. Use `AX` to plot into specific axes when creating a step plot.

`plotoptions` — Step plot options set
`TimePlotOptions` object

Step plot options set, specified as a `TimePlotOptions` object. You can use this option set to customize the step plot appearance. Use `timeoptions` to create the option set. Settings you specify in `plotoptions` overrides the preference settings in the MATLAB session in which you run `stepplot`. Therefore, `plotoptions` is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

For the list of available options, see `timeoptions`.

Output Arguments

`h` — Plot handle
`handle` object

Plot handle, returned as a `handle` object. Use the handle `h` to get and set the properties of the simulated response plot using `getoptions` and `setoptions`. For the list of available options, see the *Properties and Values Reference* section in “Customizing Response Plots from the Command Line” (Control System Toolbox).

See Also
`getoptions` | `setoptions` | `lsim` | `timeoptions`

Topics
“Customizing Response Plots from the Command Line” (Control System Toolbox)
“Working with the Linear Simulation Tool” (Control System Toolbox)
“Continuous-Discrete Conversion Methods” (Control System Toolbox)

Introduced in R2012a

mag2db

Convert magnitude to decibels (dB)

Syntax

```
ydb = mag2db(y)
```

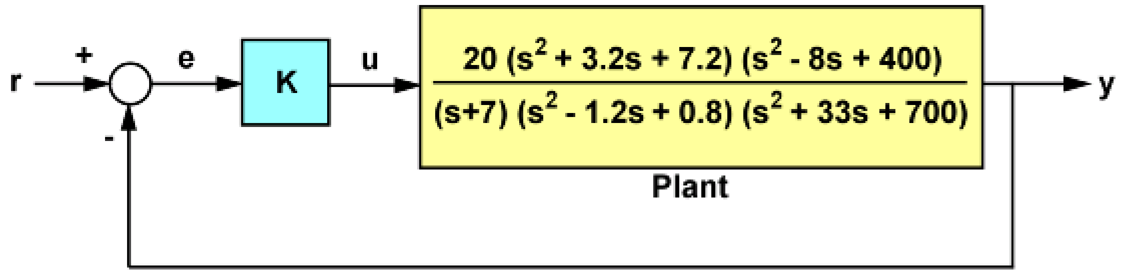
Description

`ydb = mag2db(y)` expresses in decibels (dB) the magnitude measurements specified in `y`. The relationship between magnitude and decibels is $ydb = 20 * \log_{10}(y)$

Examples

Display Gain Margins in Decibels

For this example, consider the following SISO feedback loop where the system contains multiple gain crossover or phase crossover frequencies, which leads to multiple gain or phase margin values:


r
+
-
e
K
u
$\frac{20(s^2+3.2s+7.2)(s^2-8s+400)}{(s+7)(s^2-1.2s+0.8)(s^2+33s+700)}$
Plant
y

SISO Feedback Loop

Create the transfer function.

```
G = tf(20,[1 7]) * tf([1 3.2 7.2],[1 -1.2 0.8]) * tf([1 -8 400],[1 33 700]);
```

Use the `allmargin` command to compute all stability margins.

```
m = allmargin(G)
```

```
m = struct with fields:
     GainMargin: [0.3408 3.3920]
    GMFrequency: [1.9421 16.4807]
    PhaseMargin: 68.1140
    PMFrequency: 7.0776
    DelayMargin: 0.1680
    DMFrequency: 7.0776
         Stable: 1
```

Note that gain margins are expressed as gain ratios and not in decibels (dB). Use `mag2db` to convert the values to dB.

```
GainMargins_dB = mag2db(m.GainMargin)
```

```
GainMargins_dB = 1×2

   -9.3510   10.6091
```

Input Arguments

y — Input array
scalar | vector | matrix | array

Input array, specified as a scalar, vector, matrix, or an array. When `y` is nonscalar, `mag2db` is an element-wise operation.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

ydb — Magnitude measurements in decibels
scalar | vector | matrix | array

Magnitude measurements in decibels, returned as a scalar, vector, matrix, or an array of the same size as `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)
```

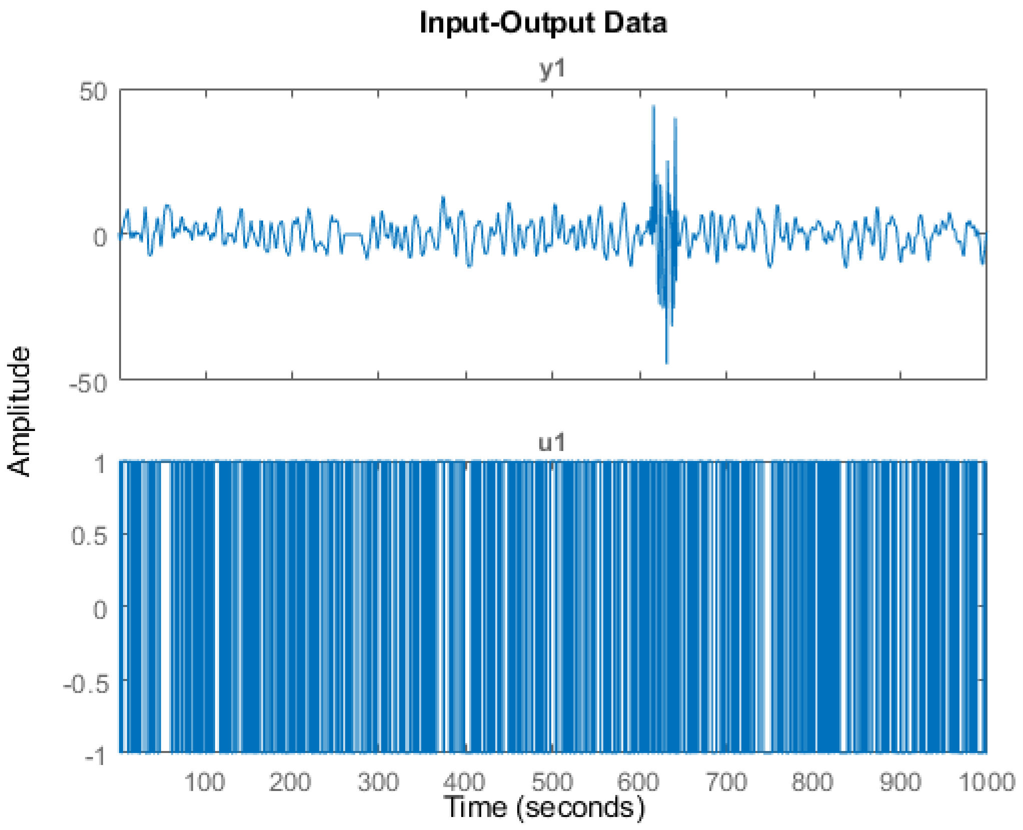
Storing multiple experiments as one `iddata` object can be very useful for handling experimental data that has been collected on different occasions, or when a data set has been split up to remove “bad” portions of the data. All the toolbox routines accept multiple-experiment data.

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);
```

See Also

`iddata` | `getexp` | `merge`

Topics

"Dealing with Multi-Experiment Data and Merging Models"
"Create Multiexperiment Data at the Command Line"

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,MaxIterations,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,MaxIterations,Tol)` specifies maximum number of iterations and tolerance. `MaxIterations` 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`.

Examples

Reconstruct Missing Data Using Specified Model

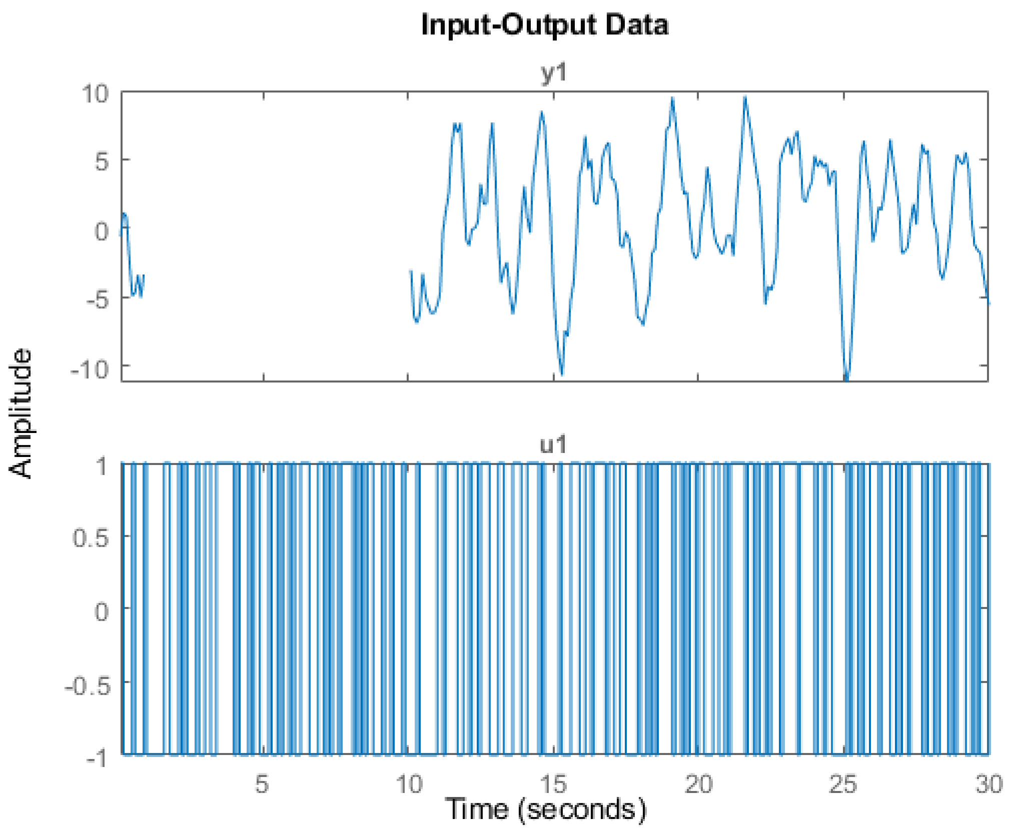
Load data with missing data points.

```
load('missing_data.mat')
```

`missing_data` is an `iddata` object containing input-output data.

Plot the data.

```
plot(missing_data)
```



The output data contains missing data between indices 10 and 100.

To reconstruct missing data using a specified model, estimate the model using measured data that has no missing samples. In this example, estimate a transfer function model with 2 poles.

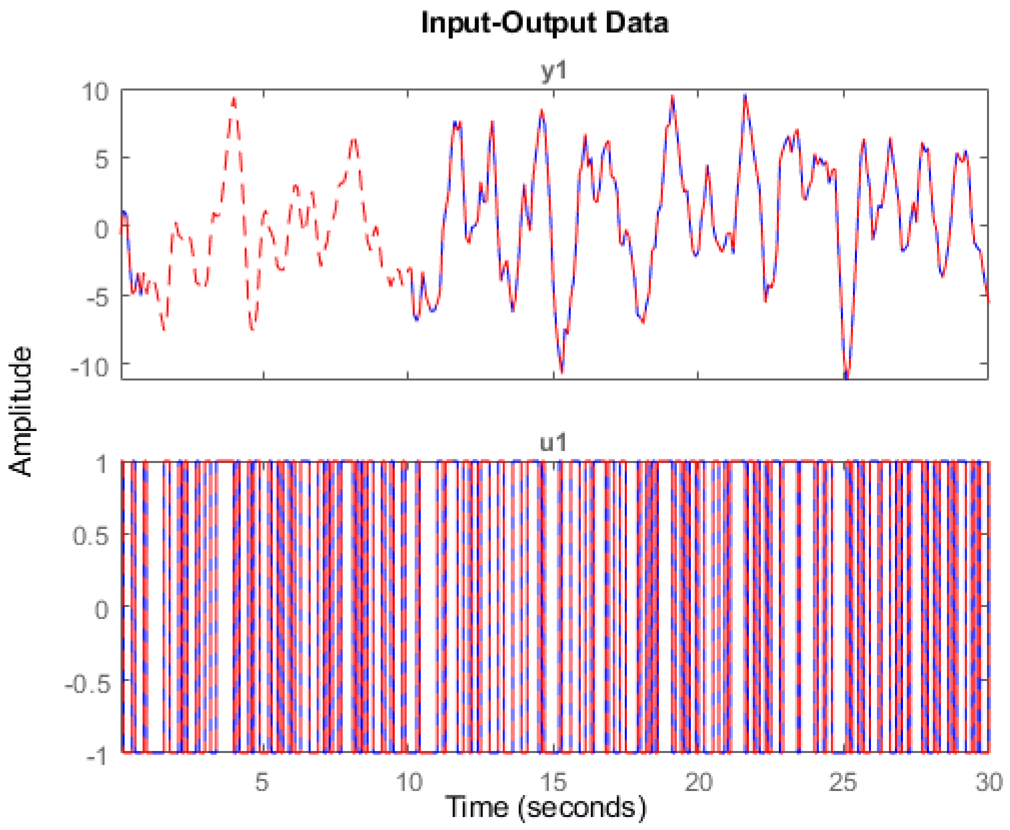
```
data2 = missing_data(101:end);
model = tfest(data2,2);
```

Reconstruct the missing data.

```
datae = misdata(missing_data,model);
```

Plot the original and reconstructed data.

```
plot(missing_data,'b',datae,'--r')
```



If you do not specify a model for reconstructing the data, the software alternates between estimating missing data and estimating models, based on the current data reconstruction.

Algorithms

For a given model, the missing data is estimated as parameters so as to minimize the output prediction errors obtained from the reconstructed data. See Section 14.2 in Ljung (1999). Treating missing outputs as parameters is not the best approach from a statistical point of view, but is a good approximation in many cases.

When no model is given, the algorithm alternates between estimating missing data and estimating models, based on the current reconstruction.

See Also

`arx` | `advice` | `pexcit` | `tfest`

Introduced before R2006a

n4sid

Estimate state-space model using subspace method with time-domain or frequency-domain data

Syntax

```
sys = n4sid(data,nx)
sys = n4sid(data,nx,Name,Value)
sys = n4sid(___,opt)
[sys,x0] = n4sid(___)
```

Description

`sys = n4sid(data,nx)` estimates a discrete-time state-space model `sys` of order `nx` using `data`, which can be time-domain or frequency-domain data. `sys` is a model of the following form:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$

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 entries of A, B, C, and K are free estimable parameters by default. For dynamic systems, D is fixed to zero by default, meaning that the system has no feedthrough. For static systems (`nx = 0`), D is an estimable parameter by default.

`sys = n4sid(data,nx,Name,Value)` incorporates additional options specified by one or more name-value pair arguments. For example, to estimate a continuous-time model, specify the sample time `'Ts'` as `0`. 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 the estimation options `opt`. These options can include the initial states, estimation objective, and subspace algorithm related choices to be used for estimation. Specify `opt` after any of the previous input-argument combinations.

`[sys,x0] = n4sid(___)` returns the value of initial states computed during estimation. You can use this syntax with any of the previous input-argument combinations.

Examples

State-Space Model

Estimate a state-space model and compare its response with the measured output.

Load the input-output data `z1`, which is stored in an `iddata` object.

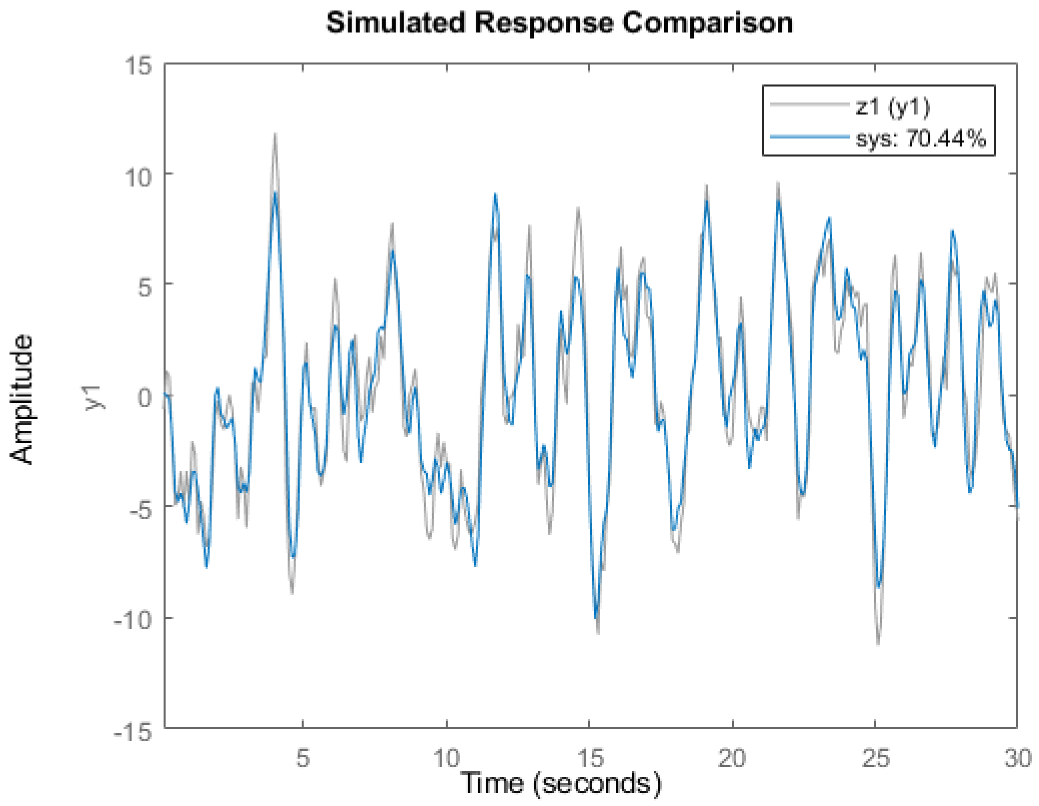
```
load iddata1 z1
```

Estimate a fourth-order state-space model.

```
nx = 4;
sys = n4sid(z1,nx)

sys =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
             x1         x2         x3         x4
   x1    0.8392    -0.3129    0.02105    0.03743
   x2    0.4768     0.6671     0.1428   0.003757
   x3  -0.01951    0.08374   -0.09761      1.046
   x4 -0.003885   -0.02914    -0.8796   -0.03171

  B =
             u1
   x1   0.02635
   x2  -0.03301
   x3 7.256e-05
   x4 0.0005861

  C =
           x1       x2       x3       x4
   y1    69.08    26.64   -2.237  -0.5601

  D =
       u1
   y1   0

  K =
             y1
   x1  0.003282
   x2  0.009339
   x3 -0.003232
   x4  0.003809

Sample time: 0.1 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 28
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using N4SID on time domain data "z1".
Fit to estimation data: 76.33% (prediction focus)
FPE: 1.21, MSE: 1.087
```

Compare the simulated model response with the measured output.

```
compare(z1,sys)
```



The plot shows that the fit percentage between the simulated model and the estimation data is greater than 70%.

You can view more information about the estimation by exploring the `idss` property `sys.Report`.

```
sys.Report
```

```
ans = 
          Status: 'Estimated using N4SID with prediction focus'
          Method: 'N4SID'
    InitialState: 'estimate'
        N4Weight: 'CVA'
       N4Horizon: [6 10 10]
             Fit: [1x1 struct]
      Parameters: [1x1 struct]
     OptionsUsed: [1x1 idoptions.n4sid]
       RandState: [1x1 struct]
        DataUsed: [1x1 struct]
```

For example, find out more information about the estimated initial state.

```
sys.Report.Parameters.X0
```

```
ans = 4×1

   -0.0085
    0.0052
   -0.0193
    0.0282
```

Determine Optimal Estimated Model Order

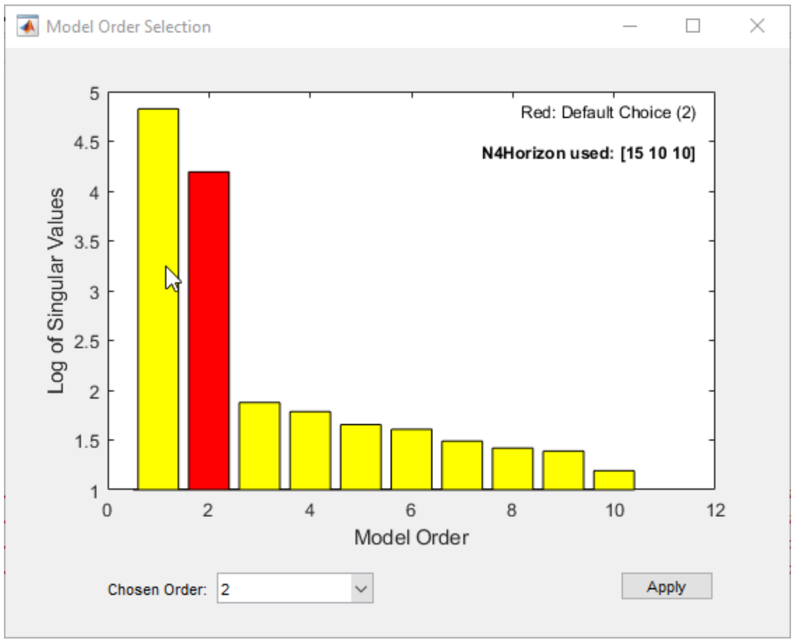
Load the input-output data `z1`, which is stored in an `iddata` object.

```
load iddata1 z1
```

Determine the optimal model order by specifying argument `nx` as a range from 1 to 10.

```
nx = 1:10;
sys = n4sid(z1,nx);
```

An automatically generated plot shows the Hankel singular values for models of the orders specified by `nx`.



States with relatively small Hankel singular values can be safely discarded. The suggested default order choice is 2.

Select the model order in the **Chosen Order** list and click **Apply**.

Specify Estimation Options

Load estimation data.

```
load iddata2 z2
```

Specify estimation options. Set the weighting scheme `'N4Weight'` to `'SSARX'` and estimation-status display option `'Display'` to `'on'`.

```
opt = n4sidOptions('N4Weight','SSARX','Display','on')
```

```
Option set for the n4sid command:

          InitialState: 'estimate'
              N4Weight: 'SSARX'
             N4Horizon: 'auto'
               Display: 'on'
           InputOffset: []
          OutputOffset: []
    EstimateCovariance: 1
          OutputWeight: []
                 Focus: 'prediction'
       WeightingFilter: []
      EnforceStability: 0
              Advanced: [1x1 struct]
```

Estimate a third-order state-space model using the updated option set.

```
nx = 3;
sys = n4sid(z2,nx,opt);
```

Modify Form, Feedthrough, and Disturbance-Model Matrices

Modify the canonical form of the A, B, and C matrices, include a feedthrough term in the D matrix, and eliminate disturbance-model estimation in the K matrix.

Load input-output data and estimate a fourth-order system using the `n4sid` default options.

```
load iddata1 z1
sys1 = n4sid(z1,4);
```

Specify the modal form and compare the `A` matrix with the default `A` matrix.

```
sys2 = n4sid(z1,4,'Form','modal');
A1 = sys1.A
```

```
A1 = 4×4

    0.8392   -0.3129    0.0211    0.0374
    0.4768    0.6671    0.1428    0.0038
   -0.0195    0.0837   -0.0976    1.0462
   -0.0039   -0.0291   -0.8796   -0.0317
```

```
A2 = sys2.A
```

```
A2 = 4×4

    0.7554    0.3779         0         0
   -0.3779    0.7554         0         0
         0         0   -0.0669    0.9542
         0         0   -0.9542   -0.0669
```

Include a feedthrough term and compare the `D` matrices.

```
sys3 = n4sid(z1,4,'Feedthrough',1);
D1 = sys1.D
```

```
D1 = 0
```

```
D3 = sys3.D
```

```
D3 = 0.0487
```

Eliminate disturbance modeling and compare the `K` matrices.

```
sys4 = n4sid(z1,4,'DisturbanceModel','none');
K1 = sys1.K
```

```
K1 = 4×1

    0.0033
    0.0093
   -0.0032
    0.0038
```

```
K4 = sys4.K
```

```
K4 = 4×1

     0
     0
     0
     0
```

Continuous-Time Canonical-Form Model

Estimate a continuous-time canonical-form model.

Load estimation data.

```
load iddata1 z1
```

Estimate the model. Set `Ts` to `0` to specify a continuous model.

```
nx = 2;
sys = n4sid(z1,nx,'Ts',0,'Form','canonical');
```

`sys` is a second-order continuous-time state-space model in the canonical form.

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 `'N4weight'` used by the N4SID algorithm. Create two option sets. For one option set, set `'N4weight'` to `'CVA'`. For the other option set, set the `'N4weight'` to `'SSARX'`.

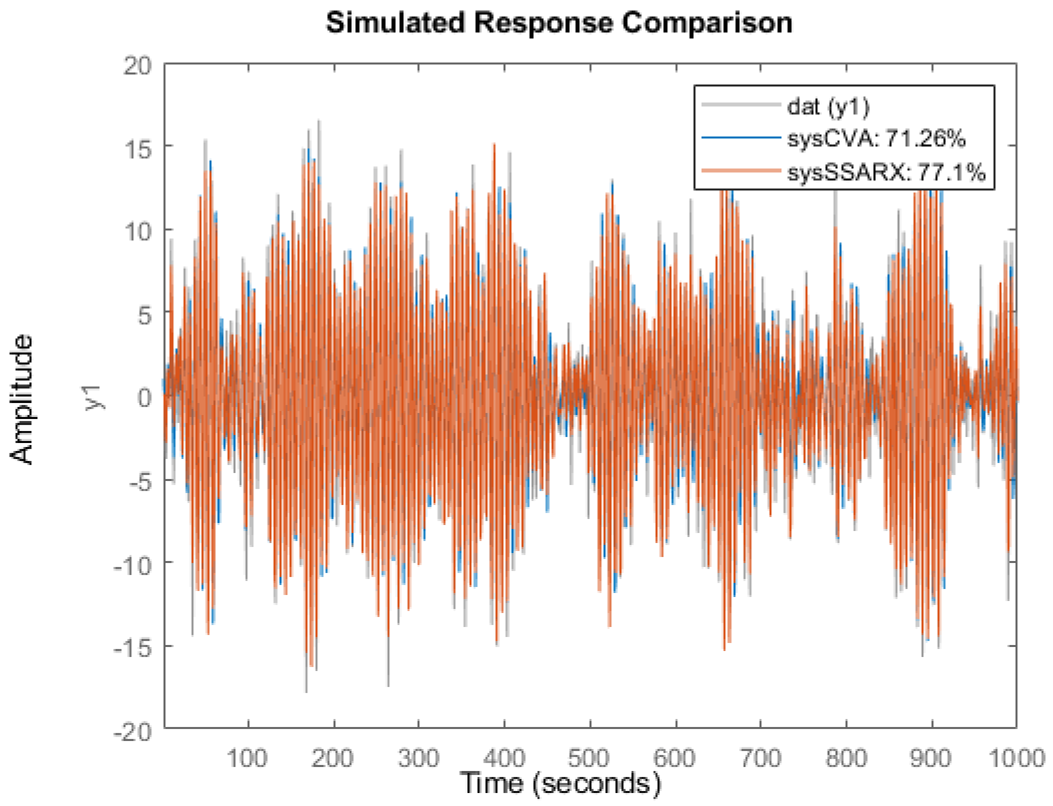
```
optCVA = n4sidOptions('N4weight','CVA');
optSSARX = n4sidOptions('N4weight','SSARX');
```

Estimate state-space models using the option 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);
```



As the plot shows, the model estimated using the SSARX algorithm produces a better fit than the model estimated using the CVA algorithm.

Input Arguments

data — Estimation data

`iddata` object | `frd` object | `idfrd` object

Estimation data, specified as an `iddata` object, an `frd` object, or an `idfrd` 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 properties specified as follows.
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — `'Frequency'`

Estimation data must be uniformly sampled. By default, the software sets the sample time of the model to the sample time of the estimation data.

For multiexperiment data, the sample times and intersample behavior of all the experiments must match.

The domain of your data determines the type of model you can estimate.

- Time-domain or discrete-time frequency-domain data — Continuous-time and discrete-time models
- Continuous-time frequency-domain data — Continuous-time models only

nx — Order of estimated model
`1:10` (default) | positive integer scalar | positive integer vector | `best` | `0`

Order of the estimated model, specified as a nonnegative integer or as a vector containing a range of positive integers.

- If you already know what order you want your estimated model to have, specify `nx` as a scalar.
- If you want to compare a range of potential orders to choose the most effective order for your estimated model, specify that range for `nx`. `n4sid` creates a Hankel singular-value plot that shows the relative energy contributions of each state in the system. States with relatively small Hankel singular values contribute little to the accuracy of the model and can be discarded with little impact. The index of the highest state you retain is the model order. The plot window includes a suggestion for the order to use. You can accept this suggestion or enter a different order. For an example, see "Determine Optimal Estimated Model Order" on page 1-962.

  If you do not specify `nx`, or if you specify `nx` as `best`, the software automatically chooses `nx` from the range `1:10`.
- If you are identifying a static system, set `nx` to `0`.

opt — Estimation options
`n4sidOptions` option set

Estimation options, specified as an `n4sidOptions` option set. Options specified by `opt` include:

- Estimation objective
- Handling of initial conditions
- Subspace algorithm-related choices

For examples showing how to specify options, see "Specify Estimation Options" on page 1-962 and "Continuous-Time Canonical-Form Model" on page 1-964.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sys = n4sid(data,nx,'Form','modal')`

Ts — Sample time of estimated model
sample time of data (`data.Ts`) (default) | `0` | positive scalar

Sample time of the estimated model, specified as the comma-separated pair consisting of `'Ts'` and either `0` or a positive scalar.

- For continuous-time models, specify `'Ts'` as `0`. In the frequency domain, using continuous-time frequency-domain data results in a continuous-time model.
- For discrete-time models, the software sets `'Ts'` to the sample time of the data in the units stored in the `TimeUnit` property.

`InputDelay` — Input delays
`0` (default) | scalar | vector

Input delay for each input channel, specified as the comma-separated pair consisting of `'InputDelay'` and a numeric vector.

- For continuous-time models, specify `'InputDelay'` in the time units stored in the `TimeUnit` property.
- For discrete-time models, specify `'InputDelay'` in integer multiples of the sample time `Ts`. For example, setting `'InputDelay'` to `3` specifies a delay of three sampling periods.

For a system with N_u inputs, set `InputDelay` to an N_u-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.

To apply the same delay to all channels, specify `'InputDelay'` as a scalar.

`Form` — Type of canonical form
`'free'` (default) | `'modal'` | `'companion'` | `'canonical'`

Type of canonical form of `sys`, specified as the comma-separated pair consisting of `'Form'` and one of the following values:

- `'free'` — Treat all entries of the matrices A, B, C, D, and K as free.
- `'modal'` — Obtain `sys` in modal form.
- `'companion'` — Obtain `sys` in companion form.
- `'canonical'` — Obtain `sys` in the observability canonical form.

For definitions of the canonical forms, see “Canonical State-Space Realizations”.

For more information about using these forms for identification, see “Estimate State-Space Models with Canonical Parameterization”.

For an example, see “Modify Form, Feedthrough, and Disturbance-Model Matrices” on page 1-963.

`Feedthrough` — Direct feedthrough from input to output
`0` (default) | `1` | logical vector

Direct feedthrough from input to output, specified as the comma-separated pair consisting of `'Feedthrough'` and a logical vector of length N_u, where N_u is the number of inputs. If you specify `'Feedthrough'` as a logical scalar, that value is applied to all the inputs. For static systems, the software always assumes `'Feedthrough'` is `1`.

For an example, see “Modify Form, Feedthrough, and Disturbance-Model Matrices” on page 1-963.

`DisturbanceModel` — Option to estimate time-domain noise component parameters
`'estimate'` | `'none'`

Option to estimate time-domain noise component parameters in the K matrix, specified as the comma-separated pair consisting of `'DisturbanceModel'` and one of the following values:

- `'estimate'` — Estimate the noise component. The *K* matrix is treated as a free parameter. For time-domain data, `'estimate'` is the default.
- `'none'` — Do not estimate the noise component. The elements of the *K* matrix are fixed at zero. For frequency-domain data, `'none'` is the default and the only acceptable value.

For an example, see “Modify Form, Feedthrough, and Disturbance-Model Matrices” on page 1-963.

Output Arguments

sys — Identified state-space model

`idss` model

Identified state-space model, returned as an `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.
`InitialState`	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 Variate 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'`.
`N4Horizon`	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.

Report Field	Description
`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: **Field** — **Description** `FitPercent` — Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE). `LossFcn` — Value of the loss function when the estimation completes. `MSE` — Mean squared error (MSE) measure of how well the response of the model fits the estimation data. `FPE` — Final prediction error for the model. `AIC` — Raw Akaike Information Criteria (AIC) measure of model quality. `AICc` — Small-sample-size corrected AIC. `nAIC` — Normalized AIC. `BIC` — Bayesian Information Criteria (BIC).
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If you did not configure any custom options, `OptionsUsed` is the set of default options. See `n4sidOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
DataUsed	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** | **Description** `Name` — Name of the data set. `Type` — Data type. `Length` — Number of data samples. `Ts` — Sample time. `InterSample` — Input intersample behavior, returned as one of the following values: • `'zoh'` — 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. `InputOffset` — Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`. `OutputOffset` — Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.

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

`x0` — Initial states computed during estimation
column vector | matrix

Initial states computed during the estimation, returned as an array containing a column vector corresponding to each experiment.

This array is also stored in the `Parameters` field of the model `Report` property.

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.

[5] McKelvey, T., H. Akcay, and L. Ljung. "Subspace-based multivariable system identification from frequency response data." *IEEE Transactions on Automatic Control*, 1996, Vol. 41, pp. 960–979.

See Also
`n4sidOptions` | `idss` | `ssest` | `tfest` | `procest` | `polyest` | `iddata` | `idfrd` | `idgrey` | `canon` | `pem`

Topics
"What Are State-Space Models?"
"Estimate State-Space Models at the Command Line"
"State-Space Model Estimation Methods"
"Estimate State-Space Models with Canonical Parameterization"

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 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 Variate 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`

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

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

If `EstimateCovariance` 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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

`n4sid` | `idpar` | `idfilt`

Topics
“Loss Function and Model Quality Metrics”

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* ≥ 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

Determine Dimensions of Model Array

Create a 3-*by*-1 array of random state-space models, each with 4 states, 1 input, and 1 output.

```
sys = rss(4,1,1,3);
```

Compute the number of dimensions of the model array.

```
ndims(sys)
```

```
ans = 4
```

The number of dimensions is 2+*p*, where *p* is the number of array dimensions. In this example, *p* is 2 because `sys` is 3-*by*-1.

See Also

`size`

Introduced in R2012a

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` | `delayest` | `idss`

Introduced before R2006a

nlarx

Estimate parameters of nonlinear ARX model

Syntax

```
sys = nlarx(data,orders)
sys = nlarx(data,regressors)
sys = nlarx( ___ ,output_fcn)

sys = nlarx(data,linmodel)
sys = nlarx(data,linmodel,output_fcn)

sys = nlarx(data,sys0)

sys = nlarx( ___ ,Options)
```

Description

Specify Regressors

`sys = nlarx(data,orders)` estimates a nonlinear ARX model to fit the given estimation data using the specified ARX model orders and the default wavelet network output function. Use this syntax when you extend an ARX linear model, or when you use only regressors that are linear with consecutive lags.

`sys = nlarx(data,regressors)` estimates a nonlinear ARX model using the specified regressor set `regressors`. Use this syntax when you have linear regressors that have non-consecutive lags, or when you also have polynomial regressors, custom regressors, or both.

`sys = nlarx( ___ ,output_fcn)` specifies the output function that maps the regressors to the model output. You can use this syntax with any of the previous input argument combinations.

Specify Linear 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. Use this syntax when you want to create a nonlinear ARX model as an extension of, or an improvement upon, an existing linear model. When you use this syntax, the software initializes the offset value to `0`. In some cases, you can improve the estimation results by overriding this initialization with the command `sys.OutputFcn.Offset.Value = NaN`.

`sys = nlarx(data,linmodel,output_fcn)` specifies the output function to use for model estimation.

Refine Existing Model

`sys = nlarx(data,sys0)` estimates or refines the parameters of the nonlinear ARX model `sys0`.

Use this syntax to:

- Estimate the parameters of a model previously created using the `idnlarx` constructor. Prior to estimation, you can configure the model properties using dot notation.

- 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.

Specify Options

`sys = nlarx(___,Options)` specifies additional configuration options for the model estimation.

Examples

Estimate Nonlinear ARX Model

Load the estimation data.

```
load twotankdata;
```

Create an `iddata` object from the estimation data with a sample time of 0.2 seconds.

```
Ts = 0.2;
z = iddata(y,u,Ts);
```

Estimate the nonlinear ARX model using ARX model orders to specify the regressors.

```
sysNL = nlarx(z,[4 4 1])
```

```
sysNL =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Wavelet network with 11 units
Sample time: 0.2 seconds

Status:
Estimated using NLARX on time domain data "z".
Fit to estimation data: 96.84% (prediction focus)
FPE: 3.482e-05, MSE: 3.431e-05
```

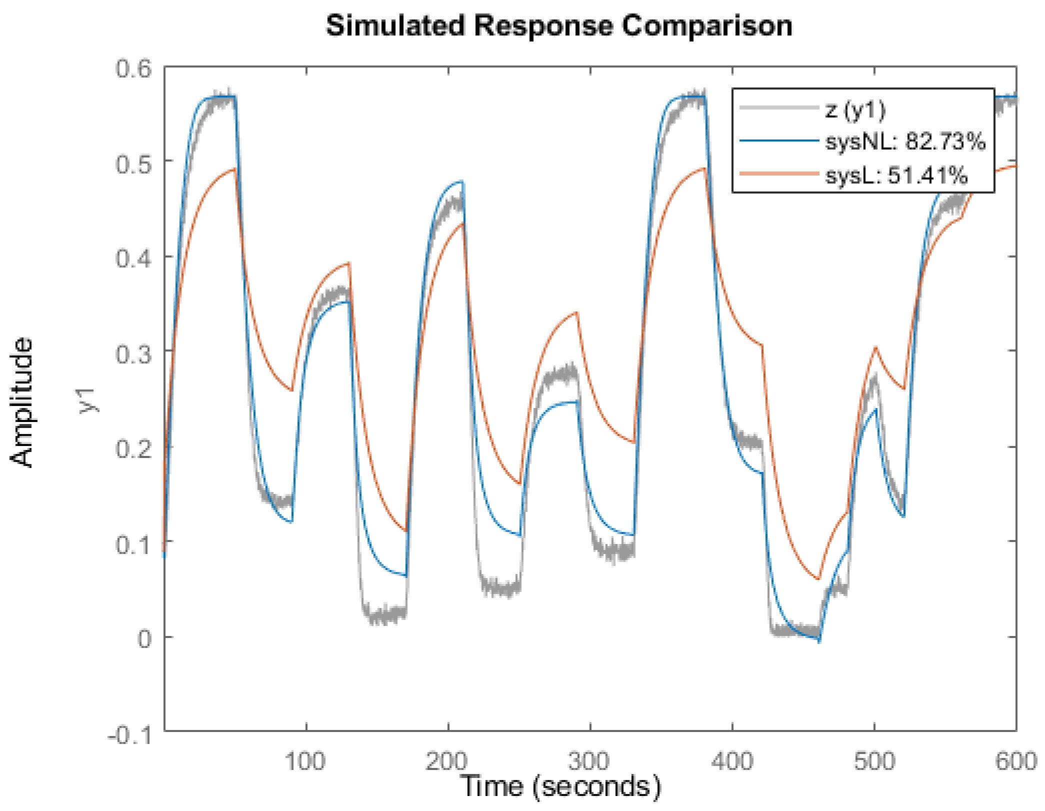
`sys` uses the default `idWaveletNetwork` function as the output function.

For comparison, compute a linear ARX model with the same model orders.

```
sysL = arx(z,[4 4 1]);
```

Compare the model outputs with the original data.

```
compare(z,sysNL,sysL)
```



The nonlinear model has a much better fit to the data than the linear model.

Estimate Nonlinear ARX Model Using Linear Regressor Set

Specify a linear regressor that is equivalent to an ARX model order matrix of `[4 4 1]`.

An order matrix of `[4 4 1]` specifies that both input and output regressor sets contain four regressors with lags ranging from 1 to 4. For example, $u_1(t-2)$ represents the second input regressor.

Specify the output and input names.

```
output_name = 'y1';
input_name = 'u1';
names = {output_name,input_name};
```

Specify the output and input lags.

```
output_lag = [1 2 3 4];
input_lag = [1 2 3 4];
lags = {output_lag,input_lag};
```

Create the linear regressor object.

```
lreg = linearRegressor(names,lags)
lreg =
Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1 2 3 4]  [1 2 3 4]}
     UseAbsolute: [0 0]
    TimeVariable: 't'

  Regressors described by this set
```

Load the estimation data and create an iddata object.

```
load twotankdata
z = iddata(y,u,0.2);
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,lreg)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Wavelet network with 11 units
Sample time: 0.2 seconds

Status:
Estimated using NLARX on time domain data "z".
Fit to estimation data: 96.84% (prediction focus)
FPE: 3.482e-05, MSE: 3.431e-05
```

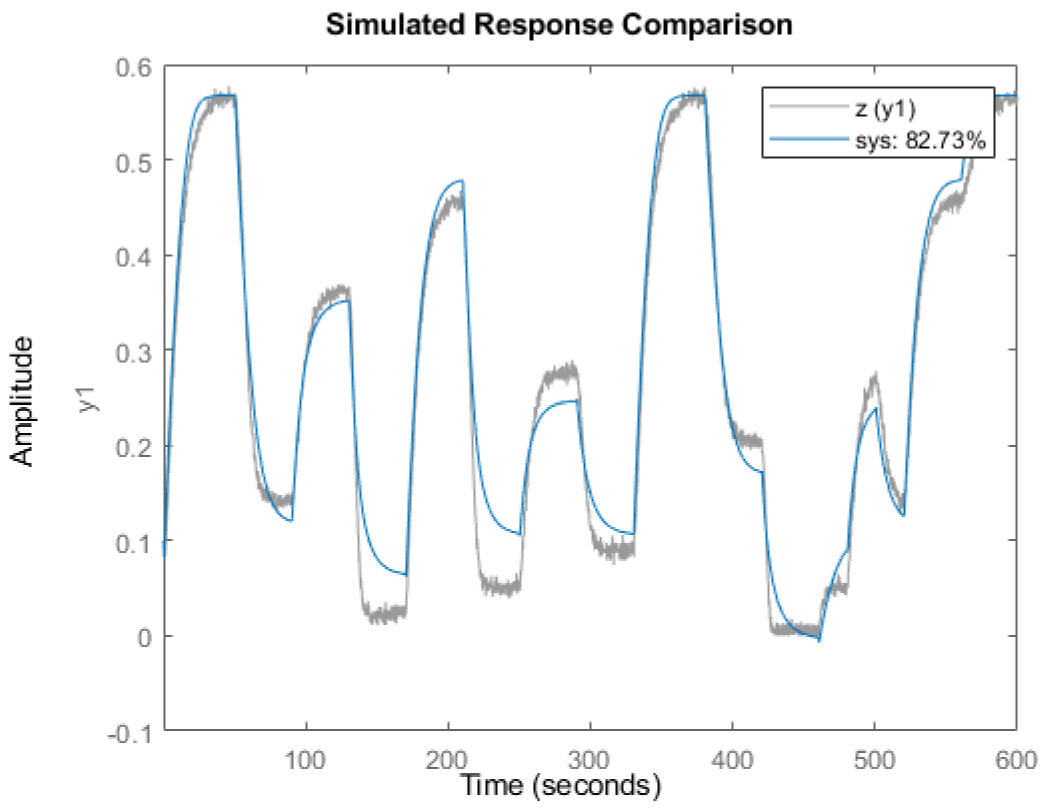
View the regressors

```
getreg(sys)
```

```
ans = 8x1 cell
    {'y1(t-1)'}
    {'y1(t-2)'}
    {'y1(t-3)'}
    {'y1(t-4)'}
    {'u1(t-1)'}
    {'u1(t-2)'}
    {'u1(t-3)'}
    {'u1(t-4)'}
```

Compare the model output to the estimation data.

```
compare(z,sys)
```



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)
```

```
sys =
Nonlinear time series model
  Outputs: y1

Regressors:
  Linear regressors in variables y1
  List of all regressors

Output function: Wavelet network with 8 units
```

```
Sample time: 0.01 seconds

Status:
Estimated using NLARX on time domain data "z".
Fit to estimation data: 92.92% (prediction focus)
FPE: 0.2568, MSE: 0.2507
```

Specify and Customize Output Function

Estimate a nonlinear ARX model that uses the mapping function `idSigmoidNetwork` as its output function.

Load the data and divide it into the estimation and validation data sets `ze` and `zv`.

```
load twotankdata.mat u y
z = iddata(y,u,'Ts',0.2);
ze = z(1:1500);
zv = z(1501:end);
```

Configure the `idSigmoidNetwork` mapping function. Fix the offset to 0.2 and the number of units to 15.

```
s = idSigmoidNetwork;
s.Offset.Value = 0.2;
s. NonlinearFcn.NumberOfUnits = 15;
```

Create a linear model regressor specification that contains four output regressors and five input regressors.

```
reg1 = linearRegressor({'y1','u1'},{1:4,0:4});
```

Create a polynomial model regressor specification that contains the squares of two input terms and three output terms.

```
reg2 = polynomialRegressor({'y1','u1'},{1:2,0:2},2);
```

Set estimation options for the search method and maximum number of iterations.

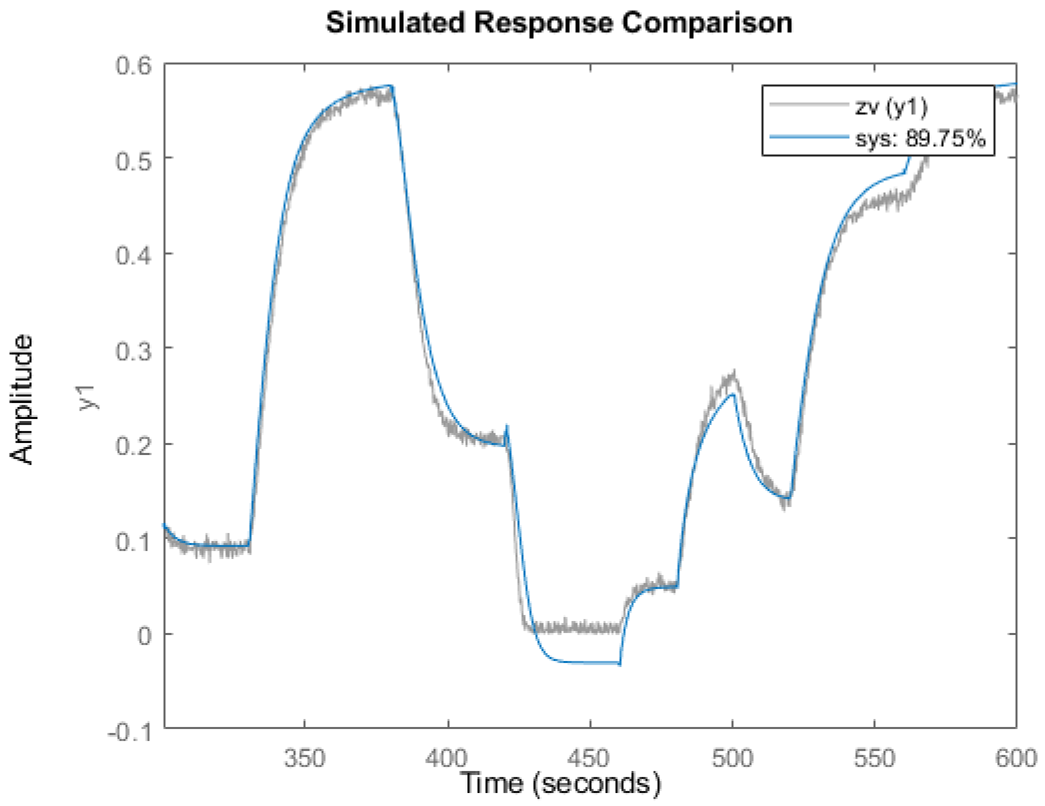
```
opt = nlarxOptions('SearchMethod','fmincon')';
opt.SearchOptions.MaxIterations = 40;
```

Estimate the nonlinear ARX model.

```
sys = nlarx(ze,[reg1;reg2],s,opt);
```

Validate `sys` by comparing the simulated model response to the validation data set.

```
compare(zv,sys)
```



Add Output Function to Extend and Improve Linear Model

Estimate a linear model and improve the model by adding a `treepartition` output function.

Load the estimation data.

```
load throttledata ThrottleData
```

Estimate a linear ARX model `linsys` with orders `[2 2 1]`.

```
linsys = arx(ThrottleData,[2 2 1]);
```

Create an `idnlarx` template model that uses `linsys` and specifies `sigmoidnet` as the output function.

```
sys0 = idnlarx(linsys,idTreePartition);
```

Fix the linear component of `sys0` so that during estimation, the linear portion of `sys0` remains identical to `linsys`. Set the offset component value to `NaN`.

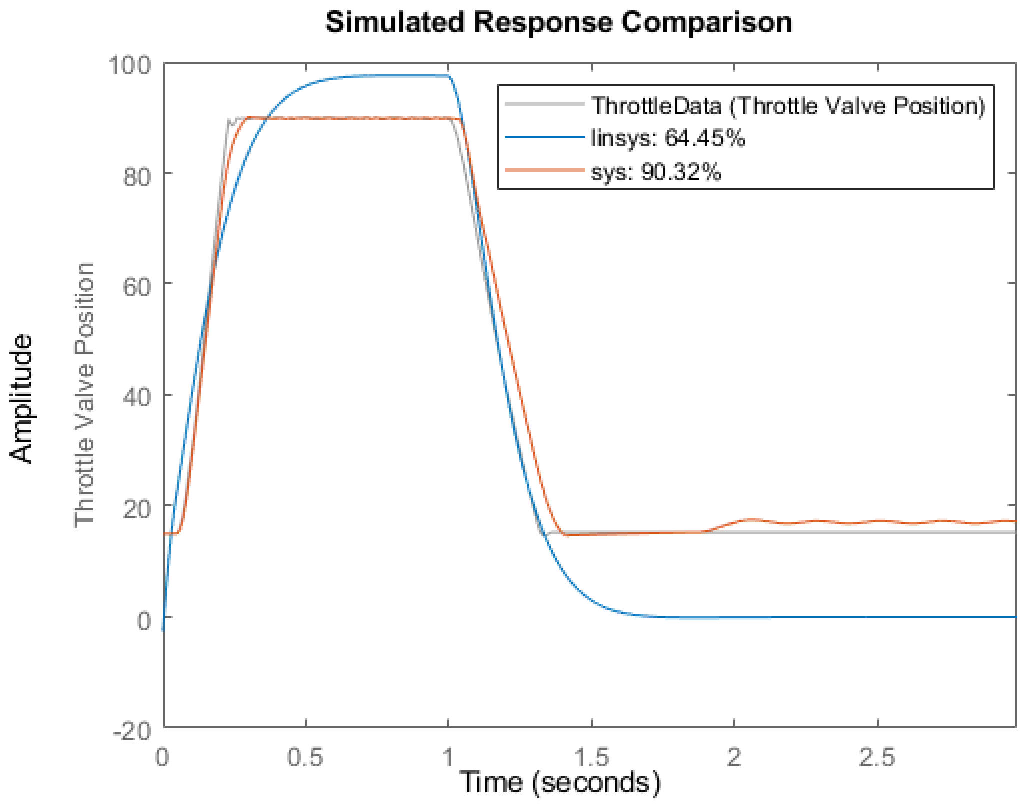
```
sys0.OutputFcn.LinearFcn.Free = false;
sys0.OutputFcn.Offset.Value = NaN;
```

Estimate the free parameters of `sys0`, which are the nonlinear-function parameters and the offset.

```
sys = nlarx(ThrottleData,sys0);
```

Compare the fit accuracies for the linear and nonlinear models.

```
compare(ThrottleData,linsys,sys)
```



Estimate Nonlinear ARX Model Using Custom Network Mapping Object

Generating a custom network mapping object requires the definition of a user-defined unit function.

Define the unit function and save it as `gaussunit.m`.

```
function [f,g,a] = gaussunit(x)
% Custom unit function nonlinearity.
%
% Copyright 2015 The MathWorks, Inc.
f = exp(-x.*x);
if nargout>1
  g = -2*x.*f;
  a = 0.2;
end
```

Create a custom network mapping object using a handle to the `gaussunit` function.

```
H = @gaussunit;
CNet = idCustomNetwork(H);
```

Load the estimation data.

```
load iddata1
```

Estimate a nonlinear ARX model using the custom network.

```
sys = nlarx(z1,[1 2 1],CNet)
```

```
sys = 

<strong>Nonlinear ARX model with 1 output and 1 input</strong>
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1

Output function: Custom Network with 10 units and "gaussunit" unit function
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 64.35% (prediction focus)
FPE: 3.58, MSE: 2.465
```

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 six inputs and two outputs.

Specify the model orders.

```
Orders = [ones(2,2),2*ones(2,6),ones(2,6)];
```

Specify different mapping functions for each output channel.

```
NL = [idWaveletNetwork(2),idLinear];
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,Orders,NL)
```

```
sys =
Nonlinear ARX model with 2 outputs and 6 inputs
  Inputs: u1, u2, u3, u4, u5, u6
  Outputs: y1, y2
```

```
Regressors:
  Linear regressors in variables y1, y2, u1, u2, u3, u4, u5, u6
  List of all regressors

Output functions:
  Output 1:  Wavelet network with 2 units
  Output 2:  None

Sample time: 0.02 seconds

Status:
Estimated using NLARX on time domain data "Motorized Camera".
Fit to estimation data: [98.72;98.77]% (prediction focus)
FPE: 0.5719, MSE: 1.061
```

Estimate MIMO Nonlinear ARX Model with Same Mapping Function for All Outputs

Load the estimation data and create an `iddata` object `z`. `z` contains two output channels and six input channels.

```
load motorizedcamera;
z = iddata(y,u,0.02);
```

Specify a set of linear regressors that uses the output and input names from `z` and contains:

- 2 output regressors with 1 lag.
- 6 input regressor pairs with 1 and 2 lags.

```
names = [z.OutputName; z.InputName];
lags = {1,1,[1,2],[1,2],[1,2],[1,2],[1,2],[1,2]};
reg = linearRegressor(names,lags);
```

Estimate a nonlinear ARX model using an `idSigmoidNetwork` mapping function with four units for all output channels.

```
sys = nlarx(z,reg,idSigmoidNetwork(4))
```

```
sys =
Nonlinear ARX model with 2 outputs and 6 inputs
  Inputs: u1, u2, u3, u4, u5, u6
  Outputs: y1, y2

Regressors:
  Linear regressors in variables y1, y2, u1, u2, u3, u4, u5, u6
  List of all regressors

Output functions:
  Output 1:  Sigmoid network with 4 units
  Output 2:  Sigmoid network with 4 units

Sample time: 0.02 seconds

Status:
Estimated using NLARX on time domain data "z".
```

```
Fit to estimation data: [98.86;98.79]% (prediction focus)
FPE: 2.641, MSE: 0.9233
```

Specify Linear, Polynomial, and Custom Regressors

Load the estimation data `z1`, which has one input and one output, and obtain the output and input names.

```
load iddata1 z1;
names = [z1.OutputName z1.InputName]
```

```
names = 1x2 cell
    {'y1'}    {'u1'}
```

Specify `L` as the set of linear regressors that represents $y_1(t-1)$, $u_1(t-2)$, and $u_1(t-5)$.

```
L = linearRegressor(names,{1,[2 5]});
```

Specify `P` as the polynomial regressor $y_1(t-1)^2$.

```
P = polynomialRegressor(names(1),1,2);
```

Specify `C` as the custom regressor $y_1(t-2)u_1(t-3)$. Use an anonymous function handle to define this function.

```
C = customRegressor(names,{2 3},@(x,y)x.*y)
```

```
C =
Custom regressor: y1(t-2).*u1(t-3)
    VariablesToRegressorFcn: @(x,y)x.*y
                  Variables: {'y1'  'u1'}
                       Lags: {[2]  [3]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

Combine the regressors in the column vector `R`.

```
R = [L;P;C]
```

```
R =
[3 1] array of linearRegressor, polynomialRegressor, customRegressor objects.
------------------------------------
1. Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1]  [2 5]}
     UseAbsolute: [0 0]
    TimeVariable: 't'

------------------------------------
2. Order 2 regressors in variables y1
               Order: 2
           Variables: {'y1'}
```

```
                Lags: {[1]}
         UseAbsolute: 0
    AllowVariableMix: 0
         AllowLagMix: 0
        TimeVariable: 't'

------------------------------------
3. Custom regressor: y1(t-2).*u1(t-3)
    VariablesToRegressorFcn: @(x,y)x.*y
                  Variables: {'y1'  'u1'}
                       Lags: {[2]  [3]}
                 Vectorized: 1
               TimeVariable: 't'


Regressors described by this set
```

Estimate a nonlinear ARX model with R.

```
sys = nlarx(z1,R)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables y1
  3. Custom regressor: y1(t-2).*u1(t-3)
  List of all regressors

Output function: Wavelet network with 1 units
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 59.73% (prediction focus)
FPE: 3.356, MSE: 3.147
```

View the full regressor set.

```
getreg(sys)
```

```
ans = 5x1 cell
    {'y1(t-1)'         }
    {'u1(t-2)'         }
    {'u1(t-5)'         }
    {'y1(t-1)^2'       }
    {'y1(t-2).*u1(t-3)'}
```

Estimate Nonlinear ARX Model with No Linear Term in Output Function

Load the estimation data.

```
load iddata1;
```

Create a sigmoid network mapping object with 10 units and no linear term.

```
SN = idSigmoidNetwork(10,false);
```

Estimate the nonlinear ARX model. Confirm that the model does not use the linear function.

```
sys = nlarx(z1,[2 2 1],SN);
sys.OutputFcn.LinearFcn.Use
```

```
ans = logical
   0
```

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)
```

```
NonlinearModel =
Nonlinear ARX model with 1 output and 1 input
  Inputs: Step Command
  Outputs: Throttle Valve Position

Regressors:
  Linear regressors in variables Throttle Valve Position, Step Command
  List of all regressors

Output function: Wavelet network with 12 units
Sample time: 0.01 seconds

Status:
Estimated using NLARX on time domain data "ThrottleData".
Fit to estimation data: 65.67% (prediction focus)
FPE: 145.7, MSE: 130
```

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:

- Use a sigmoid network mapping object.
- Assign a name.

```
sys.Nonlinearity = 'idSigmoidNetwork';
sys.Name = 'Model 1';
```

Estimate a nonlinear ARX model with the structure and properties specified in the `idnlarx` object.

```
sys = nlarx(z1,sys)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Sigmoid network with 10 units
Name: Model 1
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 69.03% (prediction focus)
FPE: 2.918, MSE: 1.86
```

Estimate Nonlinear ARX Model and Avoid Local Minima

If an estimation stops at a local minimum, you can perturb the model using `init` and re-estimate the model.

Load the estimation data.

```
load iddata1;
```

Estimate the initial nonlinear model.

```
sys1 = nlarx(z1,[4 2 1],'idSigmoidNetwork');
```

Randomly perturb the model parameters to avoid local minima.

```
sys2 = init(sys1);
```

Estimate the new nonlinear model with the perturbed values.

```
sys2 = nlarx(z1,sys1);
```

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 10 estimation iterations.

```
opt = nlarxOptions;
opt.Focus = 'simulation';
opt.SearchOptions.MaxIterations = 10;
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,[4 4 1],idSigmoidNetwork(3),opt)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Sigmoid network with 3 units
Sample time: 0.2 seconds

Status:
Estimated using NLARX on time domain data "z".
Fit to estimation data: 85.86% (simulation focus)
FPE: 3.791e-05, MSE: 0.0006853
```

Estimate Regularized Nonlinear ARX Model with Large Number of Units

Load the regularization example data.

```
load regularizationExampleData.mat nldata;
```

Create an `idSigmoidnetwork` mapping object with 30 units and specify the model orders.

```
MO = idSigmoidNetwork(30);
Orders = [1 2 1];
```

Create an estimation option set and set the estimation search method to `lm`.

```
opt = nlarxOptions('SearchMethod','lm');
```

Estimate an unregularized model.

```
sys = nlarx(nldata,Orders,MO,opt);
```

Configure the regularization `Lambda` parameter.

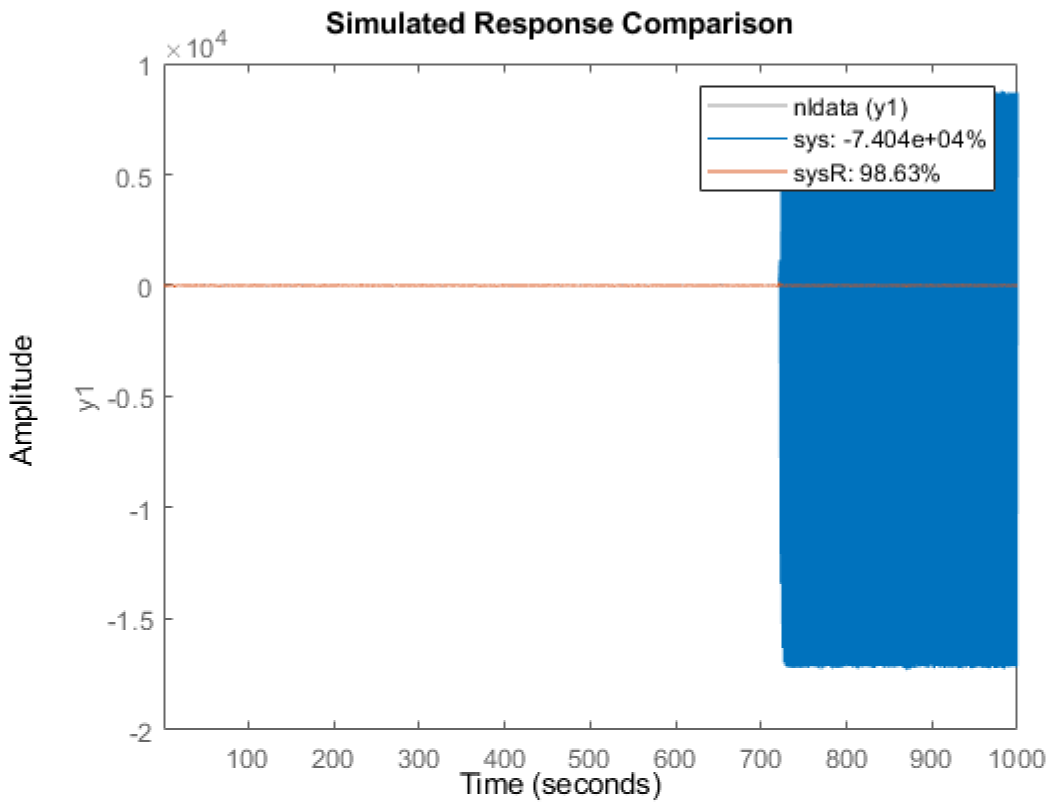
```
opt.Regularization.Lambda = 1e-8;
```

Estimate a regularized model.

```
sysR = nlarx(nldata,Orders,MO,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.

Input Arguments

`data` — Time-domain estimation data

`iddata` object | numeric matrix

Time-domain estimation data, specified as an `iddata` object or a numeric matrix.

- If `data` is an `iddata` object, then `data` can have one or more output channels and zero or more input channels.
- If `data` is a numeric matrix, then the number of columns of data must match the sum of the number of inputs (n_u) and the number of outputs (n_y.

`data` must be uniformly sampled and cannot contain missing (`NaN`) samples.

`orders` — ARX model orders

`nlarx` orders `[na nb nk]`

ARX model orders, specified as the matrix `[na nb nk]`. `na` denotes the number of delayed outputs, `nb` denotes the number of delayed inputs, and `nk` denotes the minimum input delay. The minimum output delay is fixed to `1`. For more information on how to construct the `orders` matrix, see `arx`.

When you specify `orders`, the software converts the order information into linear regressor form in the `idnlarx` `Regressors` property. For an example, see “Create Nonlinear ARX Model Using ARX Model Orders” on page 1-614.

`regressors` — Regressor specification

`linearRegressor` object | `polynomialRegressor` object | `customRegressor` object | column array of regressor specification objects

Regressor specification, specified as a column vector containing one or more regressor specification objects, which are the `linearRegressor` objects, `polynomialRegressor` objects, and `customRegressor` objects. Each object specifies a formula for generating regressors from lagged variables. For example:

- `L = linearRegressor({'y1','u1'},{1,[2 5]})` generates the regressors $y_1(t–1)$, $u_1(t–2)$, and $u_2(t–5)$.
- `P = polynomialRegressor('y2',4:7,2)` generates the regressors $y_2(t–4)^2$, $y_2(t–5)^2$, $y_2(t–6)^2$, and $y_2(t–7)^2$.
- `C = customRegressor({'y1','u1','u2'},{1 2 2},@(x,y,z)sin(x.*y+z))` generates the single regressor $\sin(y_1(t–1)u_1(t–2)+u_2(t–2))$.

When you create a regressor set to support estimation with an `iddata` object, you can use the input and output names of the object rather than create the names for the regressor function. For instance, suppose you create a linear regressor for a model, plan to use the `iddata` object `z` to estimate the model. You can use the following command to create the linear regressor.

```
L = linearRegressor([z.outputName;z.inputName],{1,[2 5]})
```

For an example of creating and using a SISO linear regressor set, see “Estimate Nonlinear ARX Model Using Linear Regressor Set” on page 1-982. For an example of creating a MIMO linear regressor set that obtains variable names from the estimation data set, see “Estimate MIMO Nonlinear ARX Model with Same Mapping Function for All Outputs” on page 1-989.

`output_fcn` — Output function

`'idWaveletNetwork'` (default) | `'idLinear'` | `[]` | `''` | `'idSigmoidNetwork'` | `'idTreePartition'` | `'idFeedforwardNetwork'` | `'idCustomNetwork'` | `'idTreePartition'` | `'idGaussianProcess'` | `'idTreeEnsemble'` | mapping object | array of mapping objects

Output function that maps the regressors of the `idnlarx` model into the model output, specified as a column array containing zero or more of the following strings or objects:

`'idWaveletNetwork'` or `idWaveletNetwork` object	Wavelet network
`'linear'` or `''` or `[]` or `idLinear` object	Linear function
`'idSigmoidNetwork'` or `idSigmoidNetwork` object	Sigmoid network
`'idTreePartition'` or `idTreePartition` object	Binary tree partition regression model
`'idGaussianProcess'` or `idGaussianProcess` object	Gaussian process regression model (requires Statistics and Machine Learning Toolbox)
`'idTreeEnsemble'` or `idTreeEnsemble`	Regression tree ensemble model requires (Statistics and Machine Learning Toolbox)
`idFeedforwardNetwork` object	Neural network — Feedforward network of Deep Learning Toolbox.
`idCustomNetwork` object	Custom network — Similar to `idSigmoidNetwork`, but with a user-defined replacement for the sigmoid function.

Use a string, such as `'idSigmoidNetwork'`, to use the default properties of the mapping function object. Use the object itself, such as `idSigmoidNetwork`, when you want to configure the properties of the mapping object.

The `idWaveletNetwork`, `idSigmoidNetwork`, `idTreePartition`, and `idCustomNetwork` objects contain both linear and nonlinear components. You can remove (not use) the linear components of `idWaveletNetwork`, `idSigmoidNetwork`, and `idCustomNetwork` by setting the `LinearFcn.Use` value to `false`.

The `idFeedforwardNetwork` function has only a nonlinear component, which is the `network` object of Deep Learning Toolbox. The `idLinear` object, as the name implies, has only a linear component.

`output_fcn` is static in that it depends only upon the data values at a specific time, but not directly on time itself. For example, if the output function $y(t)$ is equal to $y_0 + a_1\, y(t–1) + a_2\, y(t–2) + \dots\, b_1\, u(t–1) + b_2\, u(t–2) + \dots$, then `output_fcn` is a linear function that the `linear` mapping object represents.

Specifying a character vector, for example `'idSigmoidNetwork'`, creates a mapping object with default settings. Alternatively, you can specify mapping object properties in two ways:

- Create the mapping object using arguments to modify default properties.

  ```
  MO = idSigmoidNetwork(15);
  ```
- Create a default mapping object first and then use dot notation to modify properties.

  ```
  MO = idSigmoidNetwork;
  MO.NumberOfUnits = 15;
  ```

For n_y output channels, you can specify mapping objects individually for each channel by setting `output_fcn` to an array of n_y mapping objects. For example, the following code specifies `OutputFcn` using dot notation for a system with two input channels and two output channels.

```
sys = idnlarx({'y1','y2'},{'u1','u2'});
sys.OutputFcn = [idWaveletNetwork; idSigmoidNetwork];
```

To specify the same mapping for all outputs, specify `OutputFcn` as a character vector or a single mapping object.

`output_fcn` represents a static mapping function that transforms the regressors of the nonlinear ARX model into the model output. `output_fcn` is static because it does not depend on time. For example, if $y(t) = y_0 + a_1y(t-1) + a_2y(t-2) + \ldots + b_1u(t-1) + b_2u(t-2) + \ldots$, then `output_fcn` is a linear function represented by the `idLinear` object.

For an example of specifying the output function, see “Specify and Customize Output Function” on page 1-985.

`linmodel` — Discrete-time linear model
`idpoly` object | `idss` object | `idtf` object | `idproc` object

Discrete-time identified input/output linear model, specified as any linear model created using an estimator such as `arx`, `armax`, `tfest`, or `ssest`. For example, to create a state-space `idss` model, estimate the model using `ssest`.

`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 trapping the model in local minima.

  ```
  sys = init(sys);
  sys = nlarx(data,sys);
  ```
- A model previously created using the `idnlarx` constructor and with properties set using dot notation. For example, use the following to create an idnlarx object, set its properties, and estimate the model.

  ```
  sys1 = idnlarx('y1','u1',Regressors);
  sys1.OutputFcn = 'idTreePartition';
  sys1.Ts = 0.02;
  sys1.TimeUnit = 'Minutes';
  sys1.InputName = 'My Data';
  sys2 = nlarx(data,sys1);
  ```

  The preceding code is equivalent to the following nlarx command.

  ```
  sys2 = nlarx(data,Regressors,'idTreePartition','Ts',0.02,'TimeUnit','Minutes', ...
  'InputName','My Data');
  ```

`Options` — Estimation options
`nlarxOptions` option set

Estimation options for nonlinear ARX model identification, specified as an `nlarxOptions` option set.

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:

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.
`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:
	Field — **Description**
	`FitPercent` — Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
	`LossFcn` — Value of the loss function when the estimation completes.
	`MSE` — Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
	`FPE` — Final prediction error for the model.
	`AIC` — Raw Akaike Information Criteria (AIC) measure of model quality.
	`AICc` — Small-sample-size corrected AIC.
	`nAIC` — Normalized AIC.
	`BIC` — Bayesian Information Criteria (BIC).
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `nlarxOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

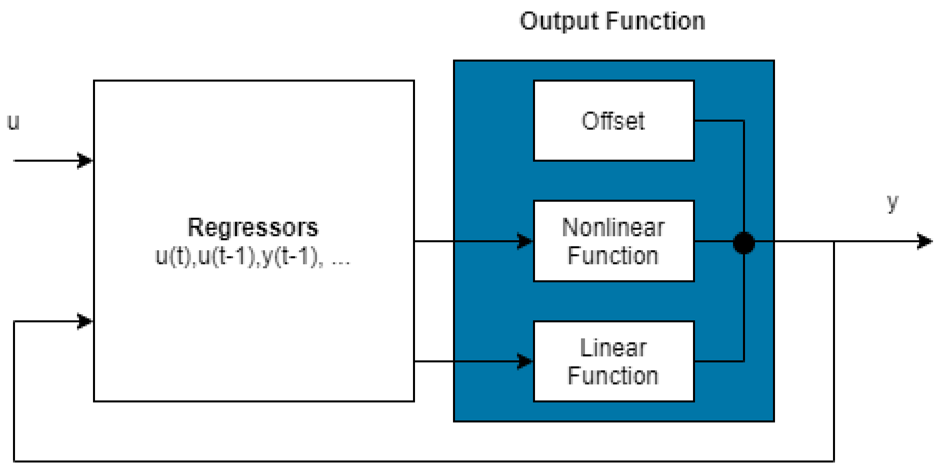
Report Field	Description
DataUsed	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** — **Description** Name — Name of the data set. Type — Data type. Length — Number of data samples. Ts — Sample time. InterSample — Input intersample behavior, returned as one of the following values: • 'zoh' — 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. InputOffset — Offset removed from time-domain input data during estimation. For nonlinear models, it is []. OutputOffset — Offset removed from time-domain output data during estimation. For nonlinear models, it is [].
Termination	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** — **Description** WhyStop — Reason for terminating the numerical search. Iterations — Number of search iterations performed by the estimation algorithm. FirstOrderOptimality — ∞-norm of the gradient search vector when the search algorithm terminates. FcnCount — Number of times the objective function was called. UpdateNorm — Norm of the gradient search vector in the last iteration. Omitted when the search method is 'lsqnonlin' or 'fmincon'. LastImprovement — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is 'lsqnonlin' or 'fmincon'. Algorithm — Algorithm used by 'lsqnonlin' or 'fmincon' search method. Omitted when other search methods are used. For estimation methods that do not require numerical search optimization, the Termination field is omitted.

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

Algorithms

Nonlinear ARX Model Structure

A nonlinear ARX model consists of model regressors and an output function. The output function includes linear and nonlinear functions that act on the model regressors to give the model output and a fixed offset for that output. This block diagram represents the structure of a nonlinear ARX model in a simulation scenario.



The software computes the nonlinear ARX model output *y* in two stages:

1. It computes regressor values from the current and past input values and the past output data.

   In the simplest case, regressors are delayed inputs and outputs, such as *u*(*t*–1) and *y*(*t*–3). These kind of regressors are called linear regressors. You specify linear regressors using the `linearRegressor` object. You can also specify linear regressors by using linear ARX model orders as an input argument. For more information, see “Nonlinear ARX Model Orders and Delay”. However, this second approach constrains your regressor set to linear regressors with consecutive delays. To create polynomial regressors, use the `polynomialRegressor` object. You can also specify custom regressors, which are nonlinear functions of delayed inputs and outputs. For example, *u*(*t*–1)*y*(*t*–3) is a custom regressor that multiplies instances of input and output together. Specify custom regressors using the `customRegressor` object.

   You can assign any of the regressors as inputs to the linear function block of the output function, the nonlinear function block, or both.
2. It maps the regressors to the model output using an output function block. The output function block can include linear and nonlinear blocks in parallel. For example, consider the following equation:

   $$F(x) = L^T(x-r) + g(Q(x-r)) + d$$

   Here, x is a vector of the regressors, and r is the mean of x. $F(x) = L^T(x-r) + y_0$ is the output of the linear function block. $g(Q(x-r)) + y_0$ represents the output of the nonlinear function block. Q is a projection matrix that makes the calculations well-conditioned. d is a scalar offset that is added to the combined outputs of the linear and nonlinear blocks. The exact form of $F(x)$ depends on your choice of output function. You can select from the available mapping objects, such as tree-partition networks, wavelet networks, and multilayer neural networks. You can also exclude either the linear or the nonlinear function block from the output function.

When estimating a nonlinear ARX model, the software computes the model parameter values, such as *L*, *r*, *d*, *Q*, and other parameters specifying *g*.

The resulting nonlinear ARX models are `idnlarx` objects that store all model data, including model regressors and parameters of the output function. For more information about these objects, see “Nonlinear Model Structures”.

Compatibility Considerations

Use of previous `idnlarx` and `idnlhw` mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the `idnlarx` and `idnlhw` objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
`wavenet`	`idWaveletNetwork`
`sigmoidnet`	`idSigmoidNetwork`
`treepartition`	`idTreePartition`
`customnet`	`idCustomNetwork`
`saturation`	`idSaturation`
`deadzone`	`idDeadZone`
`pwlinear`	`idPiecewiseLinear`
`poly1d`	`idPolynomial1D`
`unitgain`	`idUnitGain`
`linear`	`idLinear`
`neuralnet`	`idFeedforwardNetwork`

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

Use of previous `idnlarx` properties is not recommended.
Not recommended starting in R2021a

Starting in R2021a, several properties of `idnlarx` have been modified or replaced.

These changes affect the syntaxes in both `idnlarx` and `nlarx`. The use of the pre-R2021a properties in the following table is discouraged. However, the software still accepts calling syntaxes that include these properties. There are no plans to exclude these syntaxes at this time. The command syntax that uses ARX model orders continues be a recommended syntax.

Pre-R2021a Property	R2021a Property	Usage
ARX model orders `na,nb,nk`	`Regressors`, which can include `linearRegressor`, `polynomialRegressor`, and `customRegressor` objects.	`na,nb,nk` remains a valid `idnlarx` and `nlarx` input argument that the software converts to a `linearRegressor` object. You can no longer change order values in an existing `idnlarx` model by dot assignment or by using the `set` function. Create a new model object instead.
`customRegressors`	`Regressors`	Use `polynomialRegressor` or `customRegressor` to create regressor objects and add the objects to the `Regressors` array.
`NonlinearRegressors`	`RegressorUsage`	`RegressorUsage` is a table that contains regressor assignments to linear and nonlinear output components. Change assignments by modifying the corresponding `RegressorUsage`table entries.
`Nonlinearity`	`OutputFcn`	Change is in name only. Property remains an object or an array or objects that map regressor inputs to an output.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `nlarxOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`, as in the following example.

```
opt = nlarxOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

`idnlarx` | `nlarxOptions` | `isnlarx` | `goodnessOfFit` | `aic` | `fpe` | `polynomialRegressor` | `linearRegressor`

Topics

"Estimate Nonlinear ARX Models at the Command Line"
"Estimate Nonlinear ARX Models Initialized Using Linear ARX Models"

"Identifying Nonlinear ARX Models"
"Validate Nonlinear ARX Models"
"Using Nonlinear ARX Models"
"Loss Function and Model Quality Metrics"
"Regularized Estimates of Model Parameters"
"Estimation Report"

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.SearchOptions.MaxIterations = 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('IterativeWavenet','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 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.

`IterativeWavenet` — Iterative `idWaveletNetwork` estimation setting
`'auto'` (default) | `'on'` | `'off'`

Iterative `idWaveletNetwork` estimation setting, specified as the comma-separated pair consisting of `'IterativeWavenet'` 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 an `idWaveletNetwork` 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.
`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)`.
`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 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'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.

- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Options set for the search algorithm
search option set

Options set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`:

`SearchOptions` Structure When `SearchMethod` is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Tolerance`	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.	`1e-5`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see "Large-Scale vs. Medium-Scale Algorithms" (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see "Constrained Nonlinear Optimization Algorithms" (Optimization Toolbox) and "Choosing the Algorithm" (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	1e-6
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar.	1e-6
MaxIterations	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`.	100

To specify field values in `SearchOptions`, 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.SearchOptions.MaxIter = 15;
opt.SearchOptions.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
ErrorThres hold	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 `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.

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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.SearchOptions.MaxIterations = 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);
```

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`.

If you specify the `InterSample` property of `data` as `'bl'`(band-limited) and the model is continuous-time, the software treats data as first-order-hold (foh) interpolated for estimation.

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:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
`Method`	Name of the simulation solver and the search method used during estimation.
`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:

Field	Description
`FitPercent`	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
`LossFcn`	Value of the loss function when the estimation completes.
`MSE`	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
`FPE`	Final prediction error for the model.
`AIC`	Raw Akaike Information Criteria (AIC) measure of model quality.
`AICc`	Small-sample-size corrected AIC.
`nAIC`	Normalized AIC.
`BIC`	Bayesian Information Criteria (BIC).

Report Field	Description
Parameters	Estimated values of the model parameters. Structure with the following fields:

Field	Description
InitialValues	Structure with values of parameters and initial states before estimation.
ParVector	Value of parameters after estimation.
Free	Logical vector specifying the fixed or free status of parameters during estimation
FreeParCovariance	Covariance of the free parameters.
X0	Value of initial states after estimation.
X0Covariance	Covariance of the initial states.

Report Field	Description
OptionsUsed	Option set used for estimation. If no custom options were configured, this is a set of default options. See `nlgreyestOptions` for more information.
RandState	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.
DataUsed	Attributes of the data used for estimation — Structure with the following fields:

Field	Description
Name	Name of the data set.
Type	Data type — For `idnlgrey` models, this is set to `'Time domain data'`.
Length	Number of data samples.
Ts	Sample time. This is equivalent to `data.Ts`.
InterSample	Input intersample behavior. One of the following values: • `'zoh'` — 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. The value of `Intersample` has no effect on estimation results for discrete-time models.
InputOffset	Empty, `[]`, for nonlinear estimation methods.
OutputOffset	Empty, `[]`, for nonlinear estimation methods.

Report Field	Description
Termination	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** — **Description** WhyStop — Reason for terminating the numerical search. Iterations — Number of search iterations performed by the estimation algorithm. FirstOrderOptimality — ∞-norm of the gradient search vector when the search algorithm terminates. FcnCount — Number of times the objective function was called. UpdateNorm — Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. LastImprovement — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. Algorithm — Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used. For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.

For more information, see “Estimation Report”.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `nlgreyestOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = nlgreyestOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

`idnlgrey` | `nlgreyestOptions` | `pem` | `goodnessOfFit` | `aic` | `fpe`

Topics

“Creating IDNLGREY Model Files”
“Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation”
“Estimate Nonlinear Grey-Box Models”
“Loss Function and Model Quality Metrics”
“Regularized Estimates of Model Parameters”

“Estimation Report”

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.SearchOptions.MaxIterations = 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('EstimateCovariance',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 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
`MaxDifference`	Largest allowed parameter perturbation when computing numerical derivatives. Specified as a positive real value > `'MinDifference'`.	`Inf`
`MinDifference`	Smallest allowed parameter perturbation when computing numerical derivatives. Specified as a positive real value <`'MaxDifference'`.	`0.01*sqrt(eps)`

Field Name	Description	Default
DifferenceScheme	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'
Type	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. • 'Refined' — Simultaneously computes all numerical derivatives required to form each column of the Jacobian.	'Auto'

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.Type = 'Basic';
```

EstimateCovariance — 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 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.	`'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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for MIMO model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other available search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

SearchOptions Structure When SearchMethod Is Specified as 'lsqnonlin' or 'auto', When Optimization Toolbox Is Available

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of FunctionTolerance is the same as that of opt.SearchOptions.Advanced.TolFun.	1e-5
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of StepTolerance is the same as that of opt.SearchOptions.Advanced.TolX.	1e-6
MaxIterations	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance. The value of MaxIterations is the same as that of opt.SearchOptions.Advanced.MaxIter.	20
Advanced	Advanced search settings, specified as an option set for lsqnonlin. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use optimset('lsqnonlin') to create a default option set.

`SearchOptions` Structure When `SearchMethod` Is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`, When Optimization Toolbox Is Not Available

Field Name	Description	Default
`Tolerance`	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.	`1e-5`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until either `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	1e-6
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar.	1e-6
MaxIterations	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance.	100

To specify field values in SearchOptions, 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.SearchOptions.MaxIterations = 50;
opt.SearchOptions.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.

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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 = idSaturation('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`.

```
function [f,g,a] = gaussunit(x)
% Custom unit function nonlinearity.
%
% Copyright 2015 The MathWorks, Inc.
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 = idCustomNetwork(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,[],idSaturation);
```

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],[],'idWaveletNetwork');
sys0.B{1} = [0.8,1];
sys0.F{1} = [1,-1.2,0.5];
```

Estimate the model.

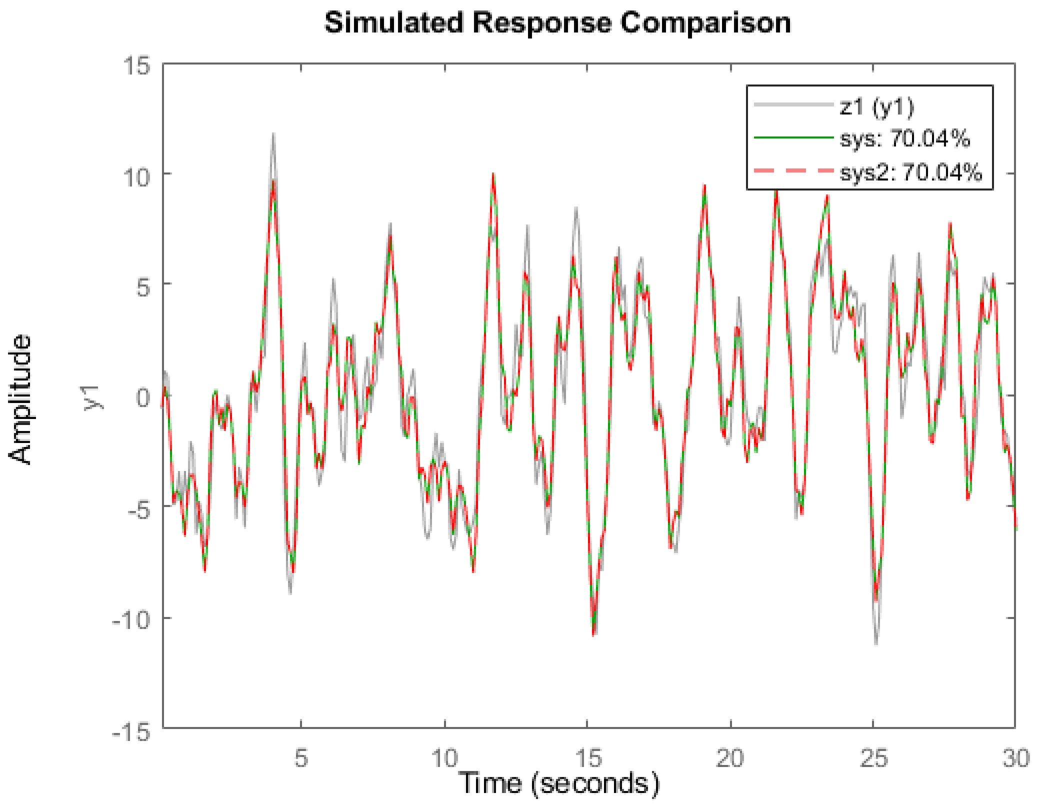
```
sys = nlhw(z1,sys0);
```

Estimate a Hammerstein-Wiener model using `nlhw` to define the model properties `B` and `F`.

```
sys2 = nlhw(z1,[2,2,0],[],'idWaveletNetwork','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],'idSigmoidNetwork','idWaveletNetwork');
```

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, use the Levenberg-Marquardt search method, and set the maximum iteration steps to `50`.

```
opt = nlhwOptions;
opt.Display = 'on';
opt.SearchMethod = 'lm';
opt.SearchOptions.MaxIterations = 50;
```

Load data and estimate the model.

```
load iddata3
sys = nlhw(z3,[4 2 1],idSigmoidNetwork,idPiecewiseLinear,opt);
```

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

`'idPiecewiseLinear'` (default) | `'idSigmoidNetwork'` | `'idWaveletNetwork'` | `'idSaturation'` | `'idDeadZone'` | `'idPolynomial1D'` | `'idUnitGain'` | nonlinearity estimator object | array of nonlinearity estimators

Input static nonlinearity estimator, specified as one of the following.

`'idPiecewiseLinear'` or `idPiecewiseLinear` object (default)	Piecewise linear function
`'idSigmoidNetwork'` or `idSigmoidNetwork` object	Sigmoid network
`'idWaveletNetwork'` or `idWaveletNetwork` object	Wavelet network
`'idSaturation'` or `idSaturation` object	Saturation
`'idDeadZone'` or `idDeadZone` object	Dead zone
`'idPolynomial1D'` or `idPolynomial1D` object	One-dimensional polynomial
`'idUnitGain'` or `[]` or`idUnitGain` object	Unit gain
`idCustomNetwork` object	Custom network — Similar to `idSigmoidNetwork`, but with a user-defined replacement for the sigmoid function.

Specifying a character vector, for example `'idSigmoidNetwork'`, creates a nonlinearity estimator object with default settings. Use object representation instead to configure the properties of a nonlinearity estimator.

```
InputNL = idWaveletNetwork;
InputNL.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
InputNL = idWaveletNetwork('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 = [idSigmoidNetwork('NumberofUnits',5); idDeadZone([-1,2])]
```

To specify the same nonlinearity for all inputs, specify a single input nonlinearity estimator.

`OutputNL` — Output static nonlinearity
`'idPiecewiseLinear'` (default) | `'idSigmoidNetwork'` | `'idWaveletNetwork'` | `'idSaturation'` | `'idDeadZone'` | `'idPolynomial1D'` | `'idUnitGain'` | nonlinearity estimator object | array of nonlinearity estimators

Output static nonlinearity estimator, specified as one of the following:

`'idPiecewiseLinear'` or `idPiecewiseLinear` object (default)	Piecewise linear function
`'idSigmoidNetwork'` or `idSigmoidNetwork` object	Sigmoid network
`'idWaveletNetwork'` or `idWaveletNetwork` object	Wavelet network
`'idSaturation'` or `idSaturation` object	Saturation
`'idDeadZone'` or `idDeadZone` object	Dead zone
`'idPolynomial1D'` or `idPolynomial1D` object	One-dimensional polynomial
`'idUnitGain'` or `[]` or`idUnitGain` object	Unit gain
`idCustomNetwork` object	Custom network — Similar to `idSigmoidNetwork`, 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 = idSigmoidNetwork;
OutputNL.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
OutputNL = idSigmoidNetwork('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.

Report Field	Description
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: **Field** — **Description** `FitPercent` — Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE). `LossFcn` — Value of the loss function when the estimation completes. `MSE` — Mean squared error (MSE) measure of how well the response of the model fits the estimation data. `FPE` — Final prediction error for the model. `AIC` — Raw Akaike Information Criteria (AIC) measure of model quality. `AICc` — Small-sample-size corrected AIC. `nAIC` — Normalized AIC. `BIC` — Bayesian Information Criteria (BIC).
Parameters	Estimated values of model parameters.
OptionsUsed	Option set used for estimation. If no custom options were configured, this is a set of default options. See `nlhwOptions` for more information.
RandState	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** | **Description** `Name` — Name of the data set. `Type` — Data type. `Length` — Number of data samples. `Ts` — Sample time. `InterSample` — Input intersample behavior, returned as one of the following values: • `'zoh'` — 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. `InputOffset` — Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`. `OutputOffset` — Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.
`Termination`	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** | **Description** `WhyStop` — Reason for terminating the numerical search. `Iterations` — Number of search iterations performed by the estimation algorithm. `FirstOrderOptimality` — ∞-norm of the gradient search vector when the search algorithm terminates. `FcnCount` — Number of times the objective function was called. `UpdateNorm` — Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `LastImprovement` — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `Algorithm` — Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used. For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.

For more information, see “Estimation Report”.

Compatibility Considerations

Use of previous idnlarx and idnlhw mapping object names is not recommended.
Not recommended starting in R2021b

Starting in R2021b, the mapping objects (also known as nonlinearities) used in the nonlinear components of the idnlarx and idnlhw objects have been renamed. The following table lists the name changes.

Pre-R2021b Name	R2021b Name
wavenet	idWaveletNetwork
sigmoidnet	idSigmoidNetwork
treepartition	idTreePartition
customnet	idCustomNetwork
saturation	idSaturation
deadzone	idDeadZone
pwlinear	idPiecewiseLinear
poly1d	idPolynomial1D
unitgain	idUnitGain
linear	idLinear
neuralnet	idFeedforwardNetwork

Scripts with the old names still run normally, although they will produce a warning. Consider using the new names for continuing compatibility with newly developed features and algorithms. There are no plans to exclude the use of these object names at this time

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the lsqnonlin search method (requires Optimization Toolbox). To enable parallel computing, use nlhwOptions, set SearchMethod to 'lsqnonlin', and set SearchOptions.Advanced.UseParallel to true.

For example:

```
opt = nlhwOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also
idnlhw | nlhwOptions | idnlhw/findop | linapp | linearize | pem | init | oe | tfest | n4sid | goodnessOfFit | aic | fpe

Topics
“Estimate Multiple Hammerstein-Wiener Models”
“Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models”

"Identifying Hammerstein-Wiener Models"
"Initialize Hammerstein-Wiener Estimation Using Linear Model"
"Loss Function and Model Quality Metrics"
"Regularized Estimates of Model Parameters"
"Estimation Report"

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, use the Levenberg-Marquardt search method, and set the maximum iteration steps to `50`.

```
opt = nlhwOptions;
opt.Display = 'on';
opt.SearchMethod = 'lm';
opt.SearchOptions.MaxIterations = 50;
```

Load data and estimate the model.

```
load iddata3
sys = nlhw(z3,[4 2 1],idSigmoidNetwork,idPiecewiseLinear,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 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.

Field Name	Description	Default
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' | 'fmincon'

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- 'gn' — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than GnPinvConstant*eps*max(size(J))*norm(J) are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 InitialGnaTolerance (see Advanced in 'SearchOptions' 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:
  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOptions` Structure When `SearchMethod` is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Tolerance`	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.	`1e-5`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	`1e-6`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`.	`100`

To specify field values in `SearchOptions`, 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.SearchOptions.MaxIterations = 50;
opt.SearchOptions.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
`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.
`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.

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$
$$e(t) = Lv(t)$$

$v(t)$ 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 = 4×1

    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`

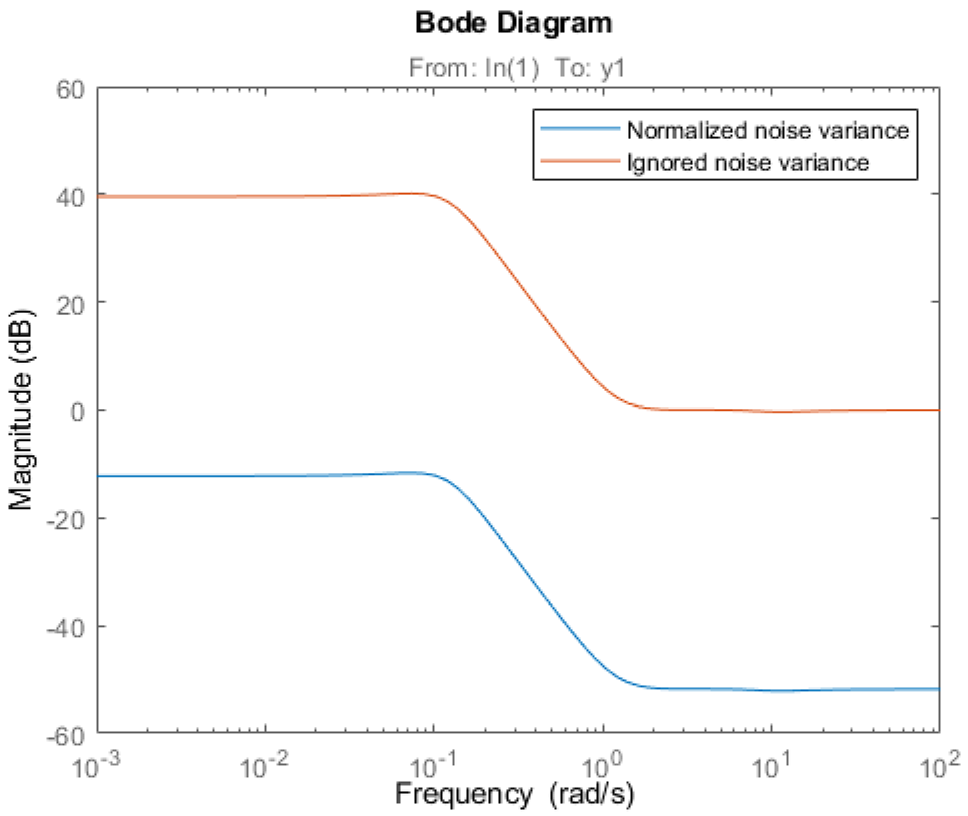
```
noise_model_normalize.B
```

```
ans = 4×1

    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

`noisecnv` | `tfdata` | `zpkdata` | `idssdata` | `spectrum`

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

`noise2meas` | `tfdata` | `zpkdata` | `idssdata`

Topics

"Treating Noise Channels as Measured Inputs"

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)
[n,fpeak] = norm(sys,Inf,tol)
```

Description

`n = norm(sys)` or `n = norm(sys,2)` returns the root-mean-squares of the impulse response of the linear dynamic system model `sys`. This value is equivalent to the H_2 norm on page 1-1061 of `sys`.

`n = norm(sys,Inf)` returns the L_∞ norm (Control System Toolbox) of `sys`, which is the peak gain of the frequency response of `sys` across frequencies. For MIMO systems, this quantity is the peak gain over all frequencies and all input directions, which corresponds to the peak value of the largest singular value of `sys`. For stable systems, the L_∞ norm is equivalent to the H_∞ norm. For more information, see `hinfnorm`.

`[n,fpeak] = norm(sys,Inf)` also returns the frequency `fpeak` at which the gain reaches its peak value.

`[n,fpeak] = norm(sys,Inf,tol)` sets the relative accuracy of the L_∞ norm to `tol`.

This command requires a Control System Toolbox license.

Examples

Compute Norm of Discrete-Time Linear System

Compute the H_2 and L_∞ norms of the following discrete-time transfer function, with sample time 0.1 second.

$$sys(z) = \frac{z^3 - 2.841z^2 + 2.875z - 1.004}{z^3 - 2.417z^2 + 2.003z - 0.5488}.$$

Compute the H_2 norm of the transfer function. The H_2 norm is the root-mean-square of the impulse response of `sys`.

```
sys = tf([1 -2.841 2.875 -1.004],[1 -2.417 2.003 -0.5488],0.1);
n2 = norm(sys)
```

```
n2 = 1.2438
```

Compute the L_∞ norm of the transfer function.

```
[ninf,fpeak] = norm(sys,Inf)
```

```
ninf = 2.5721
```

```
fpeak = 3.0178
```

Because sys is a stable system, `ninf` is the peak gain of the frequency response of `sys`, and `fpeak` is the frequency at which the peak gain occurs. Confirm these values using `getPeakGain`.

```
[gpeak,fpeak] = getPeakGain(sys)
```

```
gpeak = 2.5721
```

```
fpeak = 3.0178
```

Input Arguments

sys — Dynamic system
dynamic system model | model array

Input dynamic system, specified as any SISO or MIMO linear dynamic system model or model array. `sys` can be continuous-time or discrete-time.

`tol` — Relative accuracy
0.01 (default) | positive real scalar

Relative accuracy of the H_∞ norm, specified as a positive real scalar value.

Output Arguments

n — H_2 or L_∞ norm
scalar | array

H_2 norm or L_∞ norm of `sys`, returned as a scalar or an array.

- If `sys` is a single model, then `n` is a scalar value.
- If `sys` is a model array, then `n` is an array of the same size as `sys`, where `n(k) = norm(sys(:,:,k))`.

fpeak — Frequency of peak gain
real scalar | array of real values

Frequency at which the gain achieves the peak value `gpeak`, returned as a real scalar value or an array of real values. The frequency is expressed in units of rad/`TimeUnit`, relative to the `TimeUnit` property of `sys`.

- If `sys` is a single model, then `fpeak` is a scalar.
- If `sys` is a model array, then `fpeak` is an array of the same size as `sys`, where `fpeak(k)` is the peak gain frequency of `sys(:,:,k)`.

`fpeak` can be negative for systems with complex coefficients.

More About

H2 norm

The H_2 norm of a stable system H is 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 of a continuous-time system with transfer function $H(s)$ is given by:

$$\|H\|_2 = \sqrt{\frac{1}{2\pi}\int_{-\infty}^{\infty} \mathrm{Trace}\left[H(j\omega)^H H(j\omega)\right] d\omega}\,.$$

For a discrete-time system with transfer function $H(z)$, the H_2 norm is given by:

$$\|H\|_2 = \sqrt{\frac{1}{2\pi}\int_{-\pi}^{\pi} \mathrm{Trace}\left[H(e^{j\omega})^H H(e^{j\omega})\right] d\omega}\,.$$

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 $\omega = \infty$).

Using `norm(sys)` produces the same result as `sqrt(trace(covar(sys,1)))`.

L-infinity norm

The L_∞ norm of a SISO linear system is the peak gain of the frequency response. For a MIMO system, the L_∞ norm is the peak gain across all input/output channels.

For a continuous-time system $H(s)$, this definition means:

$$\|H(s)\|_{L_\infty} = \max_{\omega \in R} |H(j\omega)| \qquad \text{(SISO)}$$

$$\|H(s)\|_{L_\infty} = \max_{\omega \in R} \sigma_{\max}(H(j\omega)) \qquad \text{(MIMO)}$$

where $\sigma_{\max}(\cdot)$ denotes the largest singular value of a matrix.

For a discrete-time system $H(z)$, the definition means:

$$\|H(z)\|_{L_\infty} = \max_{\theta \in [0, 2\pi]} \left|H\left(e^{j\theta}\right)\right| \qquad \text{(SISO)}$$

$$\|H(z)\|_{L_\infty} = \max_{\theta \in [0, 2\pi]} \sigma_{\max}\left(H\left(e^{j\theta}\right)\right) \qquad \text{(MIMO)}$$

For stable systems, the L_∞ norm is equivalent to the H_∞ norm. For more information, see `hinfnorm`. For a system with unstable poles, the H_∞ norm is infinite. For all systems, `norm` returns the L_∞ norm, which is the peak gain without regard to system stability.

Algorithms

After converting `sys` to a state space model, `norm` uses the same algorithm as `covar` for the H_2 norm. For the L_∞ norm, `norm` uses the algorithm of [1]. `norm` computes the peak gain using the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] Bruinsma, N.A., and M. Steinbuch. "A Fast Algorithm to Compute the H_∞ Norm of a Transfer Function Matrix." *Systems & Control Letters*, 14, no.4 (April 1990): 287–93.

See Also

`freqresp` | `sigma` | `getPeakGain` | `hinfnorm`

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

`size` | `idpoly` | `idss` | `idtf` | `idproc` | `idgrey` | `idfrd`

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(sys1,sys2,...,sysN)
nyquist(sys1,LineSpec1,...,sysN,LineSpecN)
nyquist( ___ ,w)
```

```
[re,im,wout] = nyquist(sys)
[re,im,wout] = nyquist(sys,w)
[re,im,wout,sdre,sdim] = nyquist(sys,w)
```

Description

`nyquist(sys)` creates a Nyquist plot of the frequency response of a dynamic system model `sys`. The plot displays real and imaginary parts of the system response as a function of frequency.

`nyquist` plots a contour comprised of both positive and negative frequencies. The plot also shows arrows to indicate the direction of increasing frequency for each branch. `nyquist` automatically determines frequencies to plot based on system dynamics.

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

If `sys` is a model with complex coefficients, then the positive and negative branches are not symmetric.

`nyquist(sys1,sys2,...,sysN)` plots the frequency response of multiple dynamic systems on the same plot. All systems must have the same number of inputs and outputs.

`nyquist(sys1,LineSpec1,...,sysN,LineSpecN)` specifies a color, line style, and marker for each system in the plot.

`nyquist( ___ ,w)` plots system responses for frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `nyquist` plots the response at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `nyquist` plots the response at each specified frequency. The vector `w` can contain both negative and positive frequencies.

You can use `w` with any of the input-argument combinations in previous syntaxes.

`[re,im,wout] = nyquist(sys)` returns the real and imaginary parts of the frequency response at each frequency in the vector `wout`. The function automatically determines frequencies in `wout` based on system dynamics. This syntax does not draw a plot.

`[re,im,wout] = nyquist(sys,w)` returns the response data at the frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `wout` contains frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `wout` = `w`.

`[re,im,wout,sdre,sdim] = nyquist(sys,w)` also returns the estimated standard deviation of the real and imaginary parts of the frequency response for the identified model `sys`. If you omit `w`, then the function automatically determines frequencies in `wout` based on system dynamics.

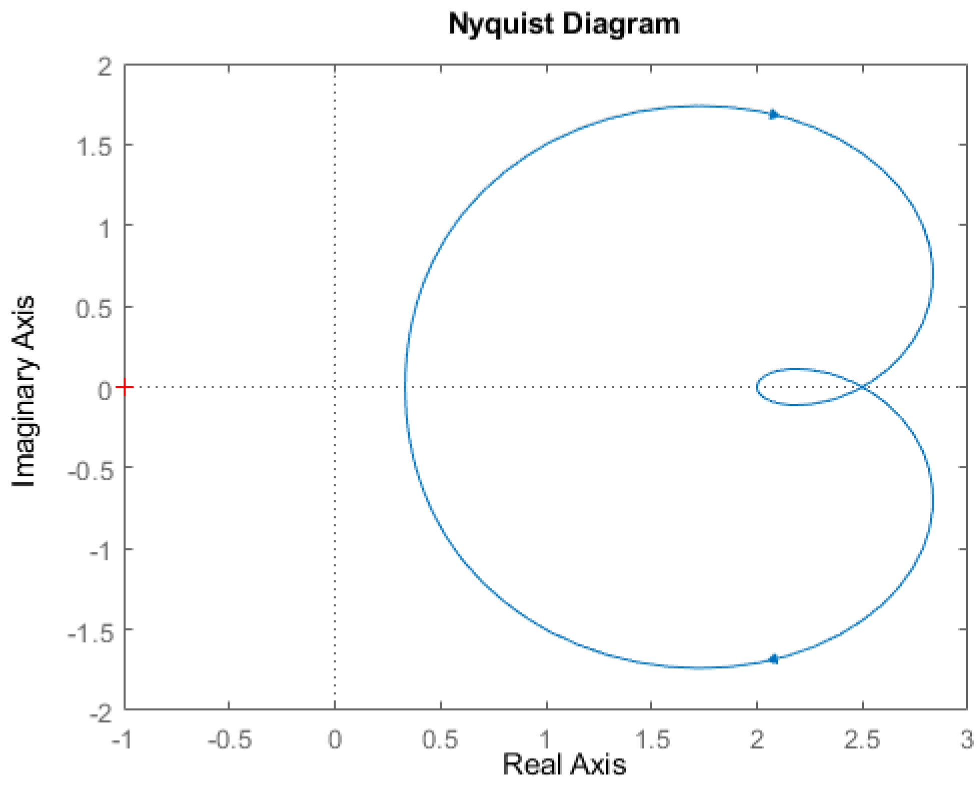
Examples

Nyquist Plot of Dynamic System

Create the following transfer function and plot its Nyquist response.

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}.$$

```
H = tf([2 5 1],[1 2 3]);
nyquist(H)
```



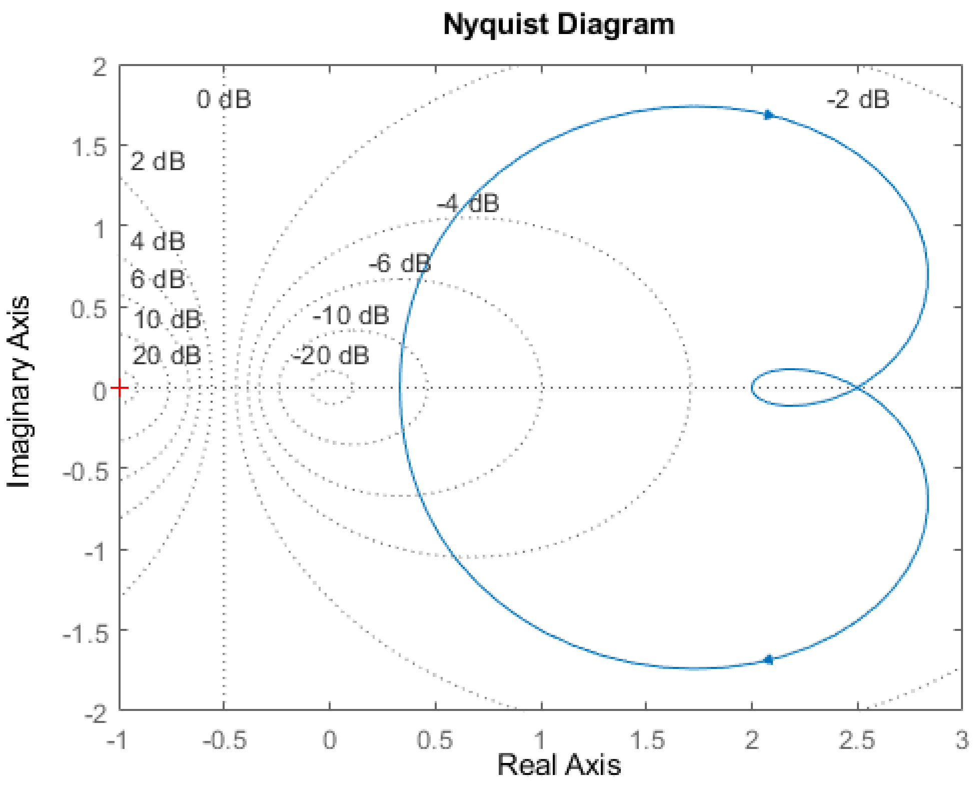
The `nyquist` function can display a grid of M-circles, which are the contours of constant closed-loop magnitude. M-circles are defined as the locus of complex numbers where the following quantity is a constant value across frequency.

$$T(j\omega) = \left|\frac{G(j\omega)}{1 + G(j\omega)}\right|.$$

Here, 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 display the grid of M-circles, right-click in the plot and select **Grid**. Alternatively, use the `grid` command.

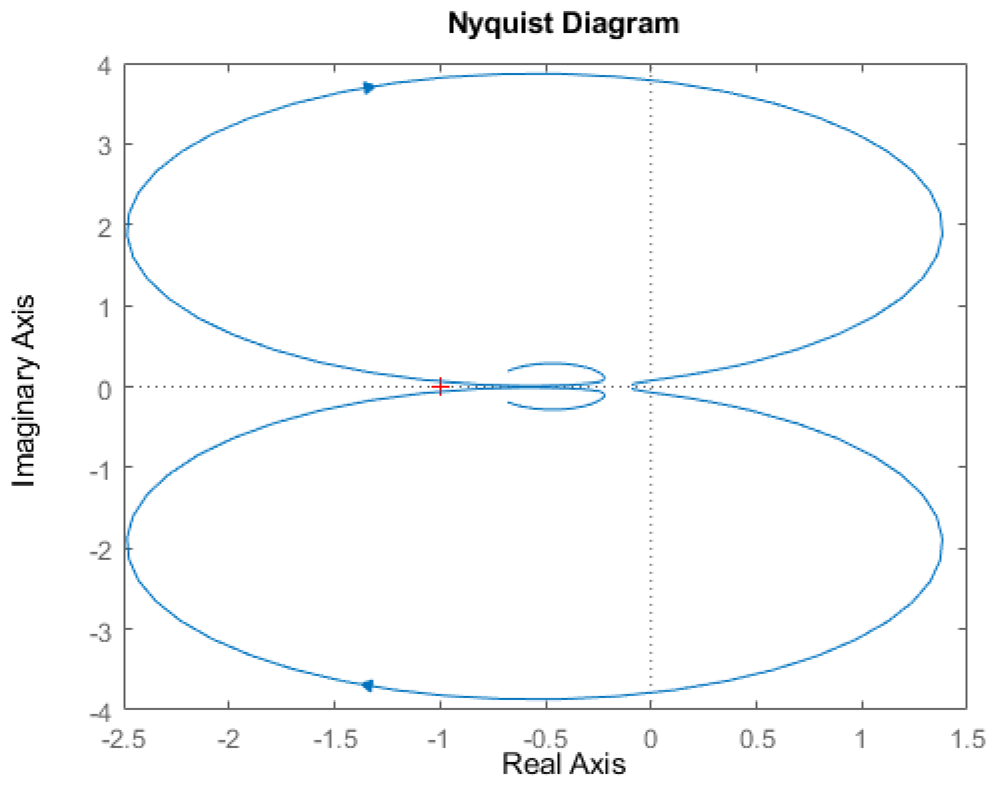
```
grid on
```



Nyquist Plot at Specified Frequencies

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

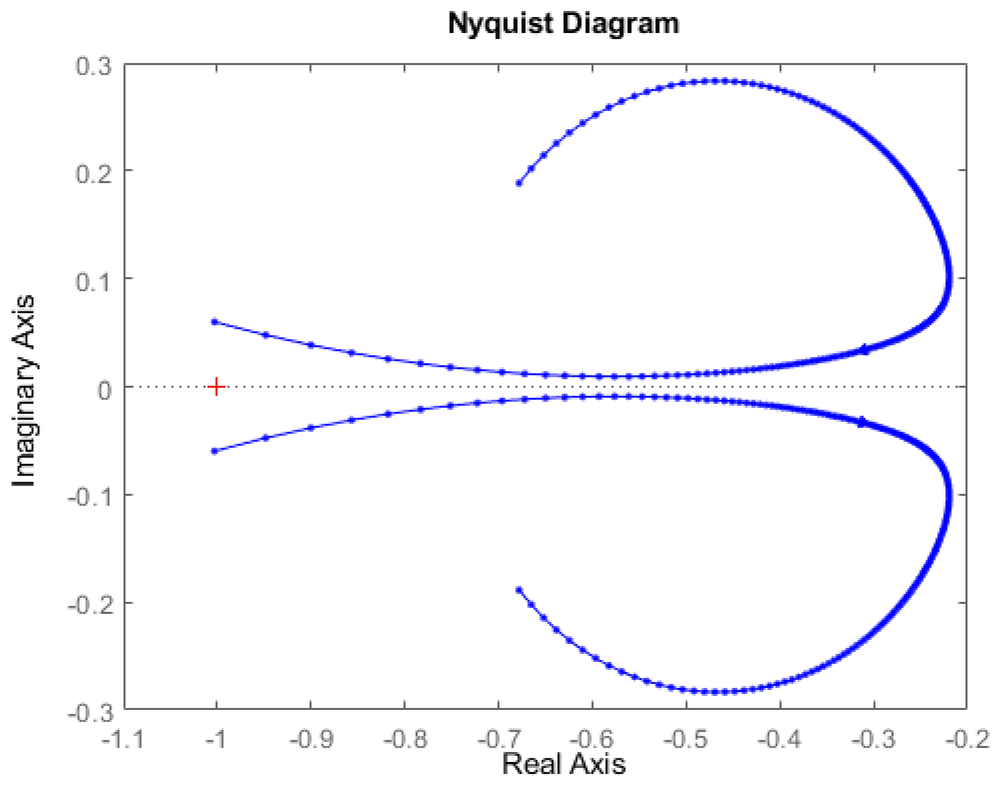
```
H = tf([-0.1,-2.4,-181,-1950],[1,3.3,990,2600]);
nyquist(H,{1,100})
```



The cell array `{1,100}` specifies a frequency range [1,100] for the positive frequency branch and [–100,–1] for the negative frequency branch in the Nyquist plot. The negative frequency branch is obtained by symmetry for models with real coefficients. When you provide frequency bounds in this way, the function selects intermediate points for frequency response data.

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

```
w = 1:0.1:30;
nyquist(H,w,'.-')
```



`nyquist` plots the frequency response at the specified frequencies.

Nyquist Plot of Several Dynamic Systems

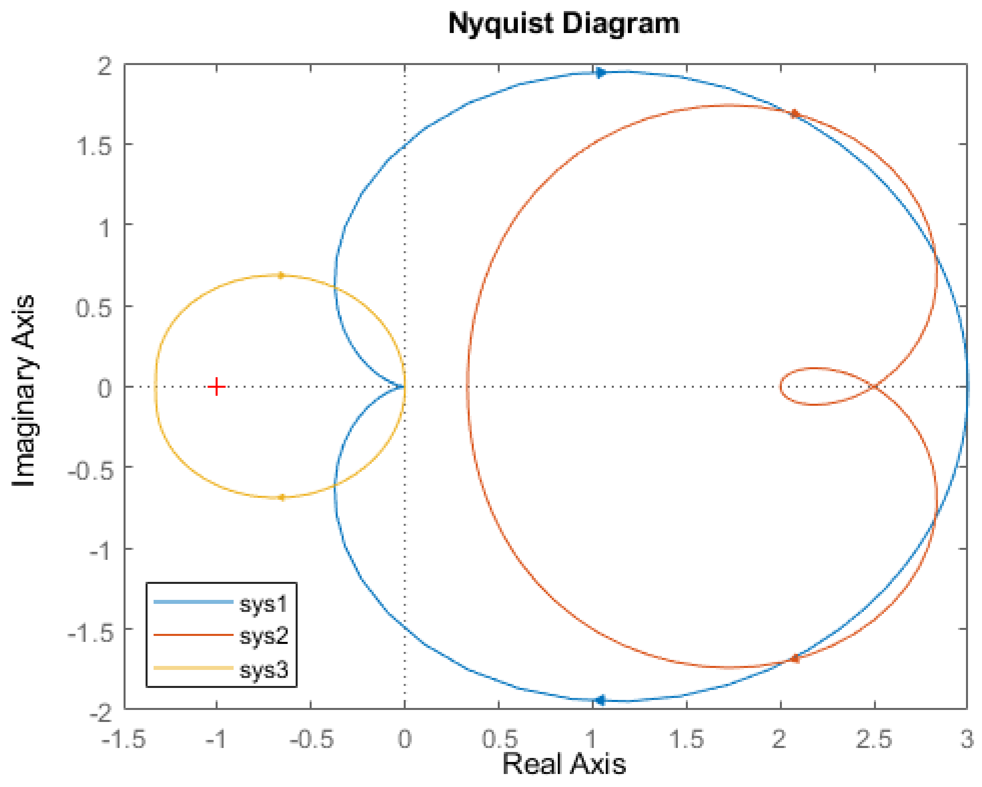
Compare the frequency response of several systems on the same Nyquist plot.

Create the dynamic systems.

```
rng(0)
sys1 = tf(3,[1,2,1]);
sys2 = tf([2 5 1],[1 2 3]);
sys3 = rss(4);
```

Create a Nyquist plot that displays all systems.

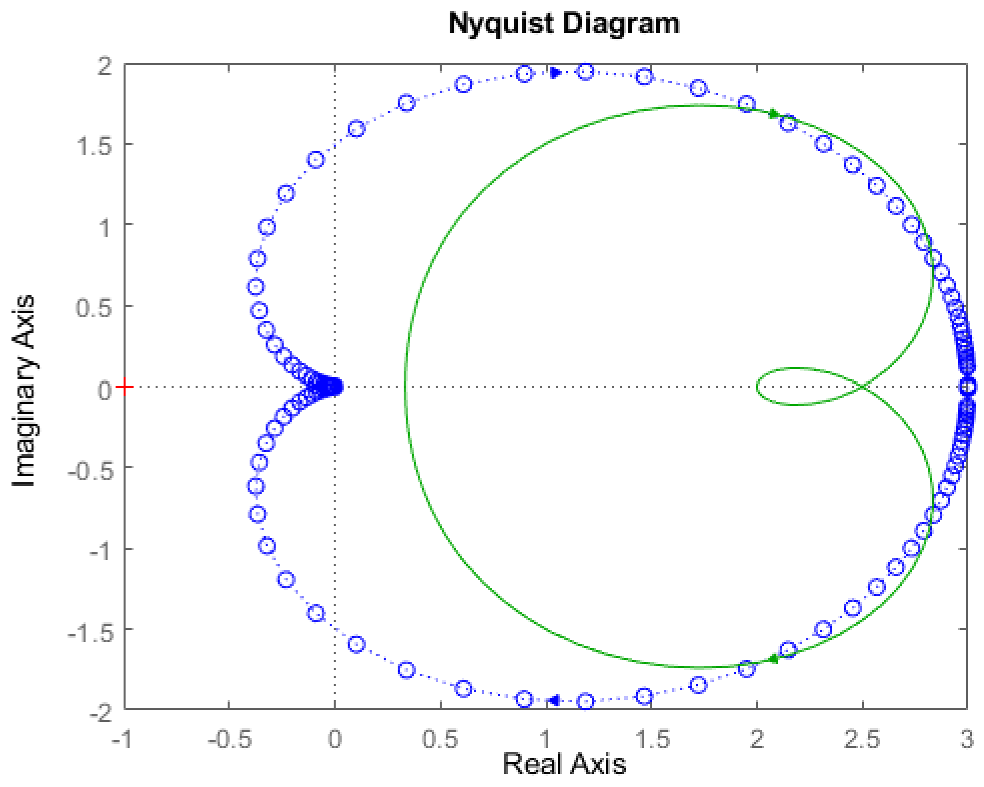
```
nyquist(sys1,sys2,sys3)
legend('Location','southwest')
```



Nyquist Plot with Specified Line Attributes

Specify the line style, color, or marker for each system in a Nyquist plot using the `LineSpec` input argument.

```
sys1 = tf(3,[1,2,1]);
sys2 = tf([2 5 1],[1 2 3]);
nyquist(sys1,'o:',sys2,'g')
```



The first LineSpec, 'o:', specifies a dotted line with circle markers for the response of sys1. The second LineSpec, 'g', specifies a solid green line for the response of sys2.

Obtain Real and Imaginary Parts of Frequency Response

Compute the real and imaginary parts of the frequency response of a SISO system.

If you do not specify frequencies, nyquist chooses frequencies based on the system dynamics and returns them in the third output argument.

```
H = tf([2 5 1],[1 2 3]);
[re,im,wout] = nyquist(H);
```

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

```
size(re)
```

```
ans = 1×3

     1     1   141
```

```
length(wout)
```

```
ans = 141
```

Thus, each entry along the third dimension of `re` gives the real part of the response at the corresponding frequency in `wout`.

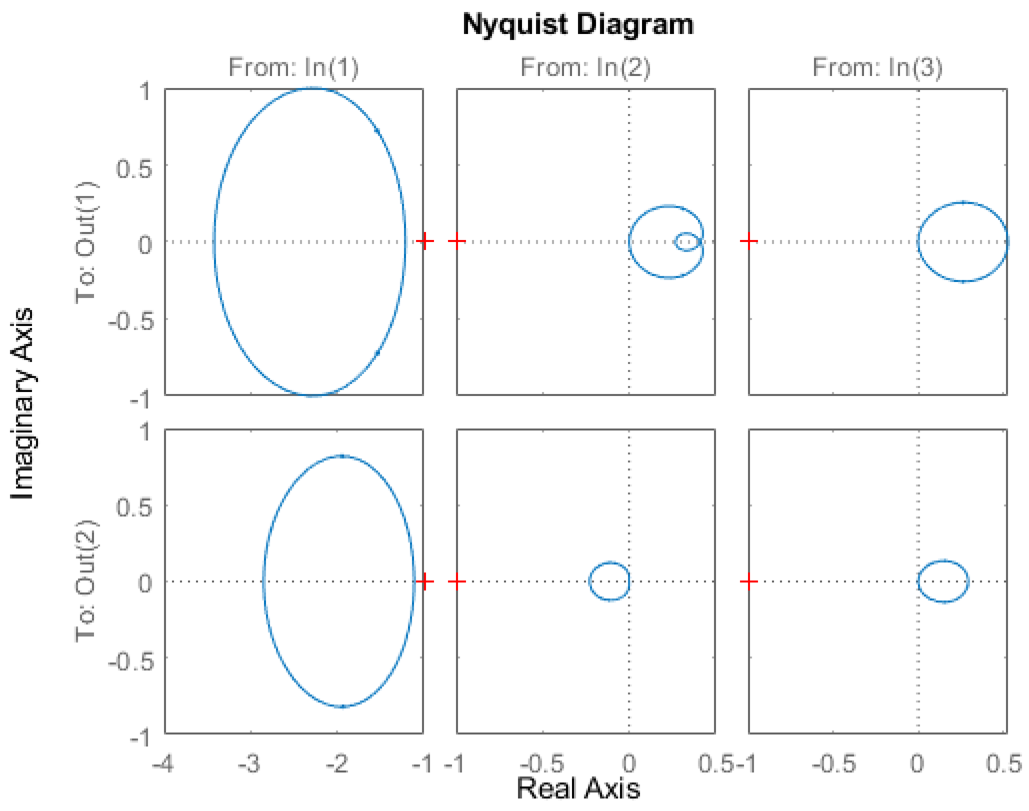
Nyquist Plot of MIMO System

For this example, create a 2-output, 3-input system.

```
rng(0,'twister');
H = rss(4,2,3);
```

For this system, `nyquist` plots the frequency responses of each I/O channel in a separate plot in a single figure.

```
nyquist(H)
```



Compute the real and imaginary parts of these responses at 20 frequencies between 1 and 10 radians.

```
w = logspace(0,1,20);
[re,im] = nyquist(H,w);
```

re and im are three-dimensional arrays, in which the first two dimensions correspond to the output and input dimensions of H, and the third dimension is the number of frequencies. For instance, examine the dimensions of re.

```
size(re)
```

```
ans = 1×3

     2     3    20
```

Thus, for example, re(1,3,10) is the real part of the response from the third input to the first output, computed at the 10th frequency in w. Similarly, im(1,3,10) contains the imaginary part of the same response.

Create Nyquist Plot of Identified Model With Response Uncertainty

Compute the standard deviations of the real and imaginary parts of the frequency response of an identified model. Use this data to create a 3σ plot of the response uncertainty.

Load the estimation data z2.

```
load iddata2 z2;
```

Identify a transfer function model using the data. Using the tfest command requires System Identification Toolbox™ software.

```
sys_p = tfest(z2,2);
```

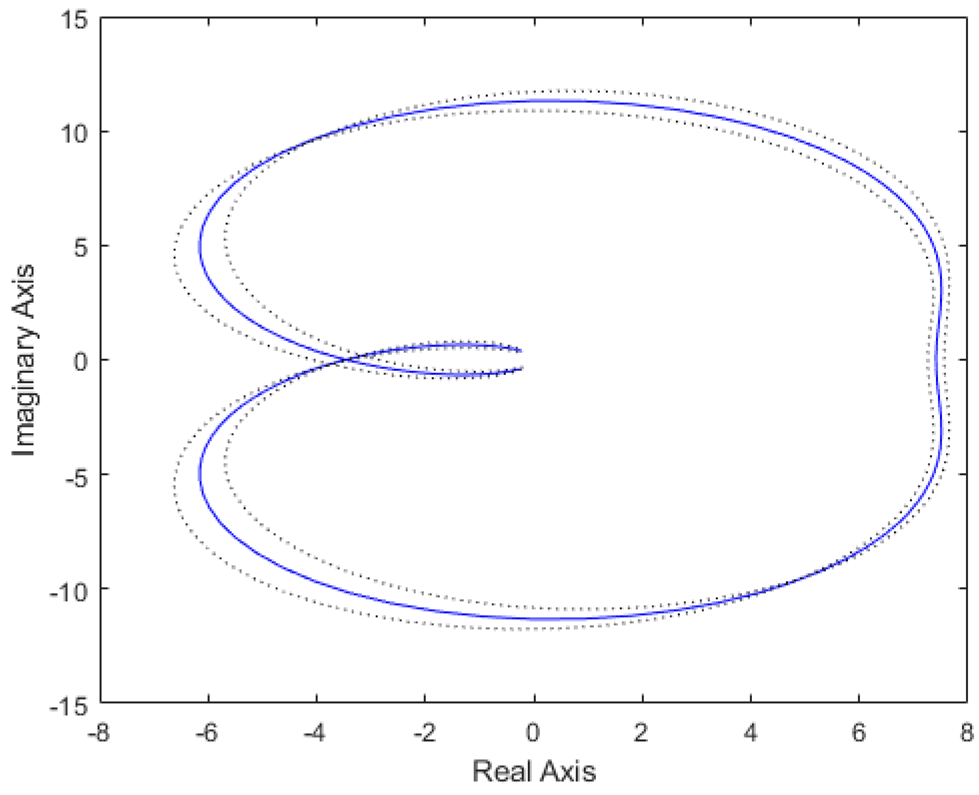
Obtain the standard deviations for the real and imaginary parts of the frequency response for a set of 512 frequencies, w.

```
w = linspace(-10*pi,10*pi,512);
[re,im,wout,sdre,sdim] = nyquist(sys_p,w);
```

re and im are the real and imaginary parts of the frequency response, and sdre and sdim are their standard deviations, respectively. The frequencies in wout are the same as the frequencies you specified in w.

Use the standard deviation data to create a 3σ plot corresponding to the confidence region.

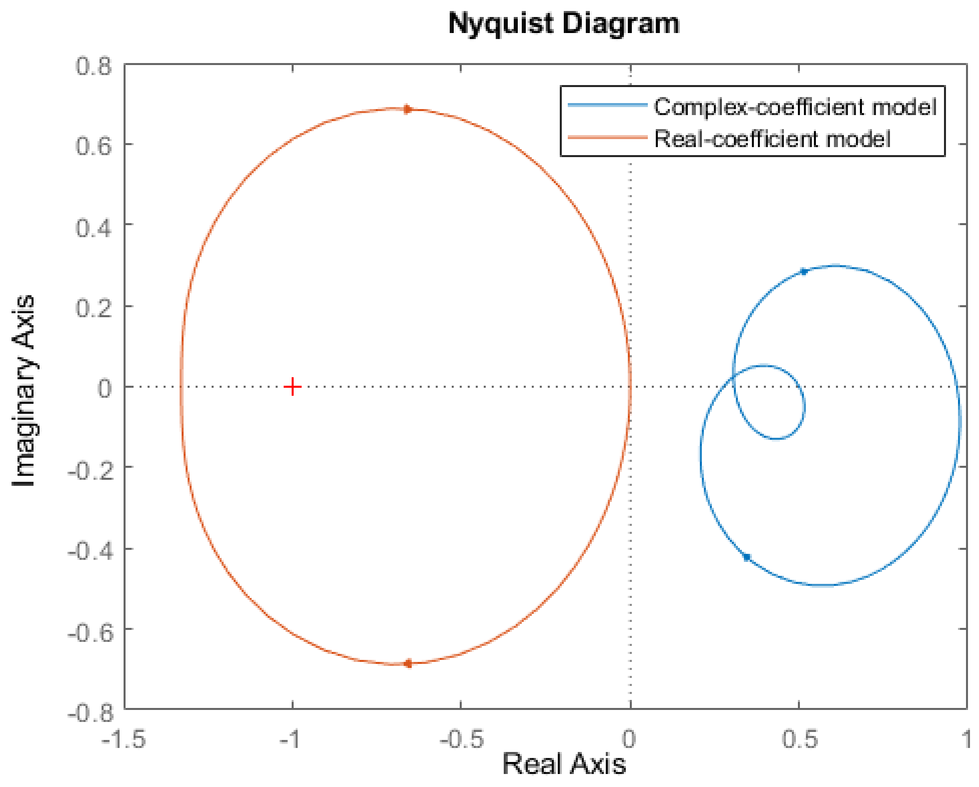
```
re = squeeze(re);
im = squeeze(im);
sdre = squeeze(sdre);
sdim = squeeze(sdim);
plot(re,im,'b',re+3*sdre,im+3*sdim,'k:',re-3*sdre,im-3*sdim,'k:')
xlabel('Real Axis');
ylabel('Imaginary Axis');
```



Nyquist Plot of Model with Complex Coefficients

Create a Nyquist plot of a model with complex coefficients and a model with real coefficients on the same plot.

```
rng(0)
A = [-3.50,-1.25-0.25i;2,0];
B = [1;0];
C = [-0.75-0.5i,0.625-0.125i];
D = 0.5;
Gc = ss(A,B,C,D);
Gr = rss(4);
nyquist(Gc,Gr)
legend('Complex-coefficient model','Real-coefficient model')
```



The Nyquist plot always shows two branches, one for positive frequencies and one for negative frequencies. The arrows indicate the direction of increasing frequency for each branch. For models with complex coefficients, the two branches are not symmetric. For models with real coefficients, the negative branch is obtained by symmetry.

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value for both plotting and returning frequency response data.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model. When you use output arguments, the function returns frequency response data for the nominal model only.
- Frequency-response data models such as `frd` models. For such models, the function plots the response at frequencies defined in the model.

- Identified LTI models, such as `idtf`, `idss`, or `idproc` models. For such models, the function can also plot confidence intervals and return standard deviations of the frequency response. See “Create Nyquist Plot of Identified Model With Response Uncertainty” on page 1-1074.

If `sys` is an array of models, the function plots the frequency responses of all models in the array on the same axes.

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a string or vector of one, two, or three characters. The characters can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line. For more information about configuring this argument, see the `LineSpec` input argument of the `plot` function.

Example: `'r--'` specifies a red dashed line

Example: `'*b'` specifies blue asterisk markers

Example: `'y'` specifies a yellow line

`w` — Frequencies
`{wmin,wmax}` | vector

Frequencies at which to compute and plot frequency response, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function computes the response at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function computes the response at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values. The vector `w` can contain both positive and negative frequencies.

If you specify a frequency range of [w_{min},w_{max}] for your plot, then the plot shows a contour comprised of both positive frequencies [w_{min},w_{max}] and negative frequencies [-w_{max},-w_{min}].

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

Output Arguments

`re` — Real part of system response
3-D array

Real part of the system response, returned as a 3-D array. The dimensions of this array are (number of system outputs)-by-(number of system inputs)-by-(number of frequency points).

- For SISO systems, `re(1,1,k)` gives the real part of the response at the `k`th frequency in `w` or `wout`. For an example, see “Obtain Real and Imaginary Parts of Frequency Response” on page 1-1072.
- For MIMO systems, `re(i,j,k)` gives the real part of the response at the `k`th frequency from the `j`th input to the `i`th output. For an example, see “Nyquist Plot of MIMO System” on page 1-1073.

`im` — Imaginary part of system response
3-D array

Imaginary part of the system response, returned as a 3-D array. The dimensions of this array are (number of system outputs)-by(number of system inputs)-by-(number of frequency points).

- For SISO systems, `im(1,1,k)` gives the imaginary part of the response at the `k`th frequency in `w` or `wout`. For an example, see "Obtain Real and Imaginary Parts of Frequency Response" on page 1-1072.
- For MIMO systems, `im(i,j,k)` gives the imaginary part of the response at the `k`th frequency from the `j`th input to the `i`th output. For an example, see "Nyquist Plot of MIMO System" on page 1-1073.

`wout` — Frequencies
vector

Frequencies at which the function returns the system response, returned as a column vector. The function chooses the frequency values based on the model dynamics, unless you specify frequencies using the input argument `w`.

`wout` also contains negative frequency values for models with complex coefficients.

Frequency values are in radians per `TimeUnit`, where `TimeUnit` is the value of the `TimeUnit` property of `sys`.

`sdre` — Standard deviation of real part
3-D array | [ ]

Estimated standard deviation of the real part of the response at each frequency point, returned as a 3-D array. `sdre` has the same dimensions as `re`.

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

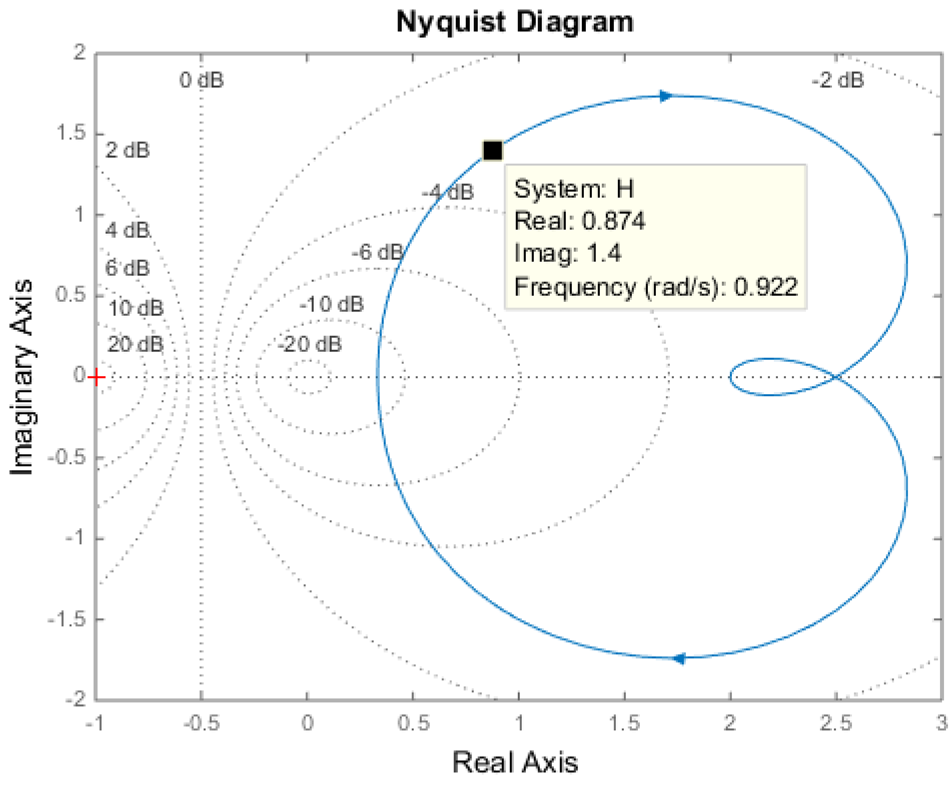
`sdim` — Standard deviation of imaginary part
3-D array | [ ]

Estimated standard deviation of the imaginary part of the response at each frequency point, returned as a 3-D array. `sdim` has the same dimensions as `im`.

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

Tips

- When you need additional plot customization options, use `nyquistplot` instead.
- Two zoom options that apply specifically to Nyquist plots are available from the right-click menu :
  - **Full View** — Clips unbounded branches of the Nyquist plot, but still includes the critical point (–1, 0).
  - **Zoom on (-1,0)** — Zooms around the critical point (–1, 0). To access critical-point zoom programmatically, use the `zoomcp` command. For more information, see `nyquistplot`.
- To activate data markers that display the real and imaginary values at a given frequency, click anywhere on the curve. The following figure shows a `nyquist` plot with a data marker.



See Also

`sigma` | `bode` | `nyquistplot`

Topics

"Plot Bode and Nyquist Plots at the Command Line"
"Dynamic System Models"

Introduced before R2006a

nyquistoptions

Create list of Nyquist plot options

Description

Use the `nyquistoptions` command to create a `NyquistPlotOptions` object to customize your Nyquist plot appearance. You can also use the command to override the plot preference settings in the MATLAB session in which you create the Nyquist plots.

Creation

Syntax

```
plotoptions = nyquistoptions
plotoptions = nyquistoptions('cstprefs')
```

Description

`plotoptions = nyquistoptions` returns a default set of plot options for use with the `nyquistplot` command. You can use these options to customize the Nyquist 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.

`plotoptions = nyquistoptions('cstprefs')` initializes the plot options with the options you selected in the Control System Toolbox and System Identification Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor”. 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.

Properties

`FreqUnits` — Frequency units
'`rad/s`' (default)

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'`

`MagUnits` — Magnitude units
'`dB`' (default) | '`abs`'

Magnitude units, specified as either '`dB`' or absolute value '`abs`'.

`PhaseUnits` — Phase units
'`deg`' (default) | '`rad`'

Phase units, specified as either '`deg`' or '`rad`' to change to degrees or radians, respectively.

`ShowFullContour` — Toggle display of the response for negative frequencies
'`on`' (default) | '`off`'

Toggle display of the response for negative frequencies, specified as either '`on`' or '`off`'.

`ConfidenceRegionNumberSD` — Number of standard deviations to use to plot the confidence region
`1` (default) | scalar

Number of standard deviations to use to plot the confidence region, specified as a scalar. This is applicable to identified models only.

`ConfidenceRegionDisplaySpacing` — Frequency spacing of the confidence ellipses
`5` (default) | scalar

Frequency spacing of the confidence ellipses to use to plot the confidence region, specified as a scalar. This is applicable to identified models only. The default value is `5`, which means the confidence ellipses are shown at every 5th frequency sample

`IOGrouping` — Grouping of input-output pairs
'`none`' (default) | '`inputs`' | '`outputs`' | '`all`'

Grouping of input-output (I/O) pairs, specified as one of the following:

- 'none' — No input-output grouping.
- 'inputs' — Group only the inputs.
- 'outputs' — Group only the outputs.
- 'all' — Group all the I/O pairs.

InputLabels — Input label style
structure (default)

Input label style, specified as a structure with the following fields:

- FontSize — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch.
- FontWeight — Character thickness, specified as 'Normal' or 'bold'. MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- FontAngle — Character slant, specified as 'Normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- Color — Text color, specified as an RGB triplet. The default color is dark grey with the RGB triplet [0.4,0.4,0.4].
- Interpreter — Text interpreter, specified as one of these values:
  - 'tex' — Interpret characters using a subset of TeX markup. This is the default value of Interpreter.
  - 'latex' — Interpret characters using LaTeX markup.
  - 'none' — Display literal characters.

OutputLabels — Output label style
structure (default)

Output label style, specified as a structure with the following fields:

- FontSize — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch.
- FontWeight — Character thickness, specified as 'Normal' or 'bold'. MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- FontAngle — Character slant, specified as 'Normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- Color — Text color, specified as an RGB triplet. The default color is dark grey with the RGB triplet [0.4,0.4,0.4].
- Interpreter — Text interpreter, specified as one of these values:
  - 'tex' — Interpret characters using a subset of TeX markup. This is the default value of Interpreter.
  - 'latex' — Interpret characters using LaTeX markup.
  - 'none' — Display literal characters.

InputVisible — Toggle display of inputs
{'on'} (default) | {'off'} | cell array

Toggle display of inputs, specified as either `{'on'}`, `{'off'}` or a cell array with multiple elements .

`OutputVisible` — Toggle display of outputs
`{'on'}` (default) | `{'off'}` | cell array

Toggle display of outputs, specified as either `{'on'}`, `{'off'}` or a cell array with multiple elements.

`Title` — Title text and style
structure (default)

Title text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a character vector. By default, the plot is titled `'Bode Diagram'`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`XLabel` — X-axis label text and style
structure (default)

X-axis label text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a character vector. By default, the axis is titled `Real Axis`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.

- 'latex' — Interpret characters using LaTeX markup.
- 'none' — Display literal characters.

YLabel — Y-axis label text and style
structure (default)

Y-axis label text and style, specified as a structure with the following fields:

- String — Label text, specified as a cell array of character vectors. By default, the axis is titled Imaginary Axis.
- FontSize — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch.
- FontWeight — Character thickness, specified as 'Normal' or 'bold'. MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- FontAngle — Character slant, specified as 'Normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- Color — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet [0,0,0].
- Interpreter — Text interpreter, specified as one of these values:
  - 'tex' — Interpret characters using a subset of TeX markup. This is the default value of Interpreter.
  - 'latex' — Interpret characters using LaTeX markup.
  - 'none' — Display literal characters.

TickLabel — Tick label style
structure (default)

Tick label style, specified as a structure with the following fields:

- FontSize — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch.
- FontWeight — Character thickness, specified as 'Normal' or 'bold'. MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- FontAngle — Character slant, specified as 'Normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- Color — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet [0,0,0].

Grid — Toggle grid display
'off' (default) | 'on'

Toggle grid display on the plot, specified as either 'off' or 'on'.

GridColor — Color of the grid lines
[0.15,0.15,0.15] (default) | RGB triplet

Color of the grid lines, specified as an RGB triplet. The default color is light grey specified by the RGB triplet [0.15,0.15,0.15].

XLimMode — X-axis limit selection mode
'auto' (default) | 'manual' | cell array

Selection mode for the x-axis limits, specified as one of these values:

- 'auto' — Enable automatic limit selection, which is based on the total span of the plotted data.
- 'manual' — Manually specify the axis limits. To specify the axis limits, set the XLim property.

YLimMode — Y-axis limit selection mode
'auto' (default) | 'manual' | cell array

Selection mode for the y-axis limits, specified as one of these values:

- 'auto' — Enable automatic limit selection, which is based on the total span of the plotted data.
- 'manual' — Manually specify the axis limits. To specify the axis limits, set the YLim property.

XLim — X-axis limits
'{[1,10]}' (default) | cell array of two-element vector of the form [min,max] | cell array

X-axis limits, specified as a cell array of two-element vector of the form [min,max].

YLim — Y-axis limits
'{[1,10]}' (default) | cell array of two-element vector of the form [min,max] | cell array

Y-axis limits, specified as a cell array of two-element vector of the form [min,max].

Object Functions

nyquistplot	Nyquist plot with additional plot customization options

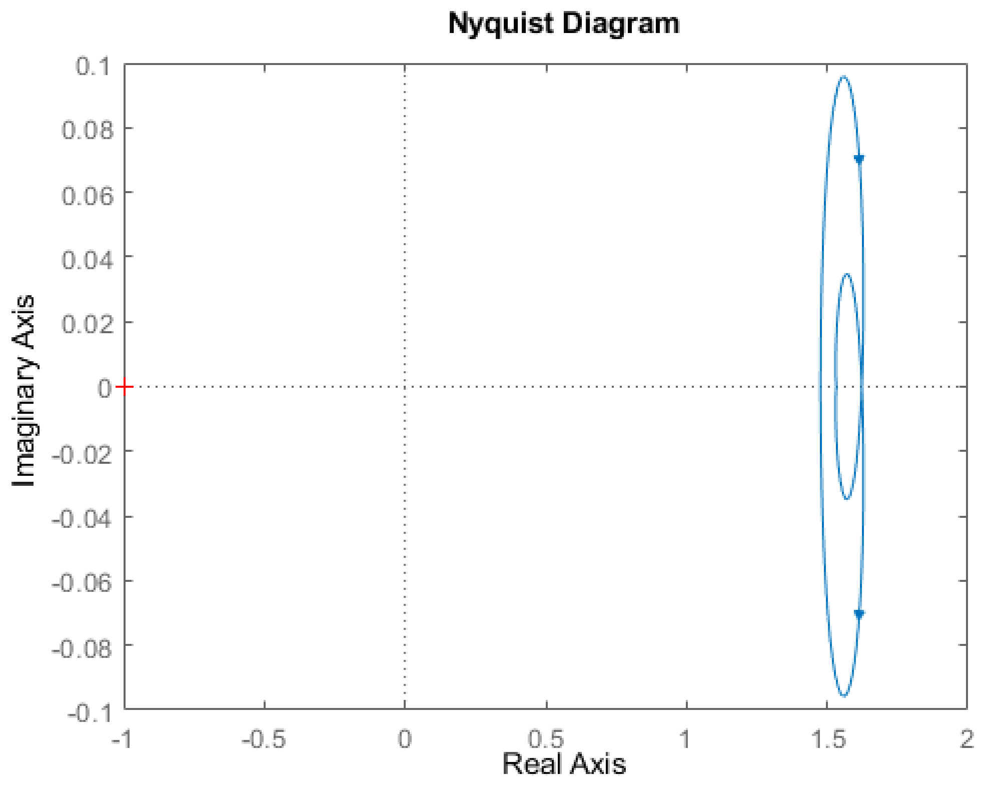
Examples

Customize Nyquist Plot using Plot Handle

For this example, use the plot handle to change the phase units to radians and to turn the grid on.

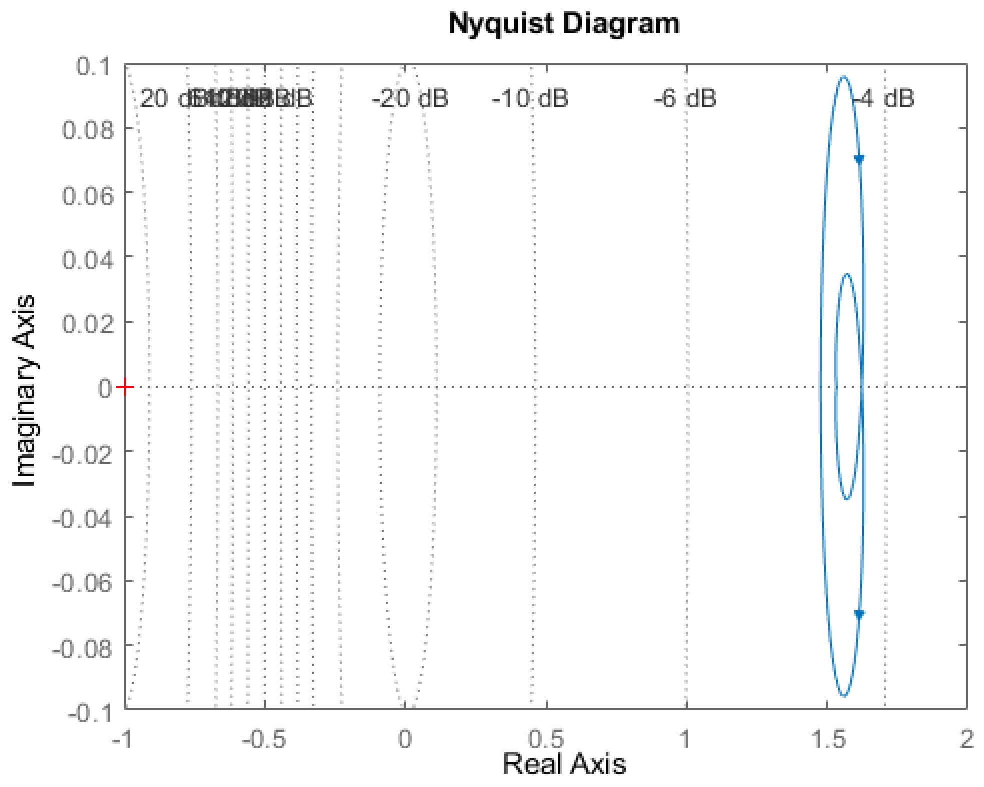
Generate a random state-space model with 5 states and create the Nyquist diagram with plot handle h.

```
rng("default")
sys = rss(5);
h = nyquistplot(sys);
```



Change the phase units to radians and turn on the grid. To do so, edit properties of the plot handle, `h` using `setoptions`.

```
setoptions(h,'PhaseUnits','rad','Grid','on');
```



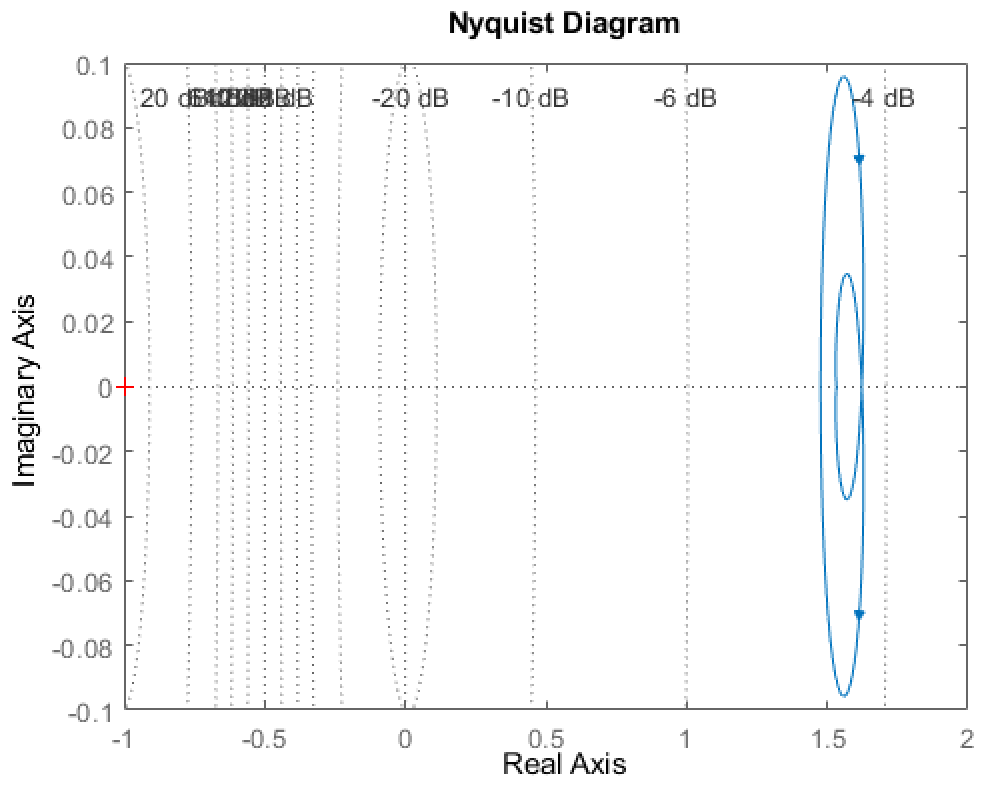
The Nyquist plot automatically updates when you call `setoptions`.

Alternatively, you can also use the `nyquistoptions` command to specify the required plot options. First, create an options set based on the toolbox preferences.

```
plotoptions = nyquistoptions('cstprefs');
```

Change properties of the options set by setting the phase units to radians and enabling the grid.

```
plotoptions.PhaseUnits = 'rad';
plotoptions.Grid = 'on';
nyquistplot(sys,plotoptions);
```



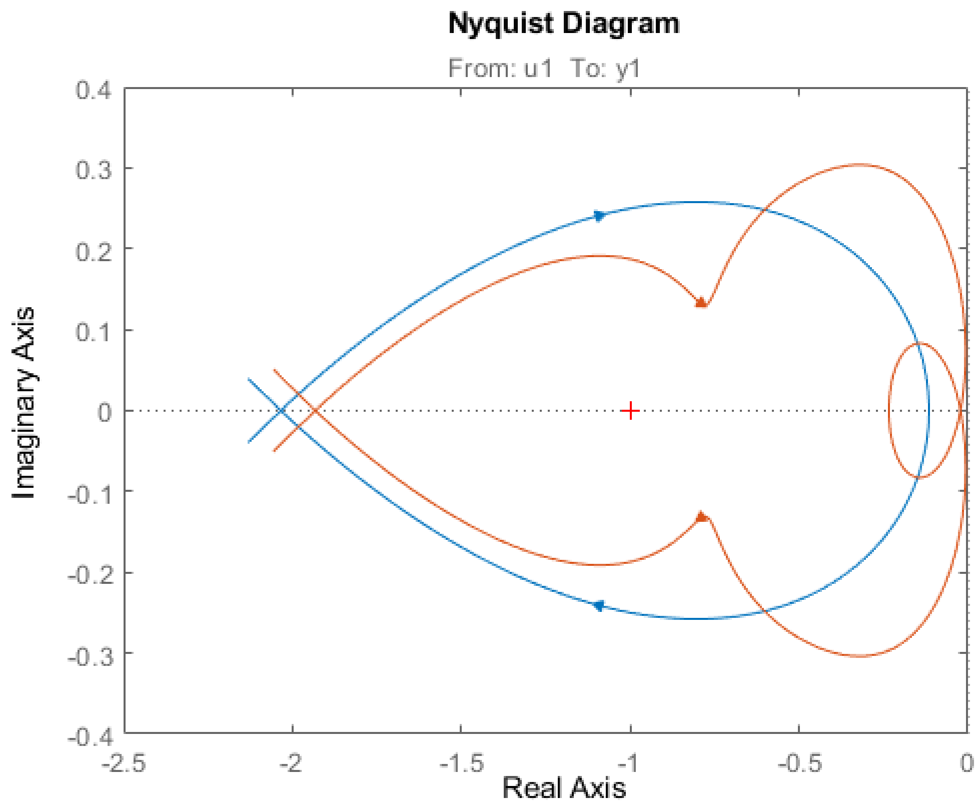
You can use the same option set to create multiple Nyquist plots with the same customization. Depending on your own toolbox preferences, the plot you obtain might look different from this plot. Only the properties that you set explicitly, in this example `PhaseUnits` and `Grid`, override the toolbox preferences.

Nyquist Plot of Identified Models with Confidence Regions at Selected Points

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.

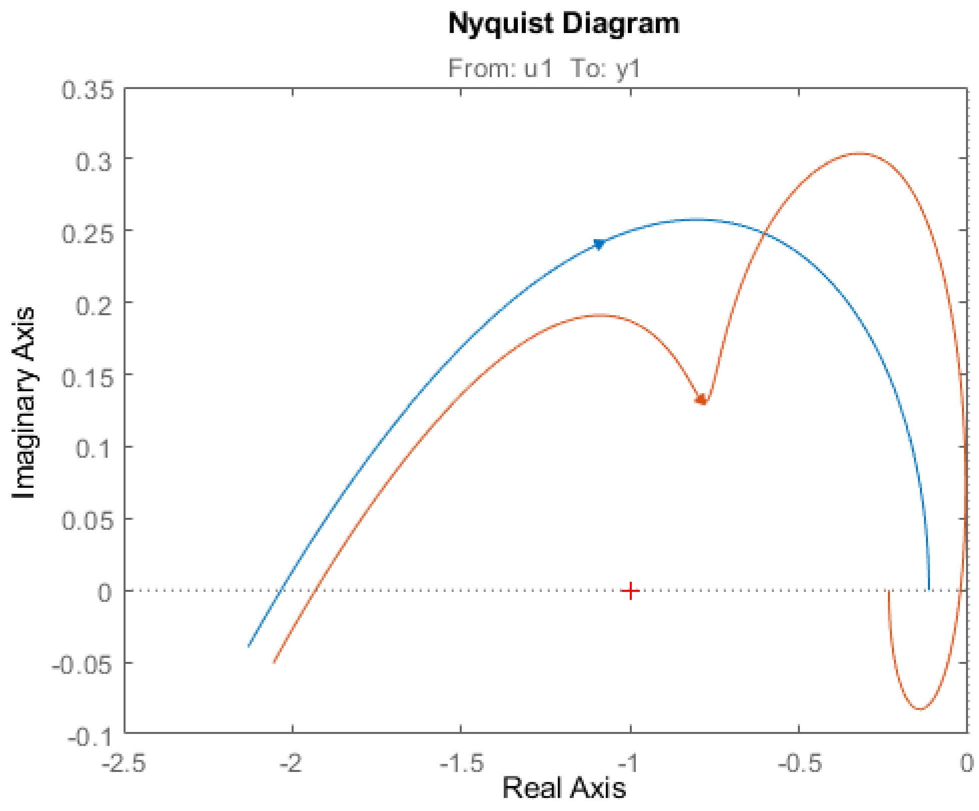
Load the identified model data and estimate the state-space models using `n4sid`. Then, plot the Nyquist diagram.

```
load iddata1
sys1 = n4sid(z1,2);
sys2 = n4sid(z1,6);
w = linspace(10,10*pi,256);
h = nyquistplot(sys1,sys2,w);
```

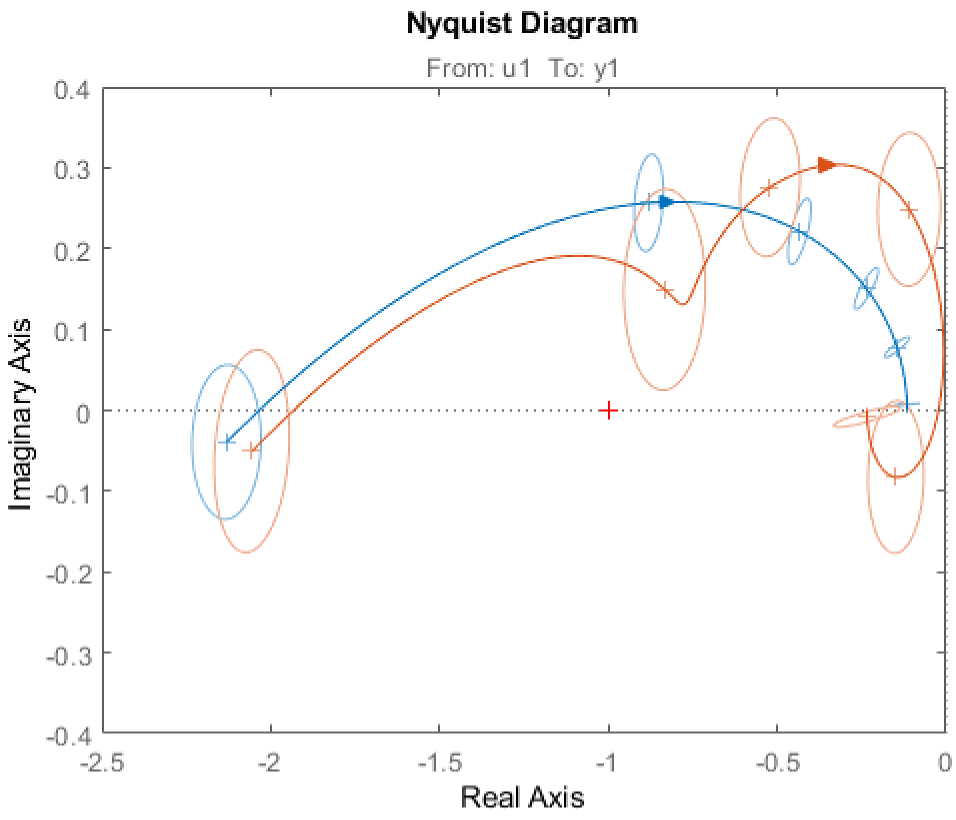


Both models produce about 76% fit to data. However, `sys2` shows higher uncertainty in its frequency response, especially close to Nyquist frequency as shown by the plot. To see this, show the confidence region at a subset of the points at which the Nyquist response is displayed.

```
setoptions(h,'ConfidenceRegionDisplaySpacing',50,...
             'ShowFullContour','off');
```



To turn on the confidence region display, right-click the plot and select **Characteristics > Confidence Region**.



Nyquist Plot with Specific Customization

For this example, consider a MIMO state-space model with 3 inputs, 3 outputs and 3 states. Create a Nyquist plot, display only the partial contour and turn the grid on.

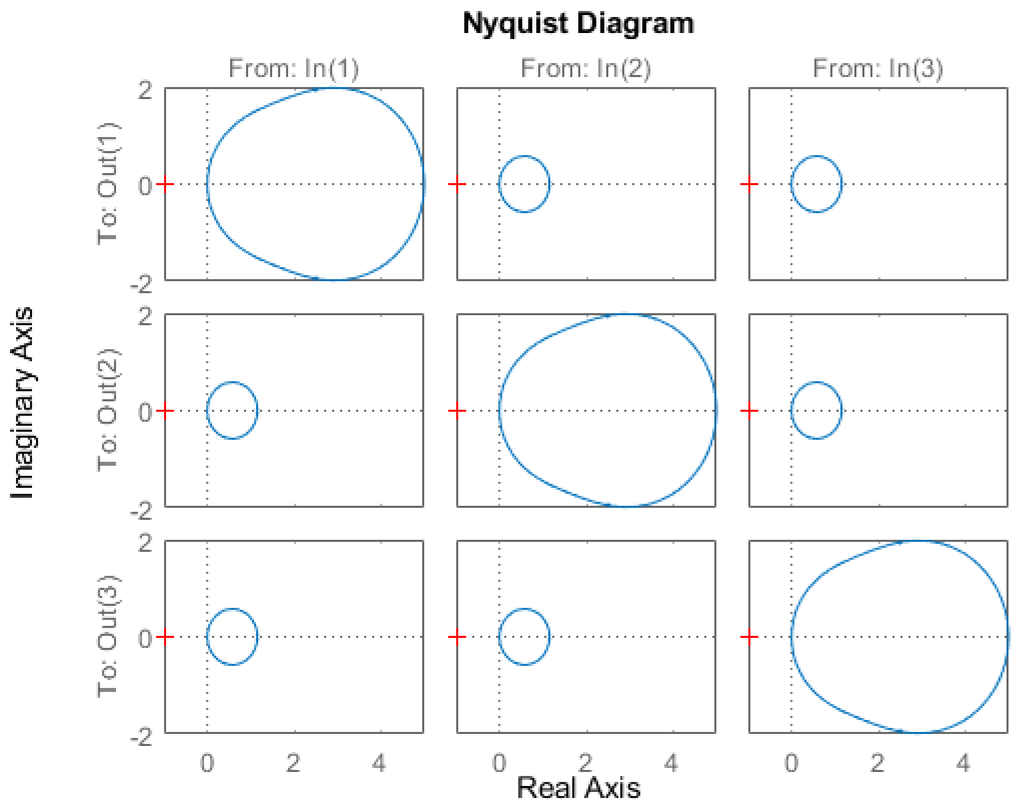
Create the MIMO state-space model `sys_mimo`.

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
size(sys_mimo)
```

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

Create a Nyquist plot with plot handle `h` and use `getoptions` for a list of the options available.

```
h = nyquistplot(sys_mimo);
```

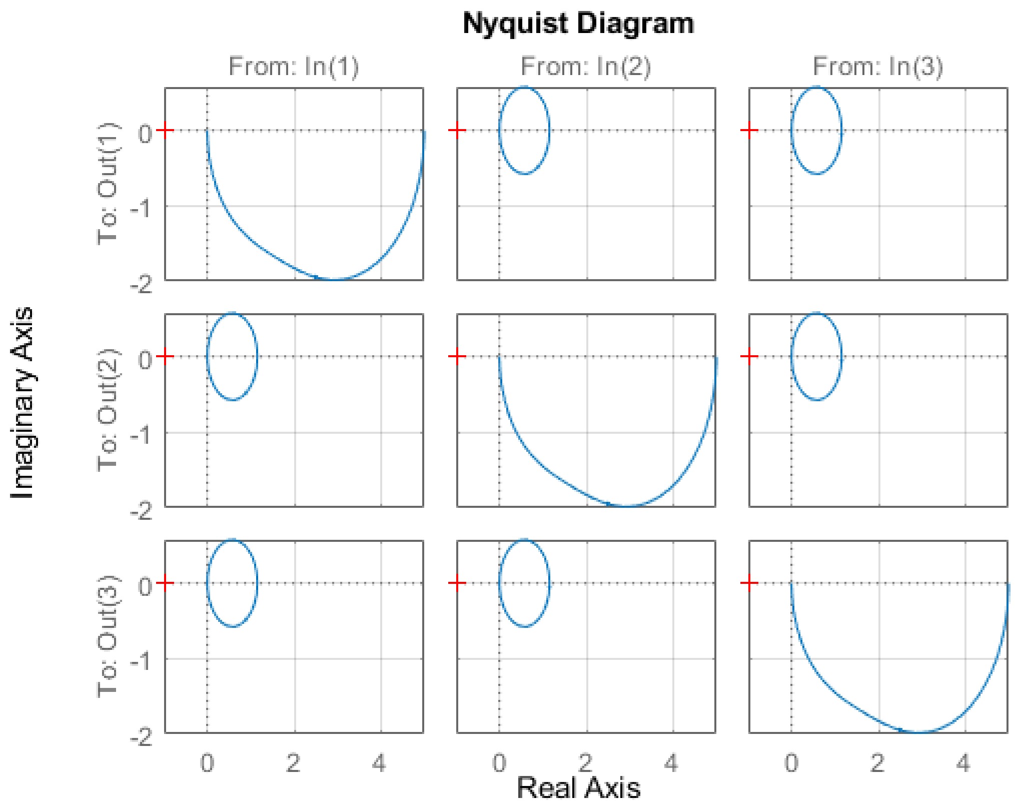


```
p = getoptions(h)
```

```
p = 

                      FreqUnits: 'rad/s'
                       MagUnits: 'dB'
                     PhaseUnits: 'deg'
                ShowFullContour: 'on'
         ConfidenceRegionNumberSD: 1
    ConfidenceRegionDisplaySpacing: 5
                     IOGrouping: 'none'
                    InputLabels: [1x1 struct]
                   OutputLabels: [1x1 struct]
                   InputVisible: {3x1 cell}
                  OutputVisible: {3x1 cell}
                          Title: [1x1 struct]
                         XLabel: [1x1 struct]
                         YLabel: [1x1 struct]
                      TickLabel: [1x1 struct]
                           Grid: 'off'
                      GridColor: [0.1500 0.1500 0.1500]
                           XLim: {3x1 cell}
                           YLim: {3x1 cell}
                       XLimMode: {3x1 cell}
                       YLimMode: {3x1 cell}
```

Use `setoptions` to update the plot with the requires customization.

```
setoptions(h,'ShowFullContour','off','Grid','on');
```



The Nyquist plot automatically updates when you call `setoptions`. For MIMO models, `nyquistplot` produces an array of Nyquist diagrams, each plot displaying the frequency response of one I/O pair.

See Also

`nyquist` | `nyquistplot` | `getoptions` | `setoptions` | `setoptions` | `showConfidence`

Topics

"Toolbox Preferences Editor"

Introduced in R2012a

nyquistplot

Nyquist plot with additional plot customization options

Syntax

```
h = nyquistplot(sys)
h = nyquistplot(sys1,sys2,...,sysN)
h = nyquistplot(sys1,LineSpec1,...,sysN,LineSpecN)
h = nyquistplot( ___ ,w)
h = nyquistplot(AX, ___ )
h = nyquistplot( ___ ,plotoptions)
```

Description

`nyquistplot` lets you plot the Nyquist diagram of a dynamic system model with a broader range of plot customization options than `nyquist`. You can use `nyquistplot` to obtain the plot handle and use it to customize the plot, such as modify the axes labels, limits and units. You can also use `nyquistplot` to draw a Nyquist diagram on an existing set of axes represented by an axes handle. To customize an existing Nyquist plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox). To create Nyquist plots with default options or to extract the standard deviation, real and imaginary parts of the frequency response data, use `nyquist`.

`h = nyquistplot(sys)` plots the Nyquist plot of the dynamic system model `sys` and returns the plot handle `h` to the plot. You can use this handle `h` to customize the plot with the `getoptions` and `setoptions` commands. If `sys` is a multi-input, multi-output (MIMO) model, then `nyquistplot` produces a grid of Nyquist plots, each plot displaying the frequency response of one I/O pair.

`h = nyquistplot(sys1,sys2,...,sysN)` plots the Nyquist plot of multiple dynamic systems `sys1,sys2,…,sysN` on the same plot. All systems must have the same number of inputs and outputs to use this syntax.

`h = nyquistplot(sys1,LineSpec1,...,sysN,LineSpecN)` sets the line style, marker type, and color for the Nyquist plot of each system. All systems must have the same number of inputs and outputs to use this syntax.

`h = nyquistplot( ___ ,w)` plots Nyquist diagram for frequencies specified by the frequencies in `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `nyquistplot` plots the Nyquist diagram at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `nyquistplot` plots the Nyquist diagram at each specified frequency.

You can use `w` with any of the input-argument combinations in previous syntaxes.

See `logspace` to generate logarithmically spaced frequency vectors.

`h = nyquistplot(AX,___)` plots the Nyquist plot on the `Axes` object in the current figure with the handle AX.

`h = nyquistplot(___,plotoptions)` plots the Nyquist plot with the options set specified in `plotoptions`. You can use these options to customize the Nyquist plot appearance using the command line. Settings you specify in `plotoptions` overrides the preference settings in the MATLAB session in which you run `nyquistplot`. Therefore, this syntax is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

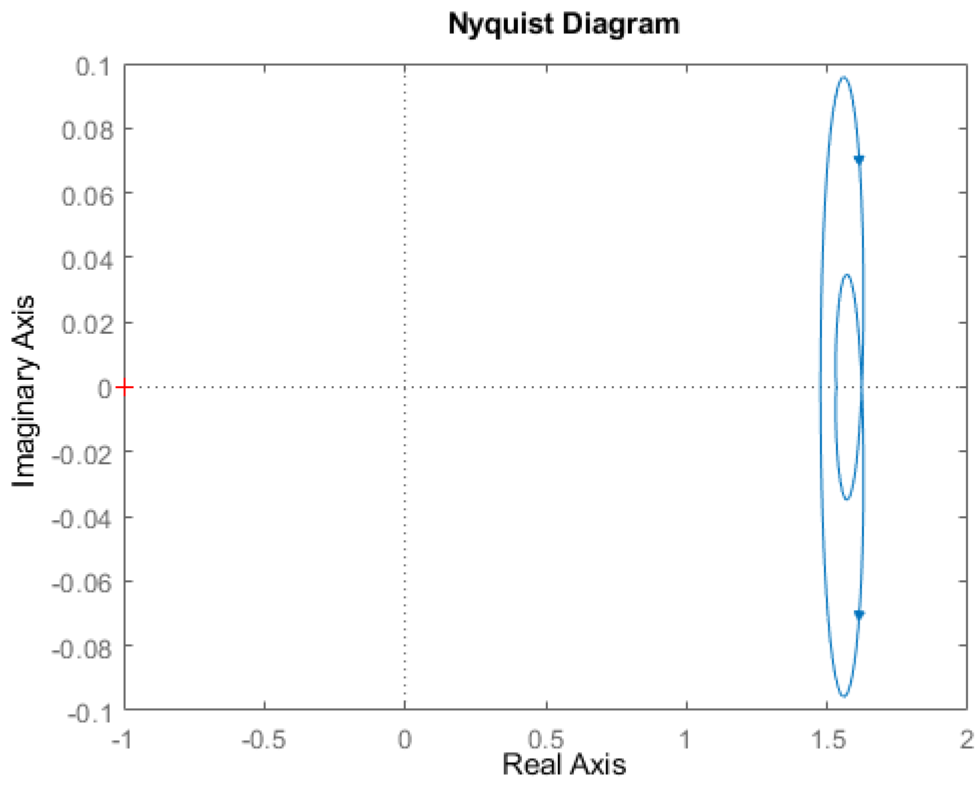
Examples

Customize Nyquist Plot using Plot Handle

For this example, use the plot handle to change the phase units to radians and to turn the grid on.

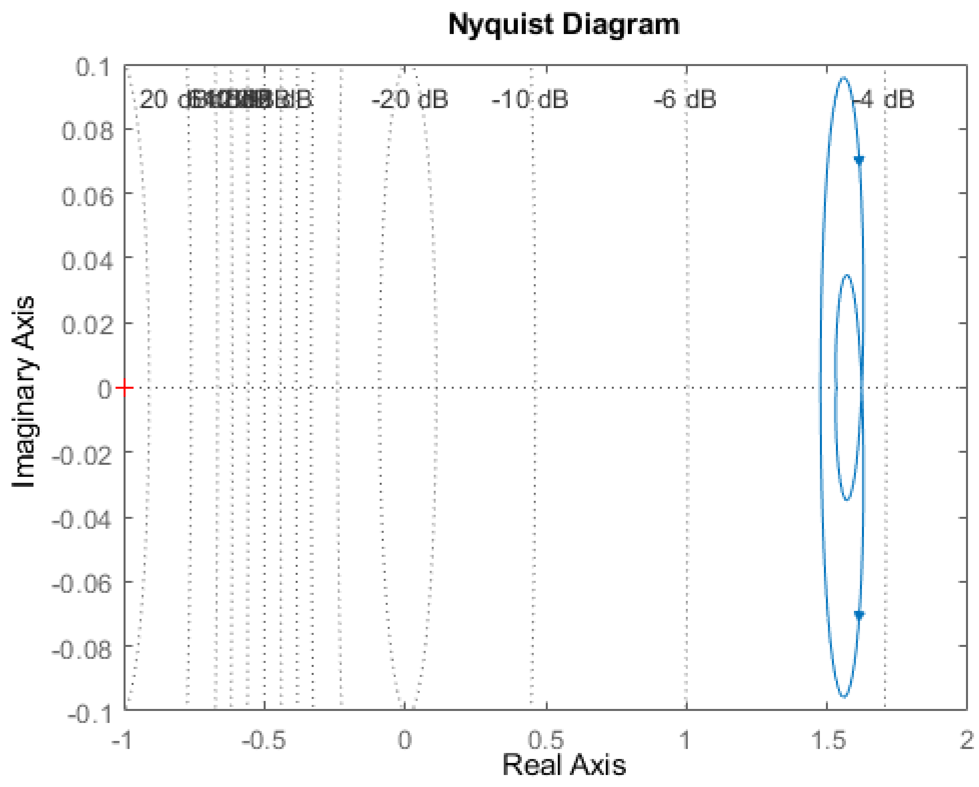
Generate a random state-space model with 5 states and create the Nyquist diagram with plot handle `h`.

```
rng("default")
sys = rss(5);
h = nyquistplot(sys);
```



Change the phase units to radians and turn on the grid. To do so, edit properties of the plot handle, `h` using `setoptions`.

```
setoptions(h,'PhaseUnits','rad','Grid','on');
```



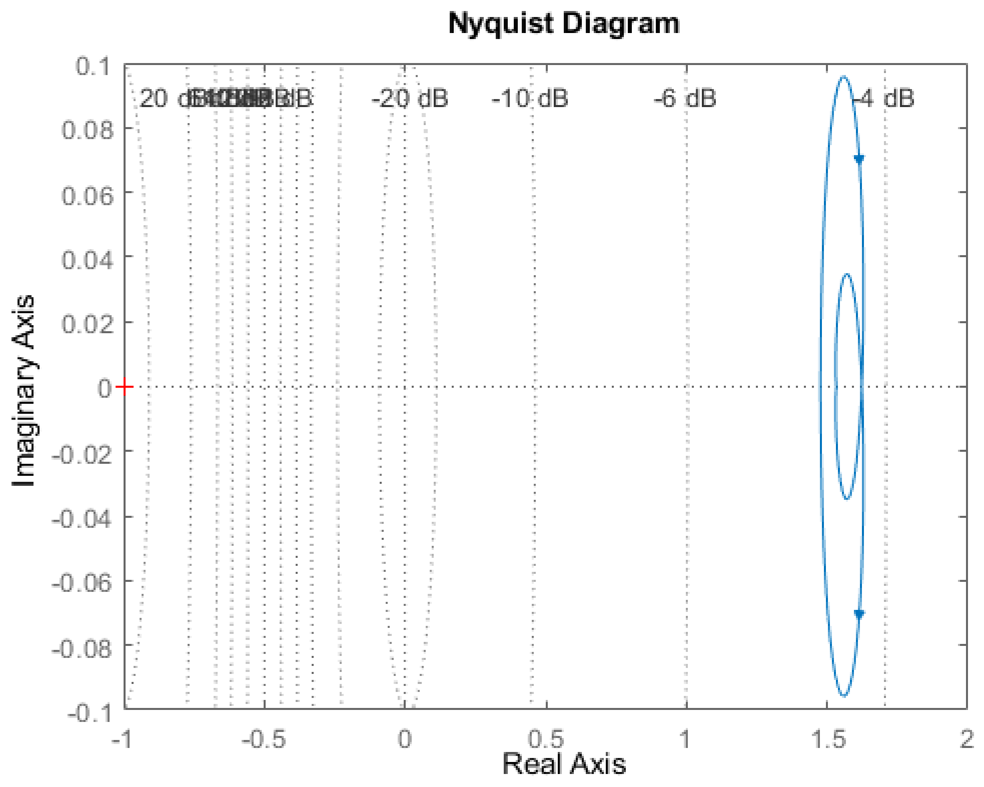
The Nyquist plot automatically updates when you call `setoptions`.

Alternatively, you can also use the `nyquistoptions` command to specify the required plot options. First, create an options set based on the toolbox preferences.

```
plotoptions = nyquistoptions('cstprefs');
```

Change properties of the options set by setting the phase units to radians and enabling the grid.

```
plotoptions.PhaseUnits = 'rad';
plotoptions.Grid = 'on';
nyquistplot(sys,plotoptions);
```

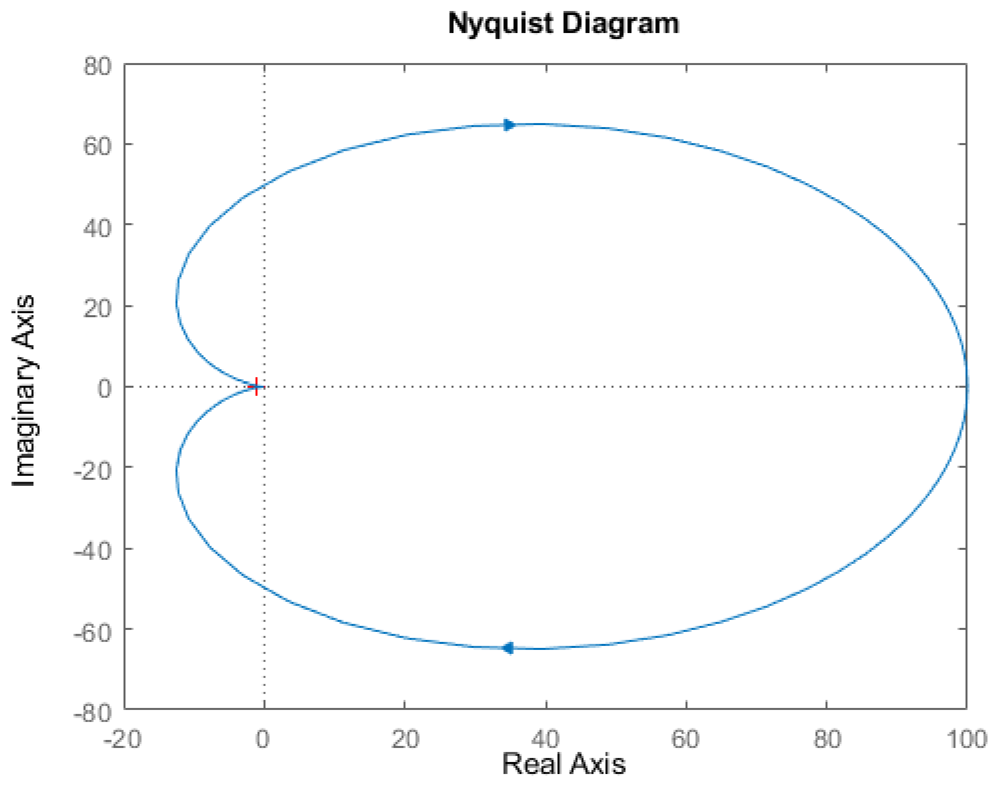


You can use the same option set to create multiple Nyquist plots with the same customization. Depending on your own toolbox preferences, the plot you obtain might look different from this plot. Only the properties that you set explicitly, in this example `PhaseUnits` and `Grid`, override the toolbox preferences.

Customize Nyquist Plot Title

Create a Nyquist plot of a dynamic system model and store a handle to the plot.

```
sys = tf(100,[1,2,1]);
h = nyquistplot(sys);
```



Change the plot title to read "Nyquist Plot of sys." To do so, use `getoptions` to extract the existing plot options from the plot handle `h`.

```
opt = getoptions(h)
```

```
opt = 

                       FreqUnits: 'rad/s'
                        MagUnits: 'dB'
                      PhaseUnits: 'deg'
                 ShowFullContour: 'on'
         ConfidenceRegionNumberSD: 1
    ConfidenceRegionDisplaySpacing: 5
                      IOGrouping: 'none'
                     InputLabels: [1x1 struct]
                    OutputLabels: [1x1 struct]
                    InputVisible: {'on'}
                   OutputVisible: {'on'}
                           Title: [1x1 struct]
                          XLabel: [1x1 struct]
                          YLabel: [1x1 struct]
                       TickLabel: [1x1 struct]
                            Grid: 'off'
                       GridColor: [0.1500 0.1500 0.1500]
                            XLim: {[-20 100]}
                            YLim: {[-80 80]}
                        XLimMode: {'auto'}
                        YLimMode: {'auto'}
```

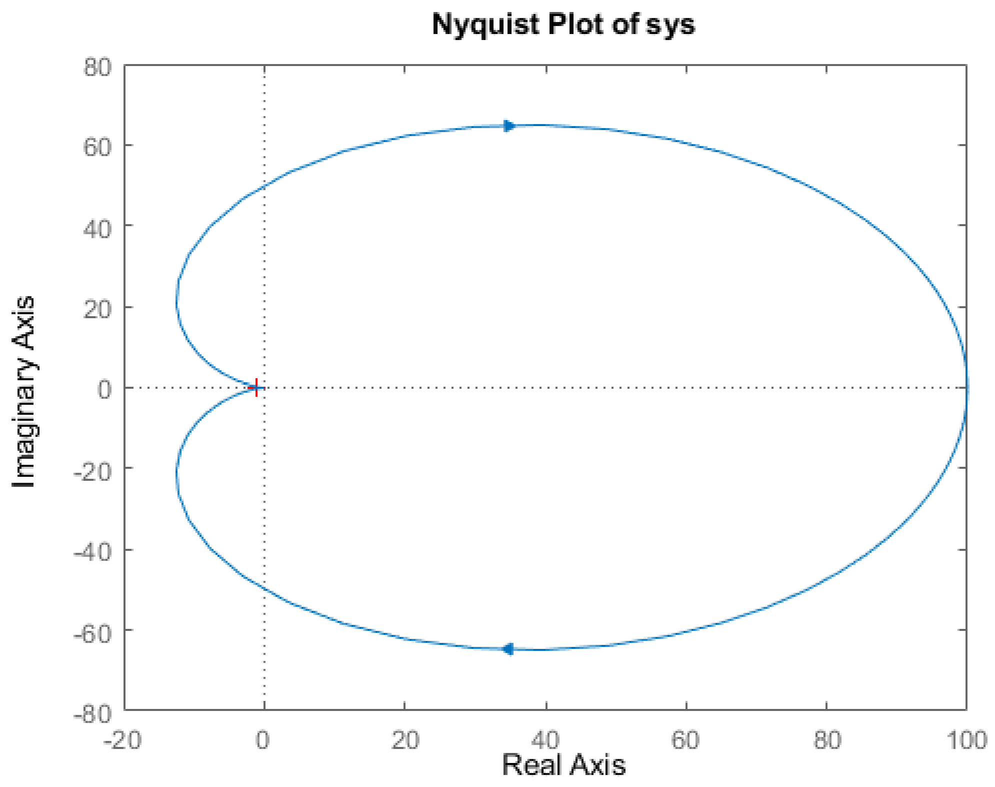
The `Title` option is a structure with several fields.

```
opt.Title
```

```
ans = struct with fields:
         String: 'Nyquist Diagram'
       FontSize: 11
     FontWeight: 'bold'
      FontAngle: 'normal'
          Color: [0 0 0]
    Interpreter: 'tex'
```

Change the `String` field of the `Title` structure, and use `setoptions` to apply the change to the plot.

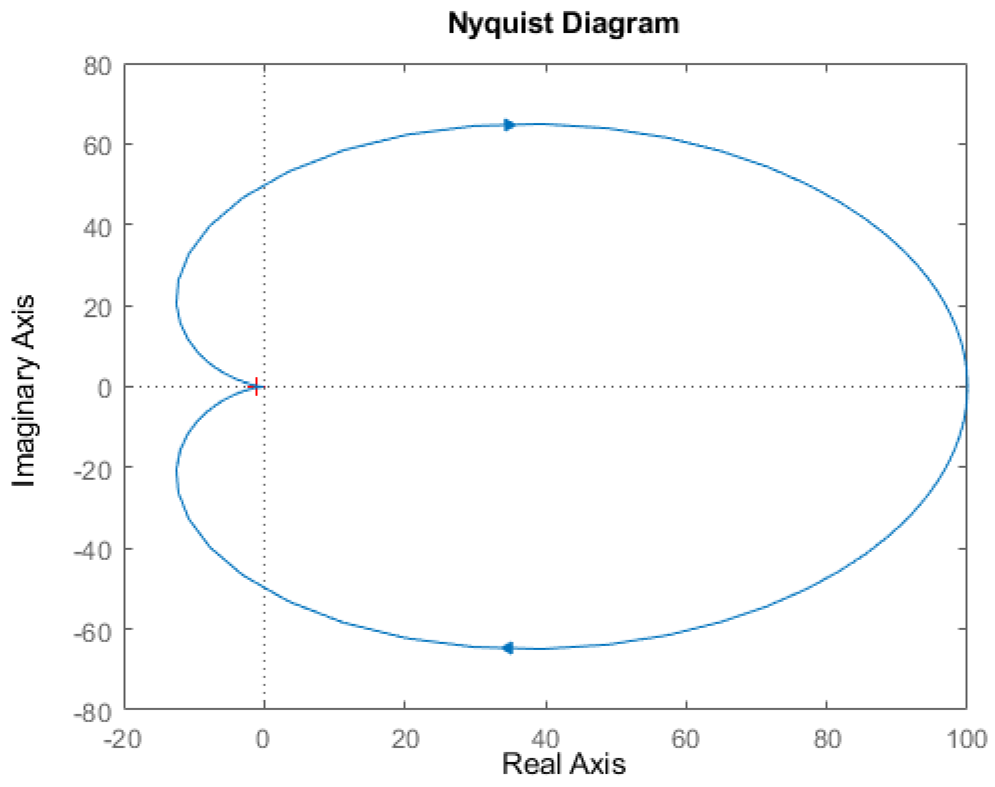
```
opt.Title.String = 'Nyquist Plot of sys';
setoptions(h,opt)
```



Zoom on Critical Point

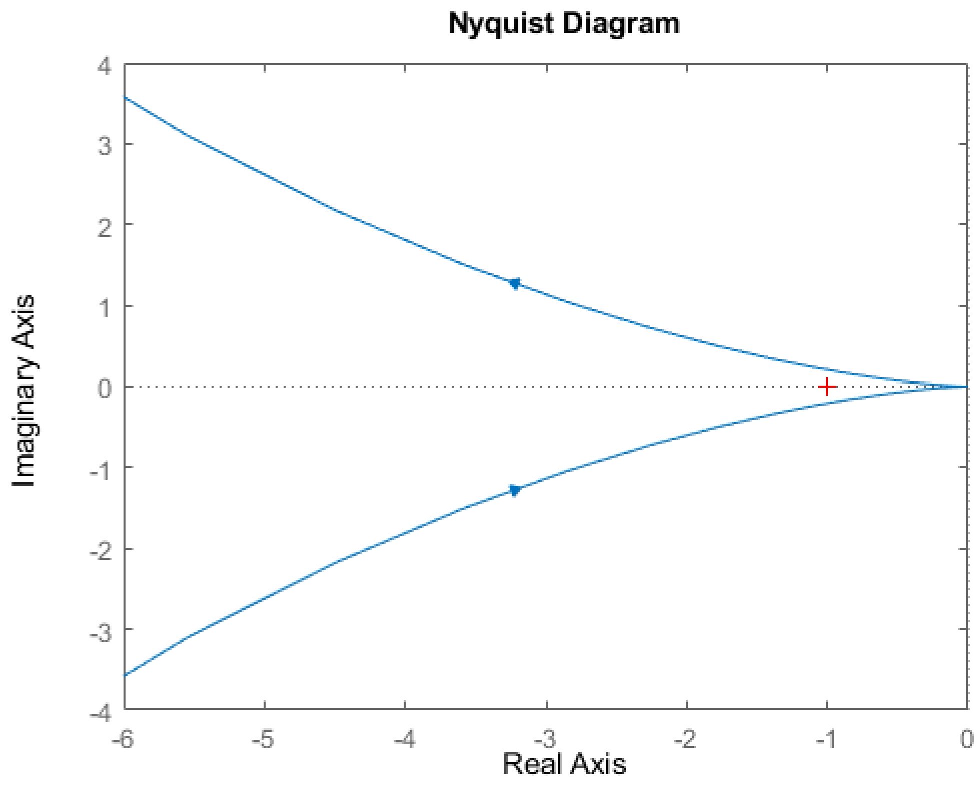
Plot the Nyquist frequency response of a dynamic system. Assign a variable name to the plot handle so that you can access it for further manipulation.

```
sys = tf(100,[1,2,1]);
h = nyquistplot(sys);
```



Zoom in on the critical point, (–1,0). You can do so interactively by right-clicking on the plot and selecting **Zoom on (-1,0)**. Alternatively, use the `zoomcp` command on the plot handle `h`.

```
zoomcp(h)
```

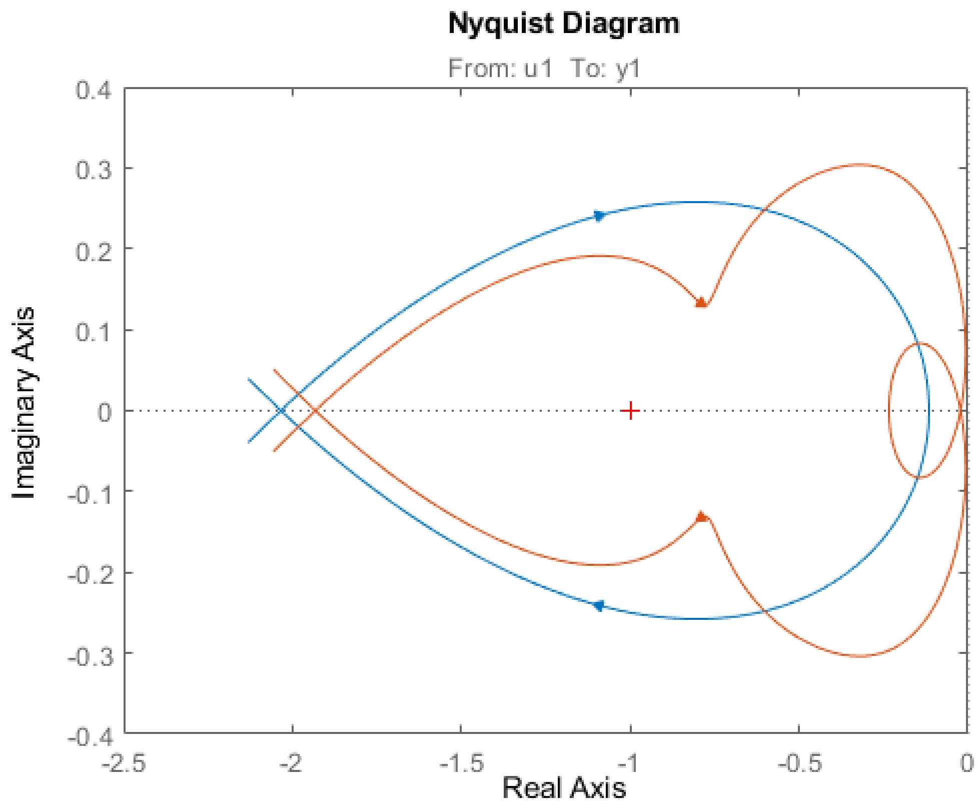


Nyquist Plot of Identified Models with Confidence Regions at Selected Points

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.

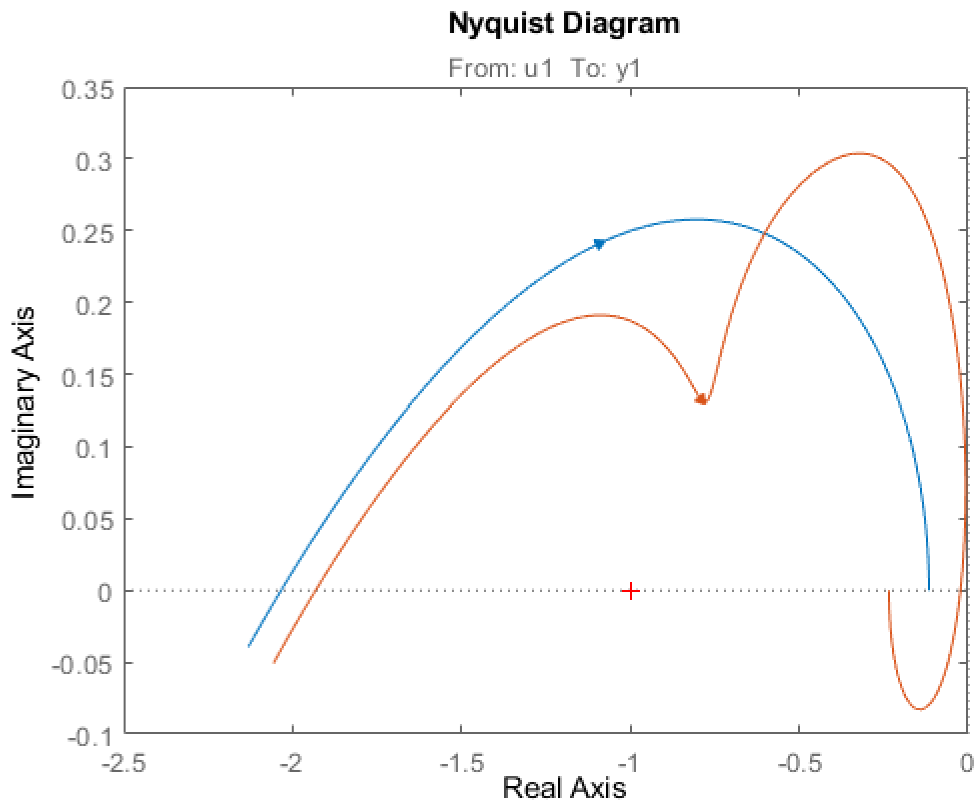
Load the identified model data and estimate the state-space models using `n4sid`. Then, plot the Nyquist diagram.

```
load iddata1
sys1 = n4sid(z1,2);
sys2 = n4sid(z1,6);
w = linspace(10,10*pi,256);
h = nyquistplot(sys1,sys2,w);
```

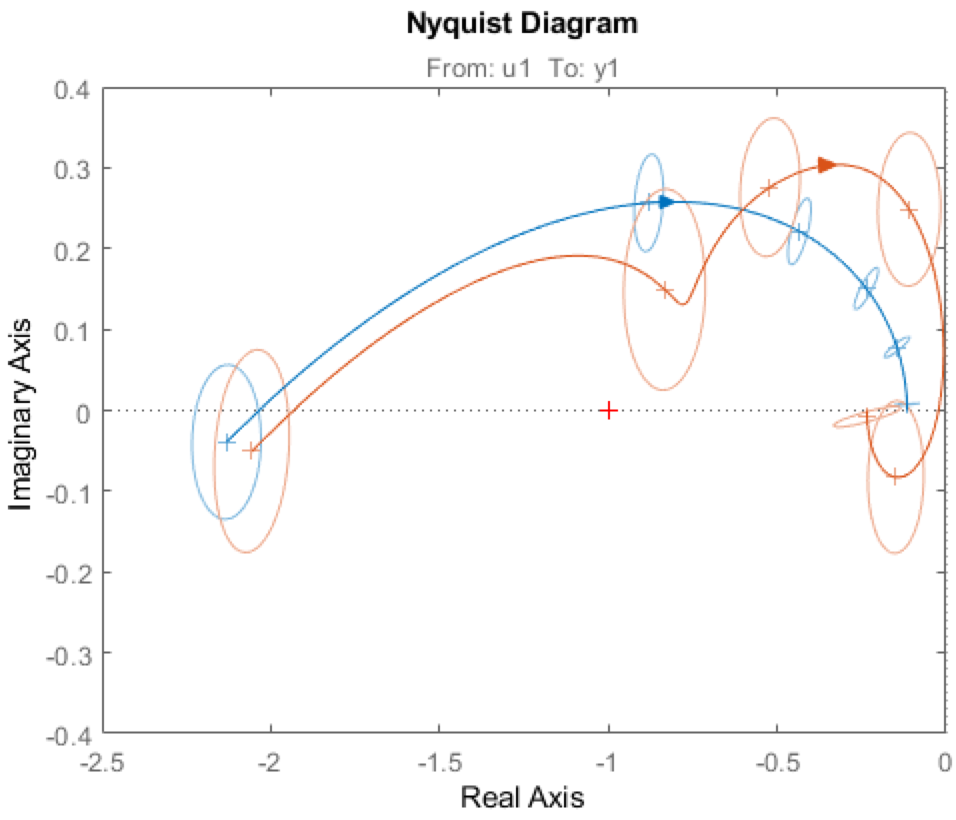


Both models produce about 76% fit to data. However, `sys2` shows higher uncertainty in its frequency response, especially close to Nyquist frequency as shown by the plot. To see this, show the confidence region at a subset of the points at which the Nyquist response is displayed.

```
setoptions(h,'ConfidenceRegionDisplaySpacing',50,...
             'ShowFullContour','off');
```



To turn on the confidence region display, right-click the plot and select **Characteristics > Confidence Region**.



Nyquist Plot with Specific Customization

For this example, consider a MIMO state-space model with 3 inputs, 3 outputs and 3 states. Create a Nyquist plot, display only the partial contour and turn the grid on.

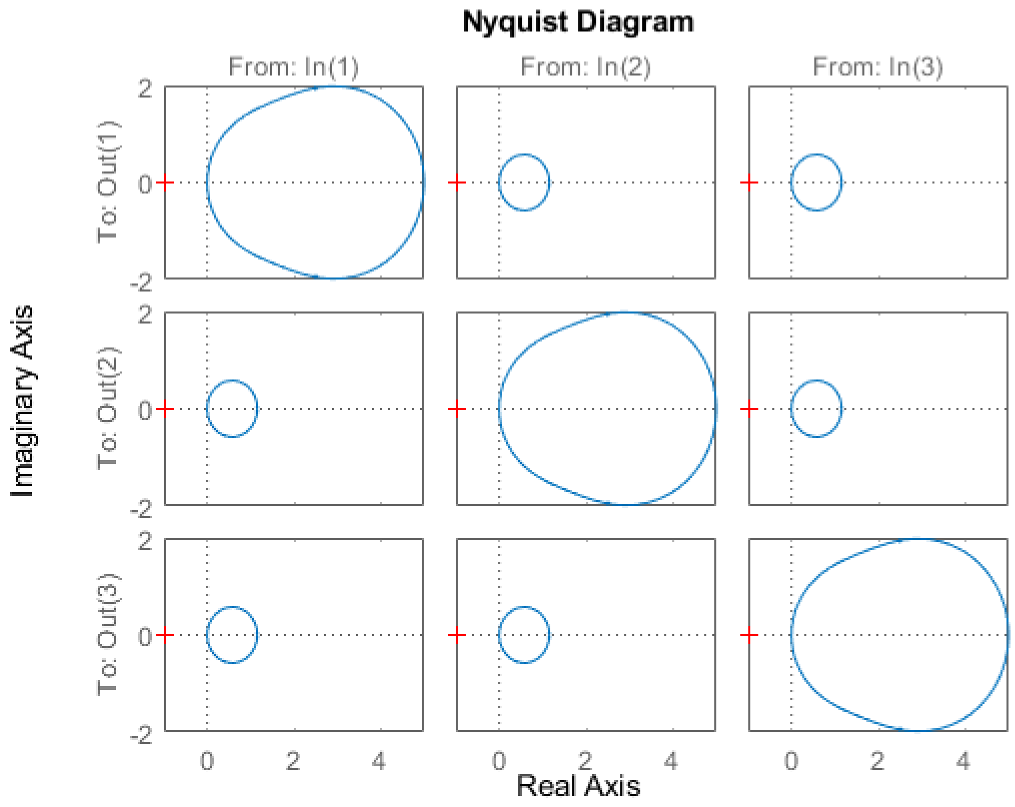
Create the MIMO state-space model `sys_mimo`.

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
size(sys_mimo)
```

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

Create a Nyquist plot with plot handle `h` and use `getoptions` for a list of the options available.

```
h = nyquistplot(sys_mimo);
```

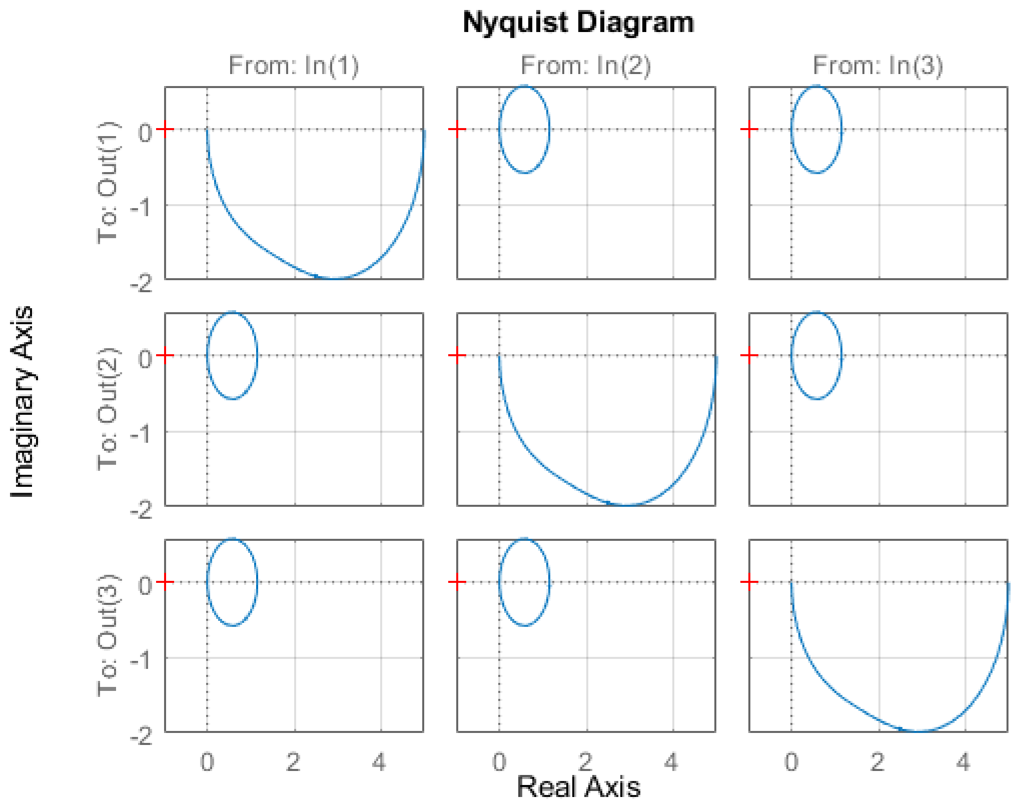


```
p = getoptions(h)
```

```
p = 

                       FreqUnits: 'rad/s'
                        MagUnits: 'dB'
                      PhaseUnits: 'deg'
                 ShowFullContour: 'on'
        ConfidenceRegionNumberSD: 1
    ConfidenceRegionDisplaySpacing: 5
                      IOGrouping: 'none'
                     InputLabels: [1x1 struct]
                    OutputLabels: [1x1 struct]
                    InputVisible: {3x1 cell}
                   OutputVisible: {3x1 cell}
                           Title: [1x1 struct]
                          XLabel: [1x1 struct]
                          YLabel: [1x1 struct]
                       TickLabel: [1x1 struct]
                            Grid: 'off'
                       GridColor: [0.1500 0.1500 0.1500]
                            XLim: {3x1 cell}
                            YLim: {3x1 cell}
                        XLimMode: {3x1 cell}
                        YLimMode: {3x1 cell}
```

Use `setoptions` to update the plot with the requires customization.

```
setoptions(h,'ShowFullContour','off','Grid','on');
```



The Nyquist plot automatically updates when you call `setoptions`. For MIMO models, `nyquistplot` produces an array of Nyquist diagrams, each plot displaying the frequency response of one I/O pair.

Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Sparse state-space models, such as `sparss` or `mechss` models. Frequency grid `w` must be specified for sparse models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value to plot the frequency response data.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model.

- Frequency-response data models such as `frd` models. For such models, the function plots the Nyquist plot at frequencies defined in the model.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models.

If `sys` is an array of models, the function plots the Nyquist responses of all models in the array on the same axes.

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'_'`	Horizontal line
`'|'`	Vertical line
`'s'`	Square
`'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'p'`	Pentagram
`'h'`	Hexagram

Color	Description
`y`	yellow
`m`	magenta
`c`	cyan

Color	Description
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

`AX` — Target axes
`Axes` object | `UIAxes` object

Target axes, specified as an `Axes` or `UIAxes` object. If you do not specify the axes and if the current axes are Cartesian axes, then `nyquistplot` plots on the current axes.

`plotoptions` — Nyquist plot options set
`NyquistPlotOptions` object

Nyquist plot options set, specified as a `NyquistPlotOptions` object. You can use this option set to customize the Nyquist plot appearance. Use `nyquistoptions` to create the option set. Settings you specify in `plotoptions` overrides the preference settings in the MATLAB session in which you run `nyquistplot`. Therefore, `plotoptions` is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

For the list of available options, see `nyquistoptions`.

`w` — Frequencies
`{wmin,wmax}` | vector

Frequencies at which to compute and plot Nyquist response, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function computes the response at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function computes the response at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values.

Specify frequencies in units of `rad/TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

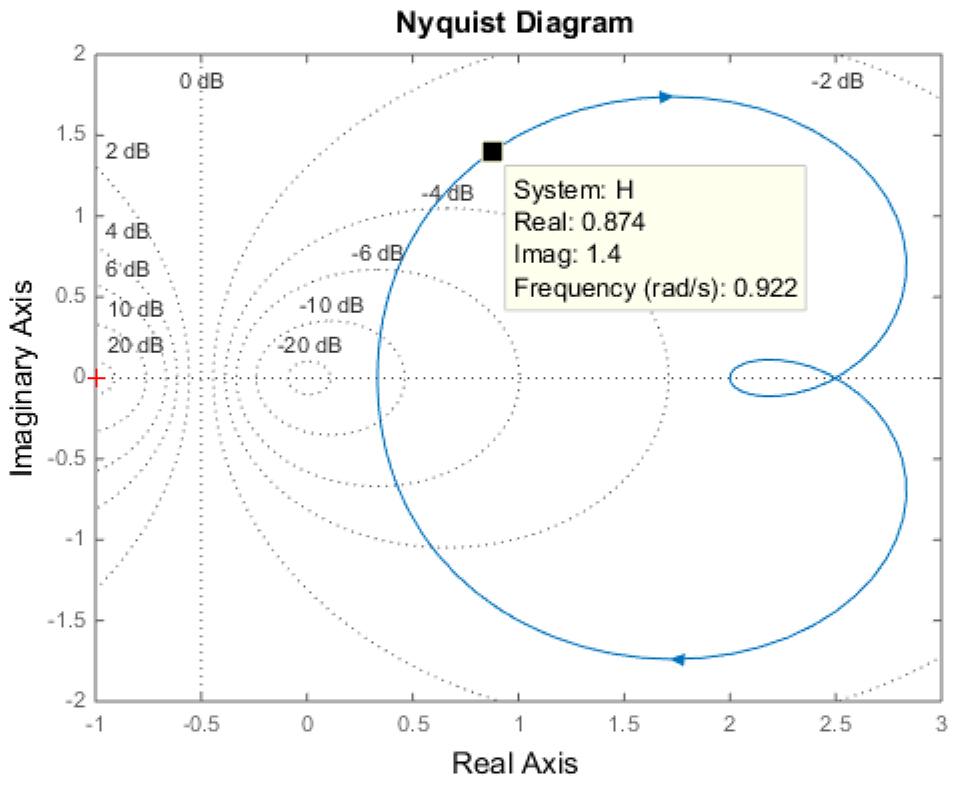
Output Arguments

`h` — Plot handle
`handle` object

Plot handle, returned as a `handle` object. Use the handle `h` to get and set the properties of the Nyquist plot using `getoptions` and `setoptions`. For the list of available options, see the *Properties and Values Reference* section in “Customizing Response Plots from the Command Line” (Control System Toolbox).

Tips

- There are two zoom options available from the right-click menu that apply specifically to Nyquist plots:
  - **Full View** — Clips unbounded branches of the Nyquist plot, but still includes the critical point (–1, 0).
  - **Zoom on (-1,0)** — Zooms around the critical point (–1,0). To access critical-point zoom programmatically, use the `zoomcp` command. See “Zoom on Critical Point” on page 1-1099.
- To activate data markers that display the real and imaginary values at a given frequency, click anywhere on the curve. The following figure shows a Nyquist plot with a data marker.



See Also

`getoptions` | `nyquist` | `setoptions` | `showConfidence` | `nyquistoptions`

Topics

“Customizing Response Plots from the Command Line” (Control System Toolbox)

Introduced in R2012a

oe

Estimate output-error polynomial model using time-domain 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)
[sys,ic] = oe( ___ )
```

Description

Output-error (OE) models are a special configuration of polynomial models, having only two active polynomials—*B* and *F*. OE models represent conventional transfer functions that relate measured inputs to outputs while also including white noise as an additive output disturbance. You can estimate OE models using time- and frequency-domain data. The `tfest` command offers the same functionality as `oe`. For `tfest`, you specify the model orders using number of poles and zeros rather than polynomial degrees. For continuous-time estimation, `tfest` provides faster and more accurate results, and is recommended.

`sys = oe(data,[nb nf nk])` estimates an OE 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.

`oe` estimates `sys` using the measured input-output data `data`, which can be in the time or the frequency domain. The orders `[nb nf nk]` define the number of parameters in each component of 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. You can use this syntax with any of the previous input-argument combinations.

`[sys,ic] = oe( ___ )` returns the estimated initial conditions as an `initialCondition` object. Use this syntax if you plan to simulate or predict the model response using the same estimation input data and then compare the response with the same estimation output data. Incorporating the initial conditions yields a better match during the first part of the simulation.

Examples

Estimate OE Polynomial Model

Estimate an OE polynomial from time-domain data using two methods to specify input delay.

Load the estimation data.

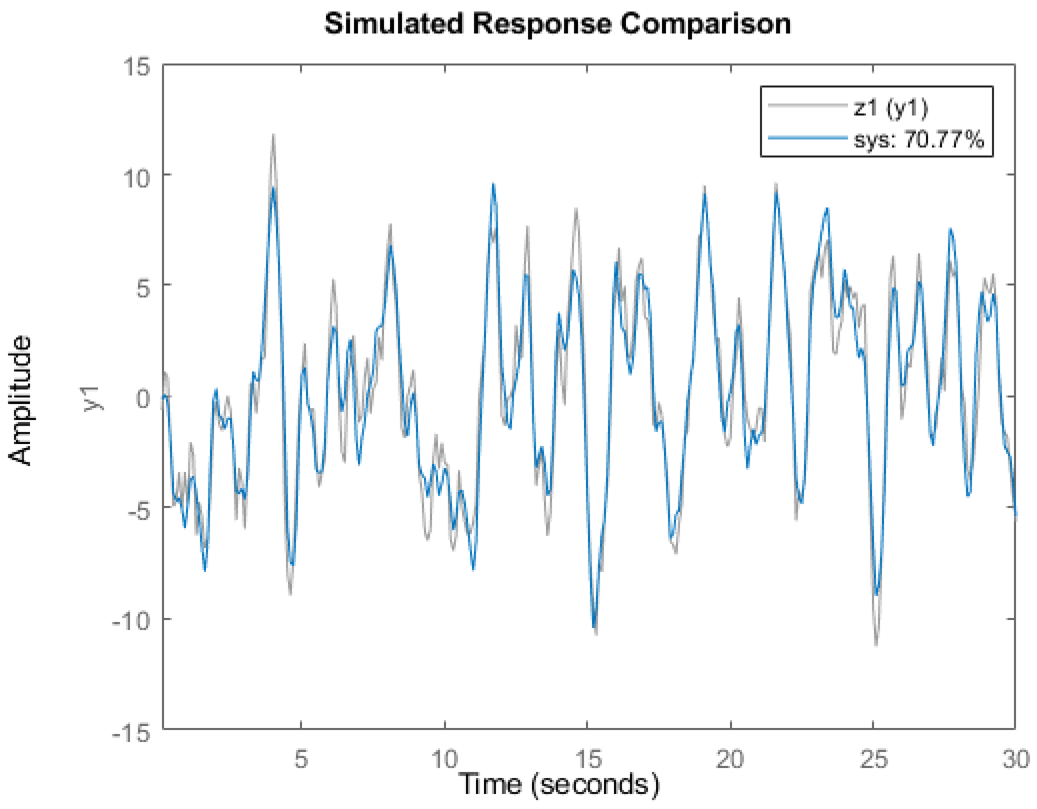
```
load iddata1 z1
```

Set the orders of the *B* and *F* polynomials `nb` and `nf`. Set the input delay `nk` to one sample. Compute the model `sys`.

```
nb = 2;
nf = 2;
nk = 1;
sys = oe(z1,[nb nf nk]);
```

Compare the simulated model response with the measured output.

```
compare(z1,sys)
```

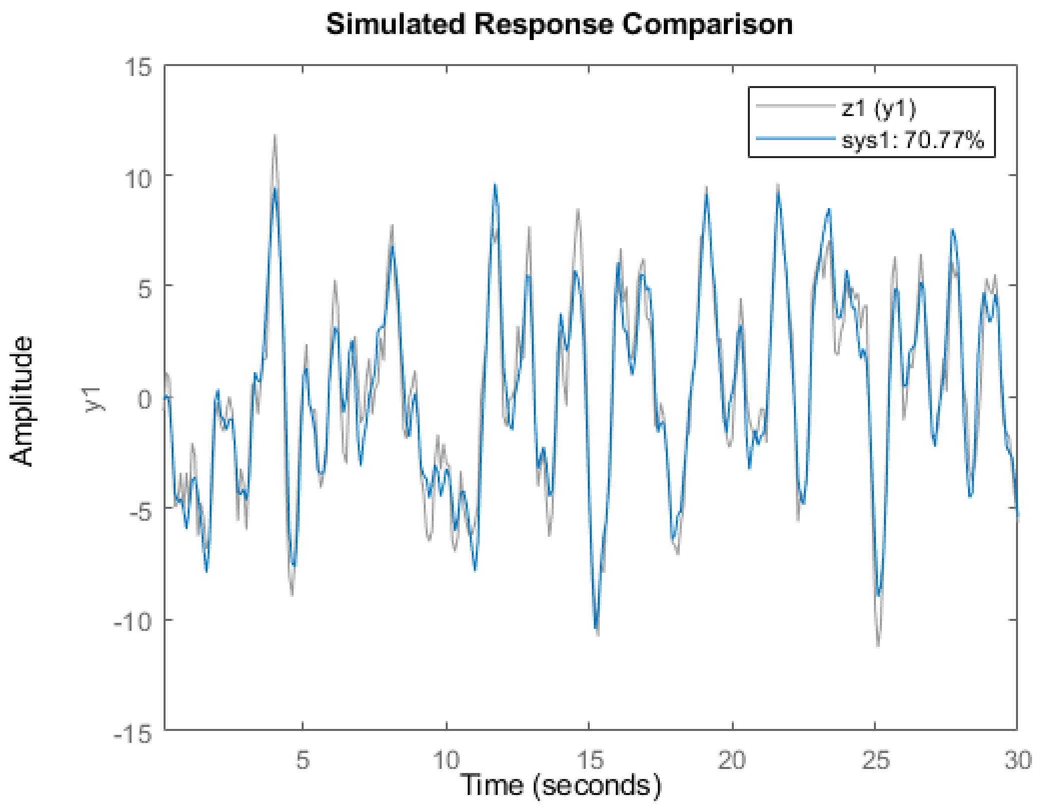


The plot shows that the fit percentage between the simulated model and the estimation data is greater than 70%.

Instead of using `nk`, you can also use the name-value pair argument `'InputDelay'` to specify the one-sample delay.

```
nk = 0;
sys1 = oe(z1,[nb nf nk],'InputDelay',1);
```

```
figure
compare(z1,sys1)
```



The results are identical.

You can view more information about the estimation by exploring the `idpoly` property `sys.Report`.

```
sys.Report
```

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

For example, find out more information about the termination conditions.

```
sys.Report.Termination
```

```
ans = struct with fields:
                 WhyStop: 'Near (local) minimum, (norm(g) < tol).'
              Iterations: 3
```

```
    FirstOrderOptimality: 0.0708
               FcnCount: 7
             UpdateNorm: 1.4809e-05
        LastImprovement: 5.1744e-06
```

The report includes information on the number of iterations and the reason the estimation stopped iterating.

Estimate Continuous-Time OE Model Using Frequency Response

Load the estimation data.

```
load oe_data1 data;
```

The `idfrd` object `data` contains the continuous-time frequency response for the following model:

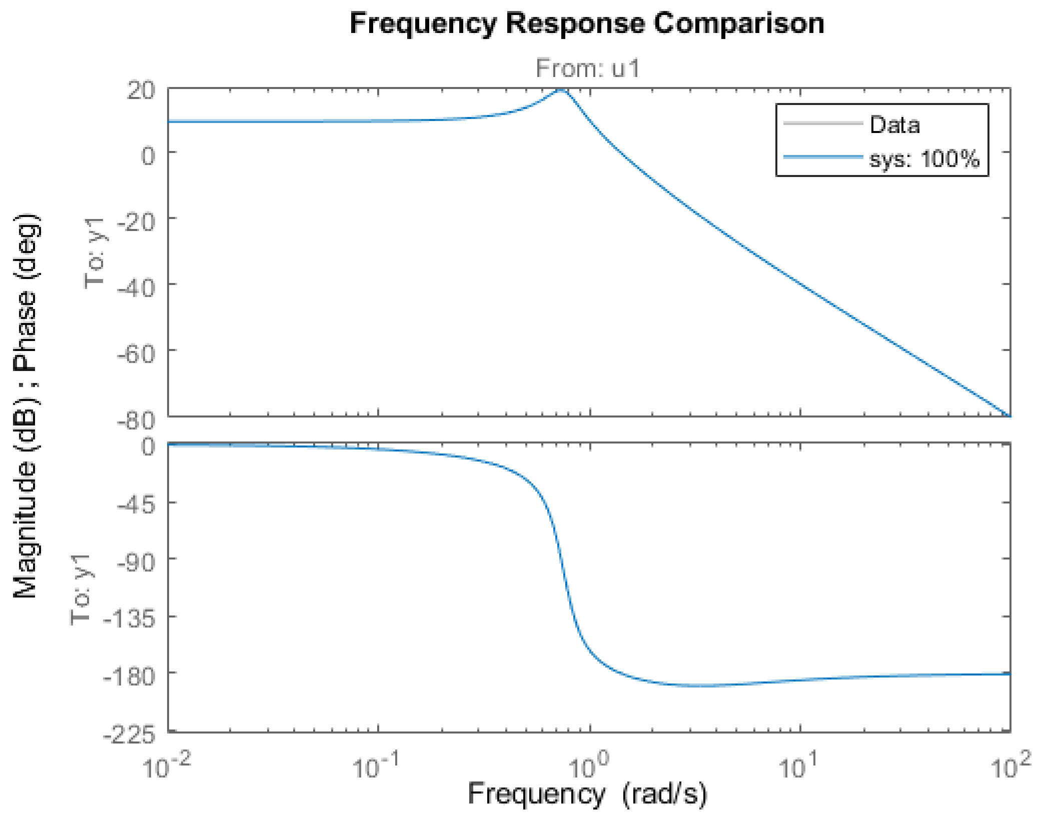
$$G(s) = \frac{s + 3}{s^3 + 2s^2 + s + 1}$$

Estimate the model.

```
nb = 2;
nf = 3;
sys = oe(data,[nb nf]);
```

Evaluate the goodness of fit.

```
compare(data,sys);
```



Estimate OE 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 the data.

```
load regularizationExampleData.mat m0simdata
```

Estimate an unregularized OE model of order 30.

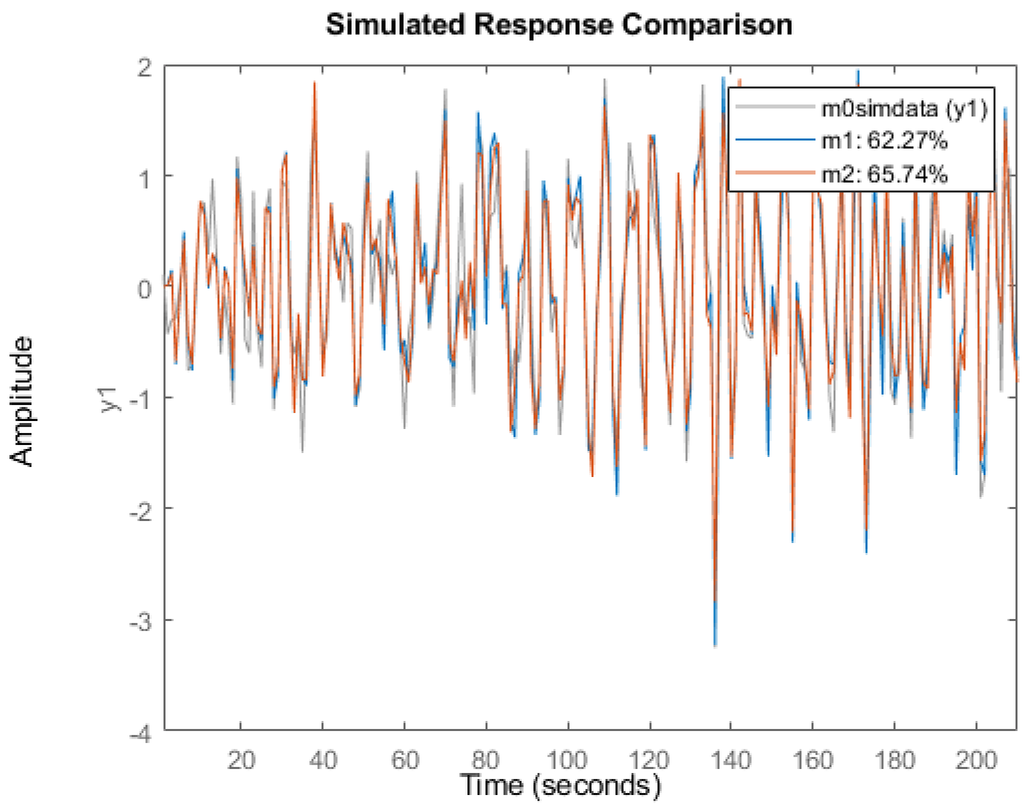
```
m1 = oe(m0simdata,[30 30 1]);
```

Obtain a regularized OE model by determining the 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.

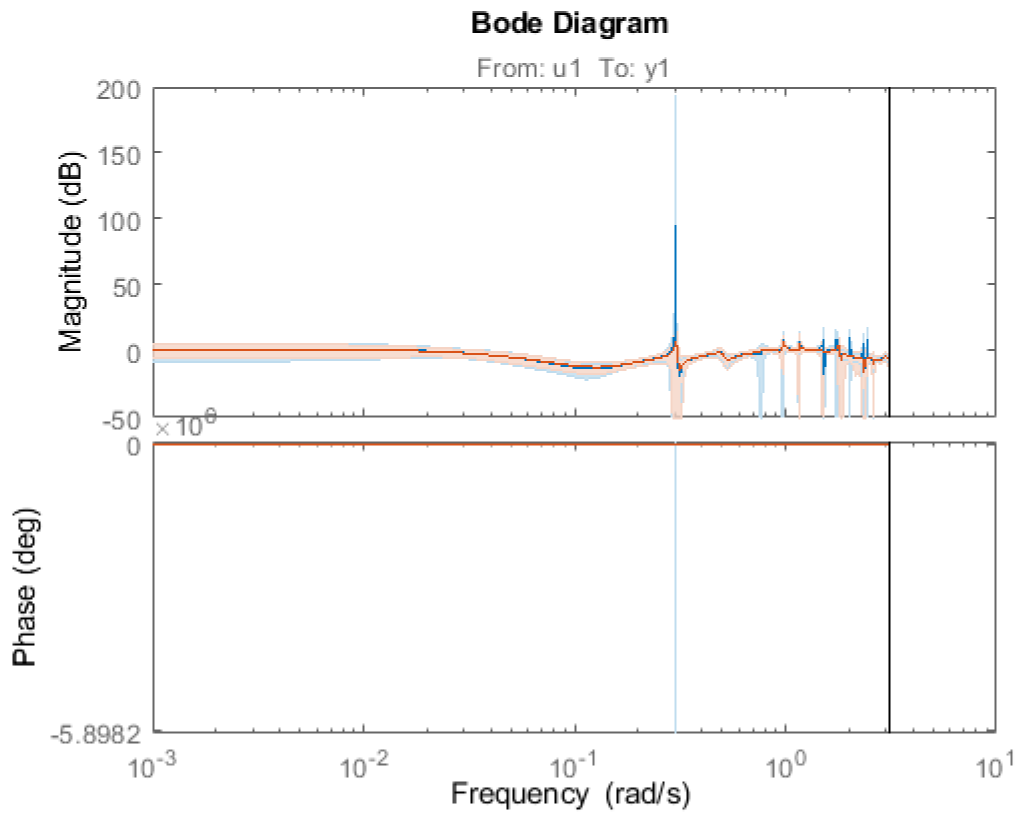
```
opt = compareOptions('InitialCondition','z');
compare(m0simdata,m1,m2,opt);
```



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 regularized model `m2` has a reduced variance compared to the unregularized model `m1`.

Estimate Continuous Model Using Band-Limited Discrete-Time Frequency-Domain Data

Load the estimation data `data` and sample time `Ts`.

```
load oe_data2.mat data Ts
```

An `iddata` object `data` contains the discrete-time frequency response for the following model:

$$G(s) = \frac{1000}{s + 500}$$

View the estimation sample time `Ts` that you loaded.

```
Ts
```

```
Ts = 1.0000e-03
```

This value matches the property `data.Ts`.

```
data.Ts
```

```
ans = 1.0000e-03
```

You can estimate a continuous model from `data` by limiting the input and output frequency bands to the Nyquist frequency. To do so, specify the estimation prefilter option `'WeightingFilter'` to define a passband from `0` to `0.5*pi/Ts` rad/s. The software ignores any response values with frequencies outside of that passband.

```
opt = oeOptions('WeightingFilter',[0 0.5*pi/Ts]);
```

Set the `Ts` property to `0` to treat `data` as continuous-time data.

```
data.Ts = 0;
```

Estimate the continuous model.

```
nb = 1;
nf = 3;
sys = oe(data,[nb nf],opt);
```

Obtain Initial Conditions

Load the data.

```
load iddata1ic z1i
```

Estimate an OE polynomial model `sys` and return the initial conditions in `ic`.

```
nb = 2;
nf = 2;
nk = 1;
[sys,ic] = oe(z1i,[nb,nf,nk]);
ic
```

```
ic = 
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [0.9428 0.4824]
    Ts: 0.1000
```

`ic` is an `initialCondition` object that encapsulates the free response of `sys`, in state-space form, to the initial state vector in `X0`. You can incorporate `ic` when you simulate `sys` with the `z1i` input signal and compare the response with the `z1i` output signal.

Input Arguments

data — Estimation data

`iddata` object | `frd` object | `idfrd` object

Estimation data, specified as an `iddata` object, an `frd` object, or an `idfrd` 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 properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — `'Frequency'`

Time-domain estimation data must be uniformly sampled. By default, the software sets the sample time of the model to the sample time of the estimation data.

For multiexperiment data, the sample times and intersample behavior of all the experiments must match.

You can compute discrete-time models from time-domain data or discrete-time frequency-domain data. Use `tfest` to compute continuous-time models.

`[nb nf nk]` — OE model orders
integer row vector | row vector of integer matrices

OE model orders, specified as a 1-by-3 vector or a vector of integer matrices.

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, the elements of `[nb nf nk]` are as follows:

- `nb` — Order of the $B(q)$ polynomial + 1, which is equivalent to the length of the $B(q)$ polynomial. `nb` is an N_y-by-N_u matrix. N_y is the number of outputs and N_u is the number of inputs.
- `nf` — Order of the F polynomial. `nf` is an N_y-by-N_u matrix.
- `nk` — Input delay, expressed as the number of samples. `nk` is an N_y-by-N_u matrix. The delay appears as leading zeros of the B polynomial.

For estimation using continuous-time frequency-domain data, specify only `[nb nf]` and omit `nk`. For an example, see “Estimate Continuous-Time OE Model Using Frequency Response” on page 1-1113.

`init_sys` — Linear system
`idpoly` model | linear model | structure

Linear system that configures the initial parameterization of `sys`, specified as an `idpoly` model, another linear model, or a structure. You obtain `init_sys` either by performing an estimation using measured data or by direct construction.

If `init_sys` is an `idpoly` model of the OE structure, `oe` uses the parameter values of `init_sys` as the initial guess for estimating `sys`. The sample time of `init_sys` must match the sample time of the data.

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 the OE structure, the software first converts `init_sys` to an OE structure model. `oe` uses the parameters of the resulting model as the initial guess for estimating `sys`.

If you do not specify `opt` and `init_sys` was obtained by estimation, then the software uses estimation options from `init_sys.Report.OptionsUsed`.

`opt` — Estimation options
`oeOptions` option set

Estimation options, specified as an `oeOptions` option set. Options specified by `opt` include:

- Estimation objective
- Handling of initial conditions
- Numerical search method and the associated options

For examples of specifying estimation options, see “Estimate Continuous Model Using Band-Limited Discrete-Time Frequency-Domain Data” on page 1-1116.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InputDelay',1`

`InputDelay` — Input delays
0 (default) | positive integer vector | integer scalar

Input delays for each input channel, specified as the comma-separated pair consisting of `'InputDelay'` and a numeric vector.

- For continuous-time models, specify `'InputDelay'` in the time units stored in the `TimeUnit` property.
- For discrete-time models, specify `'InputDelay'` in integer multiples of the sample time `Ts`. For example, setting `'InputDelay'` to `3` specifies a delay of three sampling periods.

For a system with N_u inputs, set `InputDelay` to an N_u-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.

To apply the same delay to all channels, specify `'InputDelay'` as a scalar.

For an example, see “Estimate OE Polynomial Model” on page 1-1110.

IODelay — Transport delays

0 (default) | scalar | numeric array

Transport delays for each input-output pair, specified as the comma-separated pair consisting of `'IODelay'` and a numeric array.

- For continuous-time models, specify `'IODelay'` in the time units stored in the `TimeUnit` property.
- For discrete-time models, specify `'IODelay'` in integer multiples of the sample time `Ts`. For example, setting `'IODelay'` to `4` specifies a transport delay of four sampling periods.

For a system with N_u inputs and N_y outputs, set `'IODelay'` to an N_y-by-N_u matrix. Each entry is an integer value representing the transport delay for the corresponding input-output pair.

To apply the same delay to all channels, specify `'IODelay'` as a scalar.

You can specify `'IODelay'` as an alternative to the `nk` value. Doing so simplifies the model structure by reducing the number of leading zeros in the *B* polynomial. In particular, you can represent `max(nk-1,0)` leading zeros as input-output delays using `'IODelay'` instead.

Output Arguments

sys — OE polynomial model

`idpoly` object

OE polynomial model that fits the estimation data, returned as an `idpoly` model object. This model is created using the specified model orders, delays, and estimation options. The sample time of `sys` matches the sample time of the estimation data. Therefore, `sys` is always a discrete-time model when estimated from time-domain data. For continuous-time model identification using time-domain data, use `tfest`.

The `Report` property of the model stores information about the estimation results and options used. `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.
`InitialCondition`	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'`.

Report Field	Description
`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: **Field** — **Description** `FitPercent` — Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE). `LossFcn` — Value of the loss function when the estimation completes. `MSE` — Mean squared error (MSE) measure of how well the response of the model fits the estimation data. `FPE` — Final prediction error for the model. `AIC` — Raw Akaike Information Criteria (AIC) measure of model quality. `AICc` — Small-sample-size corrected AIC. `nAIC` — Normalized AIC. `BIC` — Bayesian Information Criteria (BIC).
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `oeOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** / **Description** `Name` — Name of the data set. `Type` — Data type. `Length` — Number of data samples. `Ts` — Sample time. `InterSample` — Input intersample behavior, returned as one of the following values: • `'zoh'` — 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. `InputOffset` — Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`. `OutputOffset` — Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.
`Termination`	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** / **Description** `WhyStop` — Reason for terminating the numerical search. `Iterations` — Number of search iterations performed by the estimation algorithm. `FirstOrderOptimality` — ∞-norm of the gradient search vector when the search algorithm terminates. `FcnCount` — Number of times the objective function was called. `UpdateNorm` — Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `LastImprovement` — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `Algorithm` — Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used. For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.

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

ic — Initial conditions
initialCondition object | object array of initialCondition values

Estimated initial conditions, returned as an initialCondition object or an object array of initialCondition values.

- For a single-experiment data set, ic represents, in state-space form, the free response of the transfer function model (A and C matrices) to the estimated initial states (x_0).
- For a multiple-experiment data set with N_e experiments, ic is an object array of length N_e that contains one set of initialCondition values for each experiment.

If oe returns ic values of 0 and the you know that you have non-zero initial conditions, set the 'InitialCondition' option in oeOptions to 'estimate' and pass the updated option set to oe. For example:

```
opt = oeOptions('InitialCondition','estimate')
[sys,ic] = oe(data,np,nz,opt)
```

The default 'auto' setting of 'InitialCondition' uses the 'zero' method when the initial conditions have a negligible effect on the overall estimation-error minimization process. Specifying 'estimate' ensures that the software estimates values for ic.

For more information, see initialCondition. For an example of using this argument, see “Obtain Initial Conditions” on page 1-1117.

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:

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

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

Continuous-Time Output-Error Model

If data is continuous-time frequency-domain data, oe estimates a continuous-time model with the following 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 sample delay nk does not exist in the continuous case, and you should not specify nk when you command the estimation. Instead, express any system delay using the name-value pair argument 'IODelay' along with the system delay in the time units that are stored in the property TimeUnit. For example, suppose that your continuous system has a delay of iod seconds. Use model = oe(data,[nb nf],'IODelay',iod).

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `oeOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = oeOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

`oeOptions` | `tfest` | `arx` | `armax` | `iv4` | `n4sid` | `bj` | `polyest` | `idpoly` | `iddata` | `idfrd` | `sim` | `compare`

Topics

"What Are Polynomial Models?"
"Data Supported by Polynomial Models"
"Regularized Estimates of Model Parameters"
"Apply Initial Conditions when Simulating Identified Linear Models"

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 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`

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

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

If `EstimateCovariance` 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` — Numerical search method used for iterative parameter estimation
`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:
  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

SearchOptions Structure When SearchMethod is Specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description	Default
Tolerance	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
MaxIterations	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	1e-6
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar.	1e-6
MaxIterations	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance.	100

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

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

  - 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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
`oe` | `idfilt`

Topics
“Loss Function and Model Quality Metrics”

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

particleFilter

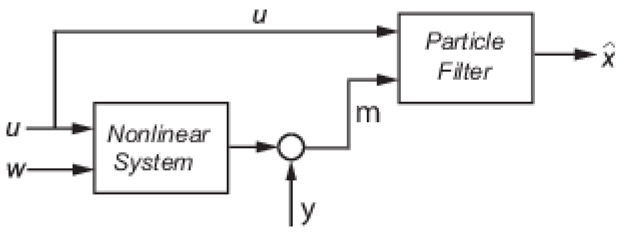
Particle filter object for online state estimation

Description

A particle filter is a recursive, Bayesian state estimator that uses discrete particles to approximate the posterior distribution of an estimated state. It is useful for online state estimation when measurements and a system model, that relates model states to the measurements, are available. The particle filter algorithm computes the state estimates recursively and involves initialization, prediction, and correction steps.

`particleFilter` creates an object for online state estimation of a discrete-time nonlinear system using the discrete-time particle filter algorithm.

Consider a plant with states *x*, input *u*, output *m*, process noise *w*, and measurement *y*. Assume that you can represent the plant as a nonlinear system.



The algorithm computes the state estimates $\hat{x}$ of the nonlinear system using the state transition and measurement likelihood functions you specify.

The software supports arbitrary nonlinear state transition and measurement models, with arbitrary process and measurement noise distributions.

To perform online state estimation, create the nonlinear state transition function and measurement likelihood function. Then construct the `particleFilter` object using these nonlinear functions. After you create the object:

1. Initialize the particles using the `initialize` command.
2. Predict state estimates at the next step using the `predict` command.
3. Correct the state estimates using the `correct` command.

The prediction step uses the latest state to predict the next state based on the state transition model you provide. The correction step uses the current sensor measurement to correct the state estimate. The algorithm optionally redistributes, or resamples, the particles in the state space to match the posterior distribution of the estimated state. Each particle represents a discrete state hypothesis of these state variables. The set of all particles is used to help determine the state estimate.

Creation

Syntax

`pf = particleFilter(StateTransitionFcn,MeasurementLikelihoodFcn)`

Object Description

`pf = particleFilter(StateTransitionFcn,MeasurementLikelihoodFcn)` creates a particle filter object for online state estimation of a discrete-time nonlinear system. `StateTransitionFcn` is a function that calculates the particles (state hypotheses) at the next time step, given the state vector at a time step. `MeasurementLikelihoodFcn` is a function that calculates the likelihood of each particle based on sensor measurements.

After creating the object, use the `initialize` command to initialize the particles with a known mean and covariance or uniformly distributed particles within defined bounds. Then, use the `correct` and `predict` commands to update particles (and hence the state estimate) using sensor measurements.

Input Arguments

`StateTransitionFcn` — State transition function
function handle

State transition function, specified as a function handle, determines the transition of particles (state hypotheses) between time steps. Also a property of the `particleFilter` object. For more information, see “Properties” on page 1-1142.

`MeasurementLikelihoodFcn` — Measurement likelihood function
function handle

Measurement likelihood function, specified as a function handle, is used to calculate the likelihood of particles (state hypotheses) from sensor measurements. Also a property of the `particleFilter` object. For more information, see “Properties” on page 1-1142.

Properties

`NumStateVariables` — Number of state variables
[ ] (default) | scalar

Number of state variables, specified as a scalar. This property is read-only and is set using `initialize`. The number of states is implicit based on the specified matrices for the initial mean of particles, or the state bounds.

`NumParticles` — Number of particles used in the filter
[ ] (default) | scalar

Number of particles used in the filter, specified as a scalar. Each particle represents a state hypothesis. You specify this property only by using `initialize`.

`StateTransitionFcn` — State transition function
function handle

State transition function, specified as a function handle, determines the transition of particles (state hypotheses) between time steps. This function calculates the particles at the next time step, including the process noise, given particles at a time step.

In contrast, the state transition function for the `extendedKalmanFilter` and `unscentedKalmanFilter` generates a single state estimate at a given time step.

You write and save the state transition function for your nonlinear system, and specify it as a function handle when constructing the `particleFilter` object. For example, if `vdpParticleFilterStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpParticleFilterStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The function signature is as follows:

```
function predictedParticles = myStateTransitionFcn(previousParticles,varargin)
```

The `StateTransitionFcn` function accepts at least one input argument. The first argument is the set of particles `previousParticles` that represents the state hypotheses at the previous time step. The optional use of `varargin` in the function enables you to input any extra parameters that are relevant for predicting the next state, using `predict`, as follows:

```
predict(pf,arg1,arg2)
```

If `StateOrientation` is 'column', then `previousParticles` is a `NumStateVariables`-by-`NumParticles` array. If `StateOrientation` is 'row', then `previousParticles` is a `NumParticles`-by-`NumStateVariables` array.

`StateTransitionFcn` must return exactly one output, `predictedParticles`, which is the set of predicted particle locations for the current time step (array with same dimensions as `previousParticles`).

`StateTransitionFcn` must include the random process noise (from any distribution suitable for your application) in the `predictedParticles`.

To see an example of a state transition function with the `StateOrientation` property set to 'column', type `edit vdpParticleFilterStateFcn` at the command line.

`MeasurementLikelihoodFcn` — Measurement likelihood function
function handle

Measurement likelihood function, specified as a function handle, is used to calculate the likelihood of particles (state hypotheses) using the sensor measurements. For each state hypothesis (particle), the function first calculates an N-element measurement hypothesis vector. Then the likelihood of each measurement hypothesis is calculated based on the sensor measurement and the measurement noise probability distribution.

In contrast, the measurement function for `extendedKalmanFilter` and `unscentedKalmanFilter` takes a single state hypothesis and returns a single measurement estimate.

You write and save the measurement likelihood function based on your measurement model, and use it to construct the object. For example, if `vdpMeasurementLikelihoodFcn.m` is the measurement likelihood function, specify `MeasurementLikelihoodFcn` as `@vdpMeasurementLikelihoodFcn`. You can also specify `MeasurementLikelihoodFcn` as a function handle to an anonymous function.

The function signature is as follows:

```
function likelihood = myMeasurementLikelihoodFcn(predictedParticles,measurement,varargin)
```

The `MeasurementLikelihoodFcn` function accepts at least two input arguments. The first argument is the set of particles `predictedParticles` that represents the predicted state hypothesis. If `StateOrientation` is 'column', then `predictedParticles` is a `NumStateVariables`-by-`NumParticles` array. If `StateOrientation` is 'row', then `predictedParticles` is a `NumParticles`-by-`NumStateVariables` array. The second argument, `measurement`, is the N-element sensor measurement at the current time step. You can provide additional input arguments using `varargin`.

The `MeasurementLikelihoodFcn` must return exactly one output, `likelihood`, a vector with `NumParticles` length, which is the likelihood of the given `measurement` for each particle (state hypothesis).

To see an example of a measurement likelihood function, type `edit vdpMeasurementLikelihoodFcn` at the command line.

`IsStateVariableCircular` — Whether the state variables have a circular distribution
[ ] (default) | logical array

Whether the state variables have a circular distribution, specified as a logical array.

This is a read-only property and is set using `initialize`.

Circular (or angular) distributions use a probability density function with a range of `[-pi,pi]`. `IsStateVariableCircular` is a row-vector with `NumStateVariables` elements. Each vector element indicates whether the associated state variable is circular.

`ResamplingPolicy` — Policy settings that determine when to trigger resampling
`particleResamplingPolicy` object

Policy settings that determine when to trigger resampling, specified as a `particleResamplingPolicy` object.

The resampling of particles is a vital step in estimating states using a particle filter. It enables you to select particles based on the current state, instead of using the particle distribution given at initialization. By continuously resampling the particles around the current estimate, you can get more accurate tracking and improve long-term performance.

You can trigger resampling either at fixed intervals or dynamically, based on the number of effective particles. The minimum effective particle ratio is a measure of how well the current set of particles approximates the posterior distribution. The number of effective particles is calculated by:

$$N_{eff} = \frac{1}{\sum_{i=1}^{N}\left(w^i\right)^2}$$

In this equation, `N` is the number of particles, and `w` is the normalized weight of each particle. The effective particle ratio is then N_{eff} / `NumParticles`. Therefore, the effective particle ratio is a function of the weights of all the particles. After the weights of the particles reach a low enough value, they are not contributing to the state estimation. This low value triggers resampling, so the particles are closer to the current state estimation and have higher weights.

The following properties of the `particleResamplingPolicy` object can be modified to control when resampling is triggered:

Property	Value	Description
TriggerMethod	'ratio' (default) | 'interval'	It is a method to determine when resampling occurs, based on the value chosen. The `'interval'` value triggers resampling at regular time steps of the particle filter operation. The `'ratio'` value triggers resampling based on the ratio of effective total particles.
SamplingInterval	1 (default) | scalar	Fixed interval between resampling, specified as a scalar. This interval determines during which correction steps the resampling is executed. For example, a value of 2 means the resampling is executed every second correction step. A value of `inf` means that resampling is never executed. This property only applies with the `TriggerMethod` is set to `'interval'`.

Property	Description
MinEffectiveParticleRatio	It is the minimum desired ratio of the effective number of particles to the total number of particles `NumParticles`. The effective number of particles is a measure of how well the current set of particles approximates the posterior distribution. A lower effective particle ratio implies that a lower number of particles are contributing to the estimation and resampling is required. If the ratio of the effective number of particles to the total number of particles `NumParticles` falls below the `MinEffectiveParticleRatio`, a resampling step is triggered.

`ResamplingMethod` — Method used for particle resampling
`'multinomial'` (default) | `'residual'` | `'stratified'` | `'systematic'`

Method used for particle resampling, specified as one of the following:

- `'multinomial'` — Multinomial resampling, also called simplified random sampling, generates `N` random numbers independently from the uniform distribution in the open interval `(0,1)` and uses them to select particles proportional to their weight.
- `'residual'` — Residual resampling consists of two stages. The first stage is a deterministic replication of each particle that have weights larger than `1/N`. The second stage consists of random sampling using the remainder of the weights (labelled as residuals).
- `'stratified'` — Stratified resampling divides the whole population of particles into subsets called strata. It pre-partitions the `(0,1)` interval into `N` disjoint sub-intervals of size `1/N`. The random numbers are drawn independently in each of these sub-intervals and the sample indices chosen in the strata.
- `'systematic'` — Systematic resampling is similar to stratified resampling as it also makes use of strata. One distinction is that it only draws one random number from the open interval `(0,1/N)` and the remaining sample points are calculated deterministically at a fixed `1/N` step size.

StateEstimationMethod — Method used for extracting a state estimate from particles
'mean' (default) | 'maxweight'

Method used for extracting a state estimate from particles, specified as one of the following:

- 'mean' - The object outputs the weighted mean of the particles, depending on the properties Weights and Particles, as the state estimate.
- 'maxweight' - The object outputs the particle with the highest weight as the state estimate.

Particles — Array of particle values
[ ] (default) | array

Array of particle values, specified as an array based on the StateOrientation property:

- If StateOrientation is 'row' then Particles is an NumParticles-by-NumStateVariables array.
- If StateOrientation is 'column' then Particles is an NumStateVariables-by-NumParticles array.

Each row or column corresponds to a state hypothesis (a single particle).

Weights — Particle weights
[ ] (default) | vector

Particle weights, defined as a vector based on the value of the StateOrientation property:

- If StateOrientation is 'row' then Weights is a NumParticles-by-1 vector, where each weight is associated with the particle in the same row in the Particles property.
- If StateOrientation is 'column' then Weights is a 1-by-NumParticles vector, where each weight is associated with the particle in the same column in the Particles property.

State — Current state estimate
[ ] (default) | vector

Current state estimate, defined as a vector based on the value of the StateOrientation property:

- If StateOrientation is 'row' then State is a 1-by-NumStateVariables vector
- If StateOrientation is 'column' then State is a NumStateVariables-by-1 vector

State is a read-only property, and is derived from Particles based on the StateEstimationMethod property. Refer to “StateEstimationMethod” on page 1-0 for details on how the value of State is determined.

State along with StateCovariance can also be determined using getStateEstimate.

StateCovariance — Current estimate of state estimation error covariance
NumStateVariables-by-NumStateVariables array (default) | [ ] | array

Current estimate of state estimation error covariance, defined as an NumStateVariables-by-NumStateVariables array. StateCovariance is a read-only property and is calculated based on the StateEstimationMethod. If you specify a state estimation method that does not support covariance, then the function returns StateCovariance as [ ].

StateCovariance and State can be determined together using getStateEstimate.

Object Functions

initialize	Initialize the state of the particle filter
predict	Predict state and state estimation error covariance at next time step using extended or unscented Kalman filter, or particle filter
correct	Correct state and state estimation error covariance using extended or unscented Kalman filter, or particle filter and measurements
getStateEstimate	Extract best state estimate and covariance from particles
clone	Copy online state estimation object

Examples

Create Particle Filter Object for Online State Estimation

To create a particle filter object for estimating the states of your system, create appropriate state transition function and measurement likelihood function for the system.

In this example, the function `vdpParticleFilterStateFcn` describes a discrete-time approximation to van der Pol oscillator with nonlinearity parameter, mu, equal to 1. In addition, it models Gaussian process noise. `vdpMeasurementLikelihood` function calculates the likelihood of particles from the noisy measurements of the first state, assuming a Gaussian measurement noise distribution.

Create the particle filter object. Use function handles to provide the state transition and measurement likelihood functions to the object.

```
myPF = particleFilter(@vdpParticleFilterStateFcn,@vdpMeasurementLikelihoodFcn);
```

To initialize and estimate the states and state estimation error covariance from the constructed object, use the `initialize`, `predict`, and `correct` commands.

Estimate States Online using Particle Filter

Load the van der Pol ODE data, and specify the sample time.

`vdpODEdata.mat` contains a simulation of the van der Pol ODE with nonlinearity parameter mu=1, using ode45, with initial conditions `[2;0]`. The true state was extracted with sample time `dt = 0.05`.

```
load ('vdpODEdata.mat','xTrue','dt')
tSpan = 0:dt:5;
```

Get the measurements. For this example, a sensor measures the first state with a Gaussian noise with standard deviation `0.04`.

```
sqrtR = 0.04;
yMeas = xTrue(:,1) + sqrtR*randn(numel(tSpan),1);
```

Create a particle filter, and set the state transition and measurement likelihood functions.

```
myPF = particleFilter(@vdpParticleFilterStateFcn,@vdpMeasurementLikelihoodFcn);
```

Initialize the particle filter at state `[2; 0]` with unit covariance, and use `1000` particles.

```
initialize(myPF,1000,[2;0],eye(2));
```

Pick the `mean` state estimation and `systematic` resampling methods.

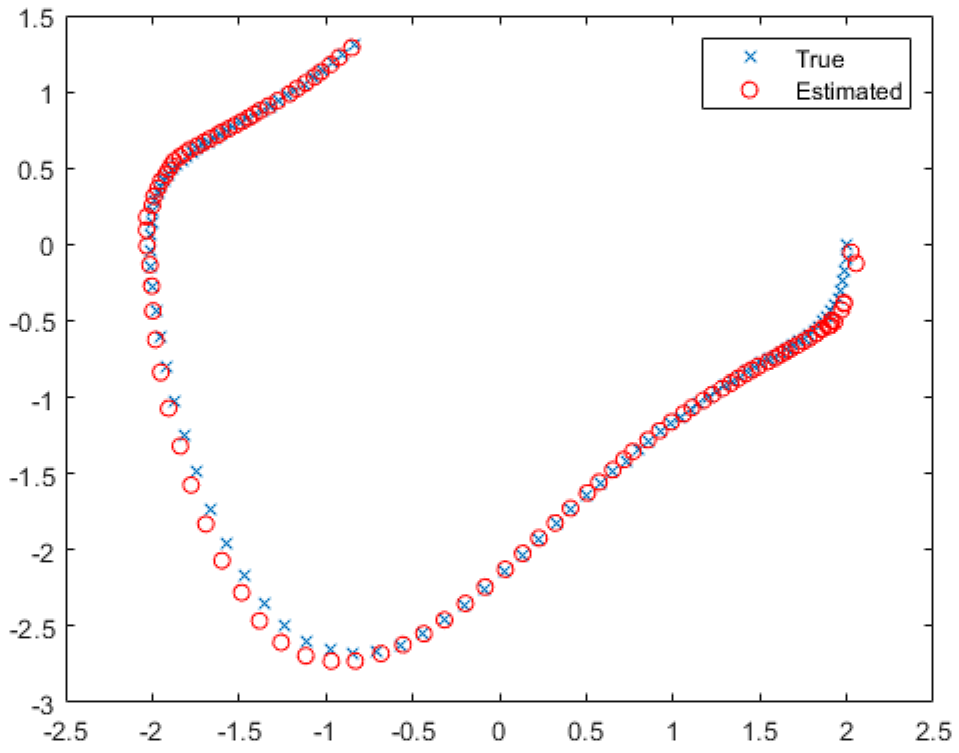
```
myPF.StateEstimationMethod = 'mean';
myPF.ResamplingMethod = 'systematic';
```

Estimate the states using the `correct` and `predict` commands, and store the estimated states.

```
xEst = zeros(size(xTrue));
for k=1:size(xTrue,1)
    xEst(k,:) = correct(myPF,yMeas(k));
    predict(myPF);
end
```

Plot the results, and compare the estimated and true states.

```
figure(1)
plot(xTrue(:,1),xTrue(:,2),'x',xEst(:,1),xEst(:,2),'ro')
legend('True','Estimated')
```



References

[1] T. Li, M. Bolic, P.M. Djuric, "Resampling Methods for Particle Filtering: Classification, implementation, and strategies," *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 70-86, May 2015.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

For more information, see “Generate Code for Online State Estimation in MATLAB”.

Supports MATLAB Function block: No

See Also

Functions
`initialize` | `predict` | `correct` | `clone` | `unscentedKalmanFilter` | `extendedKalmanFilter`

Topics
“Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter”
“Generate Code for Online State Estimation in MATLAB”
“What Is Online Estimation?”
“Validate Online State Estimation at the Command Line”
“Troubleshoot Online State Estimation”

External Websites
Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

Introduced in R2017b

pe

Prediction error for identified model

Syntax

```
err = pe(sys,data,K)
err = pe(sys,data,K,opt)
[err,ice,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,ice,sys_pred] = pe( ___ )` also returns the estimated initial conditions, `ice`, 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 1-1154.

`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 `plot`.

Output Arguments

`err`

Prediction error.

`err` is returned as an `iddata` object or matrix, depending on how you specify `data`. For example, if `data` is an `iddata` object, then so is `err`.

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.

`ice`

Estimated initial conditions.

`ice` is returned as a column vector of initial states for state-space systems and as an `initialCondition` object for transfer function and polynomial 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].

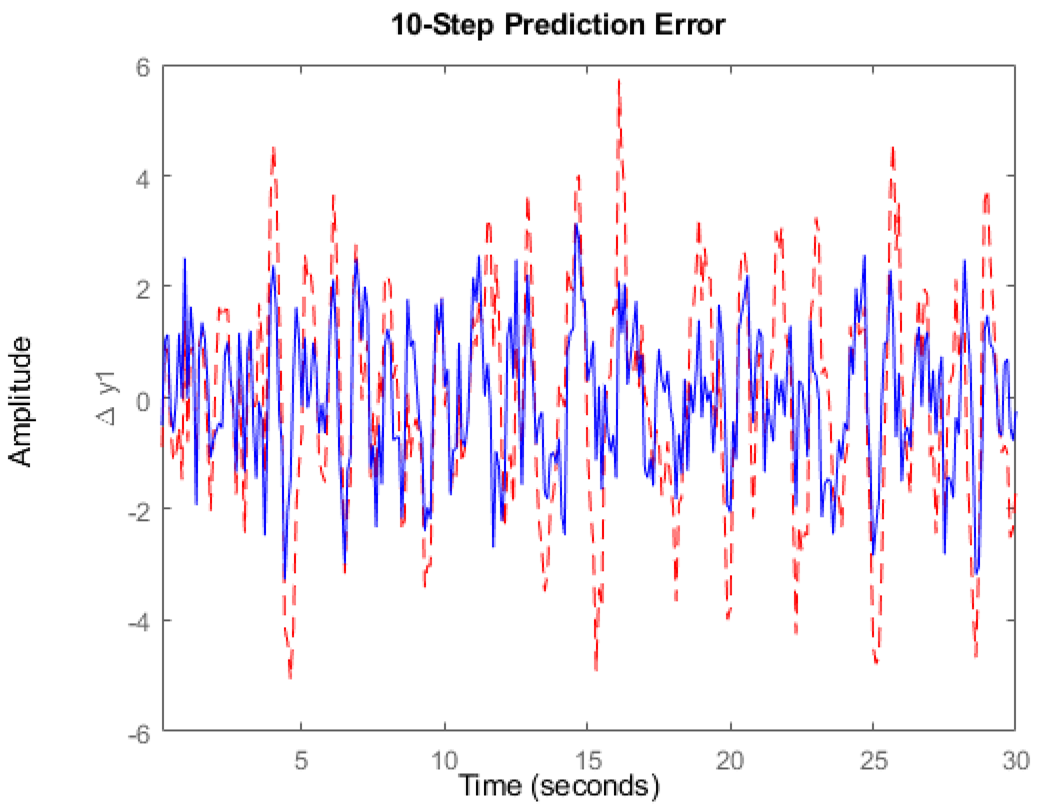
```
sys1 = arx(data,[2 2 1]);
```

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**.

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
`peOptions` | `predict` | `resid` | `sim` | `compare` | `ar` | `arx` | `n4sid` | `iddata`

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`InitialCondition`

Handling of initial conditions.

Specify `InitialCondition` as one of the following:

- `'z'` — Zero initial conditions.
- `'e'` — Estimate initial conditions such that the prediction error for observed output is minimized.

  For nonlinear grey-box models, only those initial states `i` that are designated as free in the model (`sys.InitialStates(i).Fixed = false`) are estimated. To estimate all the states of the model, first specify all the `Nx` states of the `idnlgrey` model `sys` as free.

  ```
  for i = 1:Nx
  sys.InitialStates(i).Fixed = false;
  end
  ```

  Similarly, to fix all the initial states to values specified in `sys.InitialStates`, first specify all the states as fixed in the `sys.InitialStates` property of the nonlinear grey-box model.
- `'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 linear models only.
- Vector or Matrix — Initial guess for state values, specified as a numerical column vector of length equal to the number of states. For multi-experiment data, specify a matrix with *Ne* columns, where

Ne is the number of experiments. Otherwise, use a column vector to specify the same initial conditions for all experiments. Use this option for state-space (`idss` and `idgrey`) and nonlinear models (`idnlarx`, `idnlhw`, and `idnlgrey`) only.

- `initialCondition` object — `initialCondition` object that represents a model of the free response of the system to initial conditions. For multiexperiment data, specify a 1-by-N_e array of objects, where N_e is the number of experiments.

  Use this option for linear models 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 by `pe`:

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 `[]`. The number of rows must be greater than or equal to the model order.
`Output`	Output history, specified as a matrix with *Ny* columns, where *Ny* is the number of output channels. The number of rows must be greater than or equal to the model order.

  For multi-experiment data, configure the initial conditions separately for each experiment by specifying `InitialCondition` as a structure array with *Ne* elements. To specify the same initial conditions for all experiments, use a single structure.

  The software uses `data2state` to map the historical data to states. If your model is not `idss`, `idgrey`, `idnlgrey`, or `idnlarx`, the software first converts the model to its state-space representation and then maps the data to states. If conversion of your model to `idss` is not possible, the estimated states are returned empty.
- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space (`idss` and `idgrey`) and nonlinear grey-box (`idnlgrey`) models only. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum or maximum bounds.

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

`pe` | `idpar`

Introduced in R2012a

pem

Prediction error minimization for refining linear and nonlinear models

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
```

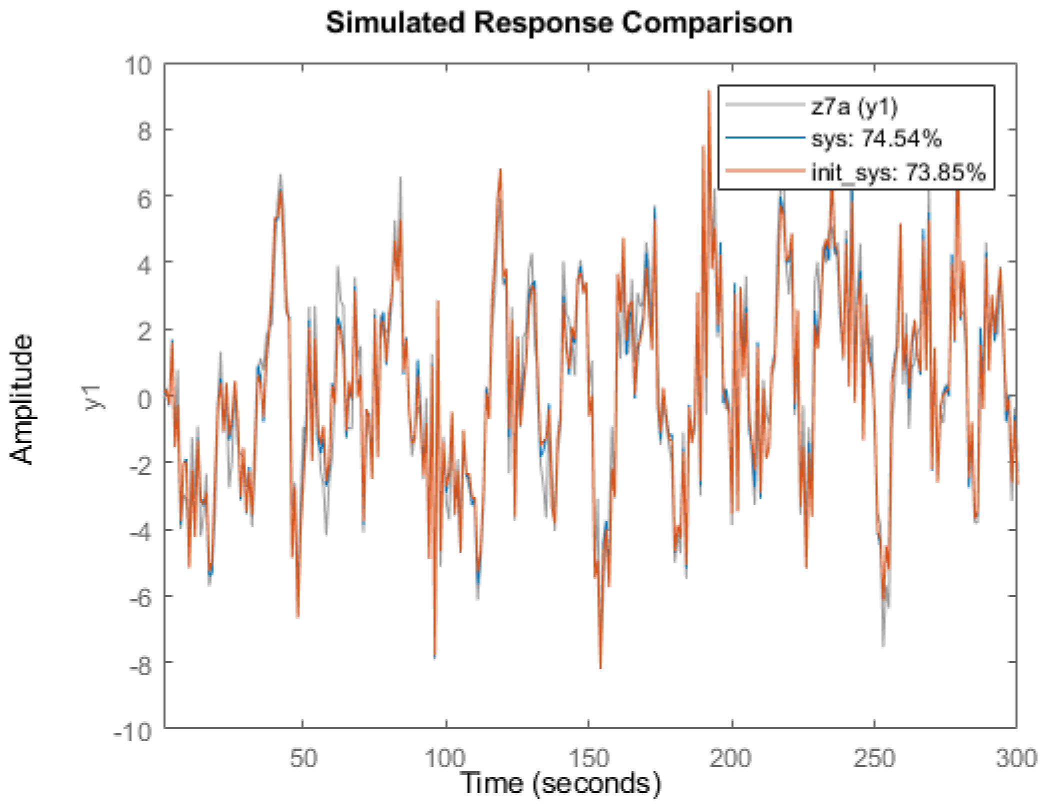
```
ans = 73.8490
```

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.

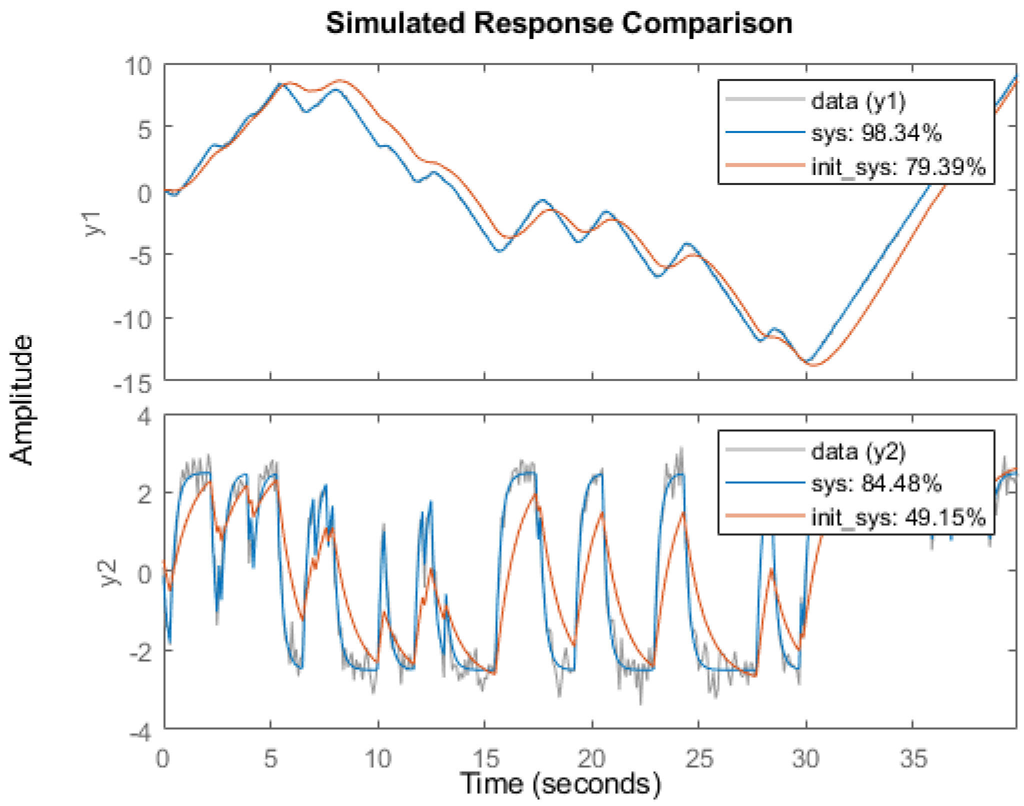
Estimate the model parameters and initial states.

```
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);
```

```
Process Model Identification

Estimation data: Time domain data z1
Data has 1 outputs, 1 inputs and 300 samples.
Model Type:
    {'P2DUZ'}



Algorithm: Levenberg-Marquardt search
 <br>
------------------------------------------------------------------------------------------
 <br>
                          Norm of      First-order    Improvement (%) <br> Iteration       Cost
     0        21.2201           -             414         3.8            -          -
     1        19.4048        1.15             323         3.8         8.55          7
     2        14.8743        2.48             814        4.41         23.3          0
     3        6.84305       0.873             451        4.43           54         11
     4        5.20355       0.977        1.49e+03        8.75           24          7
     5        1.83911       0.973             473          13         64.7          0
     6        1.67582       0.225            20.3        4.98         8.88          0
     7        1.67335       0.062            6.57      0.0829        0.147          0
     8        1.67334     0.00494          0.0555    0.000374     0.000648          0
------------------------------------------------------------------------------------------
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 8, Number of function evaluations: 42

Status: Estimated using PEM
Fit to estimation data: 70.63%, FPE: 1.73006
```

Examine the model fit.

```
sys.Report.Fit.FitPercent
```

```
ans = 70.6330
```

`sys` provides a 70.63% fit to the measured data.

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.

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:

$$e(t) = H^{-1}(q)[y(t) - G(q)u(t)]$$

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.

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.

See Also

`tfest` | `ssest` | `n4sid` | `procest` | `polyest` | `armax` | `oe` | `bj` | `greyest` | `nlhw` | `nlarx` | `nlgreyest`

Topics

"Refine Linear Parametric Models"

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

plot

Plot input and output channels of `iddata` object

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 function plots the outputs 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 is plotted accordingly.
- For frequency-domain data, the magnitude and phase of each input and output signal are plotted over the available frequency span.

To plot a subset of the data, use subreferencing:

- `plot(data(201:300))` plots the samples 201 to 300 in the data set `data`.
- `plot(data(201:300,'Altitude',{'Angle_of_attack','Speed'}))` plots the specified samples of the output named `Altitude` and the 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 is 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 specify options for only some data sets. For example, `plot(data1,data2,'k',data3)` specifies black as the plot color for `data2`.

`plot(axes_handle, ___ )` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`). Use this syntax with any of the input argument combinations in the previous syntaxes.

`plot( ___ ,plotoptions)` specifies the plot options.

`h = plot( ___ )` returns the handle to the plot. You can use this handle to customize the plot with `getoptions` and `setoptions`.

Examples

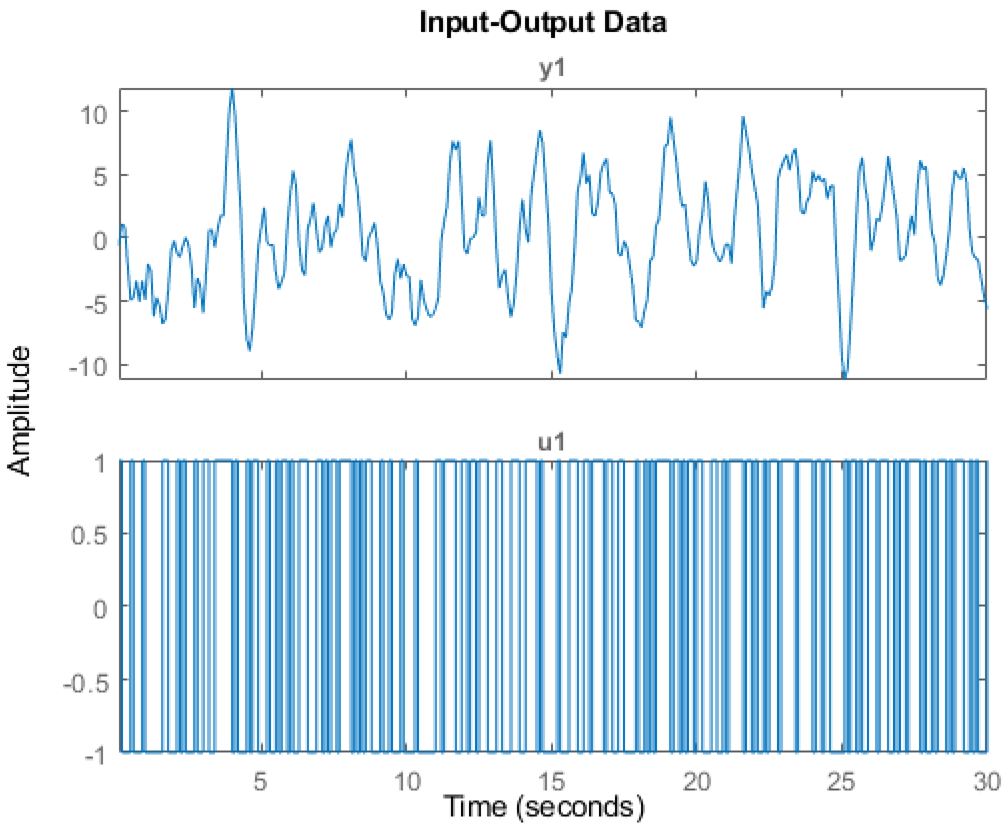
Plot Time-Domain Input-Output Data

Load the data.

```
load iddata1 z1;
```

Plot the data.

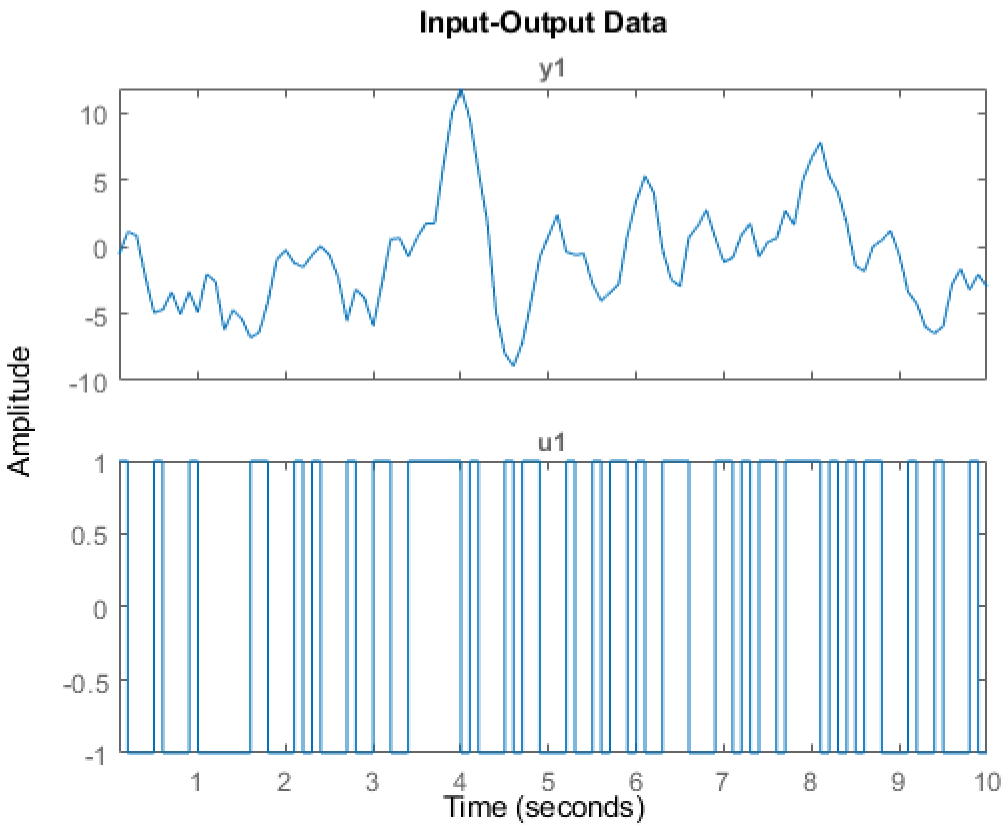
```
plot(z1)
```



The function plots the output on the top axes and the input on the bottom axes.

Plot the first 100 samples.

```
plot(z1(1:100))
```



Only the first 100 samples appear in the plot.

You can right-click the plot to explore characteristics such as peak and mean values.

Plot Frequency-Domain Input-Output Data

Load the data.

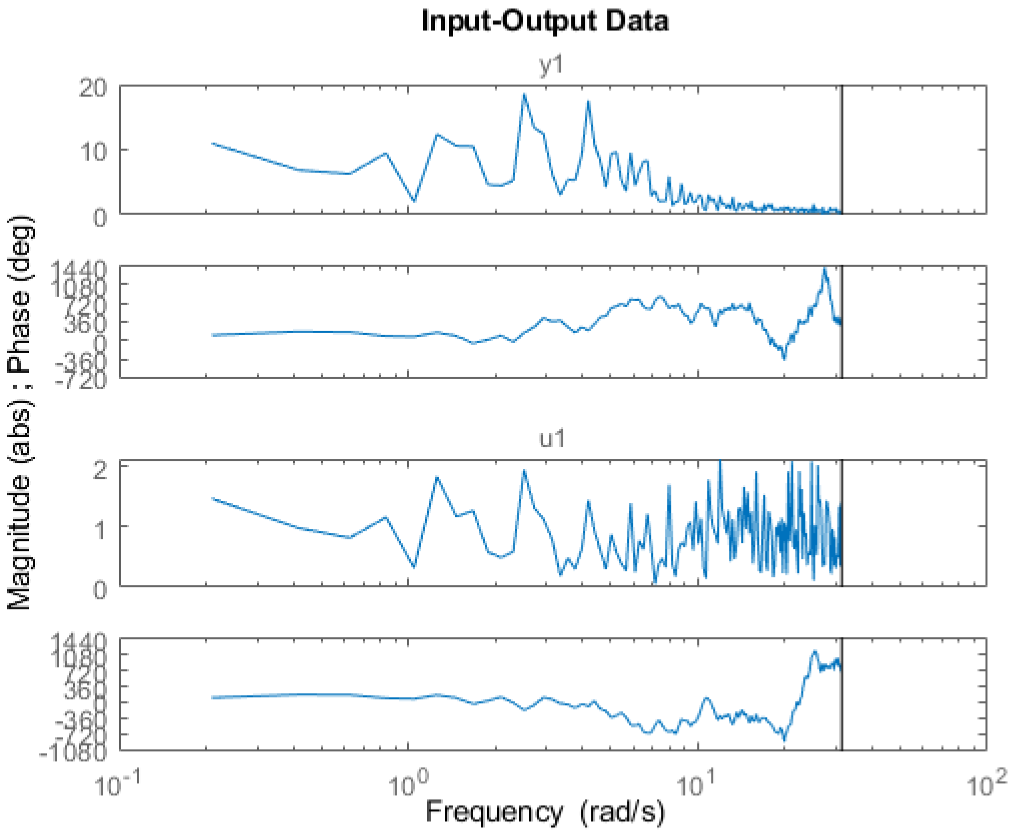
```
load iddata1 z1
```

Convert the data to the frequency domain.

```
zf = fft(z1);
```

Plot the data.

```
plot(zf);
```



Plot Input Data, Output Data, and Input-Output Data

Generate input data.

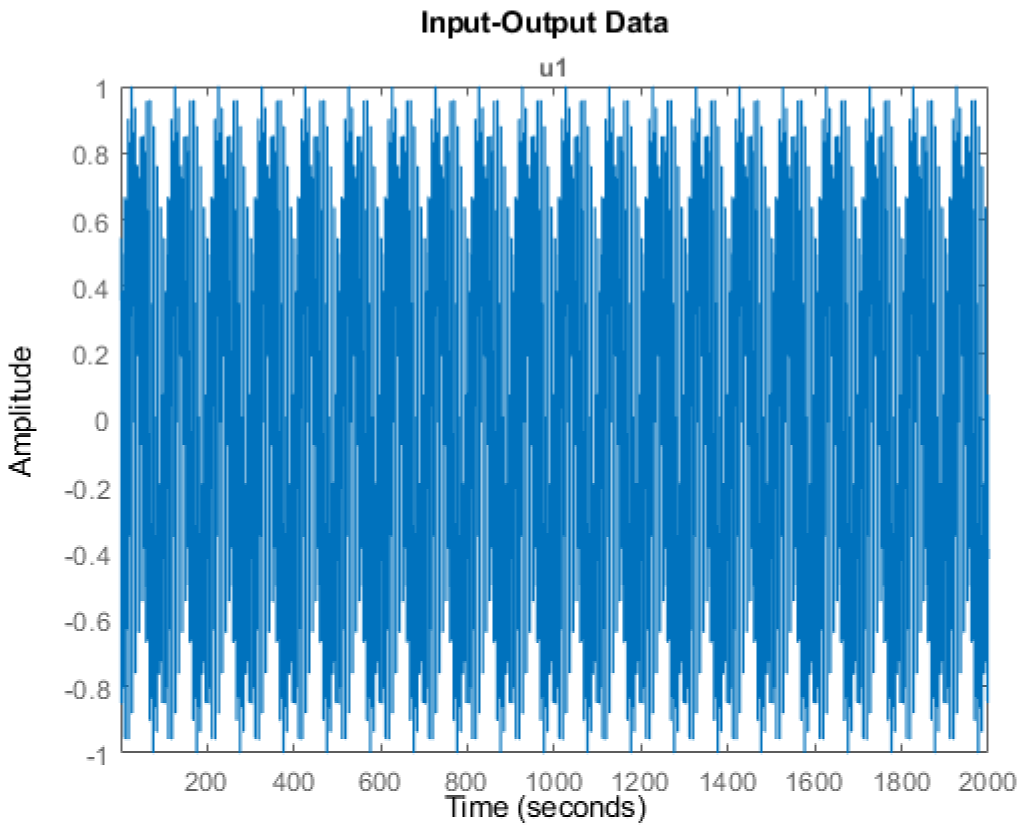
```
u = idinput([100 1 20],'sine',[],[],[5 10 1]);
u = iddata([],u,1,'per',100);
```

Generate output data.

```
sys = idtf(1,[1 2 1]);
y = sim(sys,u);
```

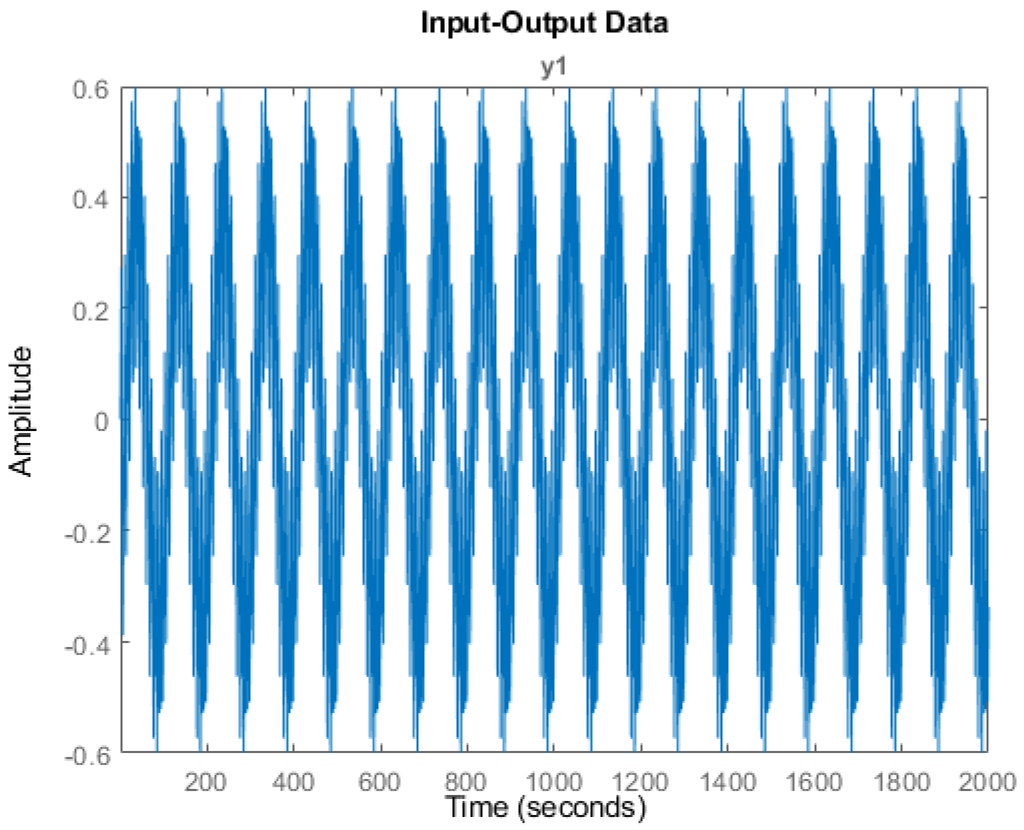
Plot only the input.

```
plot(u)
```



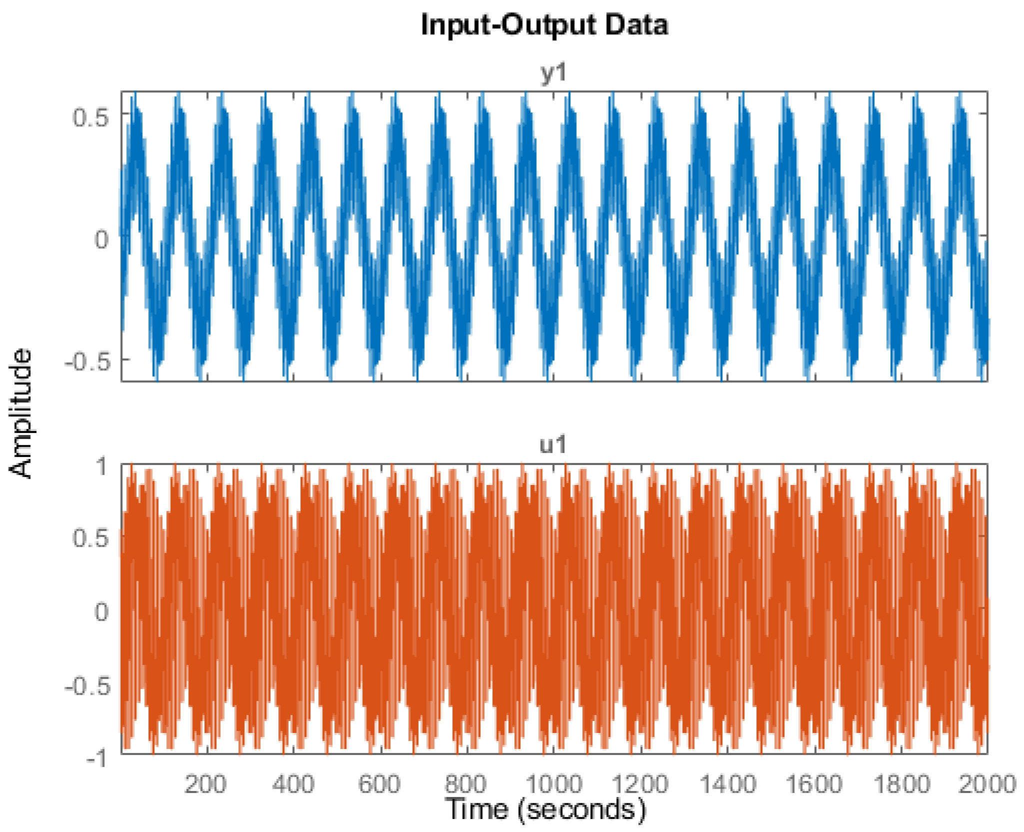
Plot only the output.

```
plot(y)
```



Plot the input and output together.

```
plot(y,u)
```



Alternatively, you can use `plot(iddata(y,u))`.

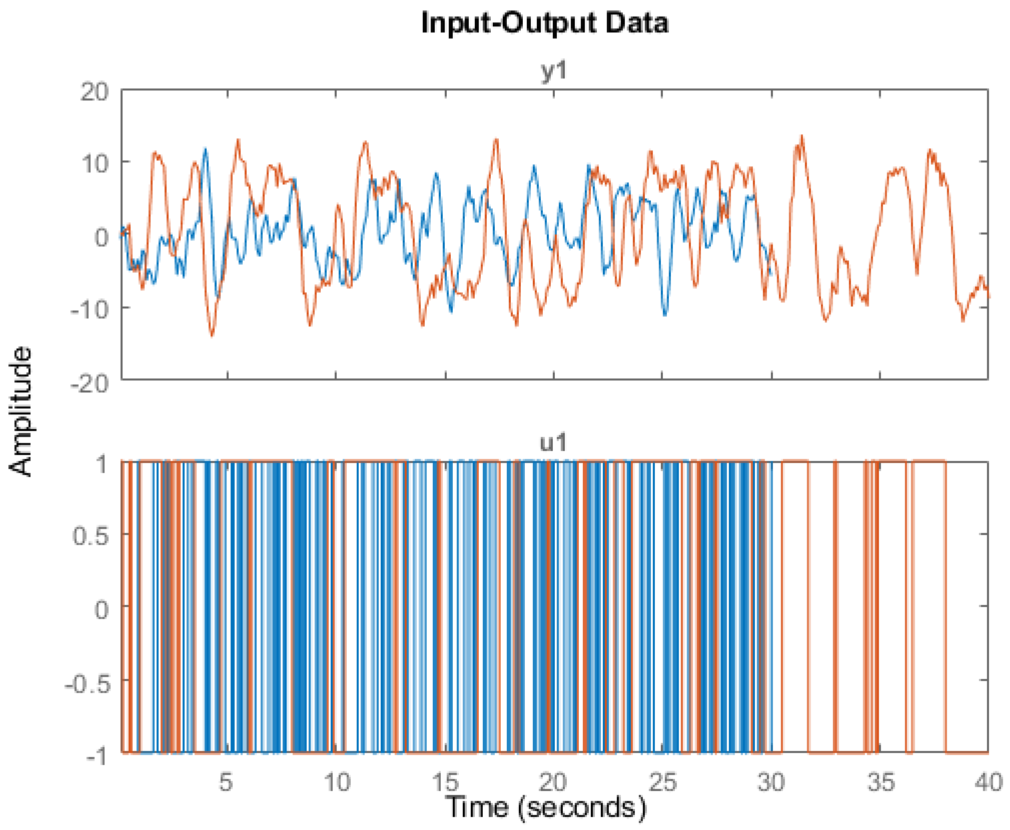
Plot Multiple Data Sets

Load two data sets.

```
load iddata1 z1
load iddata2 z2
```

Plot both datasets.

```
plot(z1,z2)
```



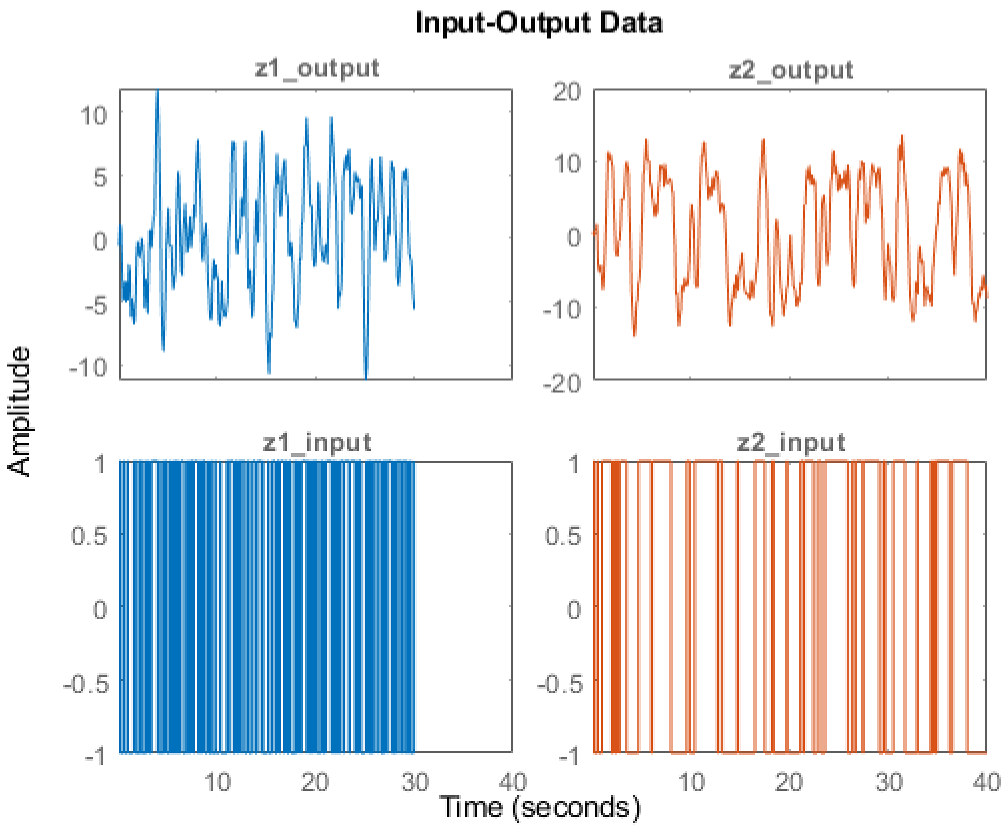
Because the data sets use the same input and output names, the function plots both data sets together.

Specify unique input and output names.

```
z1.InputName = "z1_input";
z2.InputName = "z2_input";
z1.OutputName = "z1_output";
z2.OutputName = "z2_output";
```

Plot both datasets.

```
plot(z1,z2)
```



The function plots the data sets separately.

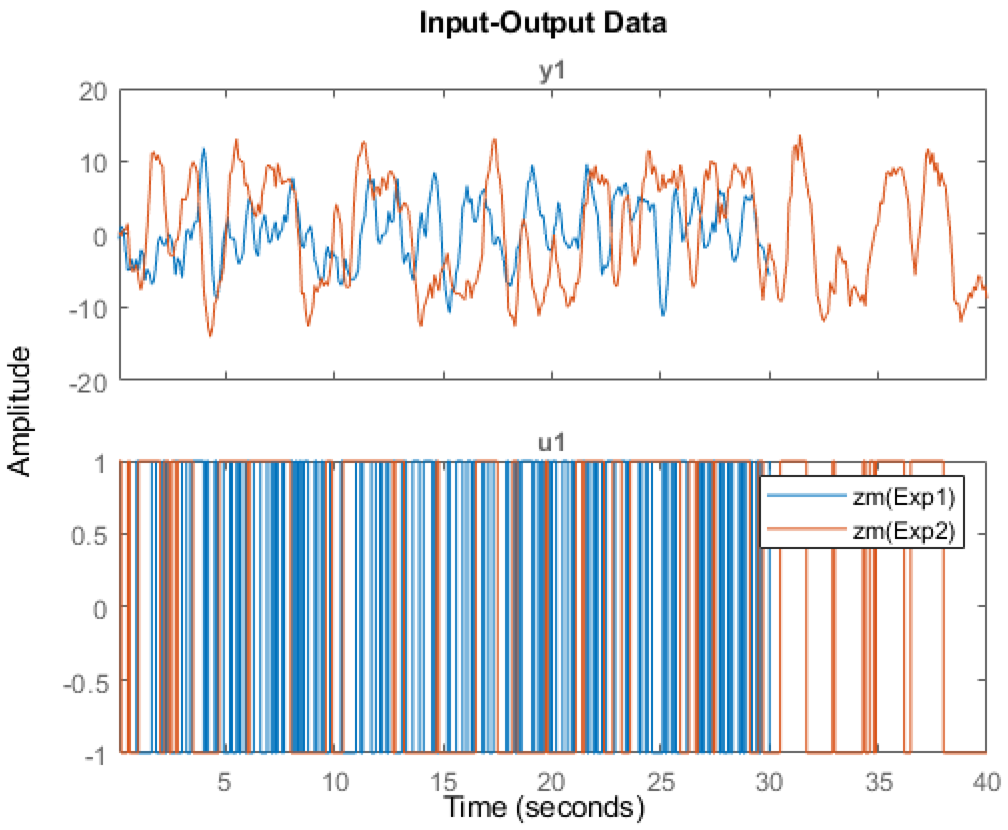
Plot Multiexperiment Data

Create a multiexperiment data set.

```
load iddata1 z1
load iddata2 z2
zm = merge(z1,z2);
```

Plot the data.

```
plot(zm)
legend('show')
```



For multiexperiment data, each experiment is treated as a separate data set. You can right-click the plots to view their characteristics.

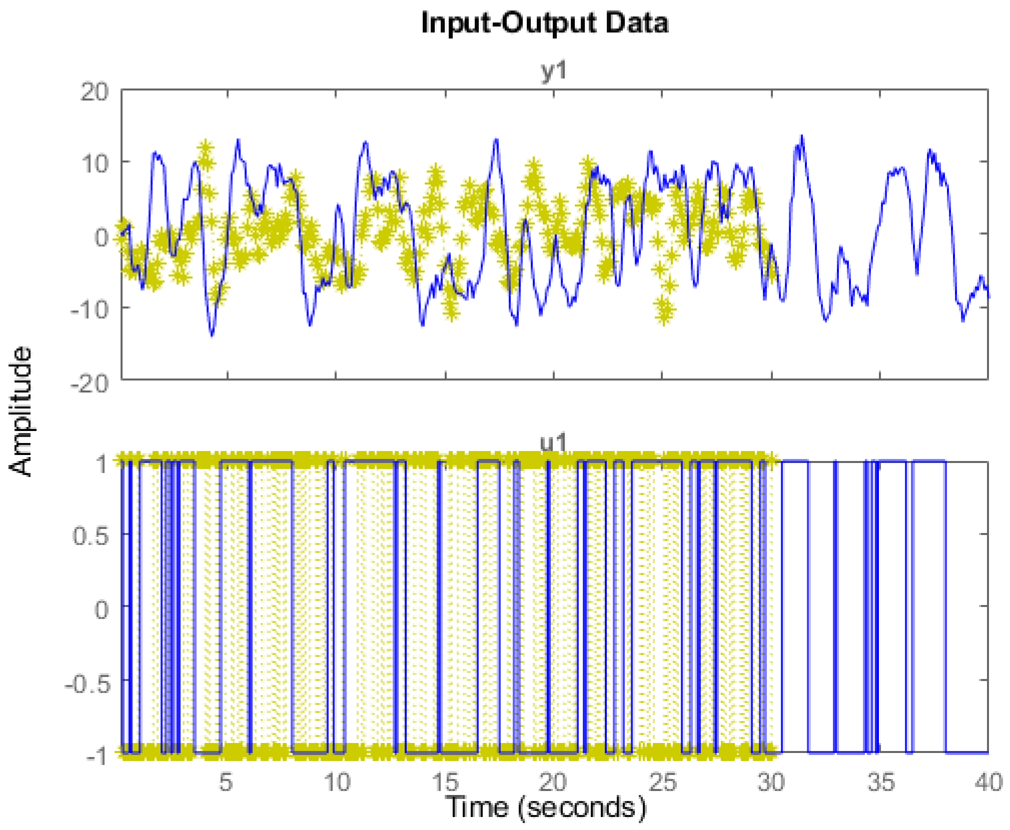
Specify Line Style, Marker Symbol, and Color

Load two data sets.

```
load iddata1 z1;
load iddata2 z2;
```

Specify the line style for both data sets.

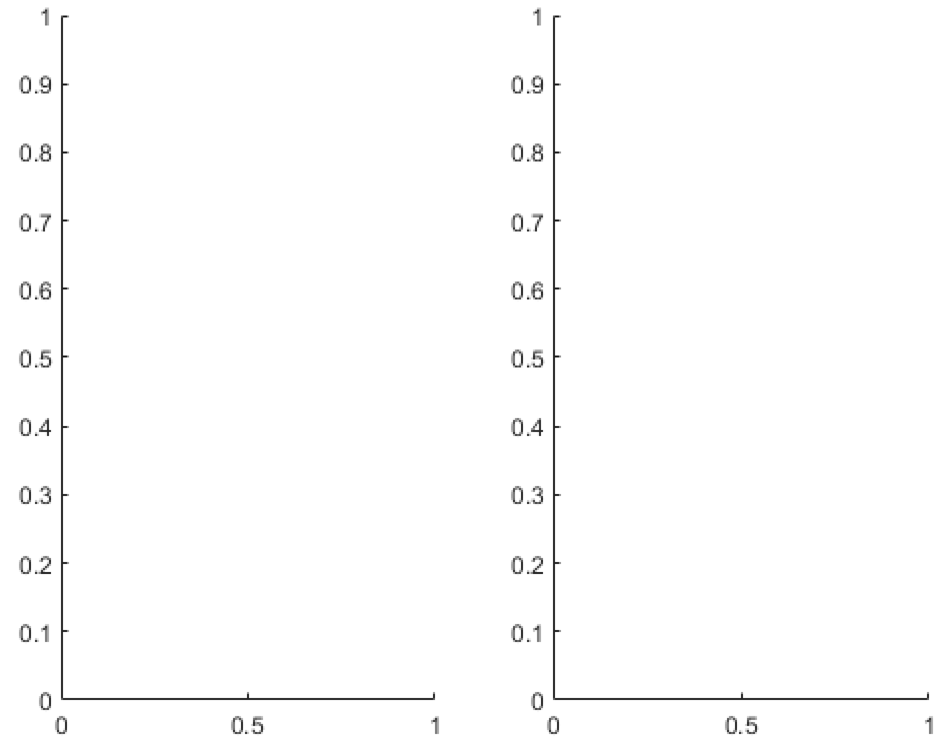
```
plot(z1,'y:*',z2,'b')
```



Specify Axes Handle

Create a figure with two subplots and return the handles for each subplot axes in `s`.

```
figure % new figure
s(1) = subplot(1,2,1); % left subplot
s(2) = subplot(1,2,2); % right subplot
```

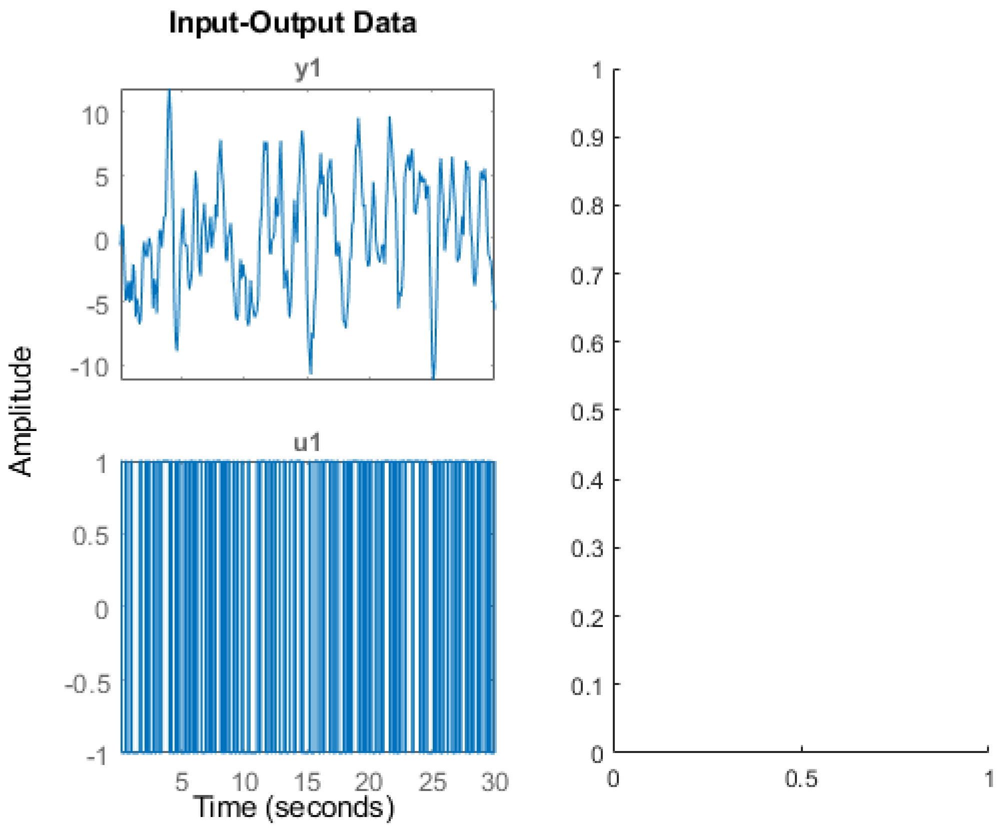


Load the data sets.

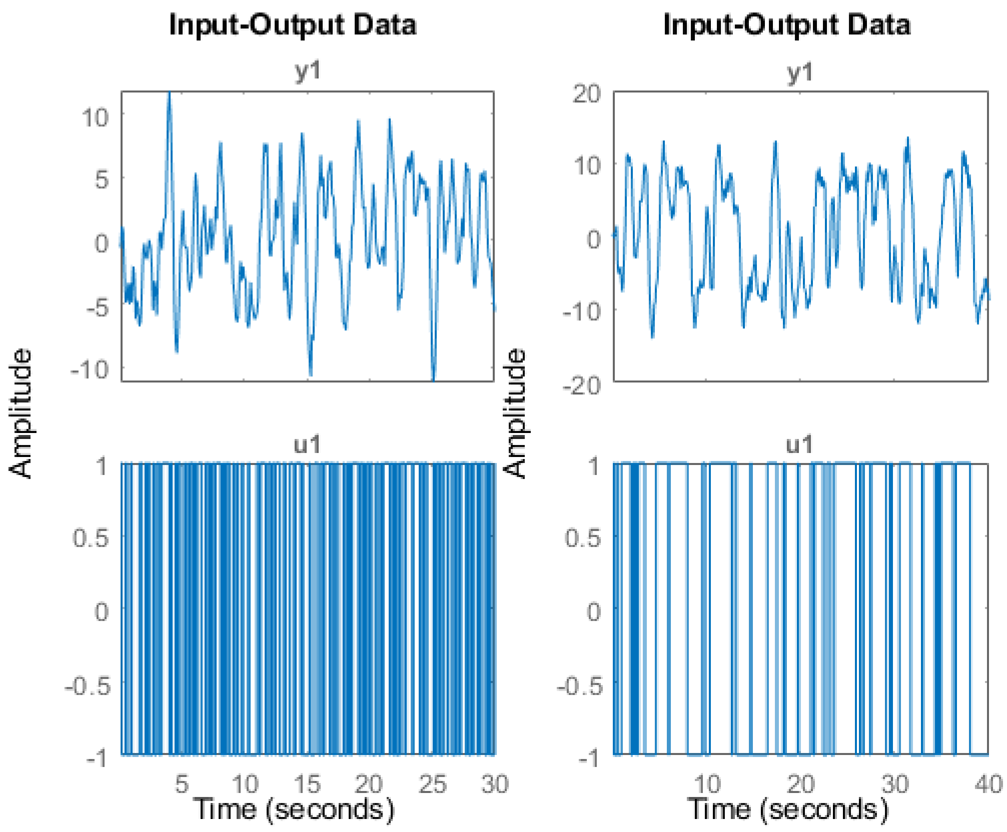
```
load iddata1;
load iddata2;
```

Create a data plot in each axes using the handles.

```
plot(s(1),z1)
```



```
plot(s(2),z2)
```

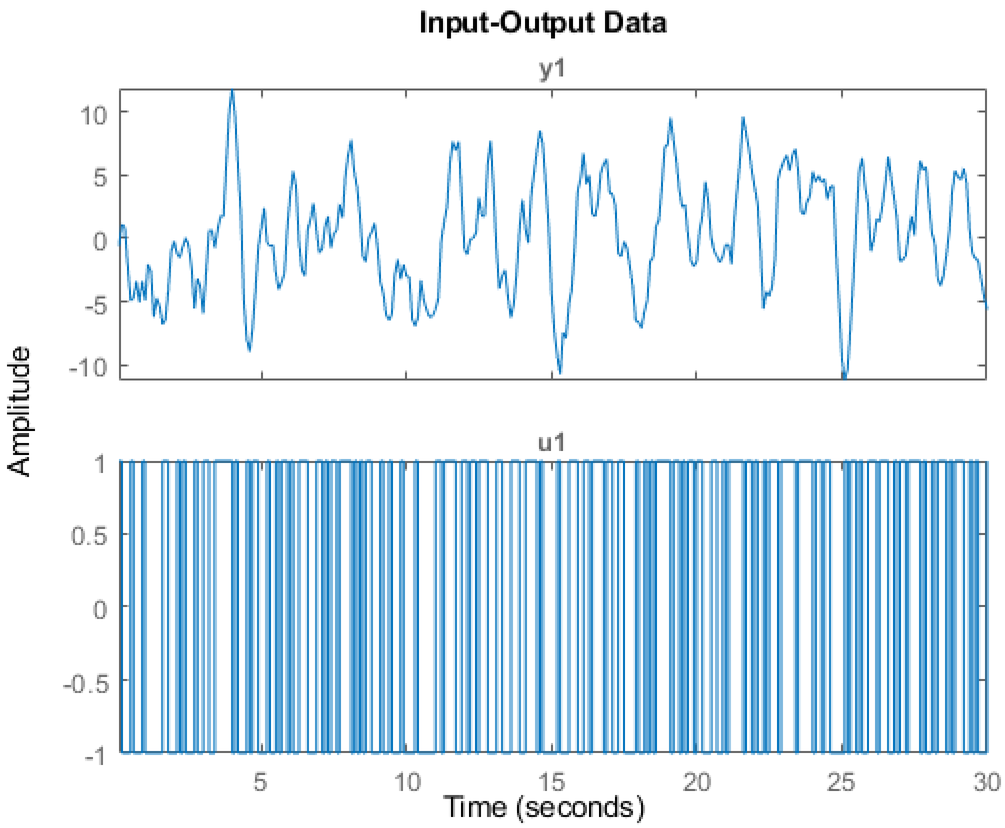


Get and Use Axes Handle

Get the handle to your current plot and modify an axis property.

Load and plot the data.

```
load iddata1 z1
plot(z1)
```



Get the axes handle for the plot.

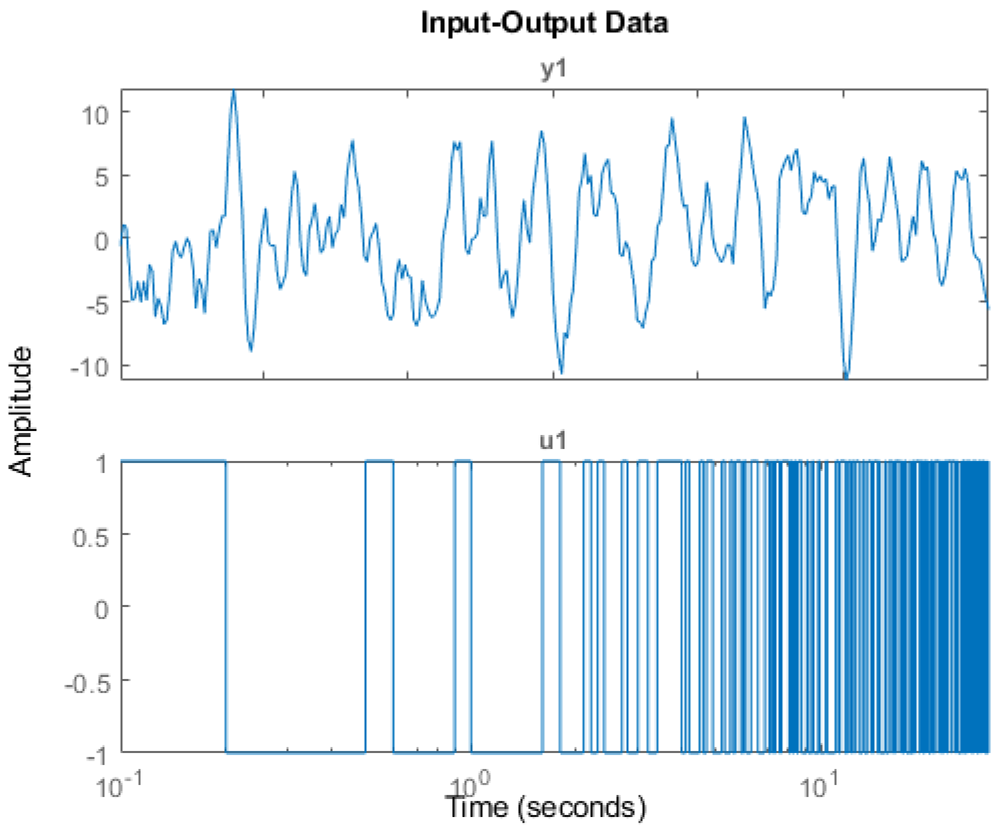
```
ah = gca
```

```
ah = 
  Axes (u1) with properties:

             XLim: [0.1000 30]
             YLim: [-1 1]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.3480]
            Units: 'normalized'

  Show all properties
```

The display shows the properties of the axes handle.

The scale of the x-axis `xScale` is `'linear'`. Change `xScale` to `'log'`.

```
ah.XScale = 'log';
```



The x-axis now displays a log scale.

Specify Plot Options

Configure a time plot.

```
opt = iddataPlotOptions('time');
```

Specify minutes as the time unit of the plot.

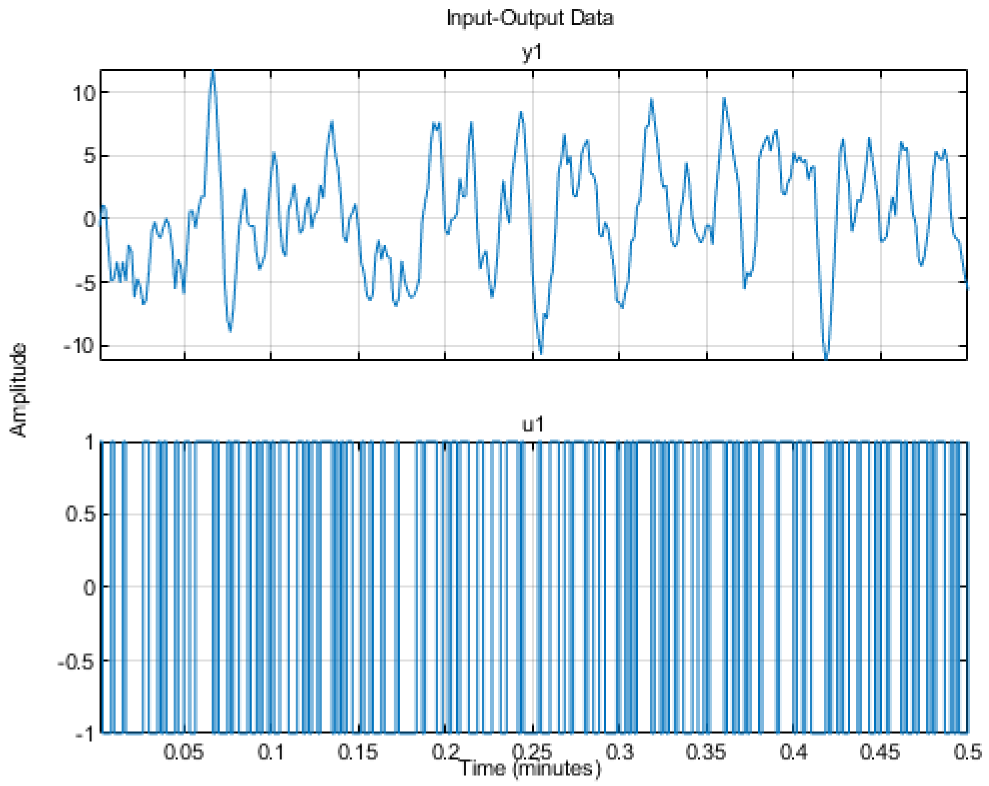
```
opt.TimeUnits = 'minutes';
```

Turn the grid on.

```
opt.Grid = 'on';
```

Create the plot with the options specified by `opt`.

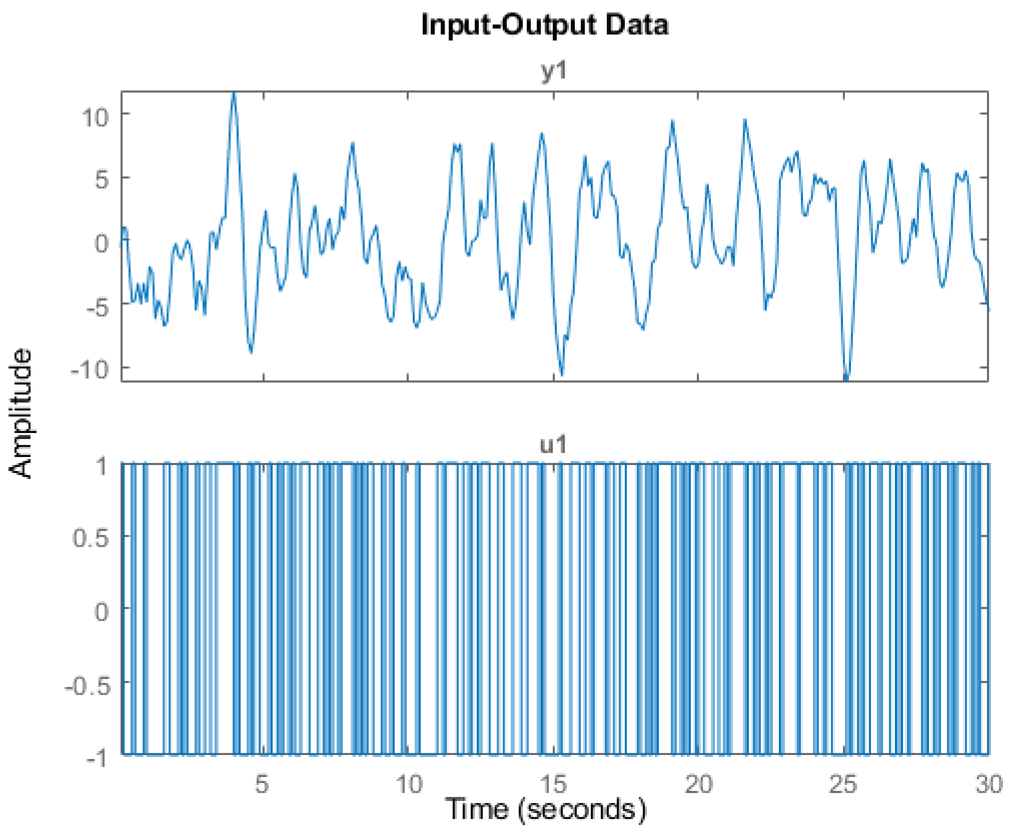
```
load iddata1 z1
plot(z1, opt);
```



Change Plot Properties Using Handle

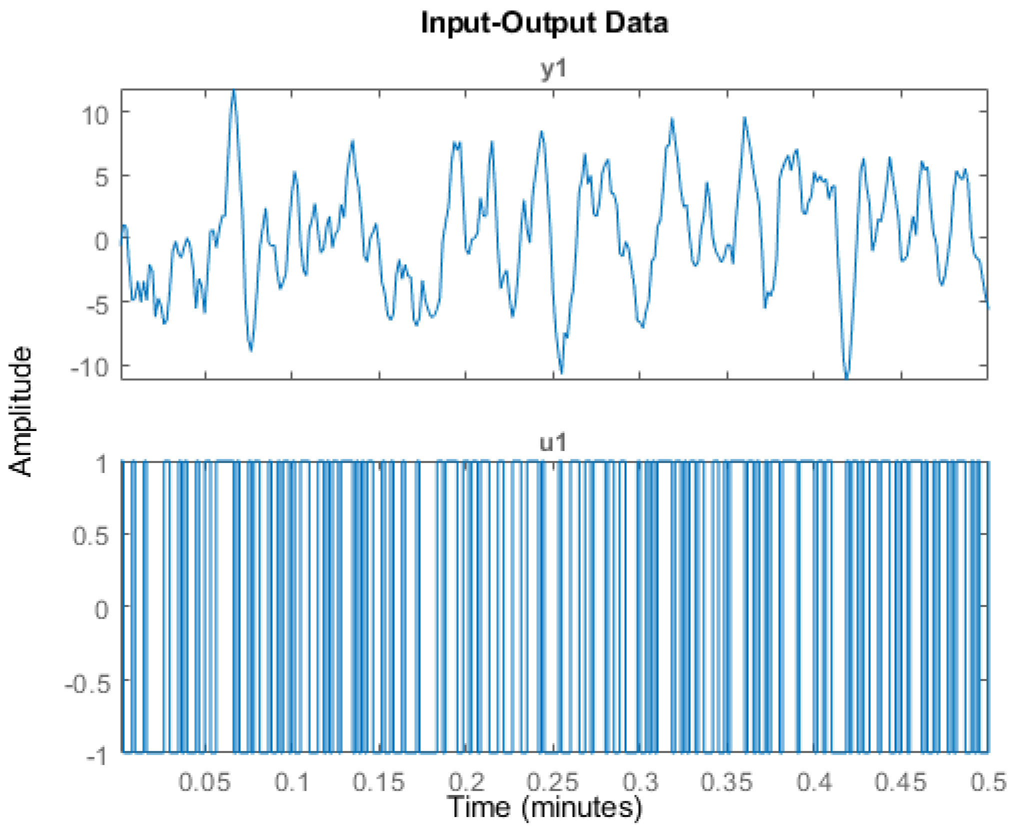
Create a data plot and return the handle.

```
load iddata1;
h = plot(z1);
```



Set the time unit of the plot.

```
setoptions(h,'TimeUnits','minutes');
```



Change Orientation of Input-Output Data Axes

Generate data with two inputs and one output.

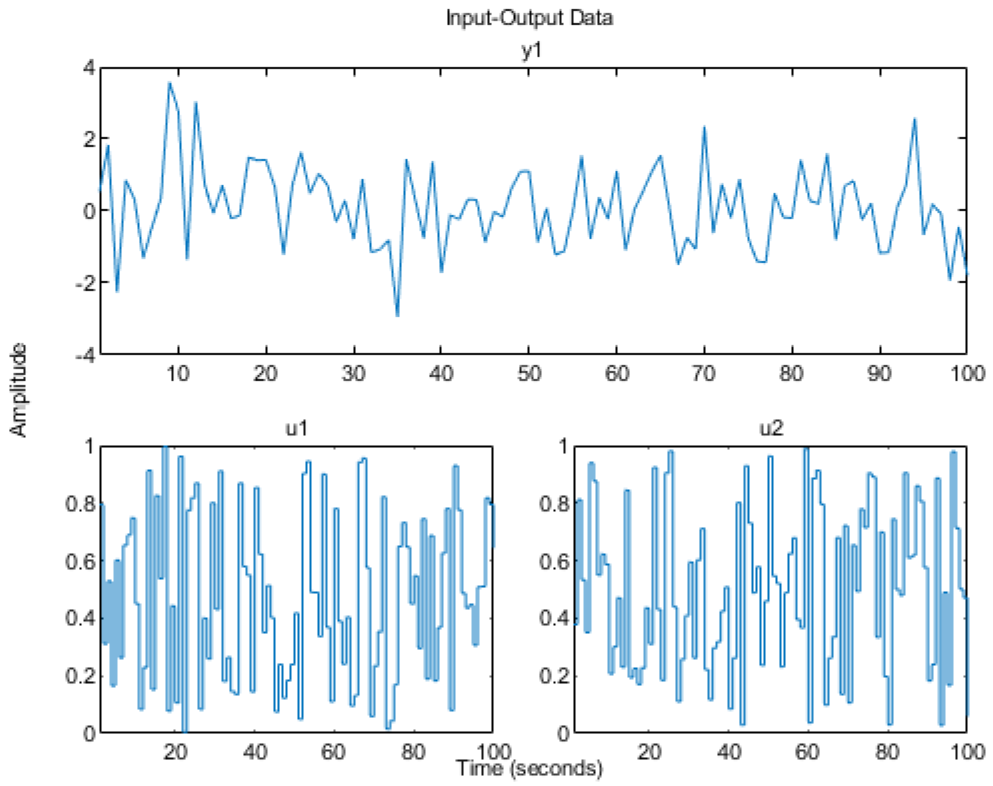
```
z = iddata(randn(100,1),rand(100,2));
```

Configure a time plot.

```
opt = iddataPlotOptions('time');
```

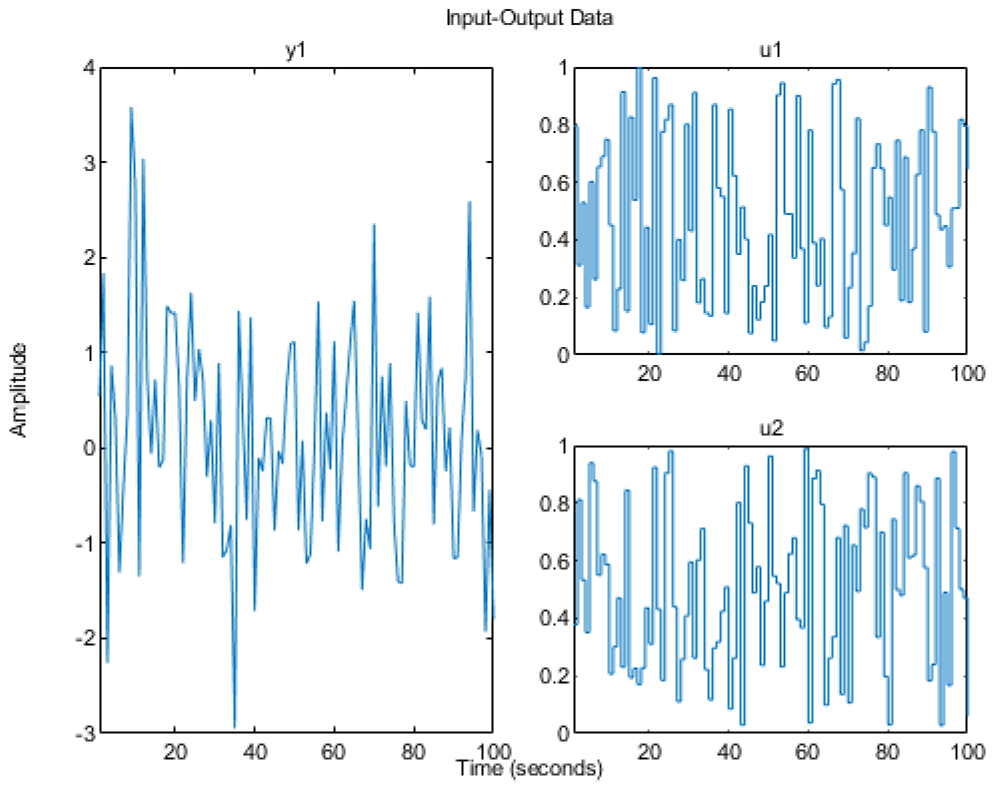
Plot the data.

```
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.

Input Arguments

data — Input-output data

`iddata` object

Input-output data, specified as an `iddata` object. The data can be in the time domain or the frequency domain. It can be a single-channel or multichannel data, and single-experiment or multiexperiment 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'` and `'b+:'`. For more information, see the `plot` reference page in the MATLAB documentation. For an example of using `LineSpec`, see “Specify Line Style, Marker Symbol, and Color” on page 1-1177.

`axes_handle` — Axes handle
handle

Axes handle, specified as a handle, and 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`. For an example of using `axes_handle` to apply a specific set of axes to the current plot, see “Specify Axes Handle” on page 1-1178. For an example of using `gca` to get your plot axes and then modifying the axes properties, see “Get and Use Axes Handle” on page 1-1181.

`plotoptions` — Plot options
structure

Plot options, specified as an option set created using `iddataPlotOptions`. For an example of using `plotoptions`, see “Specify Plot Options” on page 1-1183.

Output Arguments

`h` — Plot handle
scalar | vector

Plot handle, returned as a scalar or vector. Handles are unique identifiers that you can use to query and modify properties of a specific plot. For an example, see “Change Plot Properties Using Handle” on page 1-1184.

Tips

Right-clicking the plot opens the context menu, where you can access the following options and plot controls.

Option	Description and Suboptions
Datasets	View the datasets used in the plot.
Characteristics	**Peak Value** — View the peak value of the data. This value is useful for transient data. **Mean Value** — View the mean value of the data. This value is useful for steady-state data.

Option	Description and Suboptions
Orientation	For data with one input and one output channel: • **Single row** — Plot all inputs and outputs in one row. • **Single column**— Plot all inputs and outputs in one column. For data with more than one input or output channel: • **Output row and input row** — Plot all outputs in one row and all inputs in a second row. • **Output column and input column** — Plot all outputs in one column and all inputs in a second column.
I/O Grouping	Group input and output channels on the plot. Use this option with datasets with more than one input or output channel.
I/O Selector	Select a subset of the input and output channels to plot. By default, all input and output channels are plotted. Use this option with data sets with more than one input or output channel.
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 on page 1-1195. 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.

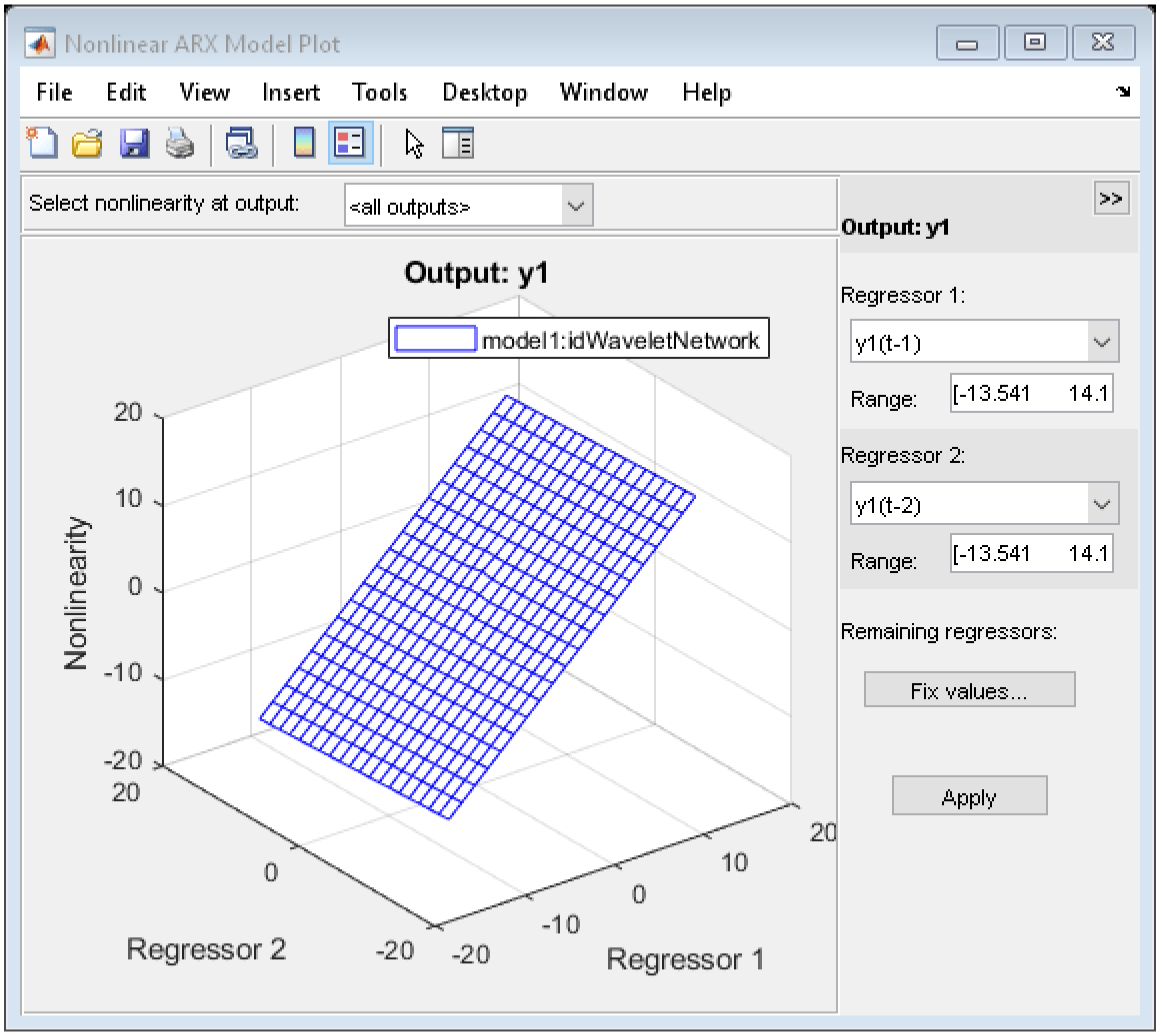
`plot( ___ ,'NumberofSamples',N)` specifies the number of samples to use to grid the regressor space on each axis. This syntax can include any of the input argument combinations in the previous syntaxes.

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],'idWaveletNetwork','nlr',[1:3]);
plot(model1)
```

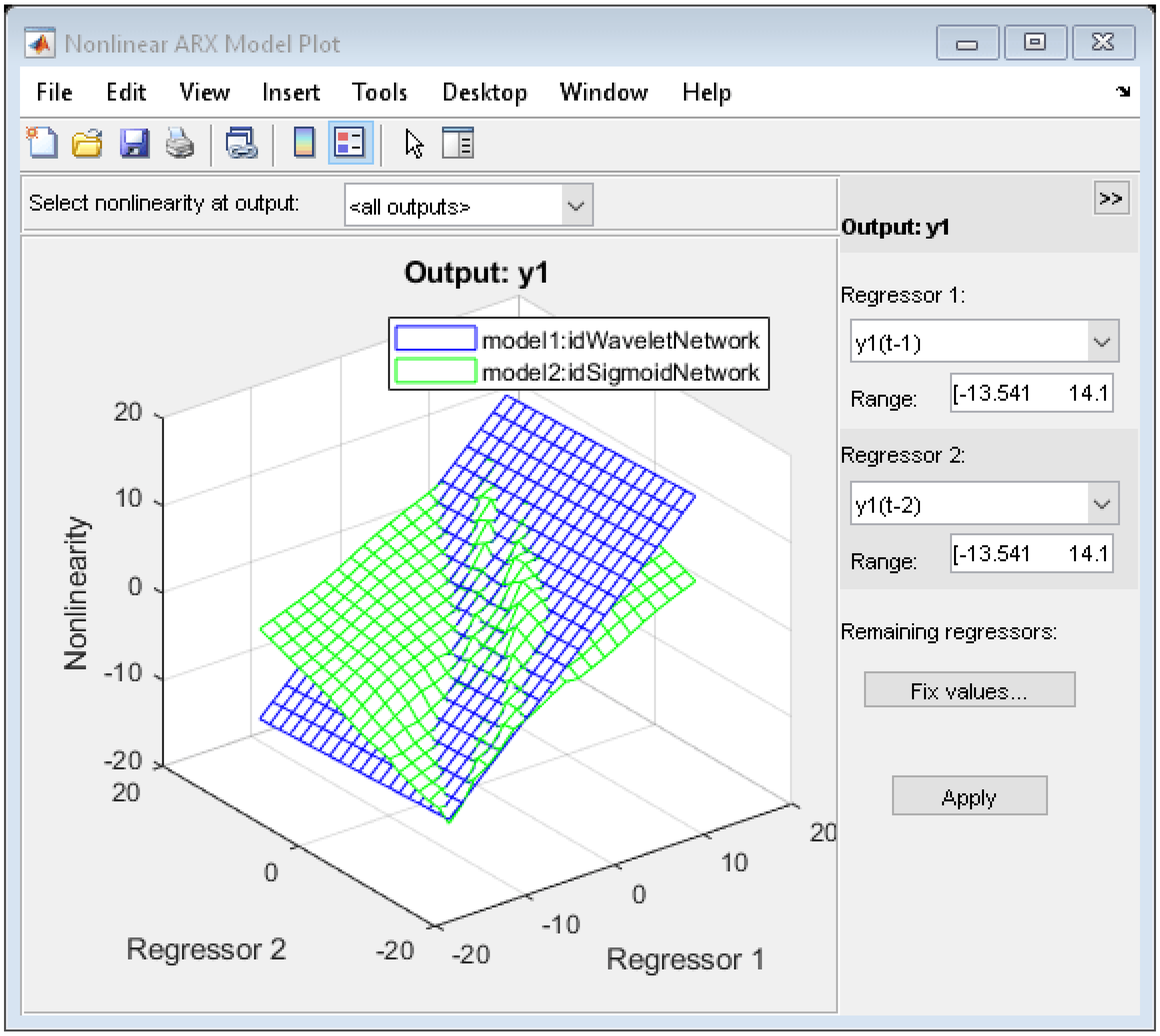


In the plot window, you can choose:

- 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.

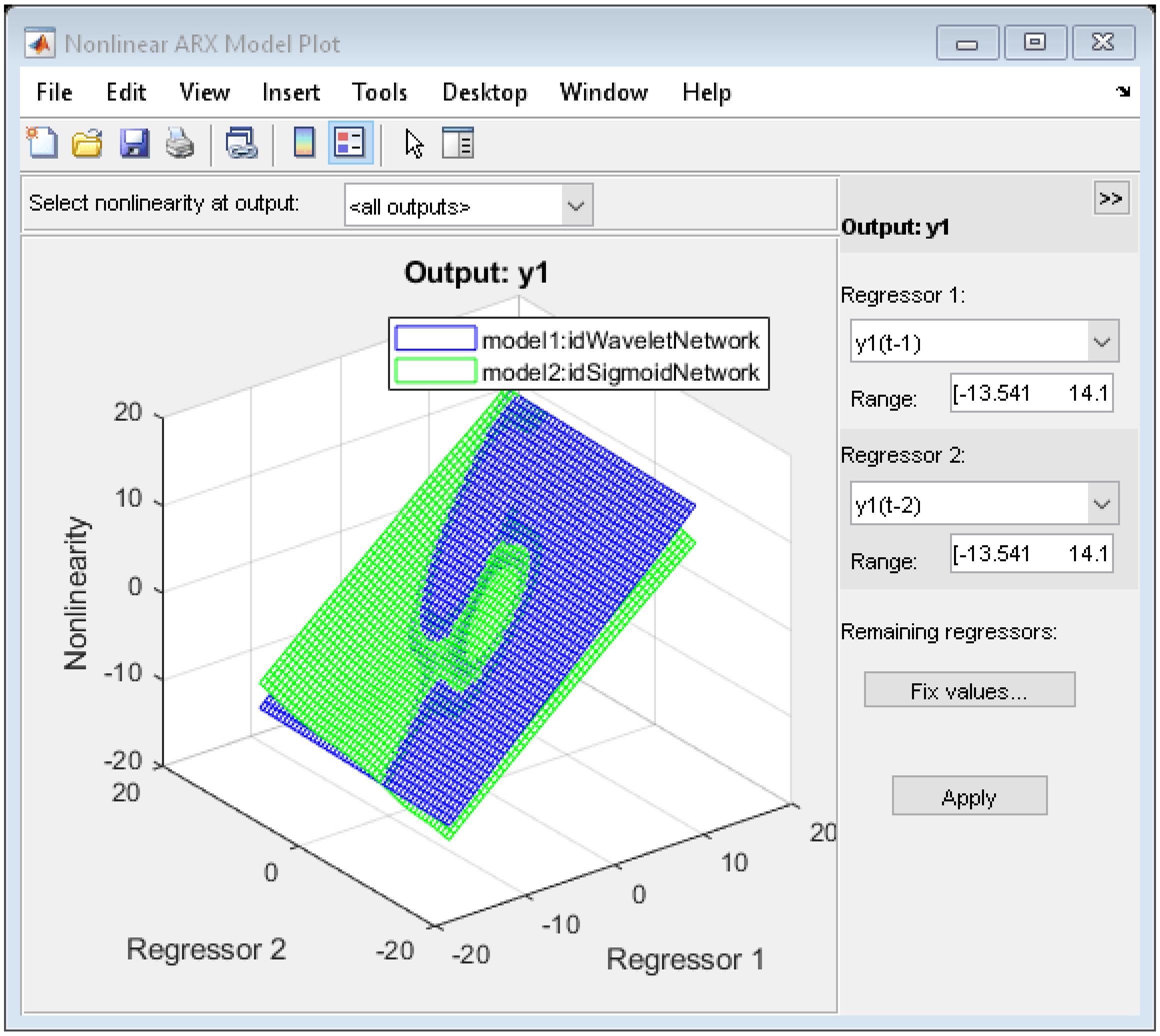
Specify Line Style for Multiple Models

```
load iddata1
model1 = nlarx(z1,[4 2 1],'idwave','nlr',[1:3]);
model2 = nlarx(z1,[4 2 1],'idSigmoidNetwork','nlr',[1:3]);
plot(model1,'b', model2, 'g')
```



Specify Number of Samples

```
load iddata1
model1 = nlarx(z1,[4 2 1],idWaveletNetwork);
model2 = nlarx(z1,[4 2 1],idSigmoidNetwork);
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

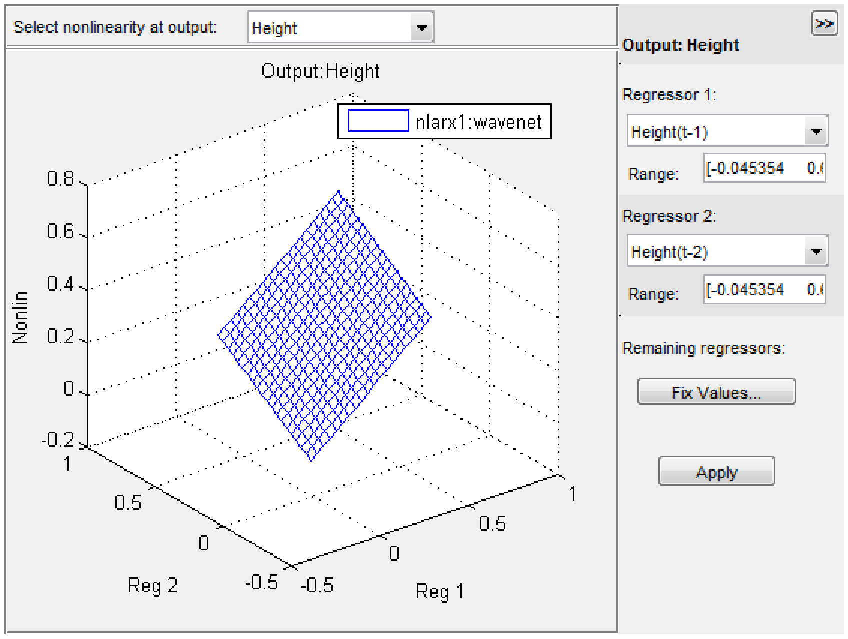
Number of points used on the regressor axis to display the regressor samples, specified as a positive integer.

Data Types: `double`

More About

What is a Nonlinear ARX Plot?

A nonlinear ARX plot displays the evaluated model nonlinearity for a chosen model output as a function of one or two model regressors. For a model `M`, the model nonlinearity (`M.Nonlinearity`) is a nonlinearity estimator function, such as `idWaveletNetwork`, `idSigmoidNetwork`, or `idTreePartition`, that uses model regressors as its inputs.



To understand what is plotted, suppose that `{r1,r2,…,rN}` are the `N` regressors used by a nonlinear ARX model `M` with nonlinearity `nl` corresponding to a model output. You can use `getreg(M)` to view these regressors. The expression `Nonlin = evaluate(nl,[v1,v2,...,vN])` returns the model output for given values of these regressors, that is, `r1 = v1`, `r2 = v2`, ..., `rN = vN`. For plotting the nonlinearities, you select one or two of the `N` regressors, for example, `rsub = {r1,r4}`. The software varies the values of these regressors in a specified range, while fixing the value of the remaining regressors, and generates the plot of `Nonlin` vs. `rsub`. By default, the software sets the values of the remaining fixed regressors to their estimated means, but you can change these values. The regressor means are stored in the `Nonlinearity.Parameters.RegressorMean` property of the model.

Examining a nonlinear ARX plot can help you gain insight into which regressors have the strongest effect on the model output. Understanding the relative importance of the regressors on the output can help you decide which regressors to include in the nonlinear function for that output. If the shape of the plot looks like a plane for all the chosen regressor values, then the model is probably linear in those regressors. In this case, you can remove the corresponding regressors from nonlinear block, and repeat the estimation.

Furthermore, you can create several nonlinear models for the same data using different nonlinearity estimators, such a `idWaveletNetwork` network and `idTreePartition`, and then compare the nonlinear surfaces of these models. Agreement between plots for various models increases the confidence that these nonlinear models capture the true dynamics of the system.

To learn more about configuring the plot, see "Configuring a Nonlinear ARX Plot".

See Also

`getreg` | `idnlarx` | `nlarx` | `evaluate`

Topics
"Structure of Nonlinear ARX Models"
"Validate Nonlinear ARX Models"

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 on page 1-1203. 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.

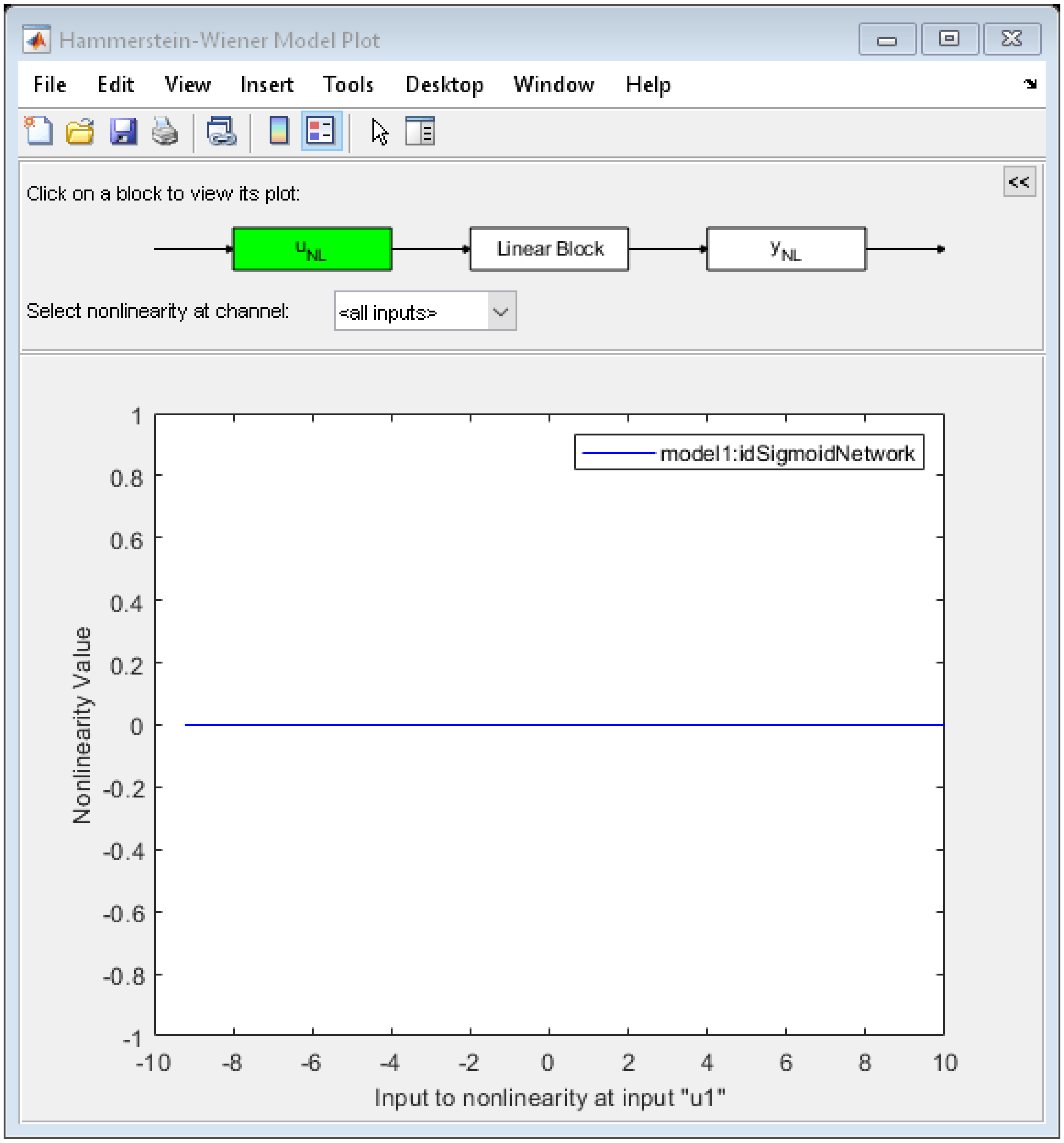
`plot( ___ ,Name,Value)` specifies plot properties using additional options specified by one or more `Name,Value` pair arguments. This syntax can include any of the input argument combinations in the previous syntaxes.

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],'idSigmoidNetwork','idDeadZone');
plot(model1)
```

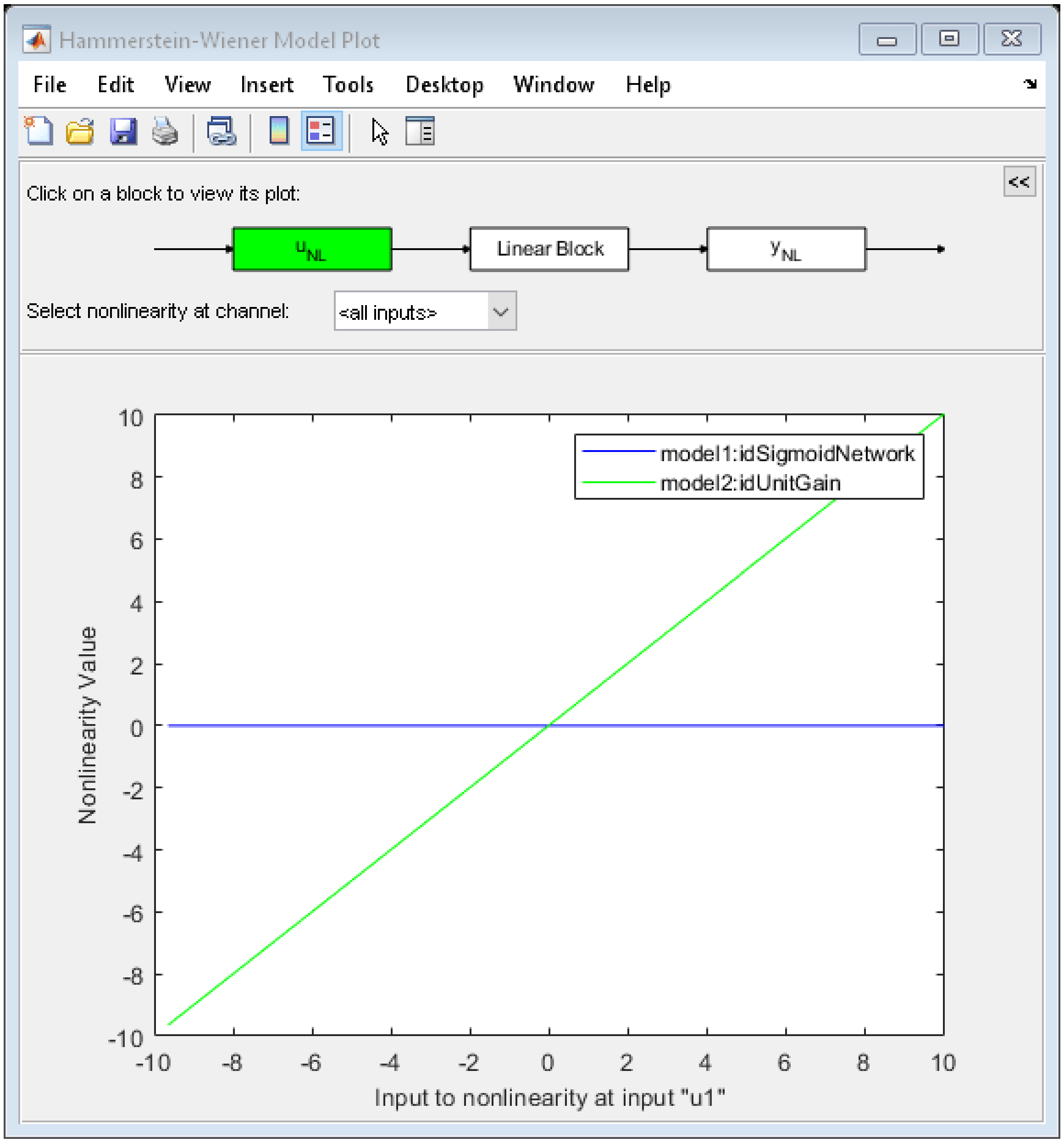

Hammerstein-Wiener Model Plot
File Edit View Insert Tools Desktop Window Help
Click on a block to view its plot:
u_{NL}
Linear Block
y_{NL}
Select nonlinearity at channel:
<all inputs>
<<
model1:idSigmoidNetwork
Nonlinearity Value
Input to nonlinearity at input "u1"
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-8
-6
-4
-2
0
2
4
6
8
10

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

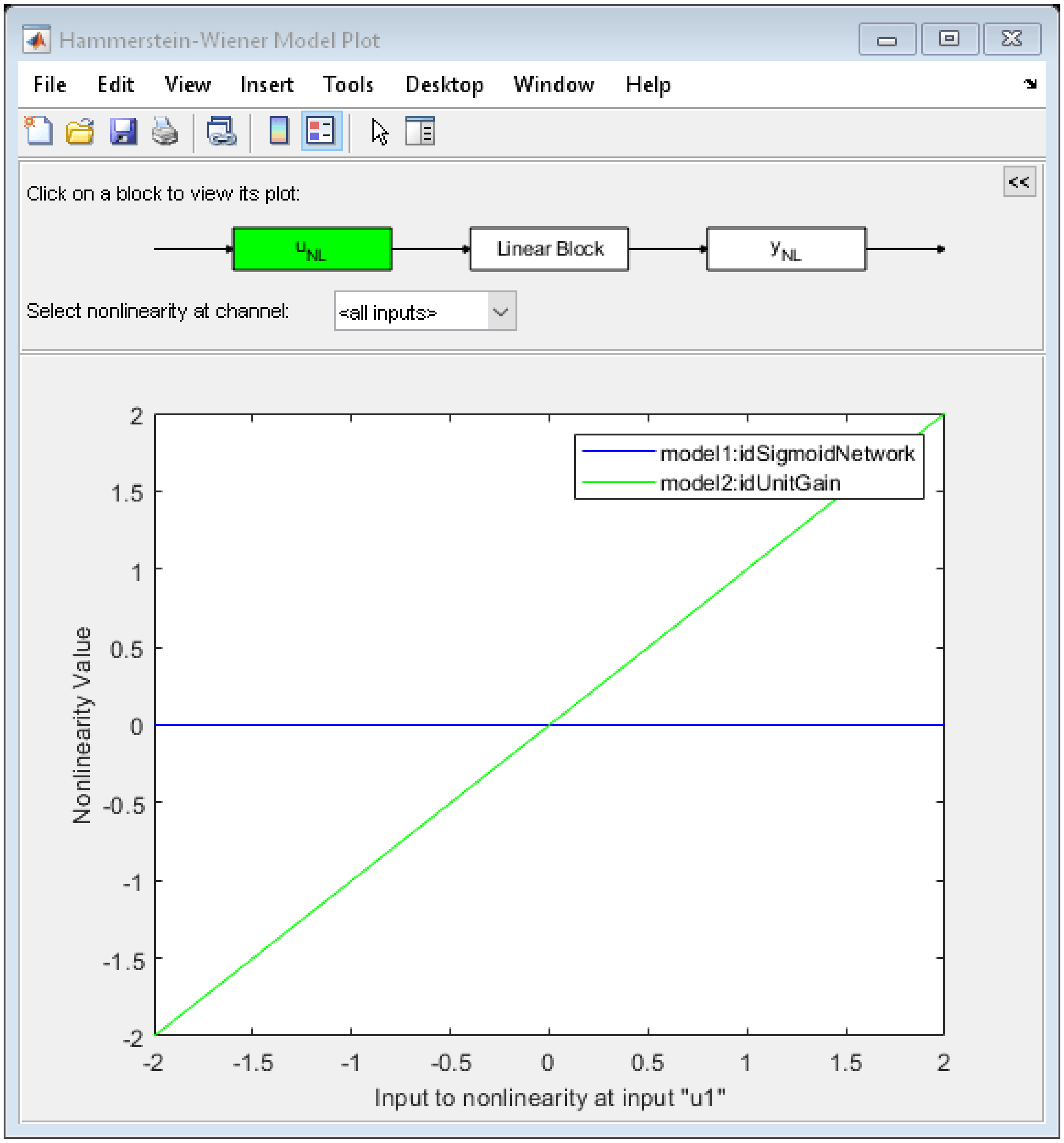
```
load iddata3
model1 = nlhw(z3,[4 2 1],'idSigmoidNetwork','idDeadZone');
model2 = nlhw(z3, [4 2 1],[],'idSigmoidNetwork');
plot(model1,'b-',model2,'g')
```



Specify Number of Samples, Time Samples, and Range of Input Nonlinearity

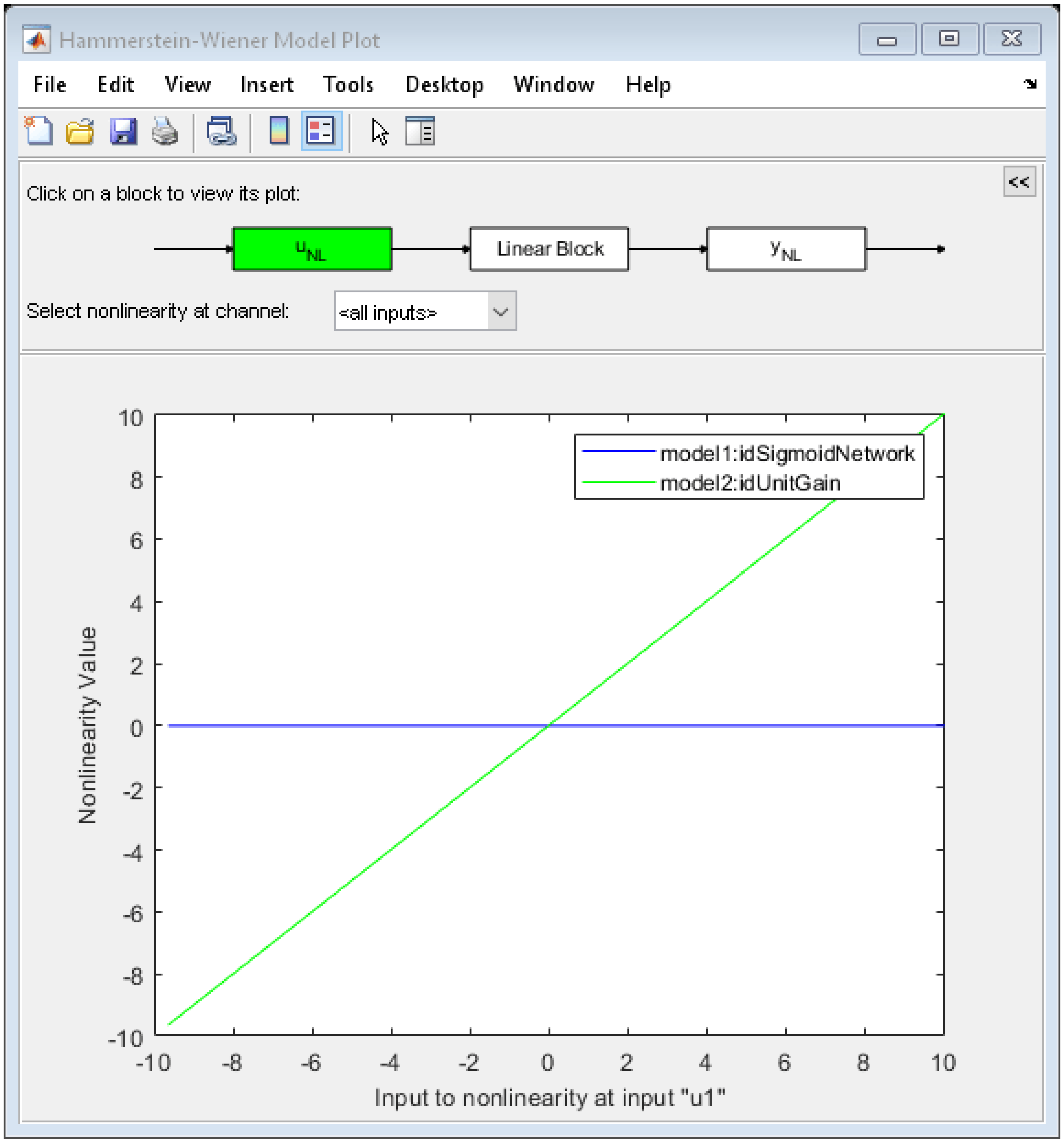
```
load iddata3
model1 = nlhw(z3,[4 2 1],idSigmoidNetwork,idDeadZone);
```

```
model2 = nlhw(z3, [4 2 1],[],idSigmoidNetwork);
plot(model1,'b-',model2,'g','NumberOfSamples',50,'time',10,'InputRange',[-2 2]);
```


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:idSigmoidNetwork
model2:idUnitGain
Nonlinearity Value
Input to nonlinearity at input "u1"

Specify Time Samples, Frequency, and Range of Output Nonlinearity

```
load iddata3
model1 = nlhw(z3,[4 2 1],idSigmoidNetwork, idDeadZone);
model2 = nlhw(z3, [4 2 1],[],idSigmoidNetwork);
plot(model1,model2,'time',1:500,'freq',{0.01,100},'OutputRange',[0 1000]);
```


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:idSigmoidNetwork
model2:idUnitGain
Nonlinearity Value
Input to nonlinearity at input "u1"

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 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 static input and output nonlinearities and linear responses of a Hammerstein-Wiener model.

Examining a Hammerstein-Wiener plot can help you determine whether you have selected a complicated nonlinearity for modeling your system. For example, suppose you use a piecewise-linear input nonlinearity to estimate your model, but the plot indicates saturation behavior. You can estimate a new model using the simpler saturation nonlinearity instead. For multivariable systems, you can use the Hammerstein-Wiener plot to determine whether to exclude nonlinearities for specific channels. If the nonlinearity for a specific input or output channel does not exhibit strong nonlinear behavior, you can estimate a new model after setting the nonlinearity at that channel to unit gain.

You can generate these plots in the **System Identification** app and at the command line. In the plot window, you can view the nonlinearities and linear responses by clicking one of the three blocks that represent the model:

- u_{NL} (*input nonlinearity*)— Click this block to view the static nonlinearity at the input to the `Linear Block`. The plot displays `evaluate(M.InputNonlinearity,u)` where `M` is the Hammerstein-Wiener model, and `u` is the input to the input nonlinearity block. For information about the blocks, see “Structure of Hammerstein-Wiener Models”.
- `Linear Block` — Click this block to view the Step, impulse, Bode, and pole-zero response plots of the embedded linear model (`M.LinearModel`). By default, a step plot of the linear model is displayed.
- y_{NL} (*output nonlinearity*) — Click this block to view the static nonlinearity at the output of the `Linear Block`. The plot displays `evaluate(M.OutputNonlinearity,x)`, where `x` is the output of the linear block.

To learn more about how to configure the linear and nonlinear blocks plots, see “Configuring a Hammerstein-Wiener Plot”.

See Also
`idnlhw` | `nlhw`

Topics
"Structure of Hammerstein-Wiener Models"
"Validating Hammerstein-Wiener Models"

Introduced in R2014a

pole

Poles of dynamic system

Syntax

```
P = pole(sys)
P = pole(sys,J1,...,JN)
```

Description

`P = pole(sys)` returns the poles of the SISO or MIMO dynamic system model `sys`. The output is expressed as the reciprocal of the time units specified in `sys.TimeUnit`. The poles of a dynamic system determine the stability and response of the system.

An open-loop linear time-invariant system is stable if:

- In continuous-time, all the poles of the transfer function have negative real parts. When the poles are visualized on the complex s-plane, then they must all lie in the left-half plane (LHP) to ensure stability.
- In discrete-time, all the poles must have a magnitude strictly smaller than one, that is they must all lie inside the unit circle.

`P = pole(sys,J1,...,JN)` returns the poles `P` of the entries in model array `sys` with subscripts `(J1,...,JN)`.

Examples

Poles of Discrete-Time Transfer Function

Compute the poles of the following discrete-time transfer function:

$$sys(z) = \frac{0.0478z - 0.0464}{z^2 - 1.81z + 0.9048}$$

```
sys = tf([0.04798 0.0464],[1 -1.81 0.9048],0.1);
P = pole(sys)
```

```
P = 2×1 complex

   0.9050 + 0.2929i
   0.9050 - 0.2929i
```

For stable discrete systems, all their poles must have a magnitude strictly smaller than one, that is they must all lie inside the unit circle. The poles in this example are a pair of complex conjugates, and lie inside the unit circle. Hence, the system `sys` is stable.

Poles of Transfer Function

Calculate the poles of following transfer function:

$$sys(s) = \frac{4.2s^2 + 0.25s - 0.004}{s^2 + 9.6s + 17}$$

```
sys = tf([4.2,0.25,-0.004],[1,9.6,17]);
P = pole(sys)
```

```
P = 2×1

   -7.2576
   -2.3424
```

For stable continuous systems, all their poles must have negative real parts. `sys` is stable since the poles are negative, that is, they lie in the left half of the complex plane.

Poles of Models in an Array

For this example, load `invertedPendulumArray.mat`, which contains a 3-by-3 array of inverted pendulum models. The mass of the pendulum varies as you move from model to model along a single column of `sys`, and the length of the pendulum varies as you move along a single row. The mass values used are 100g, 200g and 300g, and the pendulum lengths used are 3m, 2m and 1m respectively.

$$\begin{array}{llll} & Column\ 1 & Column\ 2 & Column\ 3 \\ Row\ 1 & 100g, 3m & 100g, 2m & 100g, 1m \\ Row\ 2 & 200g, 3m & 200g, 2m & 200g, 1m \\ Row\ 3 & 300g, 3m & 300g, 2m & 300g, 1m \end{array}$$

```
load('invertedPendulumArray.mat','sys');
size(sys)
```

```
3x3 array of transfer functions.
Each model has 1 outputs and 1 inputs.
```

Find poles of the model array.

```
P = pole(sys);
P(:,:,2,1)
```

```
ans = 3×1

    2.1071
   -2.1642
   -0.1426
```

`P(:,:,2,1)` corresponds to the poles of the model with 200g pendulum weight and 3m length.

Input Arguments

sys — Dynamic system
dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model, or an array of SISO or MIMO dynamic system models. Dynamic systems that you can use include continuous-time or discrete-time numeric LTI models such as `tf`, `zpk`, or `ss` models.

If `sys` is a generalized state-space model `genss` or an uncertain state-space model `uss`, `pole` returns the poles of the current or nominal value of `sys`. If `sys` is an array of models, `pole` returns the poles of the model corresponding to its subscript `J1,...,JN` in `sys`. For more information on model arrays, see “Model Arrays” (Control System Toolbox).

`J1,...,JN` — Indices of models in array whose poles you want to extract
positive integer

Indices of models in array whose poles you want to extract, specified as a positive integer. You can provide as many indices as there are array dimensions in `sys`. For example, if `sys` is a 4-by-5 array of dynamic system models, the following command extracts the poles for entry (2,3) in the array.

```
P = pole(sys,2,3);
```

Output Arguments

P — Poles of the dynamic system
column vector | array

Poles of the dynamic system, returned as a scalar or an array. If `sys` is:

- A single model, then `P` is a column vector of poles of the dynamic system model `sys`
- A model array, then `P` is an array of poles of each model in `sys`

`P` is expressed as the reciprocal of the time units specified in `sys.TimeUnit`. For example, pole is expressed in 1/minute if `sys.TimeUnit = 'minutes'`.

Depending on the type of system model, poles are computed in the following way:

- 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 the denominator roots. For more information, see `roots`.
- For MIMO transfer functions (or zero-pole-gain models), the poles are returned as the union of the poles for each SISO entry. If some I/O pairs have a common denominator, the roots of such I/O pair denominator are counted only once.

Limitations

- Multiple poles are numerically sensitive and cannot be computed with high accuracy. A pole λ with multiplicity m typically results in 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`).

For more information on multiple poles, see “Sensitivity of Multiple Roots” (Control System Toolbox).

- If `sys` has internal delays, poles are obtained by first setting all internal delays to zero so that the system has a finite number of poles, thereby 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, `pole` returns an error.

  To assess the stability of models with internal delays, use `step` or `impulse`.

See Also
`damp` | `pzmap` | `zero` | `step` | `impulse` | `pzplot`

Topics
“Pole and Zero Locations” (Control System Toolbox)
“Sensitivity of Multiple Roots” (Control System Toolbox)

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:

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

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 = 1×4

    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 = 1×4

         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

`idpoly` | `iddata` | `tfdata` | `zpkdata` | `idssdata` | `polyest`

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)
[sys,ic] = polyest( ___ )
```

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.

`[sys,ic] = polyest( ___ )` returns the estimated initial conditions as an `initialCondition` object. Use this syntax if you plan to simulate or predict the model response using the same estimation input data and then compare the response with the same estimation output data. Incorporating the initial conditions yields a better match during the first part of the simulation.

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 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.
`InitialCondition`	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:

Field	Description
`FitPercent`	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
`LossFcn`	Value of the loss function when the estimation completes.
`MSE`	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
`FPE`	Final prediction error for the model.
`AIC`	Raw Akaike Information Criteria (AIC) measure of model quality.
`AICc`	Small-sample-size corrected AIC.
`nAIC`	Normalized AIC.
`BIC`	Bayesian Information Criteria (BIC).

Report Field	Description
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `polyestOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** — **Description** `Name` — Name of the data set. `Type` — Data type. `Length` — Number of data samples. `Ts` — Sample time. `InterSample` — Input intersample behavior, returned as one of the following values: • `'zoh'` — 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. `InputOffset` — Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`. `OutputOffset` — Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.
`Termination`	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** — **Description** `WhyStop` — Reason for terminating the numerical search. `Iterations` — Number of search iterations performed by the estimation algorithm. `FirstOrderOptimality` — ∞-norm of the gradient search vector when the search algorithm terminates. `FcnCount` — Number of times the objective function was called. `UpdateNorm` — Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `LastImprovement` — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `Algorithm` — Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used. For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.

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

`ic`

Estimated initial conditions, returned as an `initialCondition` object or an object array of `initialCondition` values.

- For a single-experiment data set, `ic` represents, in state-space form, the free response of the transfer function model (*A* and *C* matrices) to the estimated initial states (x_0).
- For a multiple-experiment data set with N_e experiments, `ic` is an object array of length N_e that contains one set of `initialCondition` values for each experiment.

If `polyest` returns `ic` values of `0` and the you know that you have non-zero initial conditions, set the `'InitialCondition'` option in `polyestOptions` to `'estimate'` and pass the updated option set to `polyest`. For example:

```
opt = polyestOptions('InitialCondition','estimate')
[sys,ic] = polyest(data,[nb nc nd nf nk],opt)
```

The default `'auto'` setting of `'InitialCondition'` uses the `'zero'` method when the initial conditions have a negligible effect on the overall estimation-error minimization process. Specifying `'estimate'` ensures that the software estimates values for `ic`.

For more information, see `initialCondition`. For an example of using this argument, see “Obtain Initial Conditions” on page 1-1221.

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.

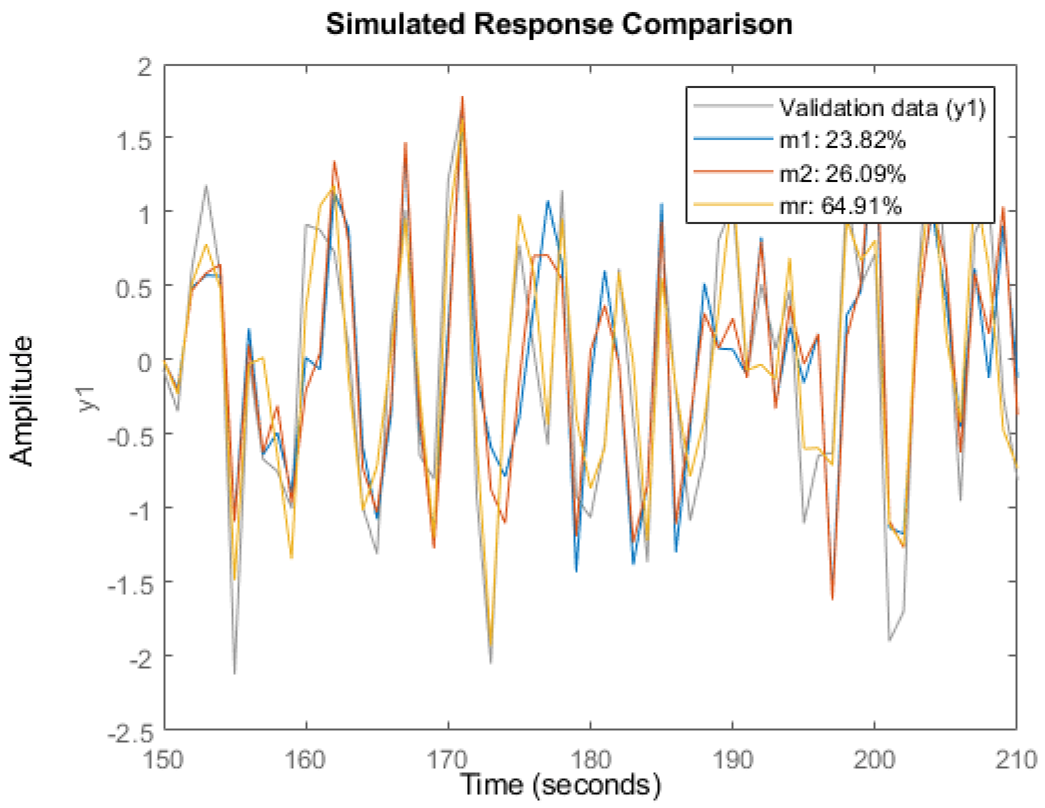
```
opt = polyestOptions;
opt.Regularization.Lambda = 1;
m2 = polyest(m0simdata(1:150),[0 20 20 20 20 1],opt);
```

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];
```

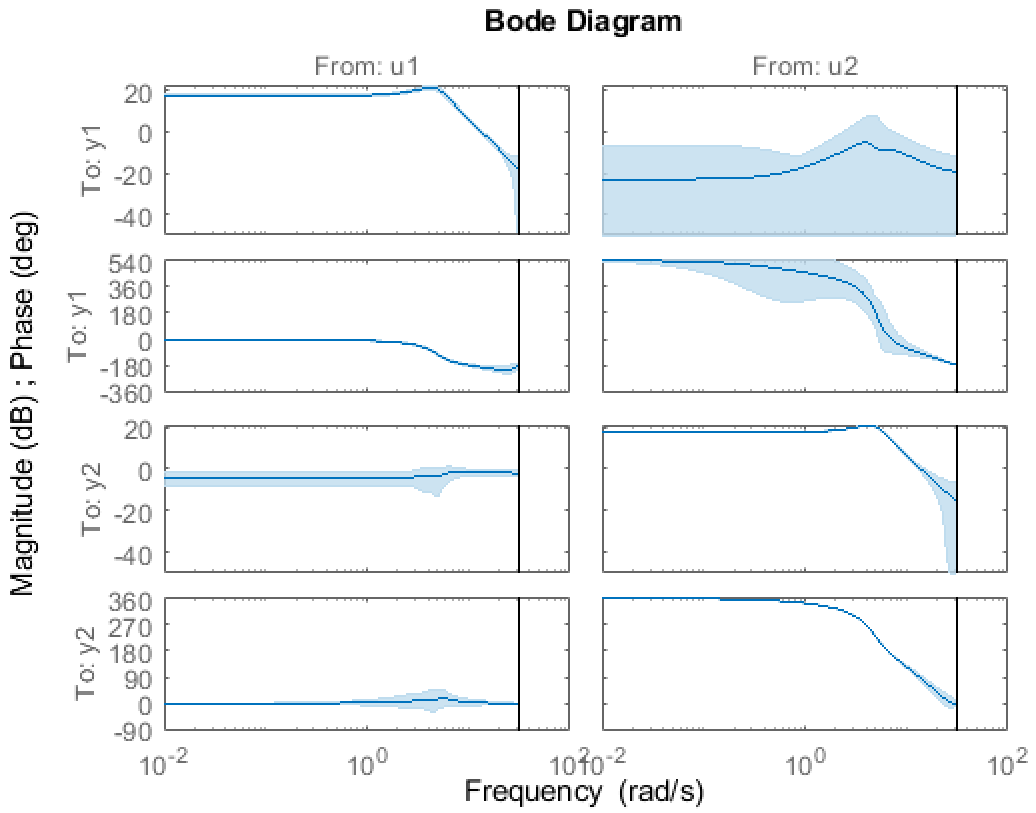
Estimate the model.

```
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.

Obtain Initial Conditions

Load the data.

```
load iddata1ic z1i
```

Estimate a polynomial model `sys` and return the initial conditions in `ic`.

```
na = 2;
nb = 2;
nc = 3;
nd = 3;
nf = 2;
nk = 1;
[sys,ic] = polyest(z1i,[na nb nc nd nf nk]);
ic
```

```
ic = 
  initialCondition with properties:

      A: [7x7 double]
     X0: [7x1 double]
      C: [0 0 0 0 0 0 1]
```

```
    Ts: 0.1000
```

ic is an initialCondition object that encapsulates the free response of sys, in state-space form, to the initial state vector in X0. You can incorporate ic when you simulate sys with the z1i input signal and compare the response with the z1i output signal.

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.

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.

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the lsqnonlin search method (requires Optimization Toolbox). To enable parallel computing, use polyestOptions, set SearchMethod to 'lsqnonlin', and set SearchOptions.Advanced.UseParallel to true.

For example:

```
opt = polyestOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

polyestOptions | idpoly | ar | arx | armax | oe | bj | tfest | procest | ssest | iddata | pem | forecast

Topics
“Regularized Estimates of Model Parameters”
“Apply Initial Conditions when Simulating Identified Linear Models”

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 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`

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

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

If `EstimateCovariance` 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` — Numerical search method used for iterative parameter estimation

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

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.

- Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOptions` Structure When `SearchMethod` is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Tolerance`	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`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see "Large-Scale vs. Medium-Scale Algorithms" (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see "Constrained Nonlinear Optimization Algorithms" (Optimization Toolbox) and "Choosing the Algorithm" (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	`1e-6`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`.	`100`

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

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

  - 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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`

Topics

“Loss Function and Model Quality Metrics”

Introduced in R2012a

polynomialRegressor

Specify polynomial regressor for nonlinear ARX model

Description

Polynomial regressors are polynomials that are composed of delayed input and output variables. For example, $y(t–1)^2$ and $y(t–1)\ u(t–1)$ are both polynomial regressors with orders of 2 and variable delays of one sample. A `polynomialRegressor` object encapsulates a set of polynomial regressors. Use `polynomialRegressor` objects when you create nonlinear ARX models using `idnlarx` or `nlarx`. You can specify `polynomialRegressor` objects along with `linearRegressor` and `customRegressor` objects and combine them into a single combined regressor set.

Creation

Syntax

```
pReg = polynomialRegressor(Variables,Lags)
pReg = polynomialRegressor(Variables,Lags,Order)
pReg = polynomialRegressor(Variables,Lags,Order,UseAbsolute)
pReg = polynomialRegressor(Variables,Lags,Order,UseAbsolute,AllowVariableMix)
pReg =
polynomialRegressor(Variables,Lags,Order,UseAbsolute,AllowVariableMix,AllowLa
gMix)
```

Description

`pReg = polynomialRegressor(Variables,Lags)` creates a `polynomialRegressor` object of order 2 that contains output and input names in Variables and the corresponding lags in Lags. For example, if `Variables` contains `'y'` and `lags` contains the corresponding lag vector `[2 4]`, then the regressors that use `'y'` are $y(t–2)^2$ and $y(t–4)^2$.

`pReg = polynomialRegressor(Variables,Lags,Order)` creates a `polynomialRegressor` object of order `Order` .

`pReg = polynomialRegressor(Variables,Lags,Order,UseAbsolute)` specifies in `UseAbsolute` whether to use the absolute values of the variables to create the regressors.

`pReg = polynomialRegressor(Variables,Lags,Order,UseAbsolute,AllowVariableMix)` specifies in `AllowVariableMix` whether to allow multiple variables in the regressor formulas. For example, if `Variables` is equal to `{'y','u'}`, `Lags` is equal to `{1,1}`, and `Order` is equal to `2`, then a value of `true` for `AllowVariableMix` results in the inclusion of the mixed-variable regressor $y(t–1)u(t–1)$, along with the single-variable regressors $y(t–1)^2$ and $u(t–1)^2$.

`pReg = polynomialRegressor(Variables,Lags,Order,UseAbsolute,AllowVariableMix,AllowLagMix)` specifies in `AllowLagMix` whether to allow different lags in the regressor formulas. For example, if `Variables` is equal to `{'y','u'}`, `Lags` is equal to `{2,[0 3]}`, `Order` is equal to `2`, and `AllowVariableMix` is equal to `false`, then a value of `true` for `AllowLagMix` results in the inclusion of the mixed-lag regressor $u(t)u(t–3)$, along with the unique-lag regressors $y(t–2)^2$, $u(t)^2$, and $u(t–3)^2$. Note that if you set `AllowVariableMix` to `true`, then the regressor set will also include $y(t–2)u(t)$ and $y(t–2)u(t–3)$.

Properties

`Variables` — Output and input variable names

cell array of strings | `iddata` object properties

Output and input variable names, specified as a cell array of strings or a cell array that references the `OutputName` and `InputName` properties of an `iddata` object. Each entry must be a string with no special characters other than white space. For an example of using this property, see “Estimate Nonlinear ARX Model with Polynomial Regressors” on page 1-1238.

Example: `{'y1','u1'}`

Example: `[z.OutputName; z.InputName]'`

Lags — Lags in each variable

cell array of non-negative integers

Lags in each variable, specified as a 1-by-n_v cell array of non-negative integer row vectors, where n_v is the total number of regressor variables. Each row vector contains n_r integers that specify the n_r regressor lags for the corresponding variable. For instance, suppose that you want the following regressors:

- Output variable y_1: $y_1(t–1)^2$ and $y_1(t–2)^2$
- Input variable u_1: $u_1(t–3)^2$

To obtain these lags, set `Lags` to `{[1 2],3}`.

If a lag corresponds to an output variable of an `idnlarx` model, the minimum lag must be greater than or equal to 1.

For an example of using this property, see “Estimate Nonlinear ARX Model with Polynomial Regressors” on page 1-1238.

Example: `{1 1}`

Example: `{[1 2],[1,3,4]}`

`UseAbsolute` — Absolute value indicator

`false` (default) | logical vector

Absolute value indicator that determines whether to use the absolute value of a regressor variable instead of the signed value, specified as a logical vector with a length equal to the number of variables.

For an example of setting this property, see “Use Absolute Value in Polynomial Regressor Set” on page 1-1240.

Example: `[true,false]`

`AllowVariableMix` — Mixed variables indicator

`false` (default) | logical vector

Mixed variables indicator that determines whether to use multiple variables in regressor formulas such as *y*(*t*–1)*u*(*t*–1), specified as a logical vector with a length equal to the number of variables.

For an example of setting this property, see “Use Multiple Variables in Polynomial Regressor Term” on page 1-1240.

Example: `[true,false]`

`AllowLagMix` — Mixed lag indicator
`false` (default) | logical vector

Mixed lag indicator that determines whether to use different lags in regressor formulas such as *u*(*t*)*u*(*t*–3), specified as a logical vector with a length equal to the number of variables.

To set this property for an existing nonlinear ARX model `sys`, use dot notation, as shown in the following command.

For an example of setting this property, see “Use Mixed Lags in Polynomial Regressor Term” on page 1-1241.

Example: `[true,false]`

`TimeVariable` — Name of time variable
`'t'` (default) | character array | string

Name of the time variable, specified as a valid MATLAB variable name that is distinct from values in `Variables`.

Example: `'ClockTime'`

Examples

Estimate Nonlinear ARX Model with Polynomial Regressors

Load the data and create an `iddata` object `z` with a sample time of 0.1 seconds.

```
load twotankdata y u
z = iddata(y,u,'Ts',0.1);
```

Specify polynomial regressors that have the forms $u(t-2)^2$, $u(t-4)^2$, and $y(t-1)^2$.

Use the properties of `z` to specify the variable names.

```
Variables = [z.OutputName;z.InputName];
```

Specify the lags.

```
Lags = {1,[2 4]};
```

Create the regressor. The default order is 2.

```
pReg = polynomialRegressor(Variables,Lags)
```

```
pReg =
Order 2 regressors in variables y1, u1
               Order: 2
```

```
            Variables: {'y1'  'u1'}
                 Lags: {[1]  [2 4]}
          UseAbsolute: [0 0]
     AllowVariableMix: 0
          AllowLagMix: 0
         TimeVariable: 't'

  Regressors described by this set
```

Use pReg to estimate the nonlinear ARX model.

```
sys = nlarx(z,pReg)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Order 2 regressors in variables y1, u1
  List of all regressors

Output function: Wavelet network with 66 units
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z".
Fit to estimation data: 95.97% (prediction focus)
FPE: 5.843e-05, MSE: 5.569e-05
```

View the regressors.

```
getreg(sys)
```

```
ans = 3x1 cell
    {'y1(t-1)^2'}
    {'u1(t-2)^2'}
    {'u1(t-4)^2'}
```

Specify Order for Polynomial Regressor

Specify the third-order polynomial regressor $u_1(t-2)^3$.

```
Variables = 'u1';
Lags = 2;
Order = 3;
pReg = polynomialRegressor(Variables,Lags,Order)
```

```
pReg =
Order 3 regressors in variables u1
               Order: 3
           Variables: {'u1'}
                Lags: {[2]}
         UseAbsolute: 0
    AllowVariableMix: 0
```

```
         AllowLagMix: 0
        TimeVariable: 't'

  Regressors described by this set
```

Use Absolute Value in Polynomial Regressor Set

Create a second-order polynomial regressor set that uses lags of 3, 10, and 100 in variable `y1` and lags of 0 and 4 in variable `u1`.

```
vars = {'y1','u1'};
lags = {[3 10 100],[0,4]};
```

Specify that the `y1` regressor use the absolute value of `y1`.

```
UseAbs = [true,false];
```

Create the polynomial regressor.

```
reg = polynomialRegressor(vars,lags,2,UseAbs)
```

```
reg = 
Order 2 regressors in variables y1, u1
               Order: 2
           Variables: {'y1'  'u1'}
                Lags: {[3 10 100]  [0 4]}
         UseAbsolute: [1 0]
    AllowVariableMix: 0
         AllowLagMix: 0
        TimeVariable: 't'

  Regressors described by this set
```

Use Multiple Variables in Polynomial Regressor Term

Create a polynomial regressor set that includes the terms $y_1(t-1)^2$, $u_1(t-1)^2$, and $y_1(t-1)u_1(t-1)$.

Specify the variables and lags.

```
vars = {'y1','u1'};
lags = {1, 1};
```

Specify that mixed-variable regressors be created.

```
mixvar = true;
```

Create a second-order polynomial regressor using `mixvar`. Set the fourth position, which represents the `UseAbsolute` property, to `false`.

```
reg = polynomialRegressor(vars,lags,2,false,mixvar)
```

```
reg = 
Order 2 regressors in variables y1, u1
```

```
               Order: 2
           Variables: {'y1'  'u1'}
                Lags: {[1]  [1]}
         UseAbsolute: [0 0]
    AllowVariableMix: 1
         AllowLagMix: 0
        TimeVariable: 't'

  Regressors described by this set
```

As an alternative, you can create the regressor specification first using the variables and lags and set the `AllowVariableMix` property afterward using dot notation.

```
reg1 = polynomialRegressor(vars,lags);
reg1.AllowVariablemix = true
```

```
reg1 =
Order 2 regressors in variables y1, u1
               Order: 2
           Variables: {'y1'  'u1'}
                Lags: {[1]  [1]}
         UseAbsolute: [0 0]
    AllowVariableMix: 1
         AllowLagMix: 0
        TimeVariable: 't'

  Regressors described by this set
```

Use `reg1` in a nonlinear ARX model.

```
load twotankdata y u;
z = iddata(y,u,'Ts',0.1);
sys = nlarx(z,reg1);
```

View the regressors.

```
getreg(sys)
```

```
ans = 3x1 cell
    {'y1(t-1)^2'      }
    {'u1(t-1)^2'      }
    {'y1(t-1)*u1(t-1)'}
```

The regressors include mixed-variable terms.

Use Mixed Lags in Polynomial Regressor Term

Specify a polynomial regressor set that includes a term of the form $u(t)u(t-3)$.

Specify the variable names and the lags.

```
vars = {'y1','u1'};
lags = {2,[0 3]};
```

Initialize a second-order polynomial regressor.

```
reg = polynomialRegressor(vars,lags);
```

Specify that the regressor use mixed lags.

```
reg.AllowLagMix = true;
```

Use the regressor set in a nonlinear ARX model.

```
load twotankdata y u;
z = iddata(y,u,'Ts',0.1);
sys = nlarx(z,reg);
```

View the regressors.

```
getreg(sys)
```

```
ans = 4x1 cell
    {'y1(t-2)^2'    }
    {'u1(t)^2'      }
    {'u1(t-3)^2'    }
    {'u1(t)*u1(t-3)'}
```

The regressors include the mixed-lag term.

Estimate Nonlinear ARX Model with Polynomial and Linear Regressors

Load the data and create an `iddata` object `z`.

```
load twotankdata y u
z = iddata(y,u,'Ts',0.1);
```

Specify polynomial regressors that have the forms $u(t-2)^2$ and $u(t-4)^2$. Also specify a linear regressor of the form $y(t-1)$.

Specify the input lag.

```
uLags = {[2 4]};
```

Specify the polynomial regressors. The default regressor order is 2.

```
pReg = polynomialRegressor(z.InputName,uLags);
```

Specify the output lag and specify the linear regressor.

```
lLags = 1;
lReg = linearRegressor(z.OutputName,lLags);
```

Estimate a nonlinear ARX model.

```
reg = [pReg;lReg]
```

```
reg = 
[2 1] array of polynomialRegressor, linearRegressor objects.
------------------------------------
1. Order 2 regressors in variables u1
```

```
               Order: 2
           Variables: {'u1'}
                Lags: {[2 4]}
         UseAbsolute: 0
    AllowVariableMix: 0
         AllowLagMix: 0
        TimeVariable: 't'

------------------------------------
2. Linear regressors in variables y1
       Variables: {'y1'}
            Lags: {[1]}
     UseAbsolute: 0
    TimeVariable: 't'


Regressors described by this set
```

```
sys = nlarx(z,reg)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1
  2. Order 2 regressors in variables u1
  List of all regressors

Output function: Wavelet network with 21 units
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z".
Fit to estimation data: 96.56% (prediction focus)
FPE: 4.133e-05, MSE: 4.059e-05
```

View the regressors.

```
getreg(sys)
```

```
ans = 3x1 cell
    {'u1(t-2)^2'}
    {'u1(t-4)^2'}
    {'y1(t-1)'  }
```

Specify Linear, Polynomial, and Custom Regressors

Load the estimation data `z1`, which has one input and one output, and obtain the output and input names.

```
load iddata1 z1;
names = [z1.OutputName z1.InputName]
names = 1x2 cell
    {'y1'}    {'u1'}
```

Specify L as the set of linear regressors that represents $y_1(t-1)$, $u_1(t-2)$, and $u_1(t-5)$.

```
L = linearRegressor(names,{1,[2 5]});
```

Specify P as the polynomial regressor $y_1(t-1)^2$.

```
P = polynomialRegressor(names(1),1,2);
```

Specify C as the custom regressor $y_1(t-2)u_1(t-3)$. Use an anonymous function handle to define this function.

```
C = customRegressor(names,{2 3},@(x,y)x.*y)
```

```
C =
Custom regressor: y1(t-2).*u1(t-3)
    VariablesToRegressorFcn: @(x,y)x.*y
                  Variables: {'y1'  'u1'}
                       Lags: {[2]  [3]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

Combine the regressors in the column vector R.

```
R = [L;P;C]
```

```
R =
[3 1] array of linearRegressor, polynomialRegressor, customRegressor objects.
------------------------------------
1. Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1]  [2 5]}
     UseAbsolute: [0 0]
    TimeVariable: 't'

------------------------------------
2. Order 2 regressors in variables y1
               Order: 2
           Variables: {'y1'}
                Lags: {[1]}
         UseAbsolute: 0
    AllowVariableMix: 0
         AllowLagMix: 0
        TimeVariable: 't'

------------------------------------
3. Custom regressor: y1(t-2).*u1(t-3)
    VariablesToRegressorFcn: @(x,y)x.*y
                  Variables: {'y1'  'u1'}
                       Lags: {[2]  [3]}
                 Vectorized: 1
               TimeVariable: 't'
```

```
Regressors described by this set
```

Estimate a nonlinear ARX model with R.

```
sys = nlarx(z1,R)
```

```
sys =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables y1
  3. Custom regressor: y1(t-2).*u1(t-3)
  List of all regressors

Output function: Wavelet network with 1 units
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 59.73% (prediction focus)
FPE: 3.356, MSE: 3.147
```

View the full regressor set.

```
getreg(sys)
```

```
ans = 5x1 cell
    {'y1(t-1)'         }
    {'u1(t-2)'         }
    {'u1(t-5)'         }
    {'y1(t-1)^2'       }
    {'y1(t-2).*u1(t-3)'}
```

See Also

`idnlarx` | `nlarx` | `getreg` | `linearRegressor` | `customRegressor`

Introduced in R2021a

polyreg

(Not recommended) Powers and products of standard regressors

Note `polyreg` is not recommended. Use `polynomialRegressor` instead to create polynomial regressor objects, and them add them direclty to the regressor `idnlarx` `Regressors` property. For more information, see “Compatibility Considerations”.

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],idWaveletNetwork,'CustomReg',R);
```

View all model regressors (standard and custom).

```
getreg(m)
```

```
ans = 8x1 cell
    {'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],idWaveletNetwork,'CustomReg',R);
```

Compatibility Considerations

`polyreg` is not recommended
Not recommended starting in R2021a

Starting in R2021a, the `polyreg` command is not recommended. Use the `polynomialRegressor` command instead to construct polynomial regressors. Doing so improves the computation speed and the accuracy of results, reduces the memory footprint of the `idnlarx` object, and improves code generation in Simulink.

After creating a polynomial regressor, add it directly to the `idnlarx` model `Regressor` property by using the syntax `model.Regressors(end+1) = new_polymomial_regressor_object`.

There are no plans to remove `polyreg` at this time.

See Also

`getreg` | `idnlarx` | `nlarx` | `polynomialRegressor`

Topics
"Identifying Nonlinear ARX Models"

Introduced in R2007a

predict

Predict K-step-ahead model output

Syntax

```
yp = predict(sys,data,K)
yp = predict(sys,data,K,opt)
[yp,ic,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

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.

`yp = predict(sys,data,K)` predicts the output of an identified model `sys`, `K` steps ahead using the measured input-output 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,ic,sys_pred] = predict( ___ )` also returns the estimated values for initial conditions `ic` and a predictor model `sys_pred`. Use this syntax 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 1-1260.

You can also plot the predicted model response using the `compare` command. The `compare` command compares the prediction results with observed data and displays a quantitative goodness of fit.

`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]);
```

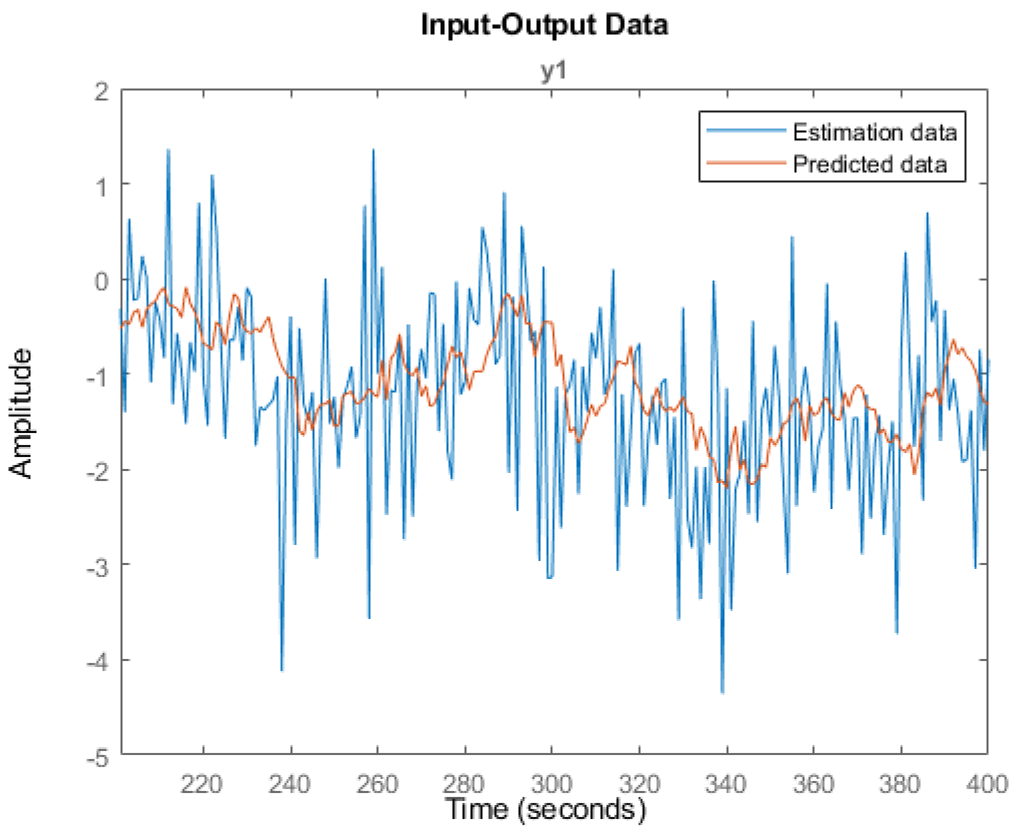
Predict the output of the model using a prediction horizon of 4.

```
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.

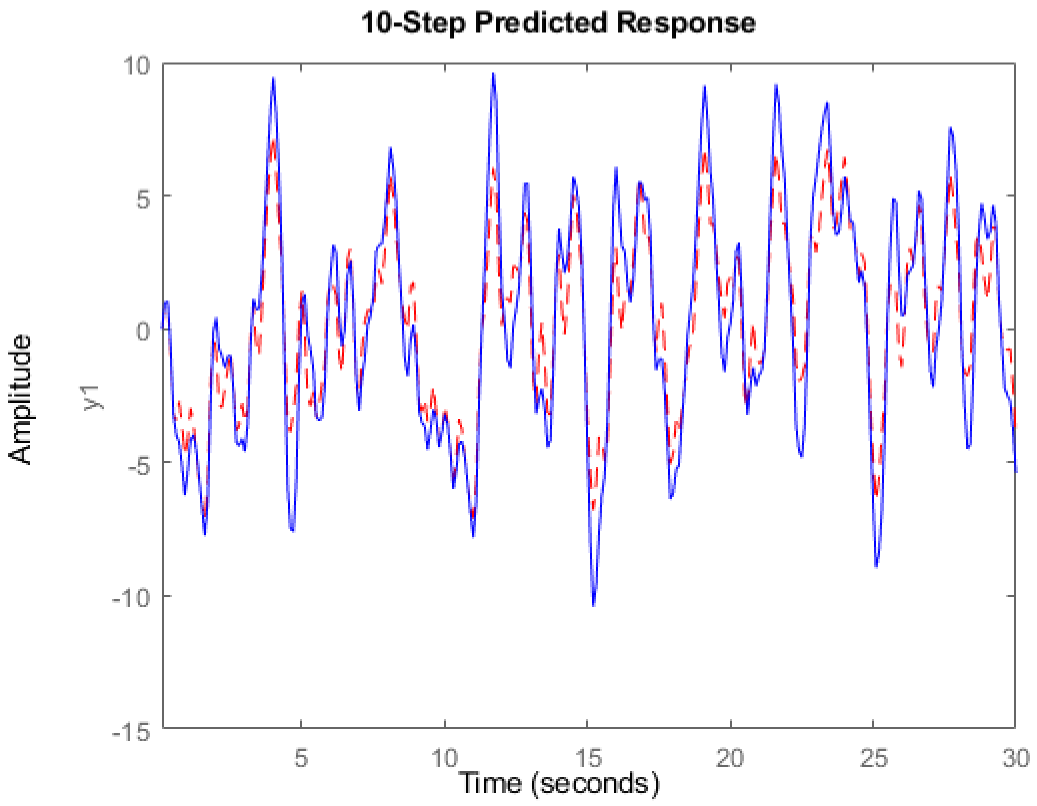
```
 sys2 = tfest(data,2);
```

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.

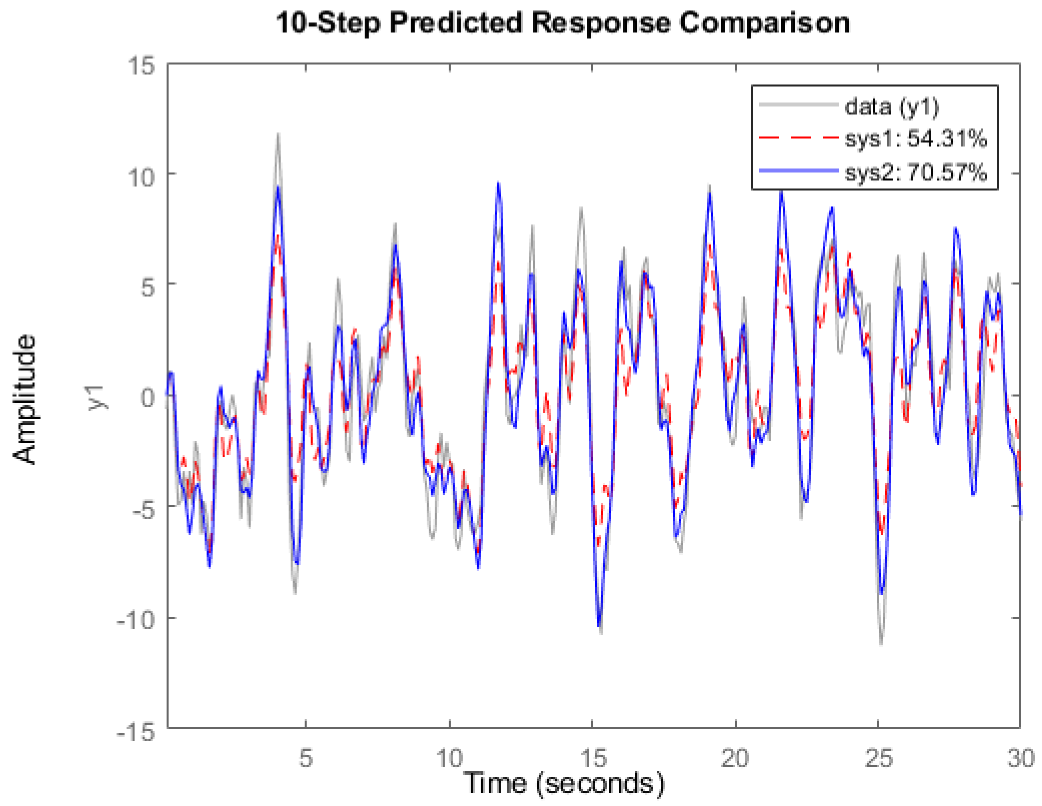
```
predict(sys1,'r--',sys2,'b',data,10,opt);
```



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,ic,sysp] = predict(sys,data,K);
```

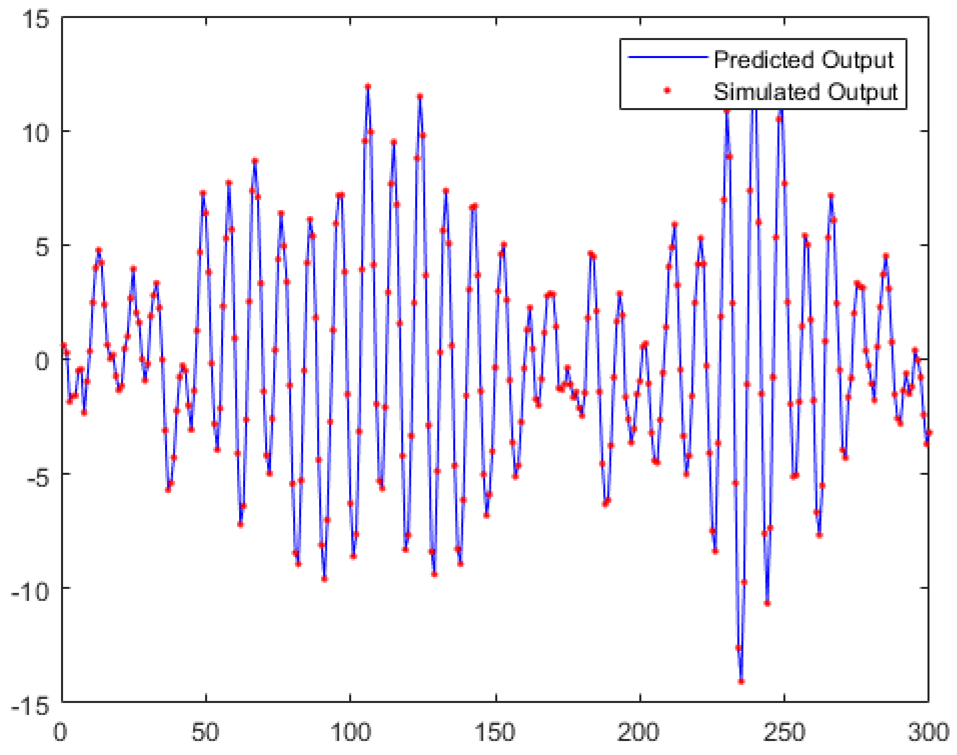
`yp` is the predicted model response, `ic` contains the estimated initial conditions, and `sysp` is the predictor model.

Simulate the predictor model with inputs `[data.OutputData,data.InputData]` and initial conditions `ic`.

```
opt = simOptions;
opt.InitialCondition = ic;
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')
```



Predict Model Using Initial Conditions Obtained During Estimation

Incorporate initial conditions that you obtained previously into your model prediction.

Load the data.

```
load iddata1ic z1i
```

Specify the ARMAX estimation option to estimate the initial state.

```
estimOpt = armaxOptions('InitialCondition','estimate');
```

Estimate an ARMAX model and return an `initialCondition` object `ic` that encapsulates the initial conditions in state-space form.

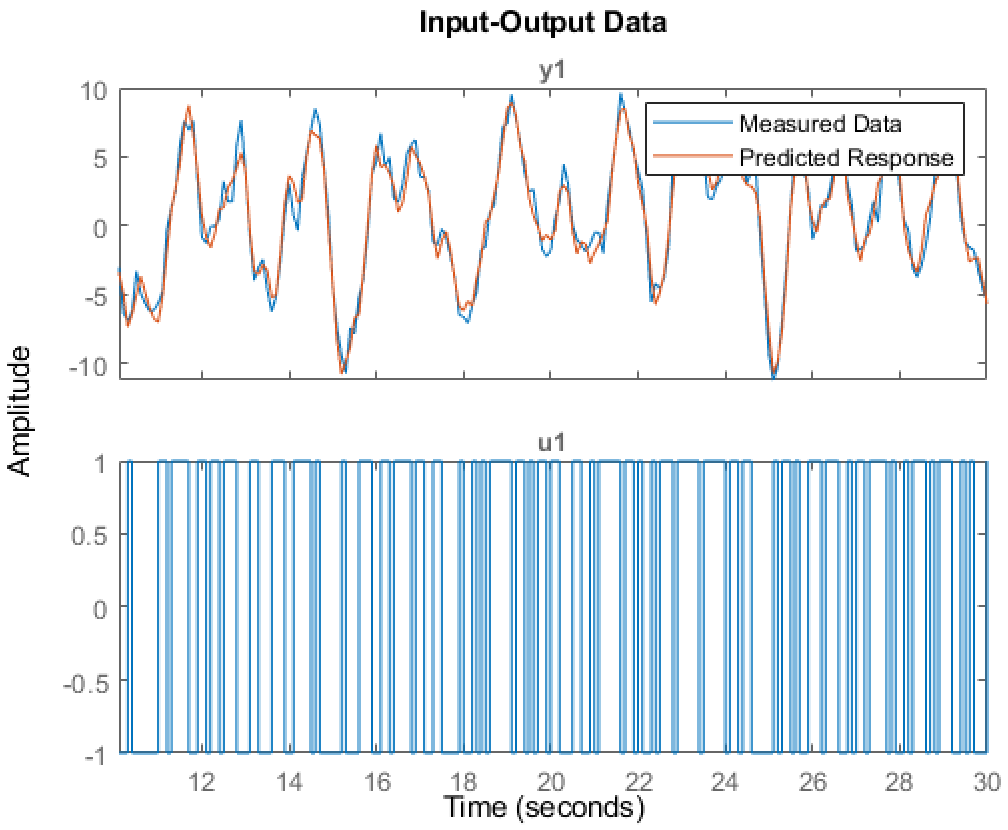
```
na = 2;
nb = 2;
nc = 2;
nk = 1;
[sys,ic] = armax(z1i,[na nb nc nk],estimOpt);
```

Specify the initial conditions for prediction.

```
predictOpt = predictOptions('InitialCondition',ic);
```

Predict the model and obtain the model response. Plot the response `y` with the measured data.

```
y = predict(sys,z1i,predictOpt);
plot(z1i,y)
legend('Measured Data','Predicted Response')
```



The measured and predicted responses show good agreement at the start of the prediction.

Understand Use of Historical Data for Model Prediction

Perform model prediction using historical data to specify initial conditions. You first predict using the `predict` command and specify the historical data using the `predictOptions` option set. You then reproduce the predicted response by manually mapping the historical data to initial states.

Load a two-input, one-output dataset.

```
load iddata7 z7
```

Identify a fifth-order state-space model using the data.

```
sys = n4sid(z7,5);
```

Split the dataset into two parts.

```
zA = z7(1:15);
zB = z7(16:end);
```

Suppose that you want to compute the 10-step-ahead prediction of the response of the identified system for data `zB`. For initial conditions, use the signal values in `zA` as the historical record. That is, the input and output values for the time immediately preceding data in `zB`.

```
IO = struct('Input',zA.InputData,'Output',zA.OutputData);
opt = predictOptions('InitialCondition',IO);
```

Generate the 10-step-ahead prediction for data `zB` using the specified initial conditions and `predict`.

```
[yp,x0,Predictor] = predict(sys,zB,10,opt);
```

`yp` is the predicted model response, `x0` are the initial states corresponding to the predictor model `Predictor`. You can simulate `Predictor` using `x0` as initial conditions to reproduce `yp.OutputData`.

Now reproduce the output by manually mapping the historical data to initial states. To do so, minimize 1-step prediction errors over the time span of `zA`.

```
x0est = data2state(sys,zA);
```

`x0est` contains the values of the five states of `sys` at the time instant immediately after the most recent data sample in `zA`.

The `Predictor` has more states than the original system due to the 10-step prediction horizon. Specify the additional states induced by the horizon to zero initial values, and then append `x0est`.

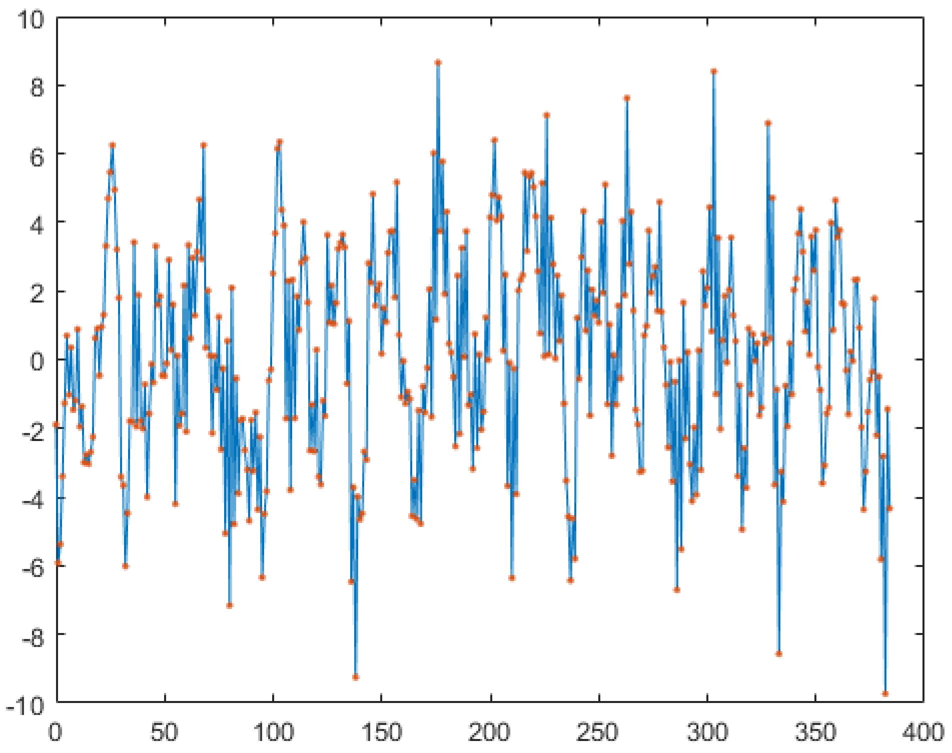
```
x0Predictor = zeros(order(Predictor),1);
x0Predictor(end-4:end) = x0est;
```

Simulate the predictor using `[zB.OutputData,zB.InputData]` as the input signal and `x0Predictor` as initial conditions.

```
uData = [zB.OutputData,zB.InputData]; % signals required for prediction
[ysim,t,xsim] = lsim(Predictor,uData,[],x0Predictor);
```

Plot the predicted output of the `predict` command `yp.OutputData` and the manually computed results `ysim`.

```
plot(t,yp.OutputData,t,ysim, '.')
```



`ysim` is the same as `yp.OutputData`.

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, $\widehat{y}(t) = y(t-1)$, 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 the Linespec argument of `plot`.

Output Arguments

yp — Predicted output response
`iddata` object | matrix of doubles

Predicted output response, returned as one of the following:

- `iddata` object — When `data` is an `iddata` object. The `OutputData` property of `yp` stores the values of the predicted output. The time variable takes values in the range represented by `data.SamplingInstants`.

- Matrix of doubles — When `data` is a matrix of doubles.

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`. In other words, the predicted response at time point `r` of measured data is stored in the `r+K-1` sample of `yp`. Note that at time `r`, the future inputs `u(r+1)`, `u(r+2)`,..., `u(r+K)` required for prediction are assumed to be known. 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.

`ic` — Estimated initial conditions
column vector | `initialCondition` object | cell array

Estimated initial conditions corresponding to the predictor model `sys_pred`, returned as a column vector, an `initialCondition` object, or a cell array.

- If `sys` is a linear transfer function or polynomial model, then `ic` is an `initialCondition` object. The `initialCondition` object encapsulates the free response of `sys`, in state-space form, with the corresponding initial state vector.
- If `sys` is any other type of linear or nonlinear dynamic model, then `ic` is an initial state vector, returned as a column vector of size equal to the number of states.
- If `data` contains multiexperiment data, then `ic` is a cell array of size *Ne*, where *Ne* is the number of experiments.

To reproduce prediction results, you can simulate `sys_pred` using `ic` as the initial conditions. For an example, see “Reproduce Prediction Results by Simulation” on page 1-1253.

If `sys` is an `idnlarx` model, `ic` 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 conditions `ic` 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 the input and `ic` as the 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 1-1253.
- 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 `ic`. For a definition of the states of `idnlhw` models, see “Definition of idnlhw States” on page 1-654.
- If `sys` is a nonlinear ARX model (`idnlarx`), `sys_pred` and `ic` 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.

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.

See Also
`predictOptions` | `compare` | `pe` | `sim` | `simsd` | `iddata` | `forecast`

Topics
"Simulate and Predict Identified Model Output"
"Simulation and Prediction at the Command Line"

Introduced before R2006a

predict

Predict state and state estimation error covariance at next time step using extended or unscented Kalman filter, or particle filter

Syntax

```
[PredictedState,PredictedStateCovariance] = predict(obj)
[PredictedState,PredictedStateCovariance] = predict(obj,Us1,...Usn)
```

Description

The `predict` command predicts the state and state estimation error covariance of an `extendedKalmanFilter`, `unscentedKalmanFilter` or `particleFilter` 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 1-1268.

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.

`[PredictedState,PredictedStateCovariance] = predict(obj)` predicts state estimate and state estimation error covariance of an extended or unscented Kalman filter, or particle filter object `obj` at the next time step.

You create `obj` using the `extendedKalmanFilter`, `unscentedKalmanFilter` or `particleFilter` 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;
```

Initialize arrays to capture results of the estimation.

```
residBuf = [];
xcorBuf = [];
xpredBuf = [];
```

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. Compute the residual between the predicted and actual measurement to assess how well the filter is performing and converging. Computing the residual is an optional step. When you use `residual`, place the command immediately before the `correct` command. If the prediction matches the measurement, the residual is zero.

After you perform the real-time commands for the time step, buffer the results so that you can plot them after the run is complete.

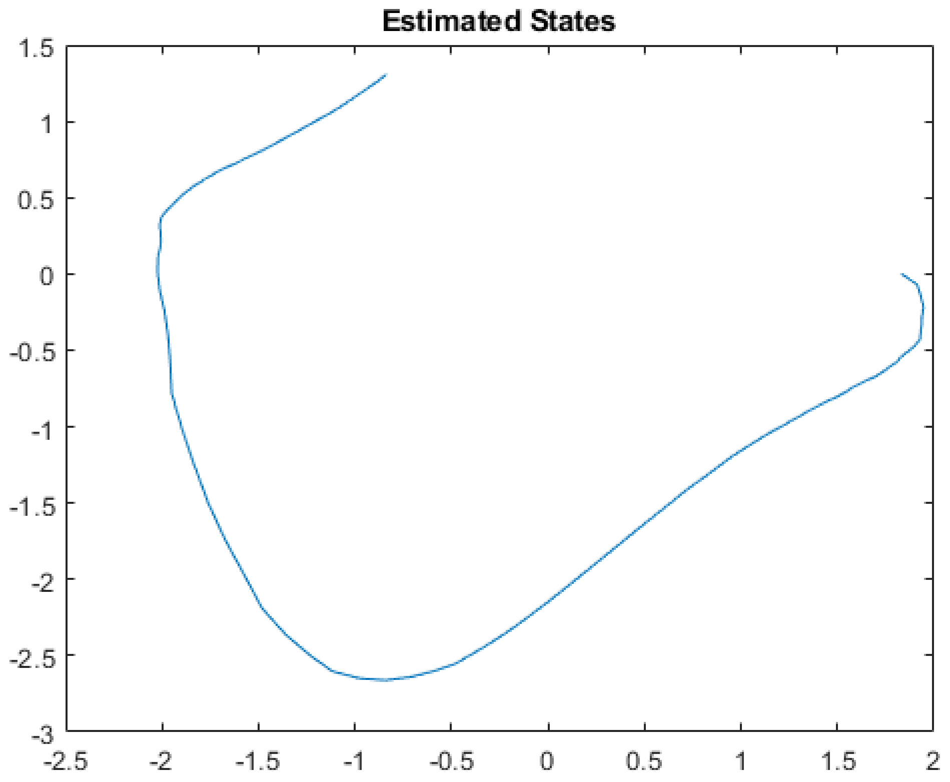
```
for k = 1:size(y)
    [Residual,ResidualCovariance] = residual(obj,y(k));
    [CorrectedState,CorrectedStateCovariance] = correct(obj,y(k));
    [PredictedState,PredictedStateCovariance] = predict(obj);

     residBuf(k,:) = Residual;
     xcorBuf(k,:) = CorrectedState';
     xpredBuf(k,:) = PredictedState';

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” on page 1-1268.

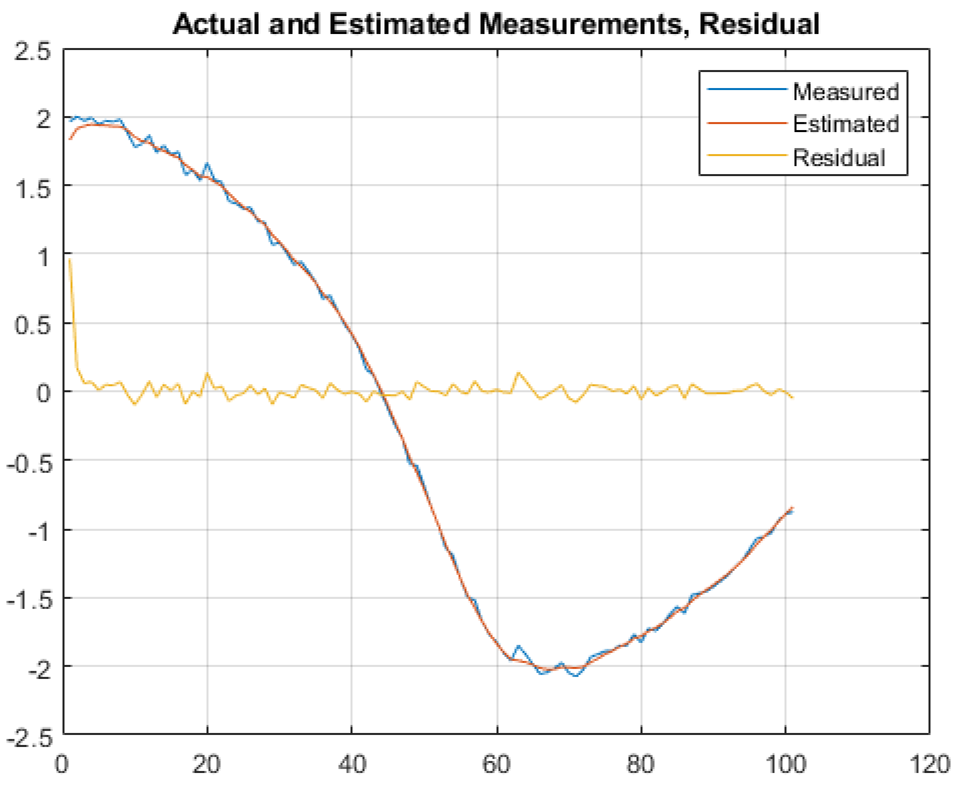
Plot the estimated states, using postcorrection values.

```
plot(xcorBuf(:,1), xcorBuf(:,2))
title('Estimated States')
```



Plot the actual measurement, the corrected estimated measurement, and the residual. For the measurement function in `vdpMeasurementFcn`, the measurement is the first state.

```
M = [y,xcorBuf(:,1),residBuf];
plot(M)
grid on
title('Actual and Estimated Measurements, Residual')
legend('Measured','Estimated','Residual')
```



The estimate tracks the measurement closely. After the initial transient, the residual remains relatively small throughout the run.

Estimate States Online using Particle Filter

Load the van der Pol ODE data, and specify the sample time.

`vdpODEdata.mat` contains a simulation of the van der Pol ODE with nonlinearity parameter mu=1, using ode45, with initial conditions `[2;0]`. The true state was extracted with sample time `dt = 0.05`.

```
load ('vdpODEdata.mat','xTrue','dt')
tSpan = 0:dt:5;
```

Get the measurements. For this example, a sensor measures the first state with a Gaussian noise with standard deviation `0.04`.

```
sqrtR = 0.04;
yMeas = xTrue(:,1) + sqrtR*randn(numel(tSpan),1);
```

Create a particle filter, and set the state transition and measurement likelihood functions.

```
myPF = particleFilter(@vdpParticleFilterStateFcn,@vdpMeasurementLikelihoodFcn);
```

Initialize the particle filter at state `[2; 0]` with unit covariance, and use `1000` particles.

```
initialize(myPF,1000,[2;0],eye(2));
```

Pick the `mean` state estimation and `systematic` resampling methods.

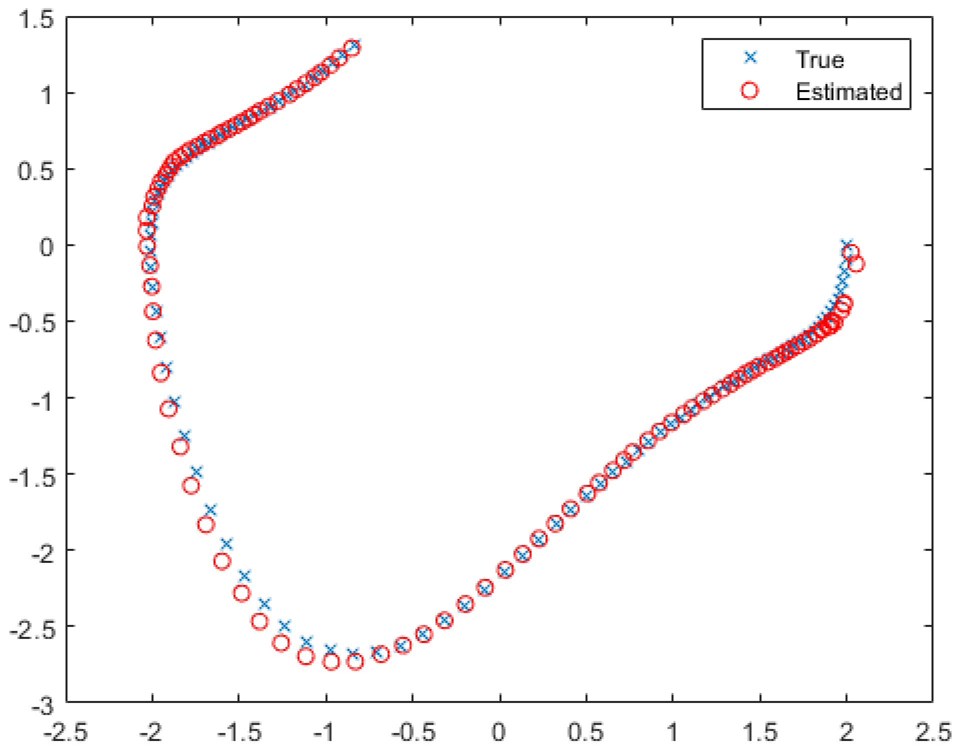
```
myPF.StateEstimationMethod = 'mean';
myPF.ResamplingMethod = 'systematic';
```

Estimate the states using the `correct` and `predict` commands, and store the estimated states.

```
xEst = zeros(size(xTrue));
for k=1:size(xTrue,1)
    xEst(k,:) = correct(myPF,yMeas(k));
    predict(myPF);
end
```

Plot the results, and compare the estimated and true states.

```
figure(1)
plot(xTrue(:,1),xTrue(:,2),'x',xEst(:,1),xEst(:,2),'ro')
legend('True','Estimated')
```



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 following 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 the next time step, given `u[k]=0.2`.

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

Retrieve the error, or *residual*, between the prediction and the measurement.

```
[Residual, ResidualCovariance] = residual(obj,0.8,0.2);
```

Input Arguments

obj — Extended or unscented Kalman filter, or particle filter object
`extendedKalmanFilter` object | `unscentedKalmanFilter` object | `particleFilter` object

Extended or unscented Kalman filter, or particle 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.
- `particleFilter` — Uses the particle 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`, `unscentedKalmanFilter` and `particleFilter` 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, or particle filter object, `obj`, to implement the estimation 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 1-1263 or “Estimate States Online using Particle Filter” on page 1-1265.

See Also

`correct` | `clone` | `extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter` | `initialize` | `residual`

Topics

“Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter”

"Generate Code for Online State Estimation in MATLAB"
"What Is Online Estimation?"
"Extended and Unscented Kalman Filter Algorithms for Online State Estimation"

Introduced in R2016b

predictOptions

Option set for `predict`

Syntax

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

Description

`opt = predictOptions` creates the default option set for `predict`. Use dot notation to modify this option set. Any options that you do not modify retain their default values.

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

Examples

Specify Output Offset for Predicting Model Response

Create a default option set for model prediction.

```
opt = predictOptions;
```

Specify the output offsets for a two-output model as 2 and 5, respectively.

```
opt.OutputOffset = [2;5];
```

The software subtracts the offset value `OutputOffset(i)` from the *i* th output signal before using the output to predict the model response. The software then adds back these offsets to the predicted response to give the final response.

Specify Zero Initial Conditions for Model Prediction

Create an option set for `predict` using zero initial conditions.

```
opt = predictOptions('InitialCondition','z');
```

Use Historical Data to Specify Initial Conditions for Model Prediction

Load a two-input, one-output dataset.

```
load iddata7 z7
```

Identify a fifth-order state-space model using the data.

```
sys = n4sid(z7,5);
```

Split the dataset into two parts.

```
zA = z7(1:15);
zB = z7(16:end);
```

Suppose that you want to compute the 10-step-ahead prediction of the response of the identified system for data `zB`. For initial conditions, use the signal values in `zA` as the historical record. That is, the input and output values for the time immediately preceding data in `zB`.

```
IO = struct('Input',zA.InputData,'Output',zA.OutputData);
opt = predictOptions('InitialCondition',IO);
```

Generate the 10-step-ahead prediction for data `zB` using the specified initial conditions.

```
[yp,x0,Predictor] = predict(sys,zB,10,opt);
```

`yp` is the predicted model response, `x0` are the initial states corresponding to the predictor model `Predictor`. You can simulate `Predictor` using `x0` as initial conditions to reproduce `yp.OutputData`.

To understand how the past data is mapped to the initial states of the model, see “Understand Use of Historical Data for Model Prediction” on page 1-1255.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `predictOptions('InitialCondition','z')` specifies zero initial conditions for the measured input-output data.

`InitialCondition` — Handling of initial conditions

`'e'` (default) | `'z'` | `'d'` | column vector | matrix | `initialCondition` object | object array | structure | `idpar` object `x0Obj`

Handling of initial conditions, specified as the comma-separated pair consisting of `'InitialCondition'` and one of the following values:

- `'z'` — Zero initial conditions.
- `'e'` — Estimate initial conditions such that the prediction error for observed output is minimized.

  For nonlinear grey-box models, only those initial states `i` that are designated as free in the model (`sys.InitialStates(i).Fixed = false`) are estimated. To estimate all the states of the model, first specify all the `Nx` states of the `idnlgrey` model `sys` as free.

  ```
  for i = 1:Nx
  sys.InitialStates(i).Fixed = false;
  end
  ```

  Similarly, to fix all the initial states to values specified in `sys.InitialStates`, first specify all the states as fixed in the `sys.InitialStates` property of the nonlinear grey-box model.

- `'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 linear models only.
- Vector or Matrix — Initial guess for state values, specified as a numerical column vector of length equal to the number of states. For multi-experiment data, specify a matrix with *Ne* columns, where *Ne* is the number of experiments. Otherwise, use a column vector to specify the same initial conditions for all experiments. Use this option for state-space (`idss` and `idgrey`) and nonlinear models (`idnlarx`, `idnlhw`, and `idnlgrey`) only.
- `initialCondition` object — `initialCondition` object that represents a model of the free response of the system to initial conditions. For multiexperiment data, specify a 1-by-N_e array of objects, where N_e is the number of experiments.

  Use this option for linear models 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 prediction:

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 `[]`. The number of rows must be greater than or equal to the model order.
`Output`	Output history, specified as a matrix with *Ny* columns, where *Ny* is the number of output channels. The number of rows must be greater than or equal to the model order.

  For an example, see “Use Historical Data to Specify Initial Conditions for Model Prediction” on page 1-1271.

  For multi-experiment data, configure the initial conditions separately for each experiment by specifying `InitialCondition` as a structure array with *Ne* elements. To specify the same initial conditions for all experiments, use a single structure.

  The software uses `data2state` to map the historical data to states. If your model is not `idss`, `idgrey`, `idnlgrey`, or `idnlarx`, the software first converts the model to its state-space representation and then maps the data to states. If conversion of your model to `idss` is not possible, the estimated states are returned empty.
- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space (`idss` and `idgrey`) and nonlinear grey-box (`idnlgrey`) models only. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum or maximum bounds.

`InputOffset` — Input signal offset

[ ] (default) | column vector | matrix

Input signal offset for time-domain data, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following values:

- `[]` — No input offsets.
- A column vector of length *Nu*, where *Nu* is the number of inputs. The software subtracts the offset value `InputOffset(i)` from the *i*th input signal before using the input to predict the model response.

- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *Nu*-by-*Ne* matrix, where *Ne* is the number of experiments. The software subtracts the offset value `InputOffset(i,j)` from the *i*th input signal of the *j*th experiment before prediction.

  If you specify a column vector of length *Nu*, then the offset value `InputOffset(i)` is subtracted from the *i*th input signal of all the experiments.

`OutputOffset` — Output signal offset
`[]` (default) | column vector | matrix

Output signal offset for time-domain data, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following values:

- `[]` — No output offsets.
- A column vector of length *Ny*, where *Ny* is the number of outputs. The software subtracts the offset value `OutputOffset(i)` from the *i*th output signal before using the output to predict the model response. After prediction, the software adds the offsets to the predicted response to give the final predicted response.
- *Ny*-by-*Ne* matrix — For multi-experiment data, specify `OutputOffset` as an *Ny*-by-*Ne* matrix, where *Ne* is the number of experiments. The software subtracts the offset value `OutputOffset(i,j)` from the *i*th output signal of the *j*th experiment before prediction.

  If you specify a column vector of length *Ny*, then the offset value `OutputOffset(i)` is subtracted from the *i*th output signal of all the experiments.

  After prediction, the software adds the removed offsets to the predicted response to give the final predicted response.

`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 values:

- `[]` — No weighting is used by the software for initial condition estimation. This option is the same as using `eye(Ny)` for the output weight, where *Ny* is the number of outputs.
- `'noise'` — The software uses the inverse of the `NoiseVariance` property of the model as the weight.
- A positive, semidefinite matrix of dimension *Ny*-by-*Ny*, where *Ny* is the number of outputs.

`OutputWeight` is relevant only for multi-output models.

Output Arguments

opt — Option set for `predict`
`predictOptions` option set

Option set for `predict`, retuned as a `predictOptions` option set.

See Also

`predict` | `absorbDelay` | `idpar`

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.
```

See Also
`getpvec` | `getcov` | `tfdata` | `ssdata` | `polydata` | `frdata` | `idssdata` | `zpkdata`

Topics
"Estimation Report"
"Loss Function and Model Quality Metrics"

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( ___ )
[sys,offset,ic] = 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.

`[sys,offset,ic] = procest( ___ )` returns the estimated initial conditions as an `initialCondition` object. Use this syntax if you plan to simulate or predict the model response using the same estimation input data and then compare the response with the same estimation output data. Incorporating the initial conditions yields a better match during the first part of the simulation.

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 | string | cell array of character vectors | string array

Process model structure, specified for SISO models as a character vector or string 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 MIMO models, use an `Ny`-by-`Nu` cell array of character vectors or string array, with one entry for each input-output pair. Here `Ny` is the number of inputs and `Nu` is the number of outputs.

For information regarding how `type` affects the structure of a process model, see `idproc`.

`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:

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.
`InitialCondition`	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'`.

Report Field	Description	
`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:	
	Field	**Description**
	`FitPercent`	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
	`LossFcn`	Value of the loss function when the estimation completes.
	`MSE`	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
	`FPE`	Final prediction error for the model.
	`AIC`	Raw Akaike Information Criteria (AIC) measure of model quality.
	`AICc`	Small-sample-size corrected AIC.
	`nAIC`	Normalized AIC.
	`BIC`	Bayesian Information Criteria (BIC).
`Parameters`	Estimated values of model parameters.	
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `procestOptions` for more information.	
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.	

Report Field	Description	
`DataUsed`	Attributes of the data used for estimation. Structure with the following fields:	
	Field	**Description**
	`Name`	Name of the data set.
	`Type`	Data type. For `idnlarx` models, this is set to `'Time domain data'`.
	`Length`	Number of data samples.
	`Ts`	Sample time. This is equivalent to `Data.Ts`.
	`InterSample`	Input intersample behavior. One of the following values: • `'zoh'` — 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. The value of `Intersample` has no effect on estimation results for discrete-time models.
	`InputOffset`	Empty, `[]`, for nonlinear estimation methods.
	`OutputOffset`	Empty, `[]`, for nonlinear estimation methods.

Report Field	Description
Termination	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** — **Description** WhyStop — Reason for terminating the numerical search. Iterations — Number of search iterations performed by the estimation algorithm. FirstOrderOptimality — ∞-norm of the gradient search vector when the search algorithm terminates. FcnCount — Number of times the objective function was called. UpdateNorm — Norm of the gradient search vector in the last iteration. Omitted when the search method is 'lsqnonlin' or 'fmincon'. LastImprovement — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is 'lsqnonlin' or 'fmincon'. Algorithm — Algorithm used by 'lsqnonlin' or 'fmincon' search method. Omitted when other search methods are used. For estimation methods that do not require numerical search optimization, the Termination field is omitted.

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.

ic — Initial conditions
initialCondition object | object array of initialCondition values

Estimated initial conditions, returned as an initialCondition object or an object array of initialCondition values.

- For a single-experiment data set, ic represents, in state-space form, the free response of the transfer function model (*A* and *C* matrices) to the estimated initial states (x_0).
- For a multiple-experiment data set with N_e experiments, ic is an object array of length N_e that contains one set of initialCondition values for each experiment.

If procest returns ic values of 0 and the you know that you have non-zero initial conditions, set the 'InitialCondition' option in procestOptions to 'estimate' and pass the updated option set to procest. For example:

```
opt = procestOptions('InitialCondition','estimate')
[sys,offset,ic] = procest(data,np,nz,opt)
```

The default `'auto'` setting of `'InitialCondition'` uses the `'zero'` method when the initial conditions have a negligible effect on the overall estimation-error minimization process. Specifying `'estimate'` ensures that the software estimates values for `ic`.

For more information, see `initialCondition`. For an example of using this argument, see “Obtain Initial Conditions” on page 1-1292.

Examples

Estimate a First Order Plus Dead Time Model

Obtain the measured input-output data.

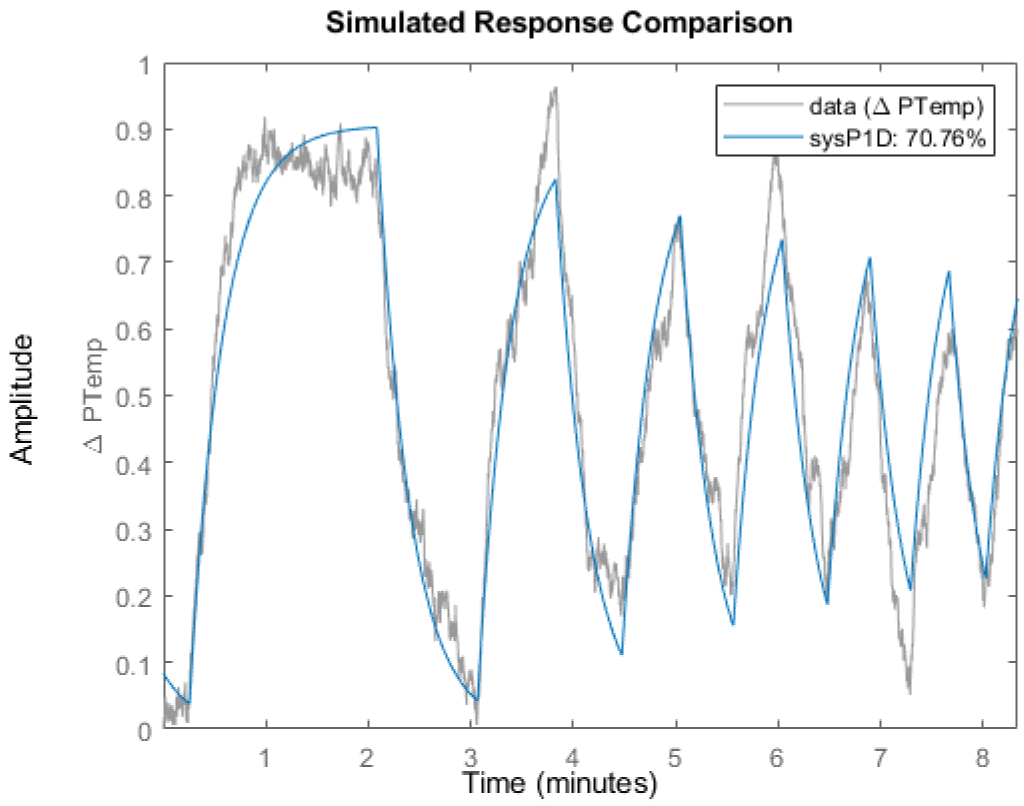
```
load iddemo_heatexchanger_data;
data = iddata(pt,ct,Ts);
data.InputName  = '\Delta CTemp';
data.InputUnit  = 'C';
data.OutputName = '\Delta PTemp';
data.OutputUnit = 'C';
data.TimeUnit   = 'minutes';
```

Estimate a first-order plus dead time process model.

```
type = 'P1D';
sysP1D = procest(data,type);
```

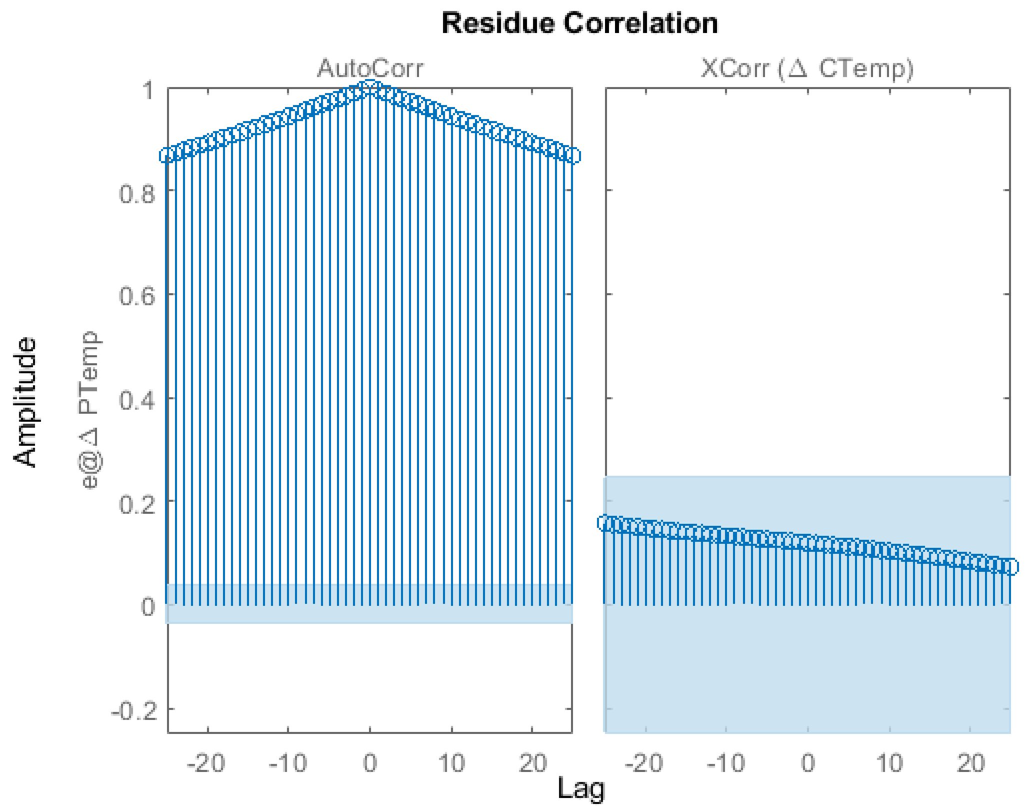
Compare the model with the data.

```
compare(data,sysP1D)
```



Plot the model residuals.

```
figure
resid(data,sysP1D);
```

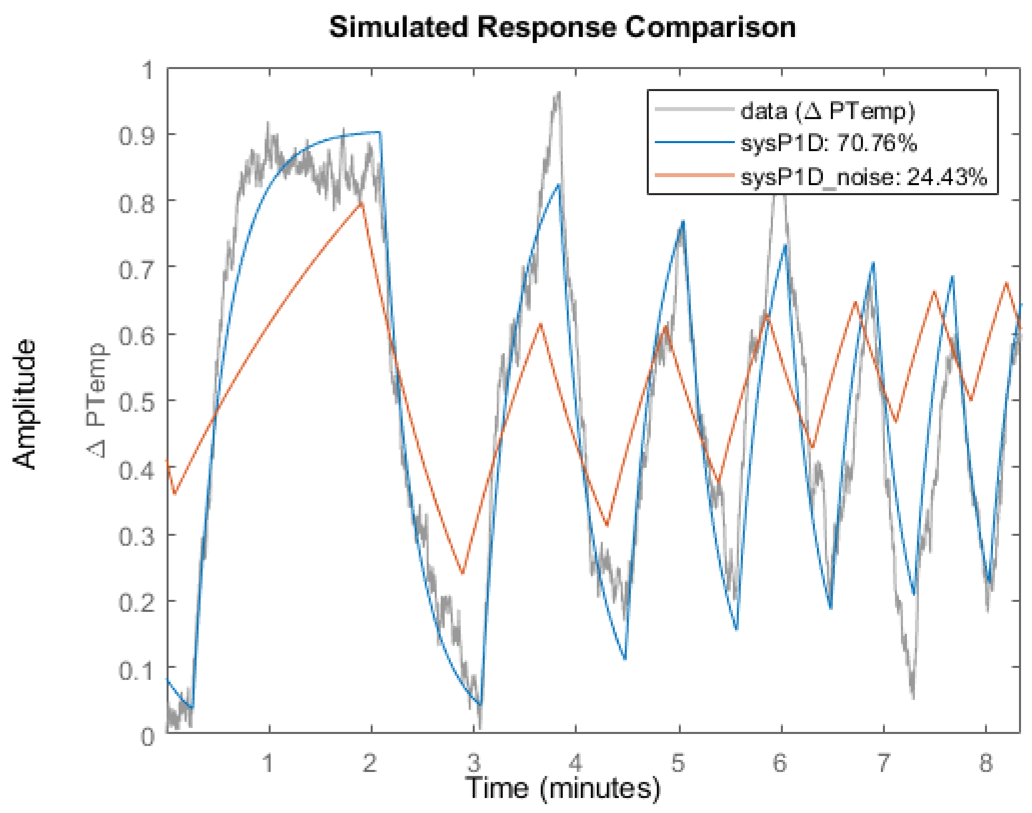


The figure shows that the residuals are correlated. To account for that, add a first order ARMA disturbance component to the process model.

```
opt = processOptions('DisturbanceModel','ARMA1');
sysP1D_noise = procest(data,'p1d',opt);
```

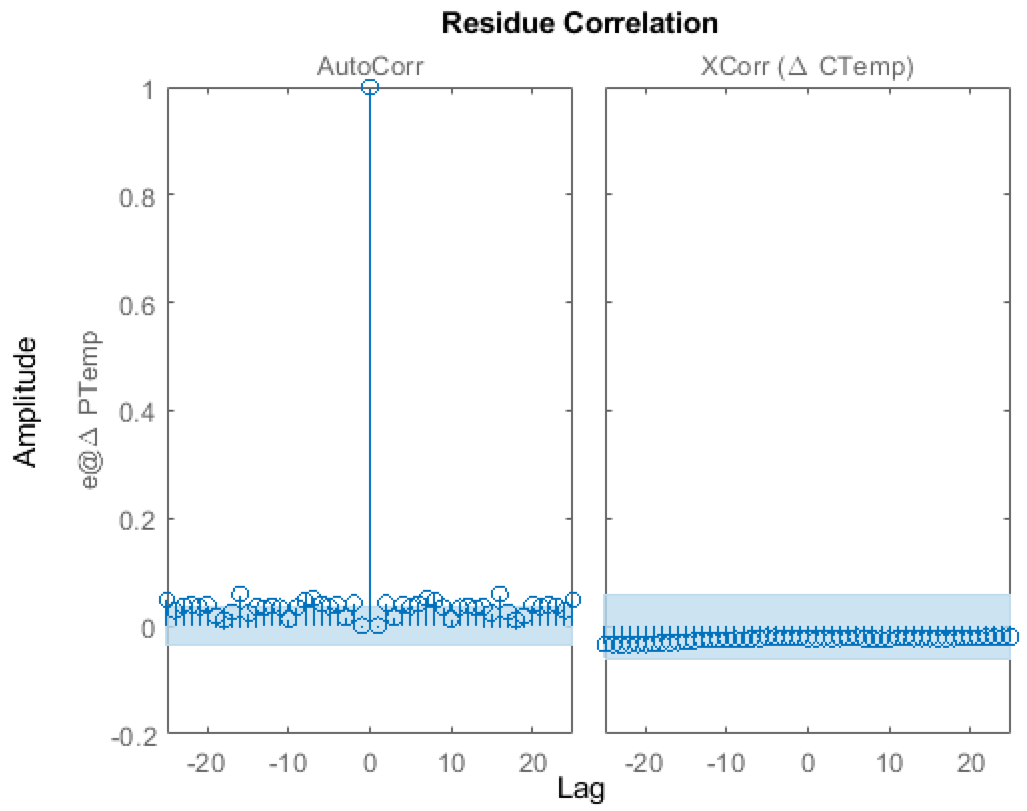
Compare the models.

```
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.

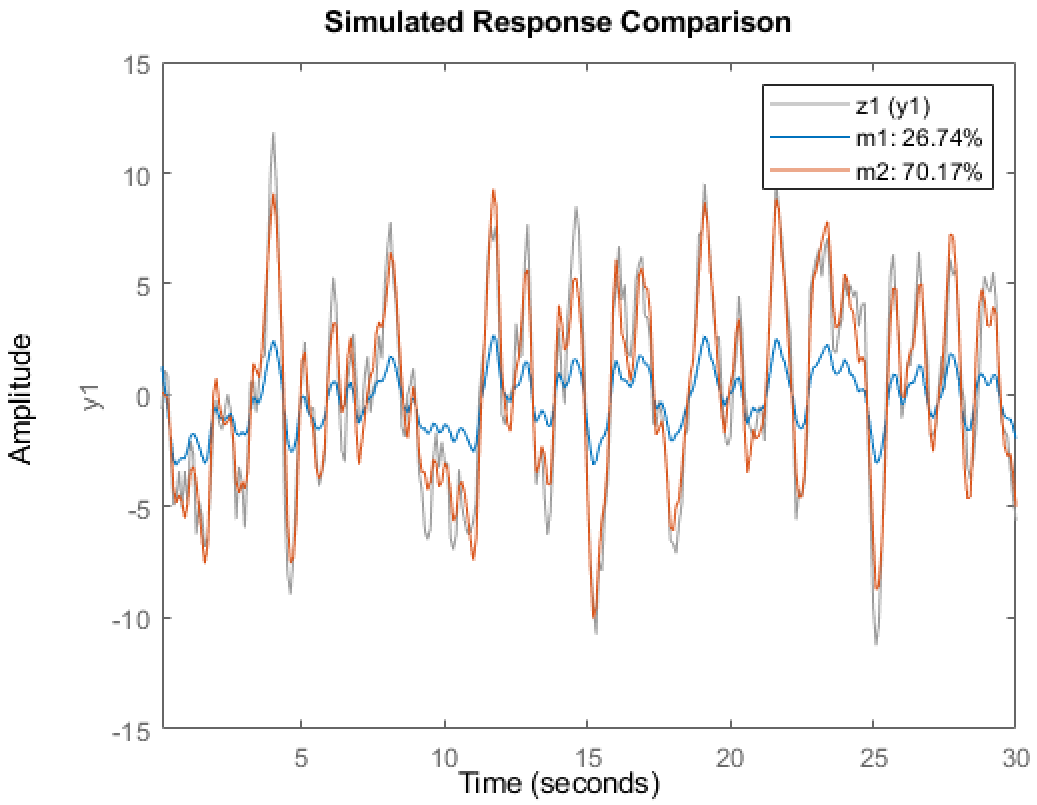
```
m = idproc('P3UZ','K',7.5,'Tw',0.25,'Zeta',0.3,'Tp3',20,'Tz',0.02);
m1 = procest(z1,m);
```

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.SearchOptions.MaxIterations = 100;
```

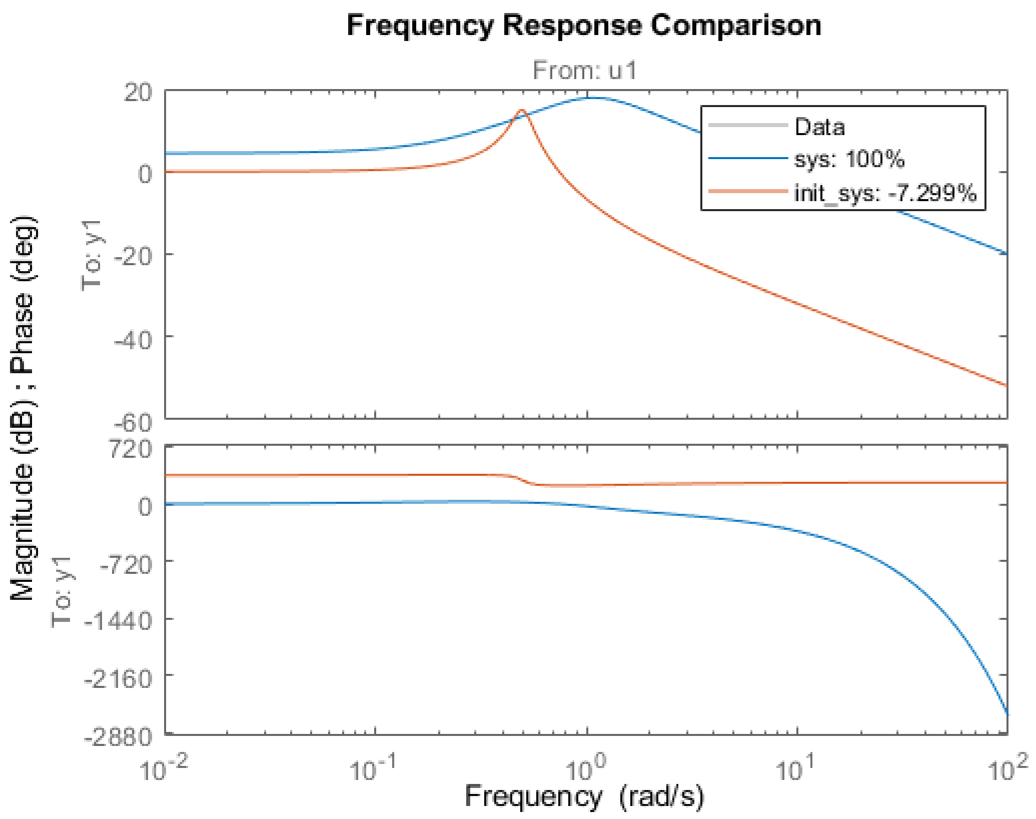
Estimate the process model.

```
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.

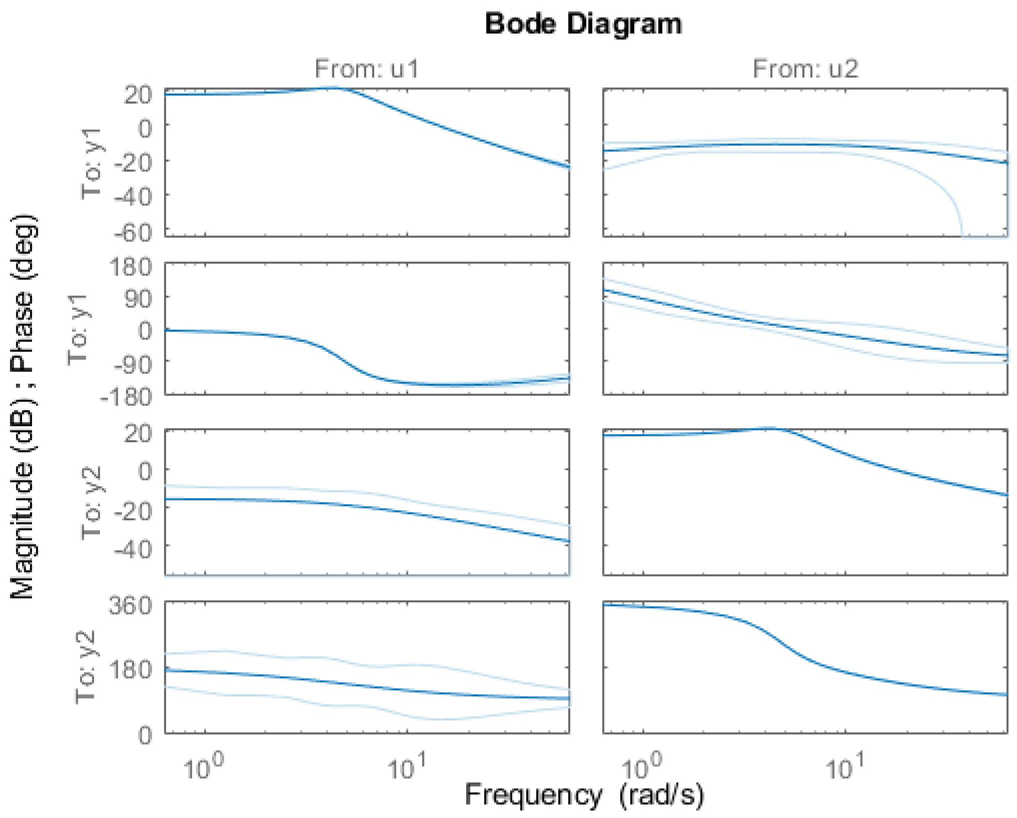
Estimate the process model.

```
type = 'P2UZ';
sys = procest(data,type);
```

The `type` variable denotes a model with complex-conjugate 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
```

Obtain Initial Conditions

Load the data.

```
load iddata1ic z1i
```

Estimate a first-order plus dead time process model `sys` and return the initial conditions in `ic`. First specify `'estimate'` for `'InitialCondition'` to force the software to estimate `ic`. The default `'auto'` setting uses the `'estimate' method` only when the influence of the initial conditions on the overall model error exceed a threshold. When the initial conditions have a negligible effect on the overall estimation-error minimization process, the `'auto`' setting uses `'zero'`.

```
opt = procestOptions('InitialCondition','estimate');
[sys,offset,ic] = procest(z1i,'P1D',opt);
ic
```

```
ic =
  initialCondition with properties:

     A: -3.8997
    X0: -1.0871
     C: 4.5652
    Ts: 0
```

`ic` is an `initialCondition` object that encapsulates the free response of `sys`, in state-space form, to the initial state vector in `X0`. You can incorporate `ic` when you simulate `sys` with the `z1i` input signal and compare the response with the `z1i` output signal.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `procestOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = procestOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also
`procestOptions` | `idproc` | `ssest` | `tfest` | `polyest` | `ar` | `arx` | `oe` | `bj`

Topics
“What Is a Process Model?”
“Regularized Estimates of Model Parameters”
“Apply Initial Conditions when Simulating Identified Linear Models”

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 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.

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

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

If `EstimateCovariance` 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'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOptions` Structure When `SearchMethod` is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Tolerance`	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`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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`

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

SearchOptions Structure When SearchMethod is Specified as 'fmincon'

Field Name	Description	Default
Algorithm	fmincon optimization algorithm, specified as one of the following: • 'sqp' — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from NaN or Inf results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • 'trust-region-reflective' — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • 'interior-point' — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from NaN or Inf results. • 'active-set' — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	'sqp'

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	1e-6
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar.	1e-6
MaxIterations	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance.	100

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

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

  - 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

References

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

[2] Wills, Adrian, B. Ninness, and S. Gibson. “On Gradient-Based Search for Multivariable System Estimates”. *Proceedings of the 16th IFAC World Congress, Prague, Czech Republic, July 3–8, 2005.* Oxford, UK: Elsevier Ltd., 2005.

See Also

`procest` | `idproc` | `idfilt`

Topics

“Loss Function and Model Quality Metrics”

Introduced in R2012a

pzmap

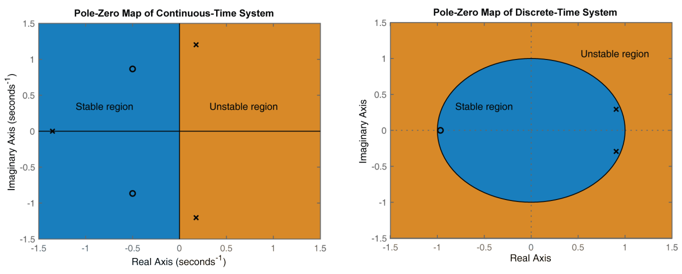
Pole-zero plot of dynamic system

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`. `x` and `o` indicates the poles and zeros respectively, as shown in the following figure.



From the figure above, an open-loop linear time-invariant system is stable if:

- In continuous-time, all the poles on the complex s-plane must be in the left-half plane (blue region) to ensure stability. The system is marginally stable if distinct poles lie on the imaginary axis, that is, the real parts of the poles are zero.
- In discrete-time, all the poles in the complex z-plane must lie inside the unit circle (blue region). The system is marginally stable if it has one or more poles lying on the unit circle.

`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. For SISO systems, `pzmap` plots the system poles and zeros. For MIMO systems, `pzmap` plots the system poles and transmission zeros.

`[p,z] = pzmap(sys)` returns the system poles and transmission zeros as column vectors `p` and `z`. The pole-zero plot is not displayed on the screen.

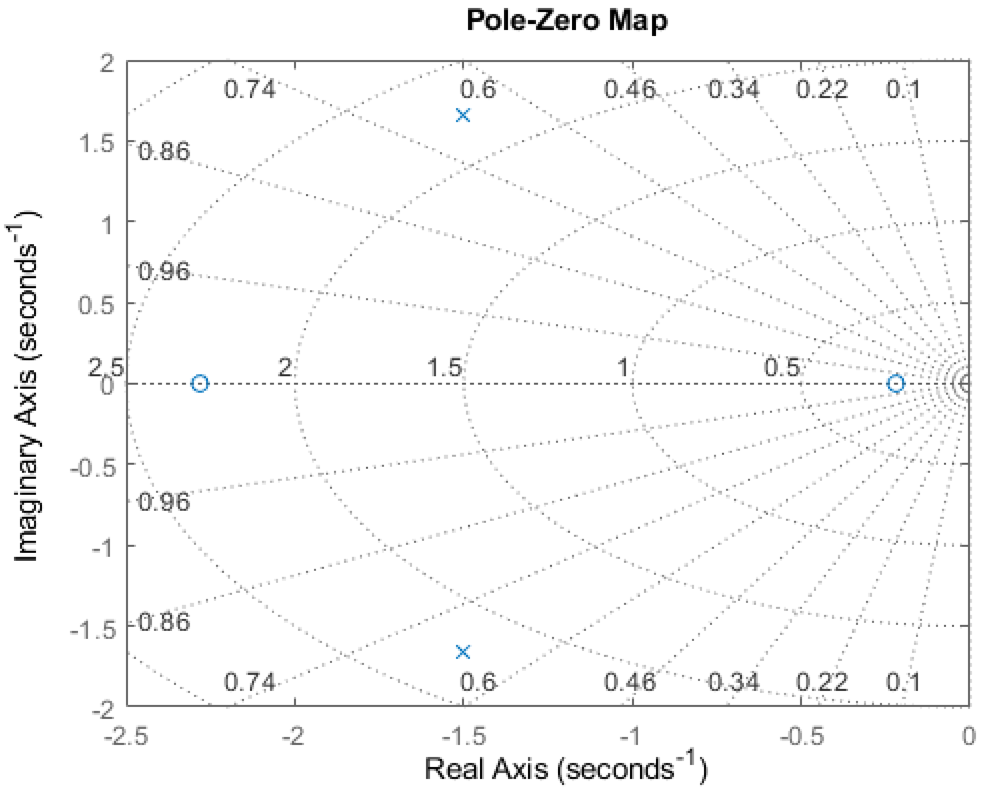
Examples

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.

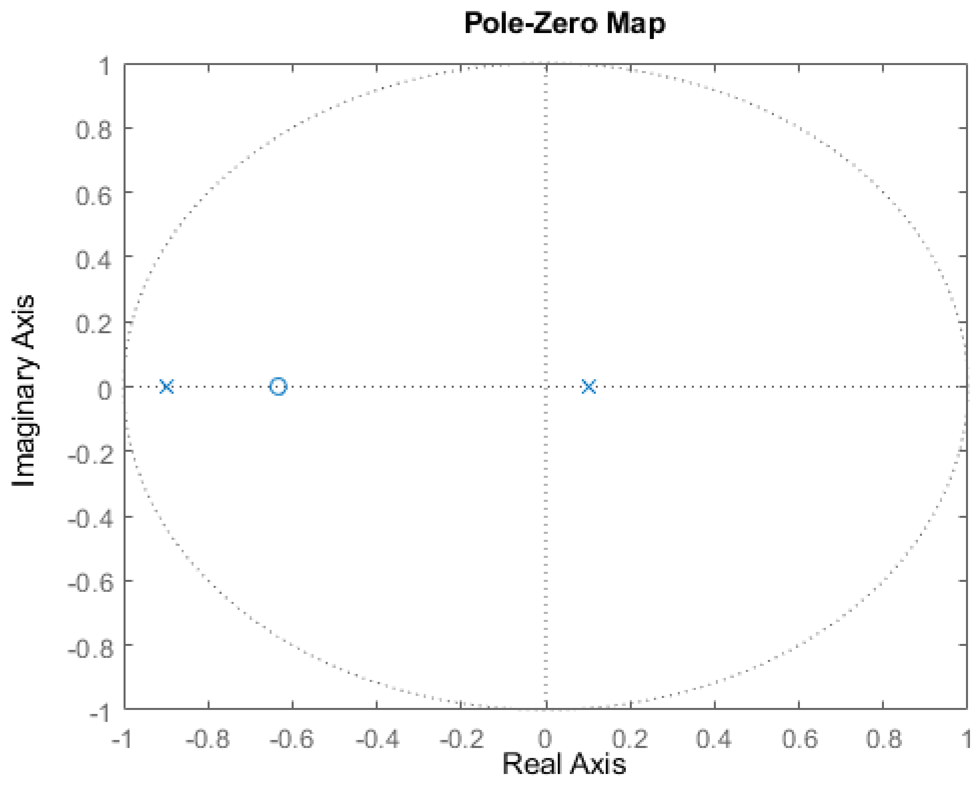
Pole-Zero Plot of Identified System

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.

Pole-Zero Map of Multiple Models

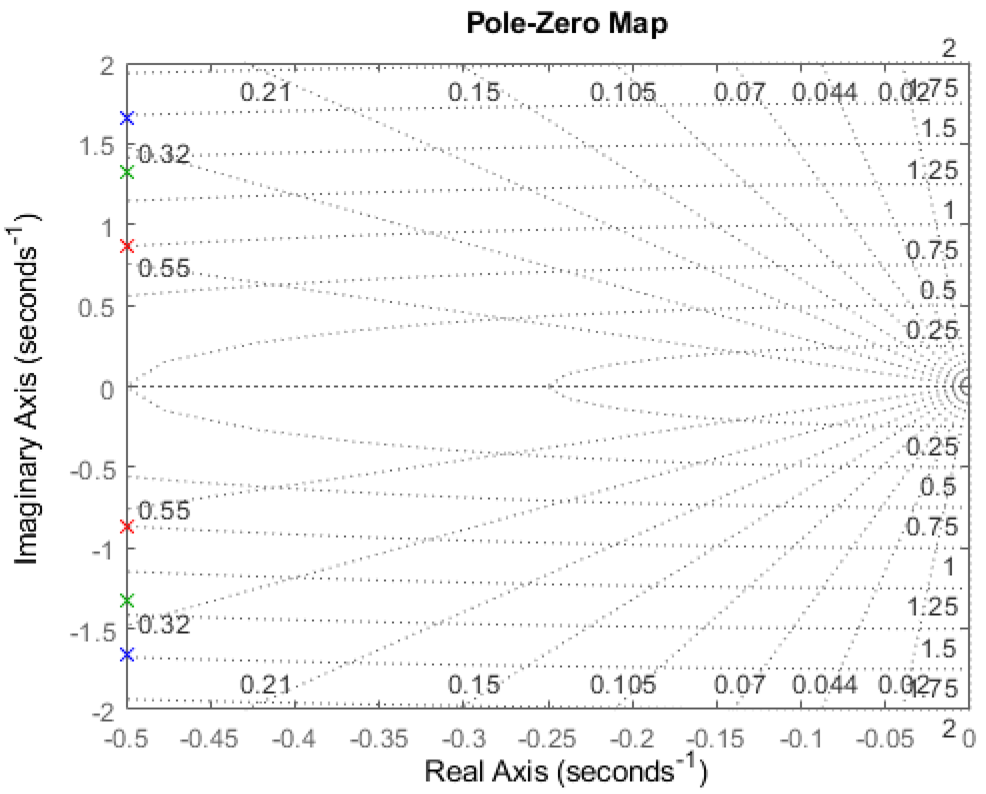
For this example, load a 3-by-1 array of transfer function models.

```
load('tfArray.mat','sys');
size(sys)
```

```
3x1 array of transfer functions.
Each model has 1 outputs and 1 inputs.
```

Plot the poles and zeros of each model in the array with distinct colors. For this example, use red for the first model, green for the second and blue for the third model in the array.

```
pzmap(sys(:,:,1),'r',sys(:,:,2),'g',sys(:,:,3),'b')
sgrid
```



`sgrid` plots lines of constant damping ratio and natural frequency in the s-plane of the pole-zero plot.

Poles and Zeros of Transfer Function

Use `pzmap` to calculate the poles and zeros of the following transfer function:

$$sys(s) = \frac{4.2s^2 + 0.25s - 0.004}{s^2 + 9.6s + 17}$$

```
sys = tf([4.2,0.25,-0.004],[1,9.6,17]);
[p,z] = pzmap(sys)
```

```
p = 2×1

   -7.2576
   -2.3424


z = 2×1

   -0.0726
    0.0131
```

Identify Near-Cancelling Pole-Zero Pairs

This example uses a model of a building with eight floors, each with three degrees of freedom: two displacements and one rotation. The I/O relationship for any one of these displacements is represented as a 48-state model, where each state represents a displacement or its rate of change (velocity).

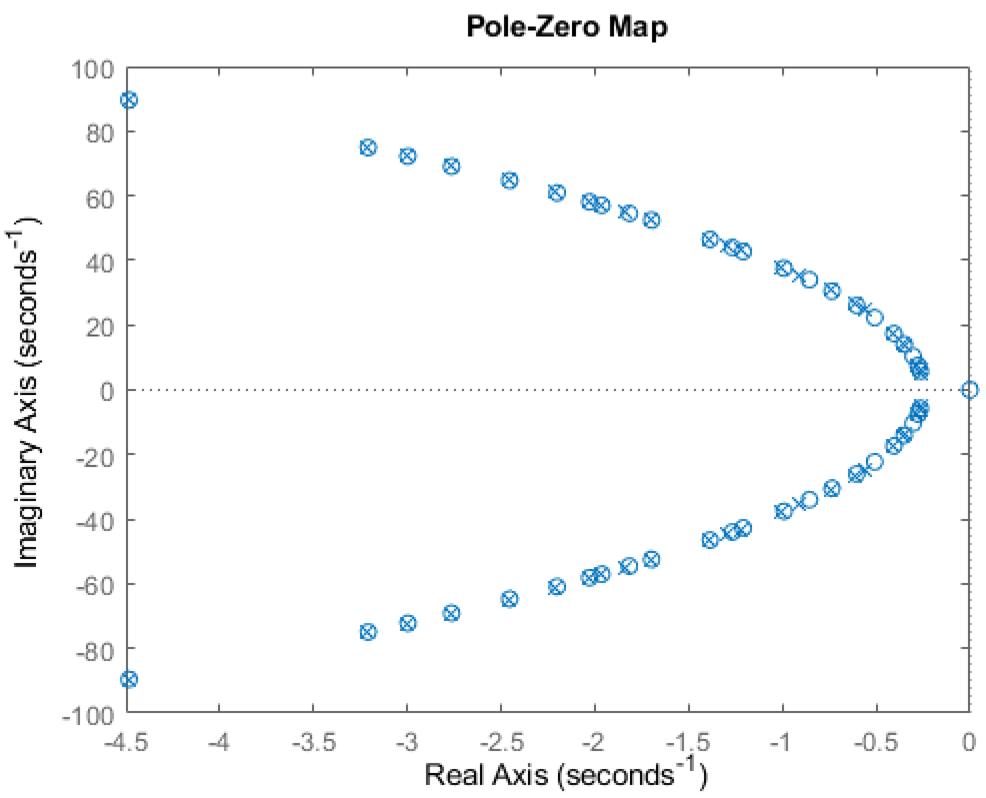
Load the building model.

```
load('building.mat');
size(G)
```

```
State-space model with 1 outputs, 1 inputs, and 48 states.
```

Plot the poles and zeros of the system.

```
pzmap(G)
```



From the plot, observe that there are numerous near-canceling pole-zero pairs that could be potentially eliminated to simplify the model, with no effect on the overall model response. `pzmap` is useful to visually identify such near-canceling pole-zero pairs to perform pole-zero simplification.

Input Arguments

sys — Dynamic system
dynamic system model | model array

Dynamic system, specified as a dynamic system model or model array. Dynamic systems that you can use include continuous-time or discrete-time numeric LTI models such as `tf`, `zpk`, or `ss` models.

If `sys` is an array of models, `pzmap` plots all the poles and zeros of every model in the array on the same plot.

Output Arguments

p — Poles of the system
column vector

Poles of the system, returned as a column vector, in order of its increasing natural frequency. `p` is the same as the output of `pole(sys)`, except for the order.

z — Transmission zeros of the system
column vector

Transmission zeros of the system, returned as a column vector. `z` is the same as the output of `tzero(sys)`.

Tips

- Use the functions `sgrid` or `zgrid` to plot lines of constant damping ratio and natural frequency in the *s*- or *z*-plane on the pole-zero plot.
- For MIMO models, `pzmap` displays 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 to customize the appearance of the pole-zero plot, use `pzplot`.

See Also

`damp` | `esort` | `dsort` | `pole` | `rlocus` | `sgrid` | `zgrid` | `zero` | `iopzmap` | `pzplot`

Introduced before R2006a

pzoptions

Create list of pole/zero plot options

Description

Use the `pzoptions` command to create a `PZMapOptions` object to customize your pole/zero plot appearance. You can also use the command to override the plot preference settings in the MATLAB session in which you create the pole/zero plots.

Creation

Syntax

```
plotoptions = pzoptions
plotoptions = pzoptions('cstprefs')
```

Description

`plotoptions = pzoptions` returns a default set of plot options for use with the `pzplot` and `iopzplot` commands. You can use these options to customize the pole/zero 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.

`plotoptions = pzoptions('cstprefs')` initializes the plot options with the options you selected in the Control System Toolbox and System Identification Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor”. 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.

Properties

`FreqUnits` — Frequency units
'`rad/s`' (default)

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'`

`TimeUnits` — Time units
`'seconds'` (default)

Time units, specified as one of the following values:

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

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 to plot the confidence region
`1` (default) | scalar

Number of standard deviations to use to plot the confidence region, specified as a scalar. This is applicable to identified models only.

`IOGrouping` — Grouping of input-output pairs
`'none'` (default) | `'inputs'` | `'outputs'` | `'all'`

Grouping of input-output (I/O) pairs, specified as one of the following:

- `'none'` — No input-output grouping.
- `'inputs'` — Group only the inputs.
- `'outputs'` — Group only the outputs.
- `'all'` — Group all the I/O pairs.

`InputLabels` — Input label style
structure (default)

Input label style, specified as a structure with the following fields:

- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as `'Normal'` or `'bold'`. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as `'Normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is dark grey with the RGB triplet `[0.4,0.4,0.4]`.
- `Interpreter` — Text interpreter, specified as one of these values:

  - `'tex'` — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - `'latex'` — Interpret characters using LaTeX markup.
  - `'none'` — Display literal characters.

`OutputLabels` — Output label style
structure (default)

Output label style, specified as a structure with the following fields:

- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as `'Normal'` or `'bold'`. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as `'Normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is dark grey with the RGB triplet `[0.4,0.4,0.4]`.
- `Interpreter` — Text interpreter, specified as one of these values:

  - `'tex'` — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - `'latex'` — Interpret characters using LaTeX markup.
  - `'none'` — Display literal characters.

`InputVisible` — Toggle display of inputs
`{'on'}` (default) | `{'off'}` | cell array

Toggle display of inputs, specified as either `{'on'}`, `{'off'}` or a cell array with multiple elements.

`OutputVisible` — Toggle display of outputs
`{'on'}` (default) | `{'off'}` | cell array

Toggle display of outputs, specified as either `{'on'}`, `{'off'}` or a cell array with multiple elements.

`Title` — Title text and style
structure (default)

Title text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a character vector. By default, the plot is titled `'Pole-Zero Map'`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`XLabel` — X-axis label text and style
structure (default)

X-axis label text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a character vector. By default, the axis is titled based on the time units `TimeUnits`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.

  - 'latex' — Interpret characters using LaTeX markup.
  - 'none' — Display literal characters.

YLabel — Y-axis label text and style
structure (default)

Y-axis label text and style, specified as a structure with the following fields:

- String — Label text, specified as a cell array of character vectors. By default, the axis is titled based on the time units TimeUnits.
- FontSize — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch.
- FontWeight — Character thickness, specified as 'Normal' or 'bold'. MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- FontAngle — Character slant, specified as 'Normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- Color — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet [0,0,0].
- Interpreter — Text interpreter, specified as one of these values:

  - 'tex' — Interpret characters using a subset of TeX markup. This is the default value of Interpreter.
  - 'latex' — Interpret characters using LaTeX markup.
  - 'none' — Display literal characters.

TickLabel — Tick label style
structure (default)

Tick label style, specified as a structure with the following fields:

- FontSize — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch.
- FontWeight — Character thickness, specified as 'Normal' or 'bold'. MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- FontAngle — Character slant, specified as 'Normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- Color — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet [0,0,0].

Grid — Toggle grid display
'off' (default) | 'on'

Toggle grid display on the plot, specified as either 'off' or 'on'.

GridColor — Color of the grid lines
[0.15,0.15,0.15] (default) | RGB triplet

Color of the grid lines, specified as an RGB triplet. The default color is light grey specified by the RGB triplet [0.15,0.15,0.15].

XLimMode — X-axis limit selection mode
'auto' (default) | 'manual' | cell array

Selection mode for the x-axis limits, specified as one of these values:

- 'auto' — Enable automatic limit selection, which is based on the total span of the plotted data.
- 'manual' — Manually specify the axis limits. To specify the axis limits, set the XLim property.

YLimMode — Y-axis limit selection mode
'auto' (default) | 'manual' | cell array

Selection mode for the y-axis limits, specified as one of these values:

- 'auto' — Enable automatic limit selection, which is based on the total span of the plotted data.
- 'manual' — Manually specify the axis limits. To specify the axis limits, set the YLim property.

XLim — X-axis limits
'{[1,10]}' (default) | cell array of two-element vector of the form [min,max] | cell array

X-axis limits, specified as a cell array of two-element vector of the form [min,max].

YLim — Y-axis limits
'{[1,10]}' (default) | cell array of two-element vector of the form [min,max] | cell array

Y-axis limits, specified as a cell array of two-element vector of the form [min,max].

Object Functions

iopzplot	Plot pole-zero map for I/O pairs with additional plot customization options
pzplot	Pole-zero plot of dynamic system model with additional plot customization options

Examples

Pole-Zero Plot with Custom Options

Plot the poles and zeros of the continuous-time system represented by the following transfer function with a custom option set:

$$sys(s) = \frac{2s^2 + 5s + 1}{s^2 + 3s + 5}.$$

Create the custom option set using pzoptions.

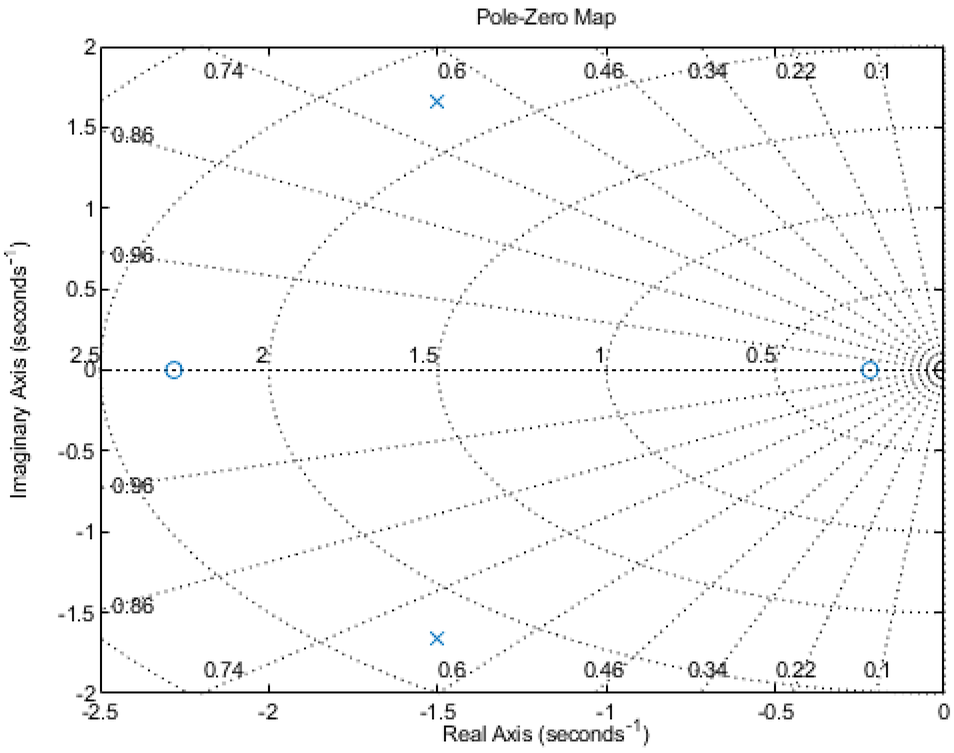
```
plotoptions = pzoptions;
```

For this example, specify the grid to be visible.

```
plotoptions.Grid = 'on';
```

Use the specified options to create a pole-zero map of the transfer function.

```
h = pzplot(tf([2 5 1],[1 3 5]),plotoptions);
```



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`.

See Also

`getoptions` | `iopzplot` | `pzplot` | `setoptions`

Topics

"Toolbox Preferences Editor"

Introduced in R2012a

pzplot

Pole-zero plot of dynamic system model with additional plot customization options

Syntax

```
h = pzplot(sys)
h = pzplot(sys1,sys2,...,sysN)
h = pzplot(sys1,LineSpec1,...,sysN,LineSpecN)
h = pzplot(ax,...)
h = pzplot(...,plotoptions)
```

Description

`pzplot` lets you plot pole-zero maps with a broader range of plot customization options than `pzmap`. You can use `pzplot` to obtain the plot handle and use it to customize the plot, such as modify the axes labels, limits and units. You can also use `pzplot` to draw a pole-zero plot on an existing set of axes represented by an axes handle. To customize an existing plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox). To create pole-zero maps with default options or to extract pole-zero data, use `pzmap`.

`h = pzplot(sys)` plots the poles and transmission zeros of the dynamic system model `sys` and returns the plot handle `h` to the plot. `x` and `o` indicates poles and zeros respectively.

`h = pzplot(sys1,sys2,...,sysN)` displays the poles and transmission zeros of multiple models on a single plot. You can specify distinct colors for each model individually.

`h = pzplot(sys1,LineSpec1,...,sysN,LineSpecN)` sets the line style, marker type, and color for the plot of each system. All systems must have the same number of inputs and outputs to use this syntax.

`h = pzplot(ax,...)` plots into the axes specified by `ax` instead of the current axis `gca`.

`h = pzplot(...,plotoptions)` plots the poles and transmission zeros with the options specified in `plotoptions`. For more information on the ways to change properties of your plots, see “Ways to Customize Plots” (Control System Toolbox).

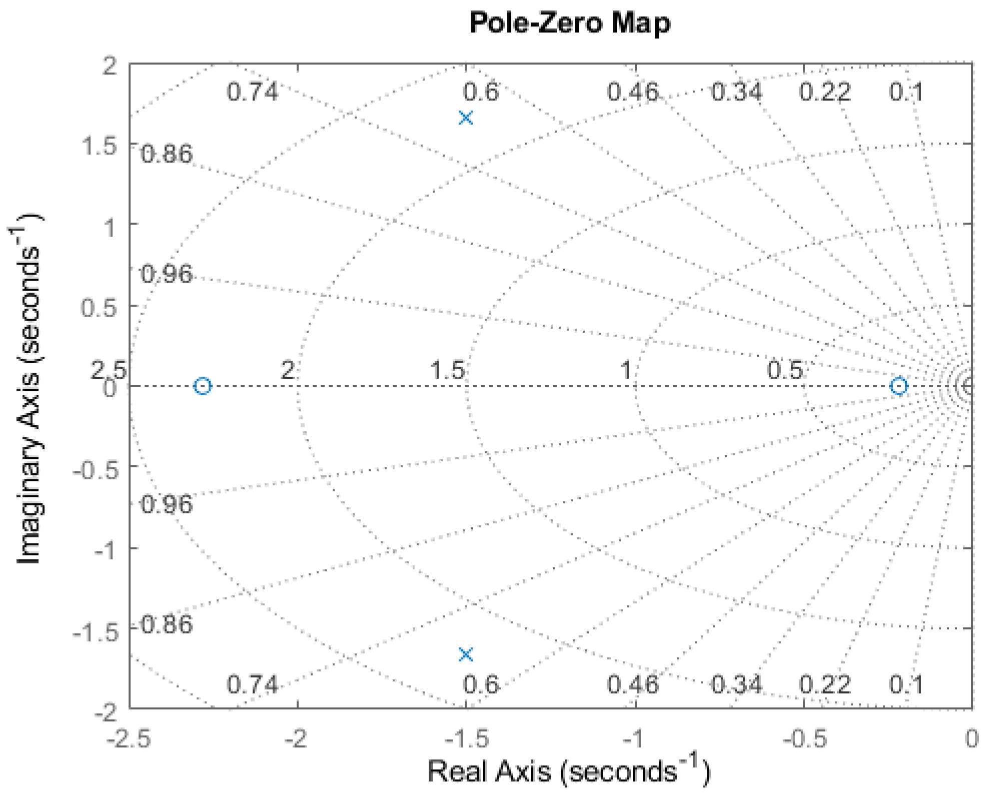
Examples

Pole-Zero Plot with Custom Plot Title

Plot the poles and zeros of the continuous-time system represented by the following transfer function:

$$sys(s) = \frac{2s^2 + 5s + 1}{s^2 + 3s + 5}.$$

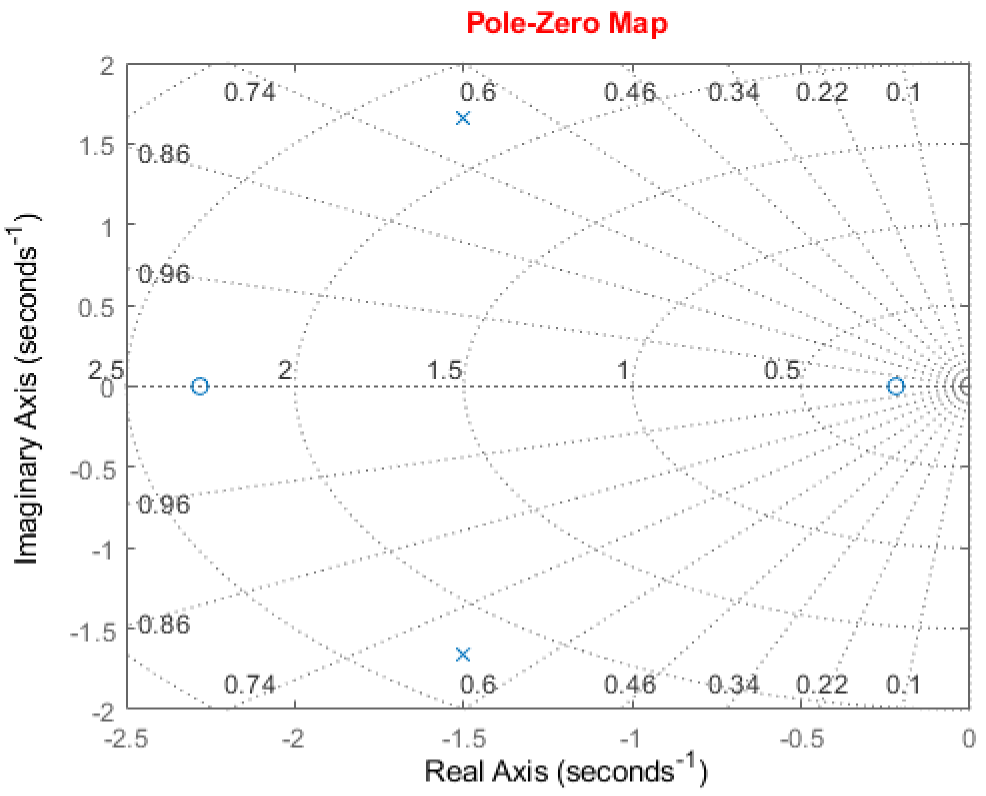
```
sys = tf([2 5 1],[1 3 5]);
h = pzplot(sys);
grid on
```



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);
```



Pole-Zero Plot of Multiple Models

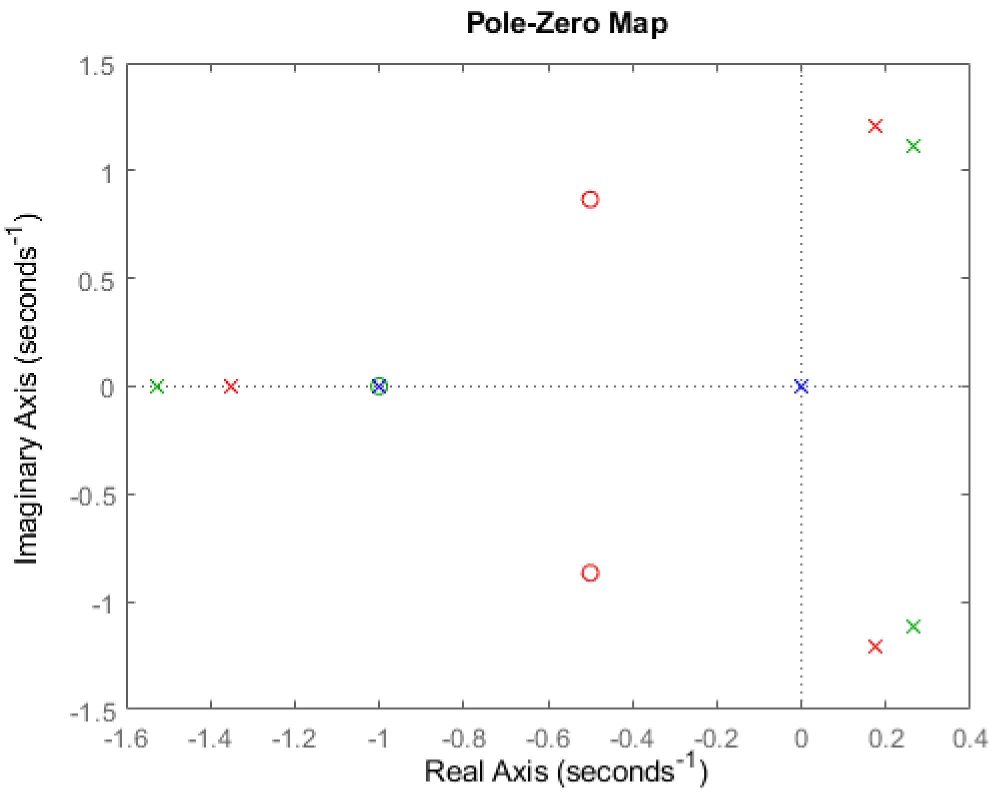
For this example, load a 3-by-1 array of transfer function models.

```
load('tfArrayMargin.mat','sys');
size(sys)
```

```
3x1 array of transfer functions.
Each model has 1 outputs and 1 inputs.
```

Plot the poles and zeros of the model array. Define the colors for each model. For this example, use red for the first model, green for the second and blue for the third model in the array.

```
pzplot(sys(:,:,1),'r',sys(:,:,2),'g',sys(:,:,3),'b');
```



Pole-Zero Plot with Custom Options

Plot the poles and zeros of the continuous-time system represented by the following transfer function with a custom option set:

$$sys(s) = \frac{2s^2 + 5s + 1}{s^2 + 3s + 5}.$$

Create the custom option set using `pzoptions`.

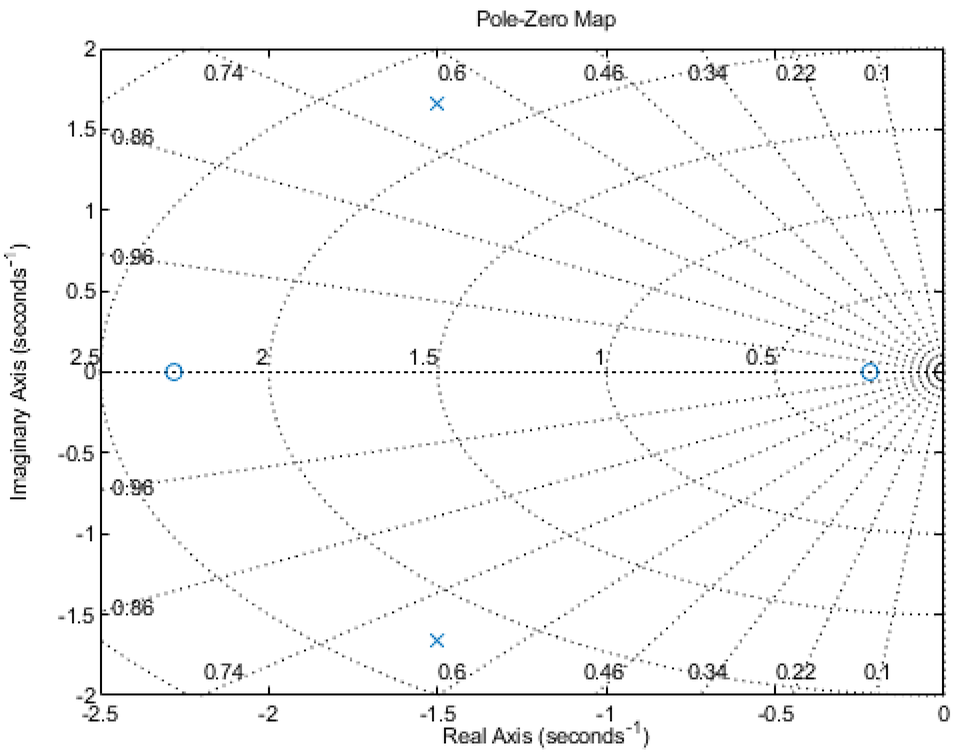
```
plotoptions = pzoptions;
```

For this example, specify the grid to be visible.

```
plotoptions.Grid = 'on';
```

Use the specified options to create a pole-zero map of the transfer function.

```
h = pzplot(tf([2 5 1],[1 3 5]),plotoptions);
```



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`.

Input Arguments

`sys` — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model, or an array of SISO or MIMO dynamic system models. Dynamic systems that you can use include continuous-time or discrete-time numeric LTI models such as `tf`, `zpk`, or `ss` models.

If `sys` is a generalized state-space model `genss` or an uncertain state-space model `uss`, `pzplot` returns the poles and transmission of the current or nominal value of `sys`. If `sys` is an array of models, `pzplot` plots the poles and zeros of each model in the array on the same diagram.

`LineSpec` — Line style, marker, and color

character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: '--or' is a red dashed line with circle markers

Line Style	Description
-	Solid line
--	Dashed line
:	Dotted line
-.	Dash-dot line

Marker	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'_'	Horizontal line
'|'	Vertical line
's'	Square
'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'p'	Pentagram
'h'	Hexagram

Color	Description
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

ax — Axes handle
axes object

Axes handle, specified as an axes object. If you do not specify the axes object, then pzplot uses the current axes gca to plot the poles and zeros of the system.

`plotoptions` — Pole-zero plot options
options object

Pole-zero plot options, specified as an options object. See `pzoptions` for a list of available plot options.

Output Arguments

`h` — Pole-zero plot options handle
scalar

Pole-zero plot options handle, returned as a scalar. Use `h` to query and modify properties of your pole-zero plot. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands.

Tips

- Use `sgrid` or `zgrid` to plot lines of constant damping ratio and natural frequency in the *s*- or *z*-plane.

See Also

`getoptions` | `pzmap` | `setoptions` | `iopzplot` | `pzoptions`

Topics
"Ways to Customize Plots" (Control System Toolbox)

Introduced before R2006a

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,

$$na:\quad A(q) = 1 + a_1q^{-1} + \ldots + a_{na}q^{-na}$$
$$nb:\quad B(q) = b_1 + b_2q^{-1} + \ldots + b_{nb}q^{-nb+1}$$
$$nc:\quad \mathrm{C}(q) = 1 + c_1q^{-1} + \ldots + c_{nc}q^{-nc}$$

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)
```

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.

See Also

`nkshift` | `recursiveARMA` | `recursiveARMAX` | `rpem` | `rplr`

Topics

“Recursive Algorithms for Online Parameter Estimation”

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.

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:

$\varphi(t) = [y(t–1),\dots,y(t–na),u(t–1),\dots,u(t–nb–nk+1)]$

Examples

Adaptive noise canceling: The signal *y* contains a component that originates from a known signal *r*. Remove this component by recursively estimating the system that relates *r* to *y* using a sixth-order FIR model and the NLMS algorithm.

```
z = [y r];
[thm,noise] = rarx(z,[0 6 1],'ng',0.1);
% noise is the adaptive estimate of the noise
% component of y
plot(y-noise)
```

If this is an online application, you can plot the best estimate of the signal `y - 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

ARX Model Structure

The general ARX model structure is:

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

The orders of the ARX model are:

$$na{:}\quad A(q) = 1 + a_1 q^{-1} + \dots + a_{na} q^{-na}$$
$$nb{:}\quad B(q) = b_1 + b_2 q^{-1} + \dots + b_{nb} q^{-nb+1}$$

Models with several inputs are defined, as follows:

$$A(q)y(t) = B_1(q)u_1(t–nk_1)+\dots+B_{\mathrm{nu}}u_{\mathrm{nu}}(t–nk_{\mathrm{nu}})+e(t)$$

See Also

`nkshift` | `recursiveAR` | `recursiveARX` | `rpem` | `rplr`

Topics

“Recursive Algorithms for Online Parameter Estimation”

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,

$$\begin{aligned} nb&: \quad B(q) = b_1 + b_2q^{-1} + \ldots + b_{nb}q^{-nb+1} \\ nc&: \quad C(q) = 1 + c_1q^{-1} + \ldots + c_{nc}q^{-nc} \\ nd&: \quad D(q) = 1 + d_1q^{-1} + \ldots + d_{nd}q^{-nd} \\ nf&: \quad F(q) = 1 + f_1q^{-1} + \ldots + f_{nf}q^{-nf} \end{aligned}$$

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`.

Algorithms

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Recursive Algorithms for Online Parameter Estimation”.

See Also

`nkshift` | `recursiveBJ` | `rpem` | `rplr`

Topics

“Recursive Algorithms for Online Parameter Estimation”

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

Syntax

```
obj = recursiveAR
obj = recursiveAR(na)
obj = recursiveAR(na,A0)
obj = recursiveAR( ___ ,Name,Value)
```

Description

Use the `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`.

`obj = recursiveAR` creates a System object for online parameter estimation of a default single output AR model structure on page 1-1346. 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`).
`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 1-1340.

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 = 1×2

    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');
```

Input Arguments

na — Model order
positive integer

Model order of the polynomial $A(q)$ of an AR model on page 1-1346, 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 parameter values are much smaller than `InitialParameterCovariance`, these initial values are given less importance during estimation. Specify a smaller initial parameter covariance if you have high confidence in the initial parameter values. This statement applies only for infinite-history estimation. Finite-history estimation does not use `InitialParameterCovariance`.

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 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 on page 1-1340 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]

InitialOutputs

Initial values of the outputs buffer in finite-history estimation, specified as 0 or as a (*W*+*na*)-by-1 vector, where *W* is the window length and *na* is the polynomial order you specify during object construction.

The InitialOutputs property provides a means of controlling the initial behavior of the algorithm.

When InitialOutputs is set to 0, the object populates the buffer with zeros.

If the initial buffer is set to 0 or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Specify InitialOutputs only when History is Finite.

InitialOutputs is a tunable property. You can change InitialOutputs when the object is in a locked state.

Default: 0

ParameterCovariance

Estimated covariance P of the parameters, returned as an *N*-by-*N* symmetric positive-definite matrix. *N* is the number of parameters to be estimated. The software computes P assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

ParameterCovariance is applicable only when EstimationMethod is 'ForgettingFactor' or 'KalmanFilter' or when History is Finite.

The interpretation of `P` depends on your settings for the `History` and `EstimationMethod` properties.

- If `History` is `Infinite`, then your `EstimationMethod` selection results in one of the following:
  - `'ForgettingFactor'` — (R_2/2)`P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.
  - `'KalmanFilter'` — R_2`P` is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Here, R_1 is the covariance matrix that you specify in `ProcessNoiseCovariance`.
- If `History` is `Finite` (sliding-window estimation) — R_2`P` is the covariance of the estimated parameters. The sliding-window algorithm does not use this covariance in the parameter-estimation process. However, the algorithm does compute the covariance for output so that you can use it for statistical evaluation.

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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”.

These methods all use an infinite data history, and are available only when `History` is `'Infinite'`.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is, after the object is locked using the `step` command.

Default: `Forgetting Factor`

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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`

`History`

Data history type defining which type of recursive algorithm you use, specified as:

- `'Infinite'` — Use an algorithm that aims to minimize the error between the observed and predicted outputs for all time steps from the beginning of the simulation.
- `'Finite'` — Use an algorithm that aims to minimize the error between the observed and predicted outputs for a finite number of past time steps.

Algorithms with infinite history aim to produce parameter estimates that explain all data since the start of the simulation. These algorithms still use a fixed amount of memory that does not grow over time. The object provides multiple algorithms of the `'Infinite'` `History` type. Specifying this option activates the `EstimationMethod` property with which you specify an algorithm.

Algorithms with finite history aim to produce parameter estimates that explain only a finite number of past data samples. This method is also called sliding-window estimation. The object provides one algorithm of the `'Finite'` type. Specifying this option activates the `WindowLength` property that sizes the window.

For more information on recursive estimation methods, see “Recursive Algorithms for Online Parameter Estimation”.

`History` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'Infinite'`

`WindowLength`

Window size determining the number of time samples to use for the sliding-window estimation method, specified as a positive integer. Specify `WindowLength` only when `History` is `Finite`.

Choose a window size that balances estimation performance with computational and memory burden. Sizing factors include the number and time variance of the parameters in your model. Always specify **Window Length** in samples, even if you are using frame-based input processing.

`WindowLength` must be greater than or equal to the number of estimated parameters.

Suitable window length is independent of whether you are using sample-based or frame-based input processing (see `InputProcessing`). However, when using frame-based processing, your window length must be greater than or equal to the number of samples (time steps) contained in the frame.

`WindowLength` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `200`

InputProcessing

Option for sample-based or frame-based input processing, specified as a character vector or string.

- `Sample-based` processing operates on signals streamed one sample at a time.
- `Frame-based` processing operates on signals containing samples from multiple time steps. Many machine sensor interfaces package multiple samples and transmit these samples together in frames. `Frame-based` processing allows you to input this data directly without having to first unpack it.

Your `InputProcessing` specification impacts the dimensions for the input and output signals when using the `step` command:

```
[theta,EstimatedOutput] = step(obj,y)
```

- `Sample-based`
  - `y` and `EstimatedOutput` are scalars.
- - `Frame-based` with *M* samples per frame
    - `y` and `EstimatedOutput` are *M*-by-1 vectors.

`InputProcessing` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'Sample-based'`

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For Simulink based workflows, use Recursive Polynomial Model Estimator.
- For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Supports MATLAB Function block: No

See Also

`step` | `release` | `reset` | `clone` | `isLocked` | Recursive Polynomial Model Estimator | `ar` | `recursiveARMA` | `recursiveARX` | `recursiveARMAX` | `recursiveBJ` | `recursiveOE` | `recursiveLS`

Topics
“Perform Online Parameter Estimation at the Command Line”
“Validate Online Parameter Estimation at the Command Line”
“What Is Online Estimation?”
“Recursive Algorithms for Online Parameter Estimation”

Introduced in R2015b

recursiveARMA

Create System object for online parameter estimation of ARMA model

Syntax

```
obj = recursiveARMA
obj = recursiveARMA(Orders)
obj = recursiveARMA(Orders,A0,C0)
obj = recursiveARMA( ___ ,Name,Value)
```

Description

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`.

`obj = recursiveARMA` creates a System object for online parameter estimation of a default single output ARMA model structure on page 1-1356. 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. 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 1-1352.

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 = 1×2

    1.0000    0.2315
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans = 2×2
10^-3 ×

    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');
```

Input Arguments

`Orders` — Model orders

1-by-2 vector of integers

Model orders of an ARMA model on page 1-1356, 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 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 on page 1-1352 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 `P` of the parameters, returned as an N-by-N symmetric positive-definite matrix. N is the number of parameters to be estimated. The software computes `P` assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1. `ParameterCovariance` is applicable only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

The interpretation of `P` depends on the estimation method:

- `'ForgettingFactor'` — R_2`/2 * P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.
- `'KalmanFilter'` — R_2`* P` is the covariance matrix of the estimated parameters, and R_1 / R_2 is the covariance matrix of the parameter changes. Where, R_1 is the covariance matrix that you specify in `ProcessNoiseCovariance`.

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements. N is the number of parameters to be estimated.

- Vector of real positive scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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,

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

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

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For Simulink-based workflows, use Recursive Polynomial Model Estimator.
- For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Supports MATLAB Function block: No

See Also

`step` | `release` | `reset` | `clone` | `isLocked` | Recursive Polynomial Model Estimator | `armax` | `recursiveAR` | `recursiveARX` | `recursiveARMAX` | `recursiveBJ` | `recursiveOE` | `recursiveLS`

Topics
“Perform Online Parameter Estimation at the Command Line”
“Validate Online Parameter Estimation at the Command Line”
“What Is Online Estimation?”
“Recursive Algorithms for Online Parameter Estimation”

Introduced in R2015b

recursiveARMAX

Create System object for online parameter estimation of ARMAX model

Syntax

```
obj = recursiveARMAX
obj = recursiveARMAX(Orders)
obj = recursiveARMAX(Orders,A0,B0,C0)
obj = recursiveARMAX( ___ ,Name,Value)
```

Description

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`.

`obj = recursiveARMAX` creates a System object for online parameter estimation of default single-input single-output (SISO) ARMAX model structure on page 1-1367. 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. 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 1-1362.

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 = 1×2

    1.0000   -0.8298
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans = 3×3

    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;
```

The `ProcessNoiseCovariance` property of `obj` is applicable only when the Kalman filter algorithm is used for estimation.

Input Arguments

`Orders` — Model orders and delays

1-by-4 vector of integers

Model orders and delays of an ARMAX model on page 1-1367, 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 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 on page 1-1362 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 `P` of the parameters, returned as an *N*-by-*N* symmetric positive-definite matrix. *N* is the number of parameters to be estimated. The software computes `P` assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1. `ParameterCovariance` is applicable only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

The interpretation of `P` depends on the estimation method:

- `'ForgettingFactor'` — R_2/2 * `P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.
- `'KalmanFilter'` — R_2* `P` is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Where, R_1 is the covariance matrix that you specify in `ProcessNoiseCovariance`.

`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, α — Covariance matrix is an *N*-by-*N* diagonal matrix, with α as the diagonal elements. *N* is the number of parameters to be estimated.
- Vector of real positive scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an *N*-by-*N* diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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 (Autoregressive Moving Average with Extra Input) 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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For Simulink-based workflows, use Recursive Polynomial Model Estimator.
- For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Supports MATLAB Function block: No

See Also
`step` | `release` | `reset` | `clone` | `isLocked` | Recursive Polynomial Model Estimator | `armax` | `recursiveAR` | `recursiveARX` | `recursiveARMA` | `recursiveBJ` | `recursiveOE` | `recursiveLS`

Topics
“Perform Online Parameter Estimation at the Command Line”
“Validate Online Parameter Estimation at the Command Line”
“What Is Online Estimation?”
“Recursive Algorithms for Online Parameter Estimation”

Introduced in R2015b

recursiveARX

Create System object for online parameter estimation of ARX model

Syntax

```
obj = recursiveARX
obj = recursiveARX(Orders)
obj = recursiveARX(Orders,A0,B0)
obj = recursiveARX( ___ ,Name,Value)
```

Description

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`.

`obj = recursiveARX` creates a System object for online parameter estimation of a default ARX model structure on page 1-1380. 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. 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 1-1373.

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 = 1×2

         0    0.7974
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans = 2×2

    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)` initial 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');
```

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 on page 1-1380, 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 *i*th polynomial $B_i(q)+1$ for the *i*th 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 *i*th 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 value 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 value 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 *i*th row of `B0` corresponds to the *i*th input and must contain `nk(i)` leading zeros, followed by `nb(i)` initial parameter values. 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.

If the initial parameter values are much smaller than `InitialParameterCovariance`, these initial values are given less importance during estimation. Specify a smaller initial parameter covariance if you have high confidence in the initial parameter values. This statement applies only for infinite-history estimation. Finite-history estimation does not use `InitialParameterCovariance`.

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 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 on page 1-1373 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 *i*th row of `B` corresponds to the *i*th 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 *i*th row of `B0` corresponds to the *i*th input and must contain `nk(i)` zeros, followed by `nb(i)` initial parameter values. 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]`

InitialOutputs

Initial values of the measured outputs buffer in finite-history estimation, specified as `0` or as a (*W*+*na*)-by-1 vector, where *W* is the window length and `na` is the order of the polynomial *A*(*q*) that you specify when constructing the object.

The `InitialOutputs` property provides a means of controlling the initial behavior of the algorithm.

When `InitialOutputs` is set to `0`, the object populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Specify `InitialOutputs` only when `History` is `Finite`.

`InitialOutputs` is a tunable property. You can change `InitialOutputs` when the object is in a locked state.

Default: `0`

`InitialInputs`

Initial values of the inputs in the finite history window, specified as `0` or as a (*W*-1+max(*nb*)+max(*nk*))-by-*nu* matrix, where *W* is the window length and *nu* is the number of inputs. *nb* is the vector of *B*(*q*) polynomial orders and *nk* is vector of input delays that you specify when constructing the `recursiveARX` object.

The `InitialInputs` property provides a means of controlling the initial behavior of the algorithm.

When the `InitialInputs` is set to `0`, the object populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Specify `InitialInputs` only when `History` is `Finite`.

`InitialInputs` is a tunable property. You can change `InitialInputs` when the object is in a locked state.

Default: `0`

`ParameterCovariance`

Estimated covariance `P` of the parameters, returned as an *N*-by-*N* symmetric positive-definite matrix. *N* is the number of parameters to be estimated. The software computes `P` assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is applicable only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'` or when `History` is `Finite`.

The interpretation of `P` depends on your settings for the `History` and `EstimationMethod` properties.

- If `History` is `Infinite`, then your `EstimationMethod` selection results in one of the following:
  - `'ForgettingFactor'` — (R_2/2)`P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.
  - `'KalmanFilter'` — R_2`P` is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Here, R_1 is the covariance matrix that you specify in `ProcessNoiseCovariance`.
- If `History` is `Finite` (sliding-window estimation) — R_2`P` is the covariance of the estimated parameters. The sliding-window algorithm does not use this covariance in the parameter-estimation process. However, the algorithm does compute the covariance for output so that you can use it for statistical evaluation.

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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”.

These methods all use an infinite data history, and are available only when `History` is `'Infinite'`.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is, after the object is locked using the `step` command.

Default: `Forgetting Factor`

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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`

`History`

Data history type defining which type of recursive algorithm you use, specified as:

- `'Infinite'` — Use an algorithm that aims to minimize the error between the observed and predicted outputs for all time steps from the beginning of the simulation.
- `'Finite'` — Use an algorithm that aims to minimize the error between the observed and predicted outputs for a finite number of past time steps.

Algorithms with infinite history aim to produce parameter estimates that explain all data since the start of the simulation. These algorithms still use a fixed amount of memory that does not grow over time. The object provides multiple algorithms of the `'Infinite'` `History` type. Specifying this option activates the `EstimationMethod` property with which you specify an algorithm.

Algorithms with finite history aim to produce parameter estimates that explain only a finite number of past data samples. This method is also called sliding-window estimation. The object provides one algorithm of the `'Finite'` type. Specifying this option activates the `WindowLength` property that sizes the window.

For more information on recursive estimation methods, see "Recursive Algorithms for Online Parameter Estimation".

`History` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'Infinite'`

`WindowLength`

Window size determining the number of time samples to use for the sliding-window estimation method, specified as a positive integer. Specify `WindowLength` only when `History` is `Finite`.

Choose a window size that balances estimation performance with computational and memory burden. Sizing factors include the number and time variance of the parameters in your model. Always specify **Window Length** in samples, even if you are using frame-based input processing.

`WindowLength` must be greater than or equal to the number of estimated parameters.

Suitable window length is independent of whether you are using sample-based or frame-based input processing (see `InputProcessing`). However, when using frame-based processing, your window length must be greater than or equal to the number of samples (time steps) contained in the frame.

`WindowLength` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `200`

`InputProcessing`

Option for sample-based or frame-based input processing, specified as a character vector or string.

- `Sample-based` processing operates on signals streamed one sample at a time.
- `Frame-based` processing operates on signals containing samples from multiple time steps. Many machine sensor interfaces package multiple samples and transmit these samples together in frames. `Frame-based` processing allows you to input this data directly without having to first unpack it.

Your `InputProcessing` specification impacts the dimensions for the input and output signals when using the `step` command:

```
[theta,EstimatedOutput] = step(obj,y,u)
```

- `Sample-based`

  - `y` and `EstimatedOutput` are scalars.
  - `u` is a 1-by-*Nu* vector, where *Nu* is the number of inputs.
- - `Frame-based` with *M* samples per frame
    - `y` and `EstimatedOutput` are *M*-by-1 vectors.
    - `u` is an *M*-by-`Nu` matrix.

`InputProcessing` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'Sample-based'`

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 :

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

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For Simulink based workflows, use Recursive Polynomial Model Estimator.
- For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Supports MATLAB Function block: No

See Also

`step` | `release` | `reset` | `clone` | `isLocked` | Recursive Polynomial Model Estimator | `arx` | `recursiveAR` | `recursiveARMAX` | `recursiveARMA` | `recursiveBJ` | `recursiveOE` | `recursiveLS`

Topics

“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”
“What Is Online Estimation?”
“Recursive Algorithms for Online Parameter Estimation”

Introduced in R2015b

recursiveBJ

Create System object for online parameter estimation of Box-Jenkins polynomial model

Syntax

```
obj = recursiveBJ
obj = recursiveBJ(Orders)
obj = recursiveBJ(Orders,B0,C0,D0,F0)
obj = recursiveBJ( ___ ,Name,Value)
```

Description

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`.

`obj = recursiveBJ` creates a System object for online parameter estimation of a default single-input single-output (SISO) Box-Jenkins polynomial model structure on page 1-1392. 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. 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 1-1387.

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 = 1×2

    1.0000   -0.6876
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans = 4×4

    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');
```

Input Arguments

`Orders` — Model orders and delays
1-by-5 vector of integers

Model orders and delays of a Box-Jenkins polynomial model on page 1-1392, 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 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 on page 1-1387 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 `P` of the parameters, returned as an N-by-N symmetric positive-definite matrix. N is the number of parameters to be estimated. The software computes `P` assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1. `ParameterCovariance` is applicable only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

The interpretation of `P` depends on the estimation method:

- `'ForgettingFactor'` — R_2/2 * `P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.
- `'KalmanFilter'` — R_2* `P` is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Where, R_1 is the covariance matrix that you specify in `ProcessNoiseCovariance`.

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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:

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

where *nu* is the number of input channels.

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

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$
$$nc: \quad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$
$$nd: \quad D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd}$$
$$nf: \quad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For Simulink-based workflows, use Recursive Polynomial Model Estimator.
- For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Supports MATLAB Function block: No

See Also

`step` | `release` | `reset` | `clone` | `isLocked` | Recursive Polynomial Model Estimator | `bj` | `recursiveAR` | `recursiveARX` | `recursiveARMA` | `recursiveARMAX` | `recursiveOE` | `recursiveLS`

Topics
“Perform Online Parameter Estimation at the Command Line”
“Validate Online Parameter Estimation at the Command Line”
“What Is Online Estimation?”
“Recursive Algorithms for Online Parameter Estimation”

Introduced in R2015b

recursiveLS

Create System object for online parameter estimation using recursive least squares algorithm

Syntax

```
obj = recursiveLS
obj = recursiveLS(Np)
obj = recursiveLS(Np,theta0)
obj = recursiveLS( ___ ,Name,Value)
```

Description

Use the `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, \`.

`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. Alternatively, you can call the object directly. For more information, see “Tips” on page 1-1408.

`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. 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 1-1402.

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
                       History: 'Infinite'
               InputProcessing: 'Sample-based'
                      DataType: 'double'
```

Estimate Parameters of System Using Recursive Least Squares Algorithm

The system has two parameters and is represented as:

$$y(t) = a_1u(t) + a_2u(t-1)$$

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 the recursive least squares algorithm.

```
obj = recursiveLS(2);
```

Load the estimation data, which for this example is a static data set.

```
load iddata3
input = z3.u;
output = z3.y;
```

Create a variable to store `u(t-1)`. This variable is updated at each time step.

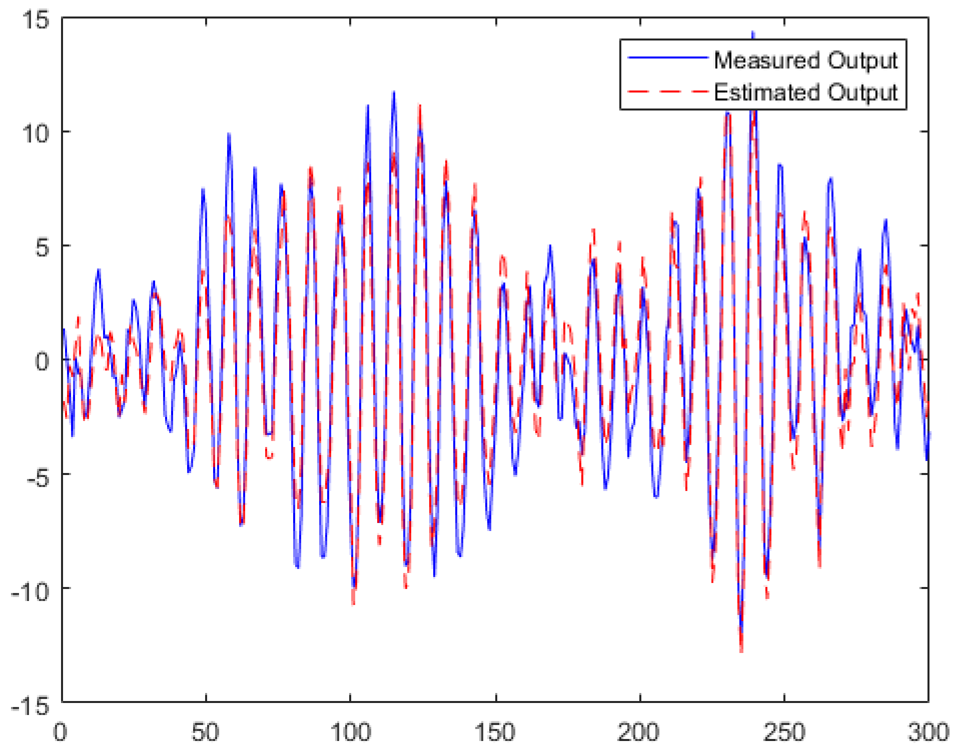
```
oldInput = 0;
```

Estimate the parameters and output using `step` and input-output data, maintaining the current regressor pair in `H`. Invoke the `step` function implicitly by calling the `obj` system object with input arguments.

```
for i = 1:numel(input)
    H = [input(i) oldInput];
    [theta, EstimatedOutput] = obj(output(i),H);
    estimatedOut(i)= EstimatedOutput;
    theta_est(i,:) = theta;
    oldInput = input(i);
end
```

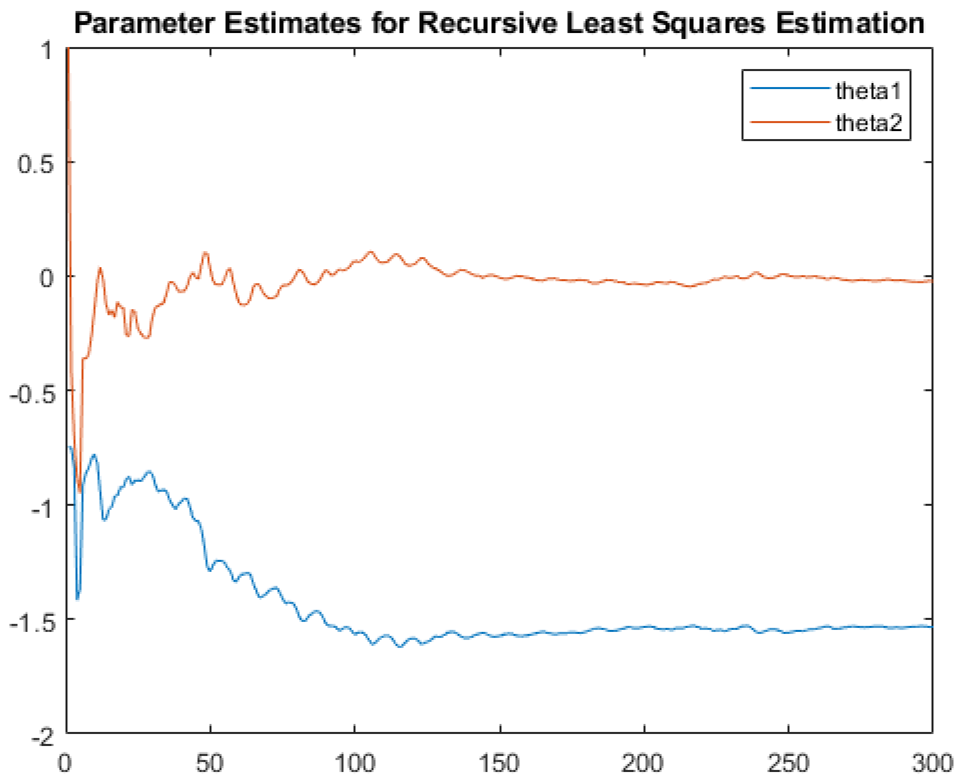
Plot the measured and estimated output data.

```
numSample = 1:numel(input);
plot(numSample,output,'b',numSample,estimatedOut,'r--');
legend('Measured Output','Estimated Output');
```



Plot the parameters.

```
plot(numSample,theta_est(:,1),numSample,theta_est(:,2))
title('Parameter Estimates for Recursive Least Squares Estimation')
legend("theta1","theta2")
```



View the final estimates.

```
theta_final = theta
```

```
theta_final = 2×1

   -1.5322
   -0.0235
```

Use Frame-Based Data for Recursive Least Squares Estimation

Use frame-based signals with the `recursiveLS` command. Machine interfaces often provide sensor data in frames containing multiple samples, rather than in individual samples. The `recursiveLS` object accepts these frames directly when you set `InputProcessing` to `Frame-based`.

The object uses the same estimation algorithms for sample-based and frame-based input processing. The estimation results are identical. There are some special considerations, however, for working with frame-based inputs.

This example is the frame-based version of the sample-based `recursiveLS` example in “Estimate Parameters of System Using Recursive Least Squares Algorithm” on page 1-1396.

The system has two parameters and is represented as:

$$y(t) = a_1 u(t) + a_2 u(t-1)$$

Here,

- u and y are the real-time input and output data, respectively.
- $u(t)$ and $u(t-1)$ are the regressors, `H`, of the system.
- a_1 and a_2 are the parameters,θ, of the system.

Create a System object for online estimation using the recursive least squares algorithm.

```
obj_f = recursiveLS(2,'InputProcessing','Frame-Based');
```

Load the data, which contains input and output time series signals. Each signal consists of 30 frames and each frame contains ten individual time samples.

```
load iddata3_frames input_sig_frame output_sig_frame
input = input_sig_frame.data;
output = output_sig_frame.data;
numframes = size(input,3)
```

```
numframes = 30
```

```
mframe = size(input,1)
```

```
mframe = 10
```

Initialize the regressor frame, which for a given frame, is of the form

$$H_f = \begin{bmatrix} u_1 & u_0 \\ u_2 & u_1 \\ \vdots & \vdots \\ u_{10} & u_9 \end{bmatrix},$$

where the most recent point in the frame is u_{10}.

```
Hframe = zeros(10,2);
```

For this first-order example, the regressor frame includes one point from the previous frame. Initialize this point.

```
oldInput = 0;
```

Estimate the parameters and output using `step` and input-output data, maintaining the current regressor frame in `Hframe`.

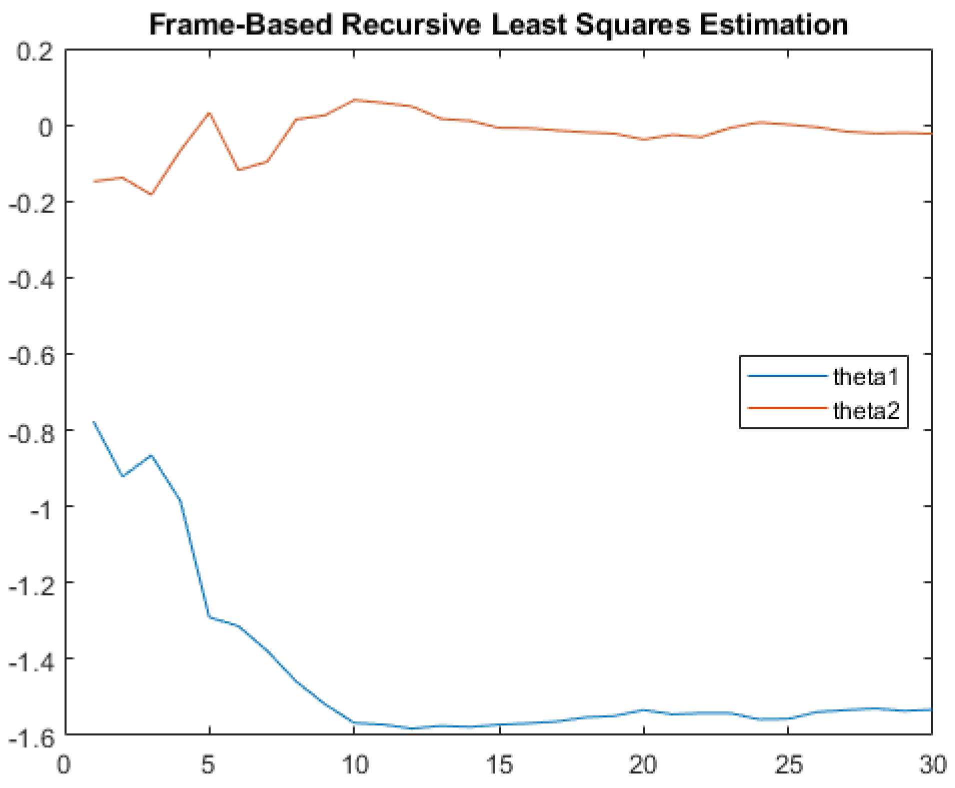
- The input and output arrays have three dimensions. The third dimension is the frame index, and the first two dimensions represent the contents of individual frames.
- Use the `circshift` function to populate the second column of `Hframe` with the past `input` value for each regressor pair by shifting the input vector by one position.
- Populate the `Hframe` element holding the oldest value, `Hframe(1,2),` with the regressor value stored from the previous frame.
- Invoke the `step` function implicitly by calling the `obj` system object with input arguments. The `step` function is compatible with frames, so no loop function within the frame is necessary.

- Save the most recent input value to use for the next frame calculation.

```
EstimatedOutput = zeros(10,1,30);
theta = zeros(2,30);
for i = 1:numframes
    Hframe = [input(:,:,i) circshift(input(:,:,i),1)];
    Hframe(1,2) = oldInput;
    [theta(:,i), EstimatedOutput(:,:,i)] = obj_f(output(:,:,i),Hframe);
    oldInput = input(10,:,i);
end
```

Plot the parameters.

```
theta1 = theta(1,:);
theta2 = theta(2,:);
iframe = 1:numframes;
plot(iframe,theta1,iframe,theta2)
title('Frame-Based Recursive Least Squares Estimation')
legend('theta1','theta2','location','best')
```



View the final estimates.

```
theta_final = theta(:,numframes)
```

```
theta_final = 2×1

   -1.5322
   -0.0235
```

The final estimates are identical to the sample-based estimation.

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.

Input Arguments

`Np` — Number of parameters
positive integer

Number of parameters in the system, specified as a positive integer.

`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)`.

The default initial value for all parameters is `1`.

Note If the initial parameter values are much smaller than `InitialParameterCovariance`, these initial values are given less importance during estimation. Specify a smaller initial parameter covariance if you have high confidence in the initial parameter values. This statement applies only for infinite-history estimation. Finite-history estimation does not use `InitialParameterCovariance`.

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 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 on page 1-1402 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 values 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 parameter values are much smaller than `InitialParameterCovariance`, these initial values are given less importance during estimation. Specify a smaller initial parameter covariance if you have high confidence in initial parameter values. This statement applies only for infinite-history estimation. Finite-history estimation does not use `InitialParameterCovariance`.

`InitialParameters` is a tunable property. You can change `InitialParameters` when the object is in a locked state.

Default: `1`

`InitialOutputs`

Initial values of the outputs buffer in finite-history estimation, specified as `0` or as a *W*-by-1 vector, where *W* is the window length.

The `InitialOutputs` property provides a means of controlling the initial behavior of the algorithm.

When `InitialOutputs` is set to `0`, the object populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Specify `InitialOutputs` only when `History` is `Finite`.

`InitialOutputs` is a tunable property. You can change `InitialOutputs` when the object is in a locked state.

Default: `0`

`InitialRegressors`

Initial values of the regressors buffer in finite-history estimation, specified as `0` or as a *W*-by-`Np` matrix, where *W* is the window length and `Np` is the number of parameters.

The `InitialRegressors` property provides a means of controlling the initial behavior of the algorithm.

When the `InitialRegressors` is set to `0`, the object populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Specify `InitialRegressors` only when `History` is `Finite`.

`InitialRegressors` is a tunable property. You can change `InitialRegressors` when the object is in a locked state.

Default: `0`

`ParameterCovariance`

Estimated covariance `P` of the parameters, returned as an *N*-by-*N* symmetric positive-definite matrix. *N* is the number of parameters to be estimated. The software computes `P` assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is applicable only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'` or when `History` is `Finite`.

The interpretation of `P` depends on your settings for the `History` and `EstimationMethod` properties.

- If `History` is `Infinite`, then your `EstimationMethod` selection results in one of the following:
  - `'ForgettingFactor'` — (R_2`/2`)`P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.
  - `'KalmanFilter'` — R_2`P` is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Here, R_1 is the covariance matrix that you specify in `ProcessNoiseCovariance`.
- If `History` is `Finite` (sliding-window estimation) — R_2`P` is the covariance of the estimated parameters. The sliding-window algorithm does not use this covariance in the parameter-

estimation process. However, the algorithm does compute the covariance for output so that you can use it for statistical evaluation.

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[\alpha_1,\ldots,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,\ldots,\alpha_N]$ 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”.

These methods all use an infinite data history, and are available only when `History` is `'Infinite'`.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is, after the object is locked using the `step` command.

Default: `Forgetting Factor`

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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`

`History`

Data history type defining which type of recursive algorithm you use, specified as:

- `'Infinite'` — Use an algorithm that aims to minimize the error between the observed and predicted outputs for all time steps from the beginning of the simulation.
- `'Finite'` — Use an algorithm that aims to minimize the error between the observed and predicted outputs for a finite number of past time steps.

Algorithms with infinite history aim to produce parameter estimates that explain all data since the start of the simulation. These algorithms still use a fixed amount of memory that does not grow over time. The object provides multiple algorithms of the `'Infinite'` `History` type. Specifying this option activates the `EstimationMethod` property with which you specify an algorithm.

Algorithms with finite history aim to produce parameter estimates that explain only a finite number of past data samples. This method is also called sliding-window estimation. The object provides one algorithm of the `'Finite'` type. Specifying this option activates the `WindowLength` property that sizes the window.

For more information on recursive estimation methods, see “Recursive Algorithms for Online Parameter Estimation”.

`History` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'Infinite'`

`WindowLength`

Window size determining the number of time samples to use for the sliding-window estimation method, specified as a positive integer. Specify `WindowLength` only when `History` is `Finite`.

Choose a window size that balances estimation performance with computational and memory burden. Sizing factors include the number and time variance of the parameters in your model. Always specify **Window Length** in samples, even if you are using frame-based input processing.

`WindowLength` must be greater than or equal to the number of estimated parameters.

Suitable window length is independent of whether you are using sample-based or frame-based input processing (see `InputProcessing`). However, when using frame-based processing, your window length must be greater than or equal to the number of samples (time steps) contained in the frame.

`WindowLength` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `200`

`InputProcessing`

Option for sample-based or frame-based input processing, specified as a character vector or string.

- `Sample-based` processing operates on signals streamed one sample at a time.
- `Frame-based` processing operates on signals containing samples from multiple time steps. Many machine sensor interfaces package multiple samples and transmit these samples together in frames. `Frame-based` processing allows you to input this data directly without having to first unpack it.

Your `InputProcessing` specification impacts the dimensions for the input and output signals when using the `step` command:

```
[theta,EstimatedOutput] = step(obj,y,H)
```

- `Sample-based`
  - `y` and `EstimatedOutput` are scalars.
  - `H` is a 1-by-`Np` vector, where `Np` is the number of parameters.
- • `Frame-based` with *M* samples per frame
  - `y` and `EstimatedOutput` are *M*-by-1 vectors.
  - `H` is an *M*-by-`Np` matrix.

`InputProcessing` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'Sample-based'`

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.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For Simulink based workflows, use Recursive Least Squares Estimator.
- For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Supports MATLAB Function block: No

See Also

`step` | `release` | `reset` | `clone` | `isLocked` | Recursive Least Squares Estimator | `mldivide` | `recursiveAR` | `recursiveARX` | `recursiveARMA` | `recursiveBJ` | `recursiveOE` | `recursiveARMAX`

Topics
“Perform Online Parameter Estimation at the Command Line”

“Validate Online Parameter Estimation at the Command Line”
“Line Fitting with Online Recursive Least Squares Estimation”
“What Is Online Estimation?”
“Recursive Algorithms for Online Parameter Estimation”

Introduced in R2015b

recursiveOE

Create System object for online parameter estimation of Output-Error polynomial model

Syntax

```
obj = recursiveOE
obj = recursiveOE(Orders)
obj = recursiveOE(Orders,B0,F0)
obj = recursiveOE( ___ ,Name,Value)
```

Description

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`.

`obj = recursiveOE` creates a System object for online parameter estimation of a default single-input-single output (SISO) Output-Error model structure on page 1-1421. 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. 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 1-1414.

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 = 1×2

    1.0000   -0.7618
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans = 2×2

    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');
```

Input Arguments

`Orders` — Model orders and delays

1-by-3 vector of integers

Model orders and delays of a Output-Error polynomial model on page 1-1421, 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 parameter values are much smaller than `InitialParameterCovariance`, these initial values are given less importance during estimation. Specify a smaller initial parameter covariance if you have high confidence in the initial parameter values. This statement applies only for infinite-history estimation. Finite-history estimation does not use `InitialParameterCovariance`.

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 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 on page 1-1414 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]

InitialOutputs

Initial values of the measured outputs buffer in finite-history estimation, specified as 0 or as a (*W*+*nf*)-by-1 vector, where *W* is the window length and nf is the order of the polynomial *F*(*q*) that you specify when constructing the object.

The InitialOutputs property provides a means of controlling the initial behavior of the algorithm.

When InitialOutputs is set to 0, the object populates the buffer with zeros.

If the initial buffer is set to 0 or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Specify InitialOutputs only when History is Finite.

`InitialOutputs` is a tunable property. You can change `InitialOutputs` when the object is in a locked state.

Default: `0`

`InitialInputs`

Initial values of the inputs in the finite history window, specified as `0` or as a (*W*+*nb*+*nk*-1)-by-1 vector, where *W* is the window length. *nb* is the vector of *B*(*q*) polynomial orders and *nk* is vector of input delays that you specify when constructing the `recursiveOE` object.

The `InitialInputs` property provides a means of controlling the initial behavior of the algorithm.

When the `InitialInputs` is set to `0`, the object populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Specify `InitialInputs` only when `History` is `Finite`.

`InitialInputs` is a tunable property. You can change `InitialInputs` when the object is in a locked state.

Default: `0`

`ParameterCovariance`

Estimated covariance `P` of the parameters, returned as an *N*-by-*N* symmetric positive-definite matrix. *N* is the number of parameters to be estimated. The software computes `P` assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is applicable only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'` or when `History` is `Finite`.

The interpretation of `P` depends on your settings for the `History` and `EstimationMethod` properties.

- If `History` is `Infinite`, then your `EstimationMethod` selection results in one of the following:
  - `'ForgettingFactor'` — (R_2/2)`P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.
  - `'KalmanFilter'` — R_2`P` is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Here, R_1 is the covariance matrix that you specify in `ProcessNoiseCovariance`.
- If `History` is `Finite` (sliding-window estimation) — R_2`P` is the covariance of the estimated parameters. The sliding-window algorithm does not use this covariance in the parameter-estimation process. However, the algorithm does compute the covariance for output so that you can use it for statistical evaluation.

`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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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”.

These methods all use an infinite data history, and are available only when `History` is `'Infinite'`.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is, after the object is locked using the `step` command.

Default: `Forgetting Factor`

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, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ 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`

`History`

Data history type defining which type of recursive algorithm you use, specified as:

- `'Infinite'` — Use an algorithm that aims to minimize the error between the observed and predicted outputs for all time steps from the beginning of the simulation.
- `'Finite'` — Use an algorithm that aims to minimize the error between the observed and predicted outputs for a finite number of past time steps.

Algorithms with infinite history aim to produce parameter estimates that explain all data since the start of the simulation. These algorithms still use a fixed amount of memory that does not grow over time. The object provides multiple algorithms of the `'Infinite'` `History` type. Specifying this option activates the `EstimationMethod` property with which you specify an algorithm.

Algorithms with finite history aim to produce parameter estimates that explain only a finite number of past data samples. This method is also called sliding-window estimation. The object provides one algorithm of the `'Finite'` type. Specifying this option activates the `WindowLength` property that sizes the window.

For more information on recursive estimation methods, see “Recursive Algorithms for Online Parameter Estimation”.

`History` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'Infinite'`

`WindowLength`

Window size determining the number of time samples to use for the sliding-window estimation method, specified as a positive integer. Specify `WindowLength` only when `History` is `Finite`.

Choose a window size that balances estimation performance with computational and memory burden. Sizing factors include the number and time variance of the parameters in your model. Always specify **Window Length** in samples, even if you are using frame-based input processing.

`WindowLength` must be greater than or equal to the number of estimated parameters.

Suitable window length is independent of whether you are using sample-based or frame-based input processing (see `InputProcessing`). However, when using frame-based processing, your window length must be greater than or equal to the number of samples (time steps) contained in the frame.

`WindowLength` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `200`

`InputProcessing`

Option for sample-based or frame-based input processing, specified as a character vector or string.

- `Sample-based` processing operates on signals streamed one sample at a time.
- `Frame-based` processing operates on signals containing samples from multiple time steps. Many machine sensor interfaces package multiple samples and transmit these samples together in frames. `Frame-based` processing allows you to input this data directly without having to first unpack it.

Your `InputProcessing` specification impacts the dimensions for the input and output signals when using the `step` command:

```
[theta,EstimatedOutput] = step(obj,y,u)
```

- `Sample-based`

  - `y`,`u`, and `EstimatedOutput` are scalars.
- • `Frame-based` with M samples per frame
  - `y`,`u`, and `EstimatedOutput` are M-by-1 vectors.

`InputProcessing` is a nontunable property. It can be set only during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'Sample-based'`

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:

$$nb\colon \quad B(q) = b_1 + b_2q^{-1} + \ldots + b_{nb}q^{-nb+1}$$
$$nf\colon \quad F(q) = 1 + f_1q^{-1} + \ldots + f_{nf}q^{-nf}$$

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,F,EstimatedOutput] = step(obj,y,u)` and `[B,F,EstimatedOutput] = obj(y,u)` perform equivalent operations.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For Simulink-based workflows, use Recursive Polynomial Model Estimator.

- For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Supports MATLAB Function block: No

See Also

`step` | `release` | `reset` | `clone` | `isLocked` | Recursive Polynomial Model Estimator | `oe` | `recursiveAR` | `recursiveARX` | `recursiveARMA` | `recursiveARMAX` | `recursiveBJ` | `recursiveLS`

Topics

“Perform Online Parameter Estimation at the Command Line”
“Validate Online Parameter Estimation at the Command Line”
“What Is Online Estimation?”
“Recursive Algorithms for Online Parameter Estimation”

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
```

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`

See Also

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

Topics
“Perform Online Parameter Estimation at the Command Line”
“Validate Online Parameter Estimation at the Command Line”
“What Is Online Estimation?”

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 SISO Transfer Function to Create MIMO Transfer Function

Create a single-input single-output (SISO) transfer function.

```
sys = tf(2,[1 3])
```

```
sys =

    2
  -----
  s + 3

Continuous-time transfer function.
```

Replicate the SISO transfer function to create a MIMO transfer function that has three inputs and two outputs.

```
rsys = repsys(sys,[2 3])
```

```
rsys =

  From input 1 to output...
          2
   1:  -----
       s + 3

          2
   2:  -----
       s + 3

  From input 2 to output...
          2
   1:  -----
       s + 3

          2
   2:  -----
       s + 3

  From input 3 to output...
          2
   1:  -----
       s + 3

          2
   2:  -----
       s + 3

Continuous-time transfer function.
```

Alternatively, you can obtain the MIMO transfer function as follows:

```
rsys = [sys sys sys; sys sys sys];
```

Replicate SISO Transfer Function to Create Array of Transfer Functions

Create a SISO transfer function.

```
sys = tf(2,[1 3]);
```

Replicate the transfer function into a 3-by-4 array of two-input, one-output transfer functions.

```
rsys = repsys(sys,[1 2 3 4]);
```

Check the size of `rsys`.

```
size(rsys)
```

```
3x4 array of transfer functions.
Each model has 1 outputs and 2 inputs.
```

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.

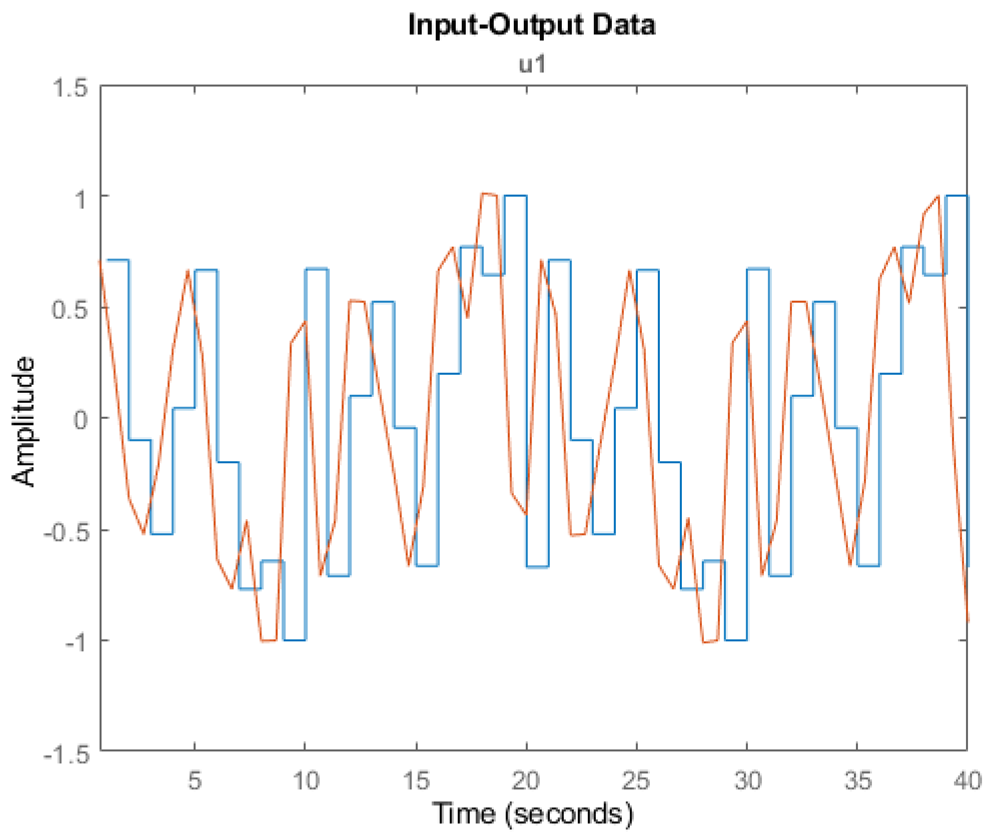
Default: `10`

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)
```



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: [2x2 double]
   InitialParameterCovariance: [2x2 double]
             EstimationMethod: 'ForgettingFactor'
            ForgettingFactor: 1
             EnableAdaptation: true
```

```
                    History: 'Infinite'
            InputProcessing: 'Sample-based'
                   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: [2x2 double]
 InitialParameterCovariance: [2x2 double]
           EstimationMethod: 'ForgettingFactor'
           ForgettingFactor: 1
           EnableAdaptation: true
                    History: 'Infinite'
            InputProcessing: 'Sample-based'
                   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:

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`

Online Estimation System Object	Estimated Parameters
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.

See Also
step | release | clone | isLocked | recursiveAR | recursiveARX | recursiveARMA | recursiveARMAX | recursiveBJ | recursiveOE | recursiveLS

Topics
"Perform Online Parameter Estimation at the Command Line"
"Validate Online Parameter Estimation at the Command Line"
"What Is Online Estimation?"

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 Shape of Model Array

Generate a 2-by-3 array of SISO models with four states each.

```
sys = rss(4,1,1,2,3);
size(sys)
```

```
2x3 array of state-space models.
Each model has 1 outputs, 1 inputs, and 4 states.
```

Change the shape of the array to create a 6-by-1 model array.

```
sys1 = reshape(sys,6,1);
size(sys1)
```

```
6x1 array of state-space models.
Each model has 1 outputs, 1 inputs, 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 autocorrelation 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 1-1442.

`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.

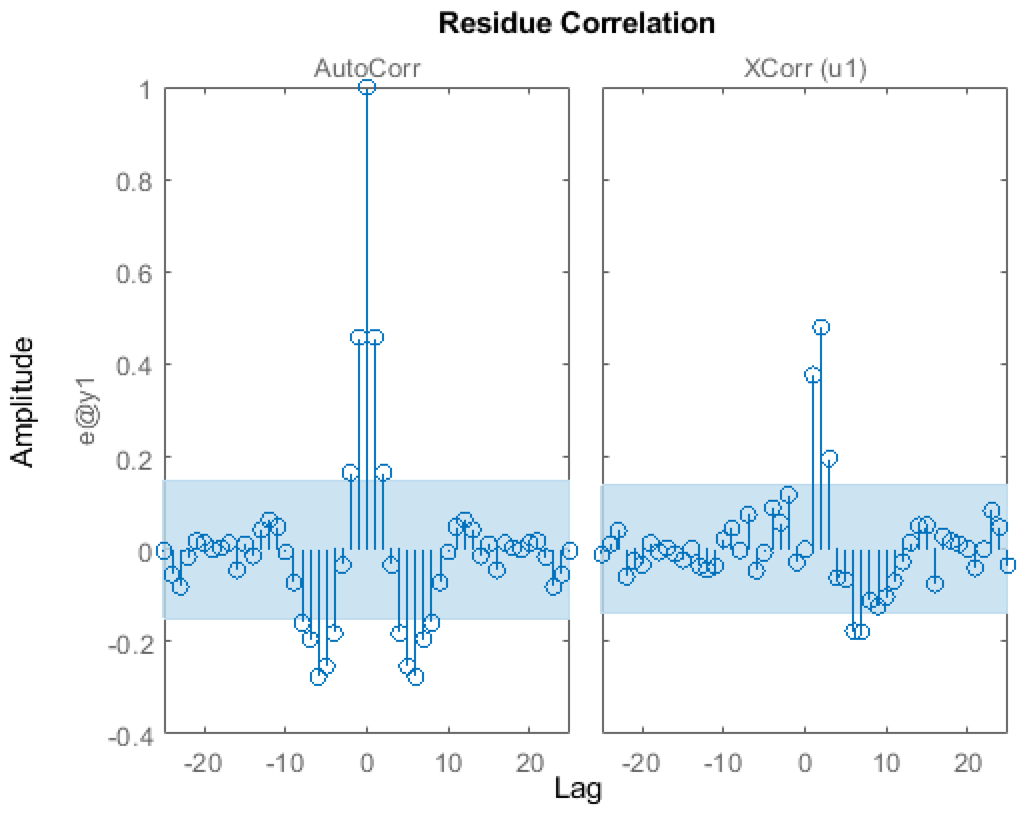
```
load iddata1
data = z1;
```

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)
```



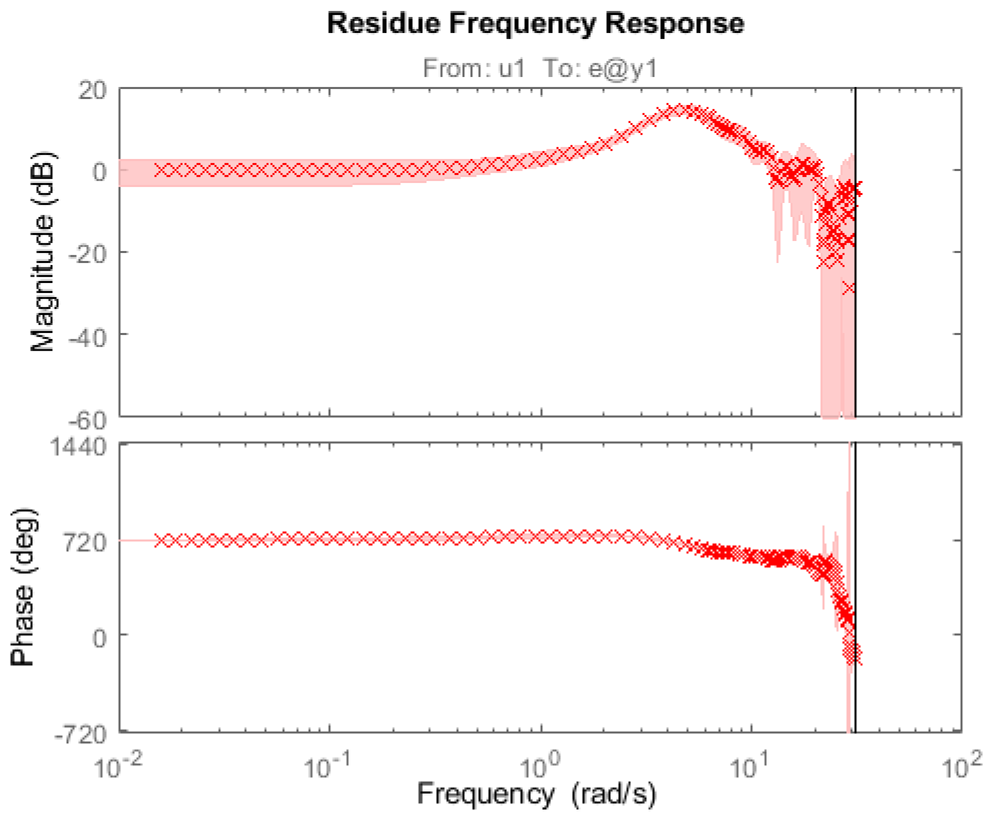
The correlations are calculated until the default maximum lag, 25. The 99% confidence region marking statistically insignificant correlations displays as a shaded region around the X-axis.

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.

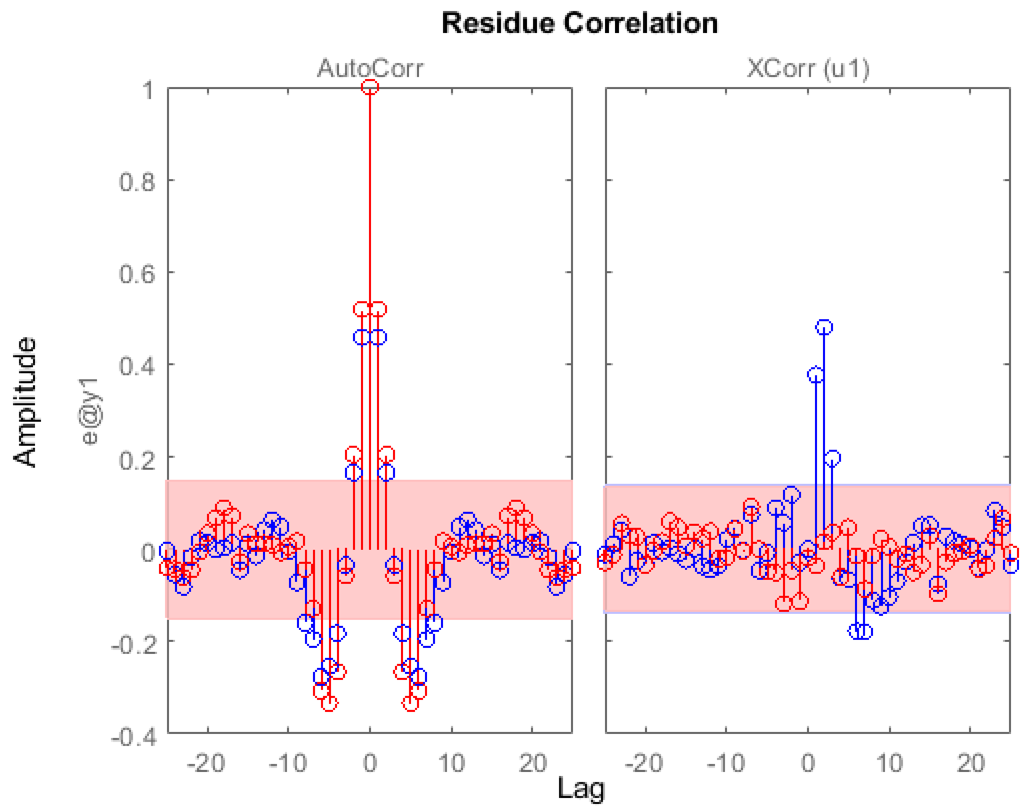
```
sys1 = arx(z1,[1 1 0]);
```

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.

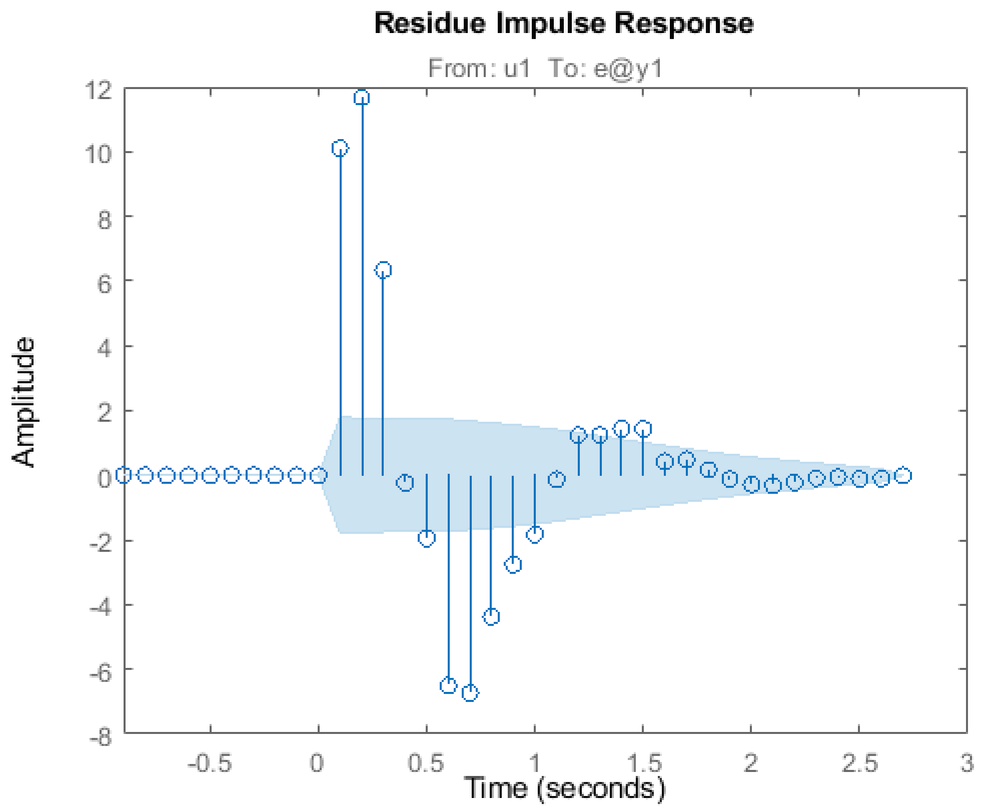
```
sys = arx(z1,[1 1 0]);
```

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);
```

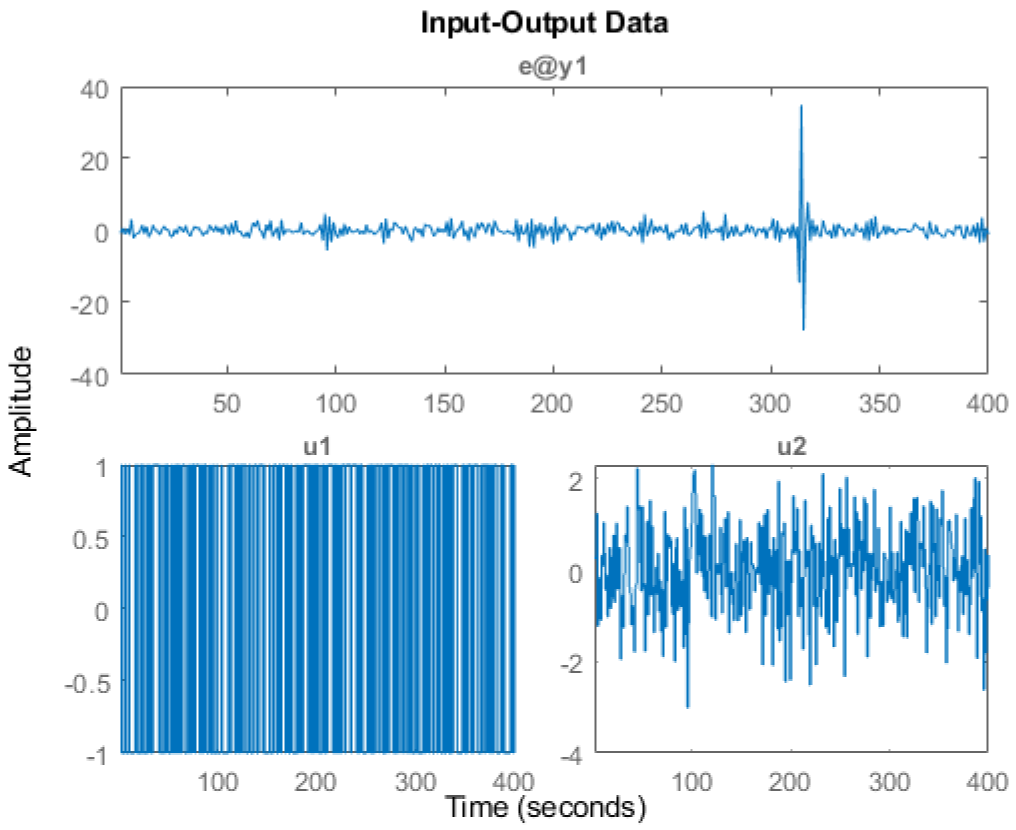
R is a 26-by-3-by-3 matrix of correlations. For example,

- `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.

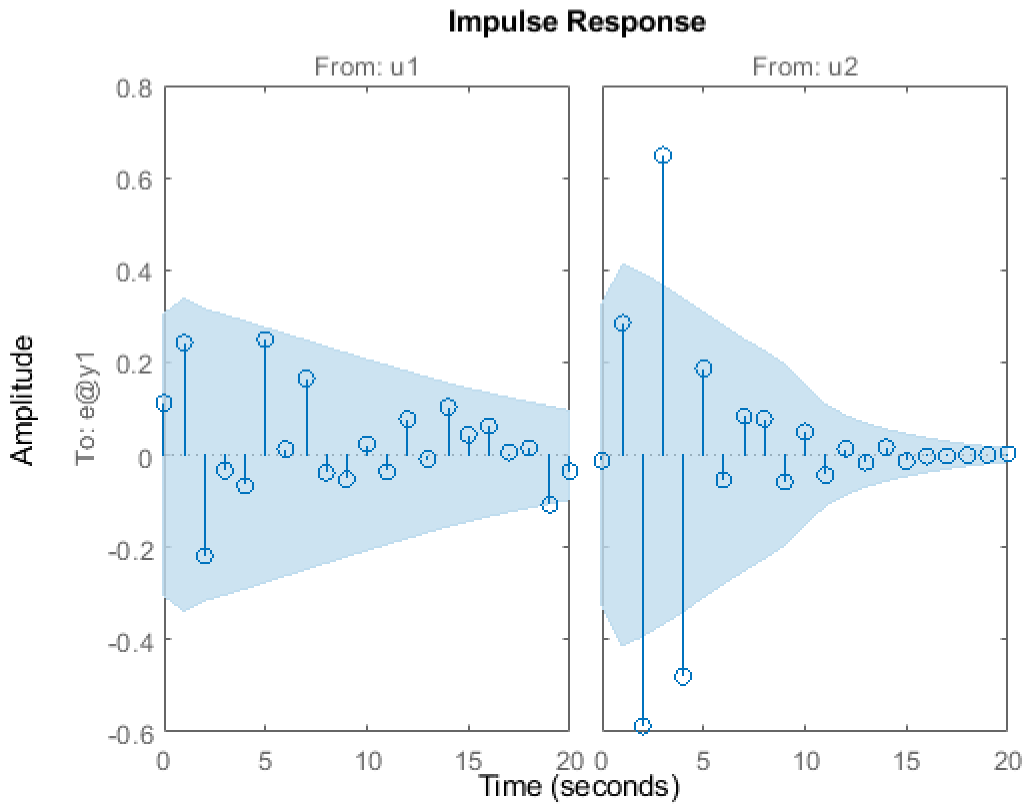
Plot the error.

```
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`. 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

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.

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` | `sim` | `simsd` | `residOptions`

Topics

"What Is Residual Analysis?"

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 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' | column vector | matrix | initialCondition object | object array | structure | idpar object x0Obj

Handling of initial conditions, specified as the comma-separated pair consisting of 'InitialCondition' and one of the following values:

- 'z' — Zero initial conditions.
- 'e' — Estimate initial conditions such that the prediction error for observed output is minimized.

  For nonlinear grey-box models, only those initial states i that are designated as free in the model (sys.InitialStates(i).Fixed = false) are estimated. To estimate all the states of the model, first specify all the Nx states of the idnlgrey model sys as free.

  ```
  for i = 1:Nx
  sys.InitialStates(i).Fixed = false;
  end
  ```

  Similarly, to fix all the initial states to values specified in sys.InitialStates, first specify all the states as fixed in the sys.InitialStates property of the nonlinear grey-box model.
- '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 linear models only.
- Vector or Matrix — Initial guess for state values, specified as a numerical column vector of length equal to the number of states. For multi-experiment data, specify a matrix with *Ne* columns, where *Ne* is the number of experiments. Otherwise, use a column vector to specify the same initial conditions for all experiments. Use this option for state-space (idss and idgrey) and nonlinear models (idnlarx, idnlhw, and idnlgrey) only.
- initialCondition object — initialCondition object that represents a model of the free response of the system to initial conditions. For multiexperiment data, specify a 1-by-N_e array of objects, where N_e is the number of experiments.

  Use this option for linear models 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 by resid:

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 []. The number of rows must be greater than or equal to the model order.
Output	Output history, specified as a matrix with *Ny* columns, where *Ny* is the number of output channels. The number of rows must be greater than or equal to the model order.

  For multi-experiment data, configure the initial conditions separately for each experiment by specifying InitialCondition as a structure array with *Ne* elements. To specify the same initial conditions for all experiments, use a single structure.

The software uses `data2state` to map the historical data to states. If your model is not `idss`, `idgrey`, `idnlgrey`, or `idnlarx`, the software first converts the model to its state-space representation and then maps the data to states. If conversion of your model to `idss` is not possible, the estimated states are returned empty.

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

`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

residual

Return measurement residual and residual covariance when using extended or unscented Kalman filter

Syntax

```
[Residual,ResidualCovariance] = residual(obj,y)
[Residual,ResidualCovariance] = residual(obj,y,Um1,...,Umn)
```

Description

The `residual` command returns the difference between the actual and predicted measurements for `extendedKalmanFilter` and `unscentedKalmanFilter` objects. Viewing the residual provides a way for you to validate the performance of the filter. Residuals, also known as innovations, quantify the prediction error and drive the correction step in the extended and unscented Kalman filter update sequence. When using `correct` and `predict` to update the estimated Kalman filter state, use the `residual` command immediately before using the `correct` command.

`[Residual,ResidualCovariance] = residual(obj,y)` returns the residual `Residual` between a measurement `y` and a predicted measurement produced by the Kalman filter `obj`. The function also returns the covariance of the residual `ResidualCovariance`.

You create `obj` using the `extendedKalmanFilter` or `unscentedKalmanFilter` commands. You specify the state transition function *f* and measurement function *h* of your nonlinear system in `obj`. The `State` property of the object stores the latest estimated state value. At each time step, you use `correct` and `predict` together to update the state *x*. The residual *s* is the difference between the actual and predicted measurements for the time step, and is expressed as $s = y - h(x)$. The covariance of the residual *S* is the sum $R + R_P$, where *R* is the measurement noise matrix set by the `MeasurementNoise` property of the filter and R_P is the state covariance matrix projected onto the measurement space.

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

Here, `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.

`[Residual,ResidualCovariance] = residual(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

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 the 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`, based on 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;
```

Initialize arrays to capture results of the estimation.

```
residBuf = [];
xcorBuf = [];
xpredBuf = [];
```

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 the 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. Compute the residual between the predicted and actual measurement to assess how well the filter is performing and converging. Computing the residual is an optional step. When you use `residual`, place the command immediately before the `correct` command. If the prediction matches the measurement, the residual is zero.

After you perform the real-time commands for the time step, buffer the results so that you can plot them after the run is complete.

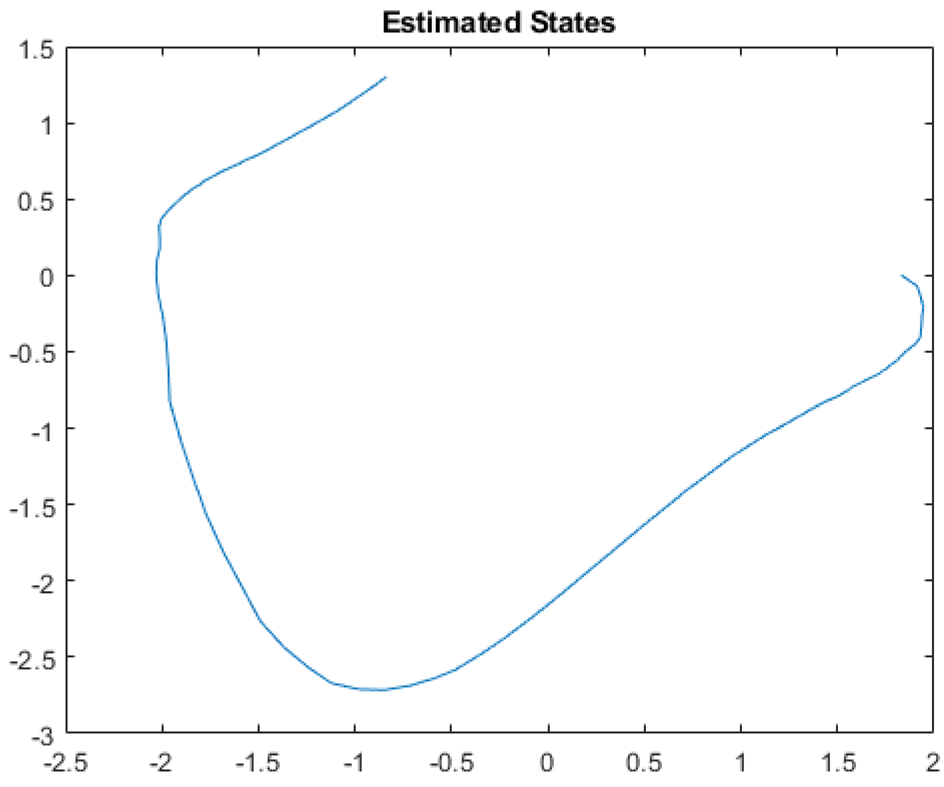
```
for k = 1:size(y)
    [Residual,ResidualCovariance] = residual(obj,y(k));
    [CorrectedState,CorrectedStateCovariance] = correct(obj,y(k));
    [PredictedState,PredictedStateCovariance] = predict(obj);

     residBuf(k,:) = Residual;
     xcorBuf(k,:) = CorrectedState';
     xpredBuf(k,:) = PredictedState';

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`. When you use the `residual` command, you do not modify any `obj` properties.

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` based on 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` based on measurements 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” on page 1-228.

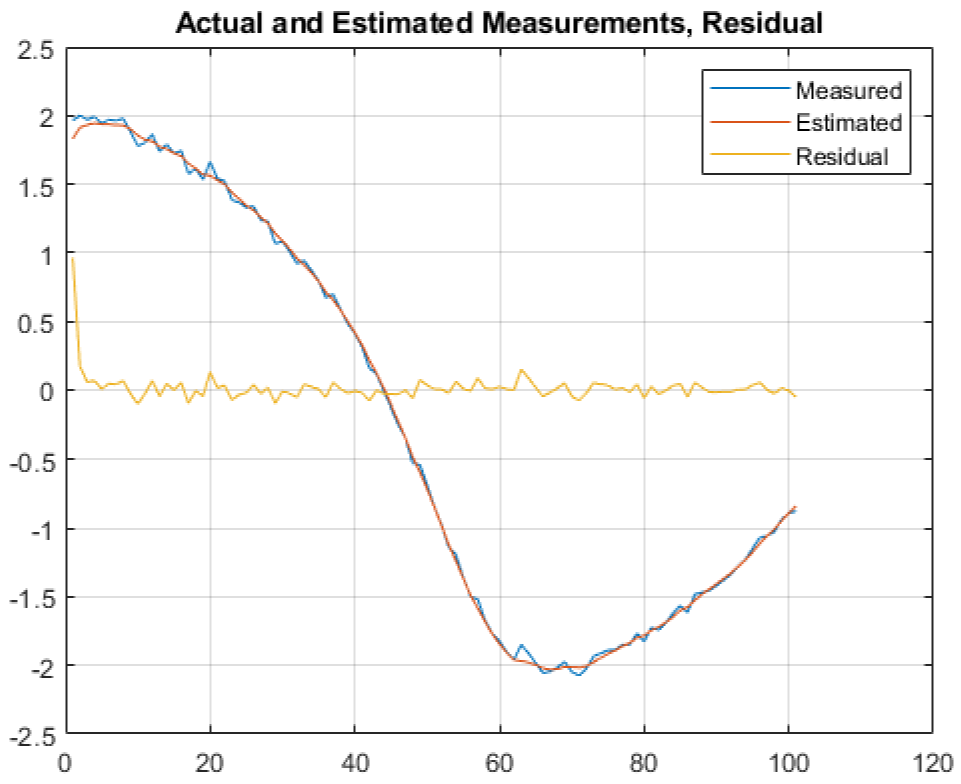
Plot the estimated states, using the postcorrection values.

```
plot(xcorBuf(:,1), xcorBuf(:,2))
title('Estimated States')
```



Plot the actual measurement, the corrected estimated measurement, and the residual. For the measurement function in `vdpMeasurementFcn`, the measurement is the first state.

```
M = [y,xcorBuf(:,1),residBuf];
plot(M)
grid on
title('Actual and Estimated Measurements, Residual')
legend('Measured','Estimated','Residual')
```



The estimate tracks the measurement closely. After the initial transient, the residual remains relatively small throughout the run.

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 following 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 the next time step, given `u[k]=0.2`.

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

Retrieve the error, or *residual*, between the prediction and the measurement.

```
[Residual, ResidualCovariance] = residual(obj,0.8,0.2);
```

Input Arguments

obj — Extended or unscented Kalman filter
`extendedKalmanFilter` object | `unscentedKalmanFilter` object

Extended or unscented Kalman filter, 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` or `MeasurementLikelihoodFcn` property of `obj`. If the function requires input arguments in addition to the state and measurement noise values, you specify these inputs in the `residual` command syntax. The `residual` command passes these inputs to the measurement or the measurement likelihood function to calculate estimated outputs. You can specify multiple arguments.

For instance, suppose that your measurement or measurement likelihood function calculates the estimated system output `y` using system inputs `u` and current time `k`, in addition to the state `x`. The U_{m1} and U_{m2} terms are therefore `u(k)` and `k`. These inputs result in the estimated output

```
y(k) = h(x(k),u(k),k)
```

Before you perform online state estimation correction at time step `k`, specify these additional inputs in the `residual` command syntax:

```
[Residual,ResidualCovariance] = residual(obj,y,u(k),k);
```

For an example showing how to use additional input arguments, see “Specify State Transition and Measurement Functions with Additional Inputs” on page 1-1451.

Output Arguments

`Residual` — Residual between current and predicted measurement
scalar | vector

Residual between current and predicted measurement, returned as a:

- Scalar for a single-output system
- Vector of size *N* for a multiple-output system, where *N* is the number of measured outputs

`ResidualCovariance` — Residual covariance
matrix

Residual covariance, returned as an *N*-by-*N* matrix where *N* is the number of measured outputs.

See Also

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

Topics
“Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter”
“Generate Code for Online State Estimation in MATLAB”
“What Is Online Estimation?”
“Extended and Unscented Kalman Filter Algorithms for Online State Estimation”
“Validate Online State Estimation at the Command Line”

Introduced in R2019b

retrend

Add offsets or trends to time-domain data signals stored in `iddata` objects

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);
```

See Also

`getTrend` | `detrend` | `TrendInfo`

Topics

"Handling Offsets and Trends in Data"

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,

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + ... + b_{nb} q^{-nb+1}$$
$$nf: \quad F(q) = 1 + f_1 q^{-1} + ... + f_{nf} q^{-nf}$$

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`.

Algorithms

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Recursive Algorithms for Online Parameter Estimation”.

See Also

`nkshift` | `recursiveOE` | `rpem` | `rplr`

Topics

“Recursive Algorithms for Online Parameter Estimation”

Introduced before R2006a

rpem

Estimate general input-output models using recursive prediction-error minimization method

Syntax

```
thm = rpem(z,nn,adm,adg)
```

```
[thm,yhat,P,phi,psi] = rpem(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

`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.

The parameters of the general linear model structure

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t - nk_1) + ... + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t - nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

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.

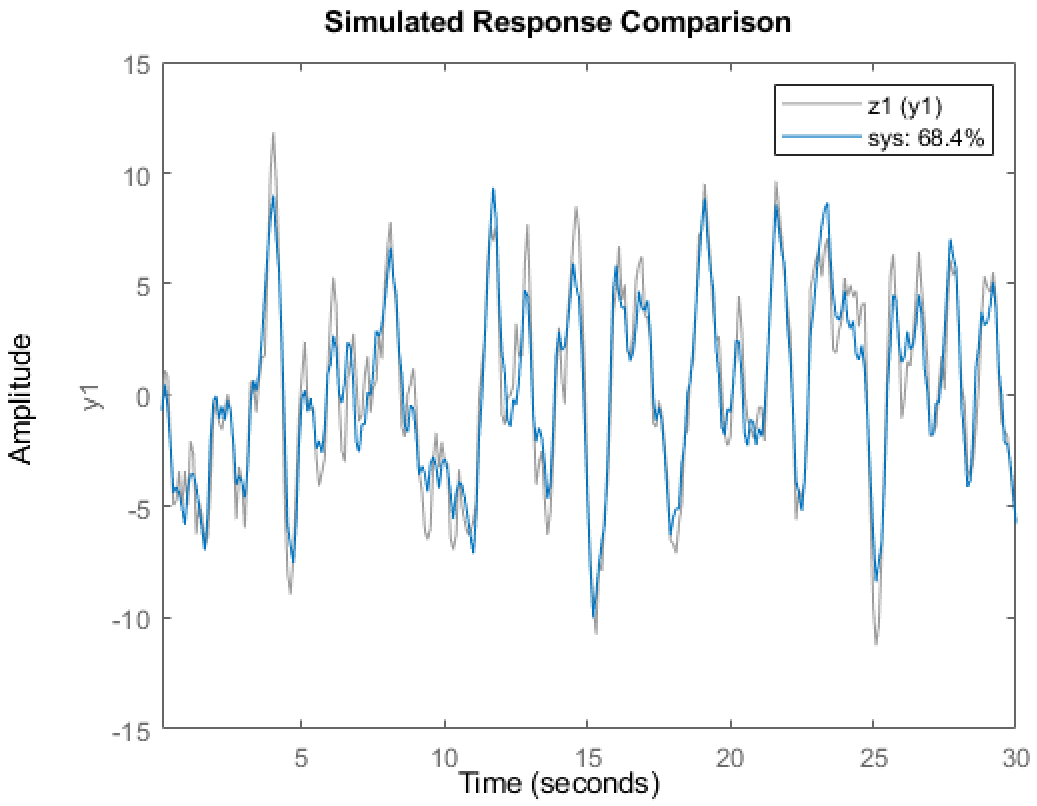
```
p = EstimatedParameters(end,:);
```

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);
```



Algorithms

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also "Recursive Algorithms for Online Parameter Estimation".

See Also

`nkshift` | `recursiveAR` | `recursiveARX` | `recursiveARMA` | `recursiveARMAX` | `recursiveBJ` | `recursiveOE` | `rplr`

Topics
"What Is Online Estimation?"
"Recursive Algorithms for Online Parameter Estimation"

Introduced before R2006a

rplr

Estimate general input-output models using recursive pseudolinear regression method

Syntax

```
thm = rplr(z,nn,adm,adg)
```

```
[thm,yhat,P,phi] = rplr(z,nn,adm,adg,th0,P0,phi0)
```

Description

`rplr` is not compatible with MATLAB Coder or MATLAB Compiler.

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);
```

References

For more information about HARF and SHARF, see Chapter 11 in Ljung (1999).

See Also

`nkshift` | `recursiveAR` | `recursiveARX` | `recursiveARMA` | `recursiveARMAX` | `recursiveBJ` | `recursiveOE` | `rpem`

Topics

"What Is Online Estimation?"
"Recursive Algorithms for Online Parameter Estimation"

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.

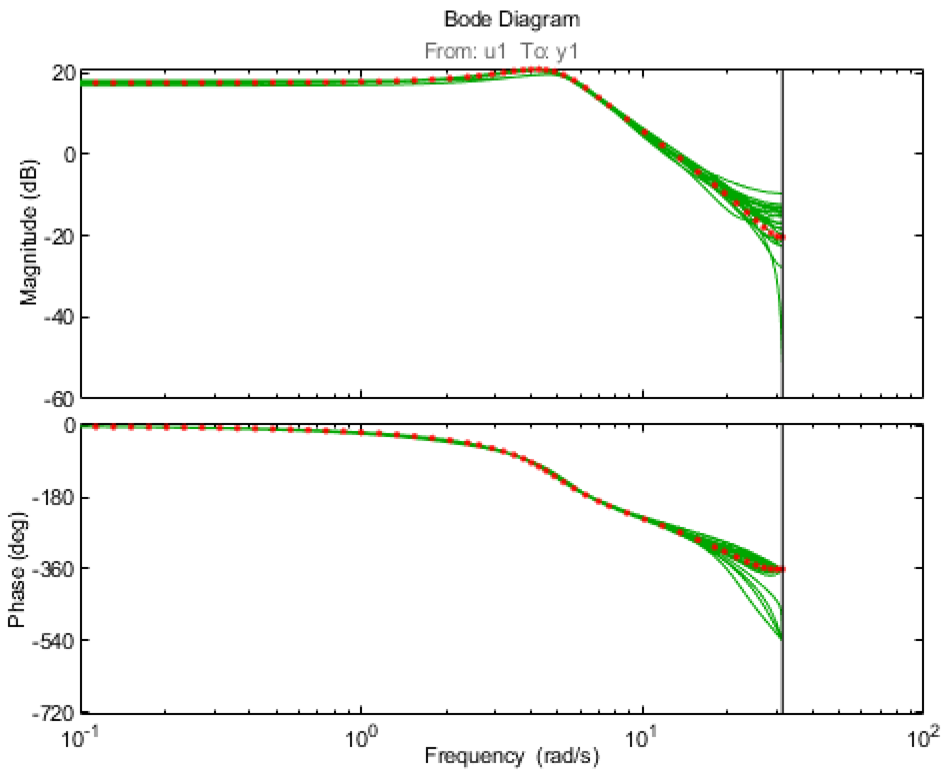
```
load iddata2 z2;
sys = n4sid(z2,3);
```

Randomly sample the estimated model.

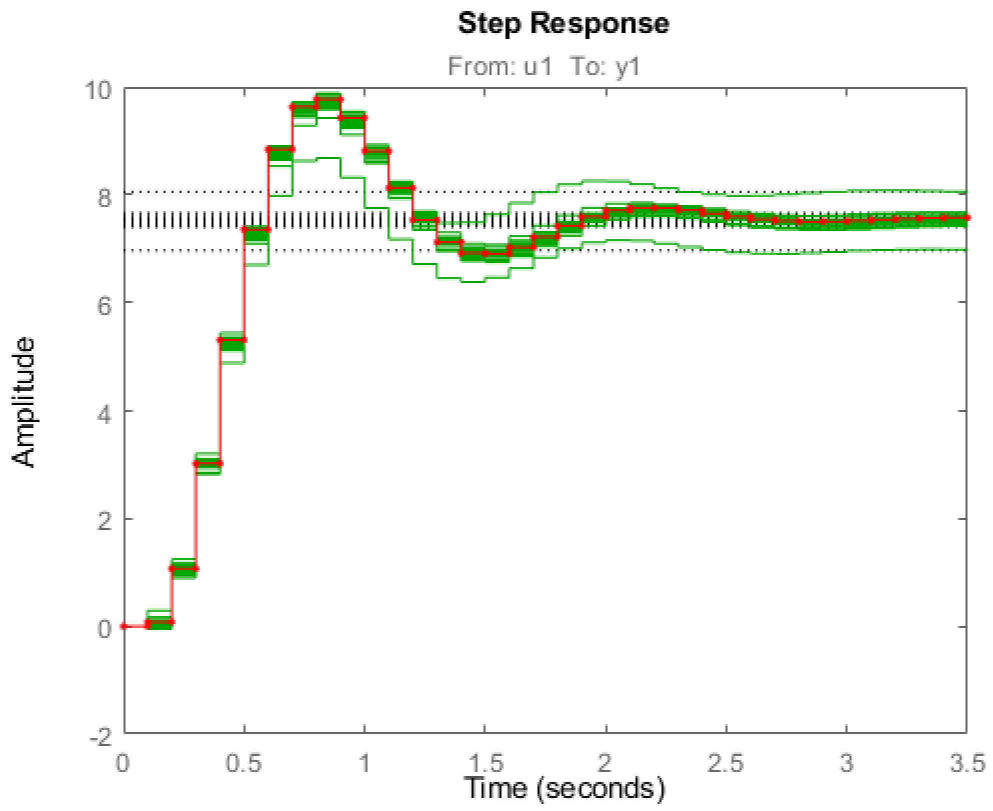
```
N = 20;
sys_array = rsample(sys,N);
```

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.-')
```



Specify Standard Deviation Level for Parameter Perturbation

Estimate a third-order, discrete-time, state-space model.

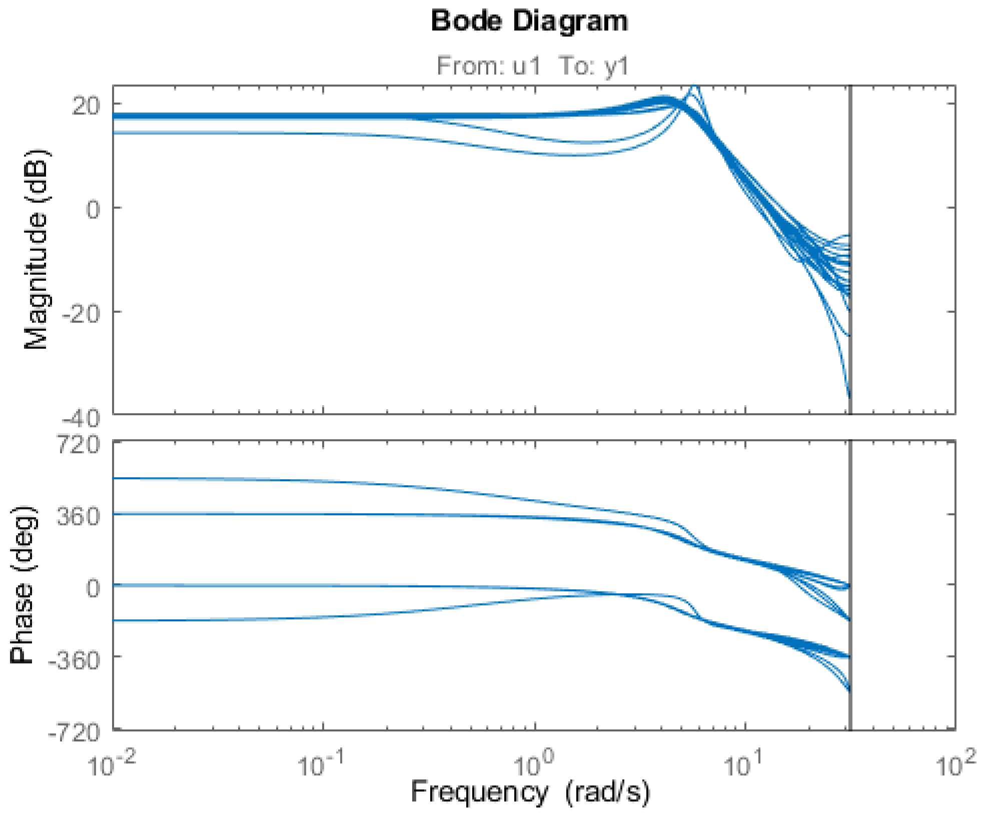
```
load iddata2 z2;
sys = n4sid(z2,3);
```

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.

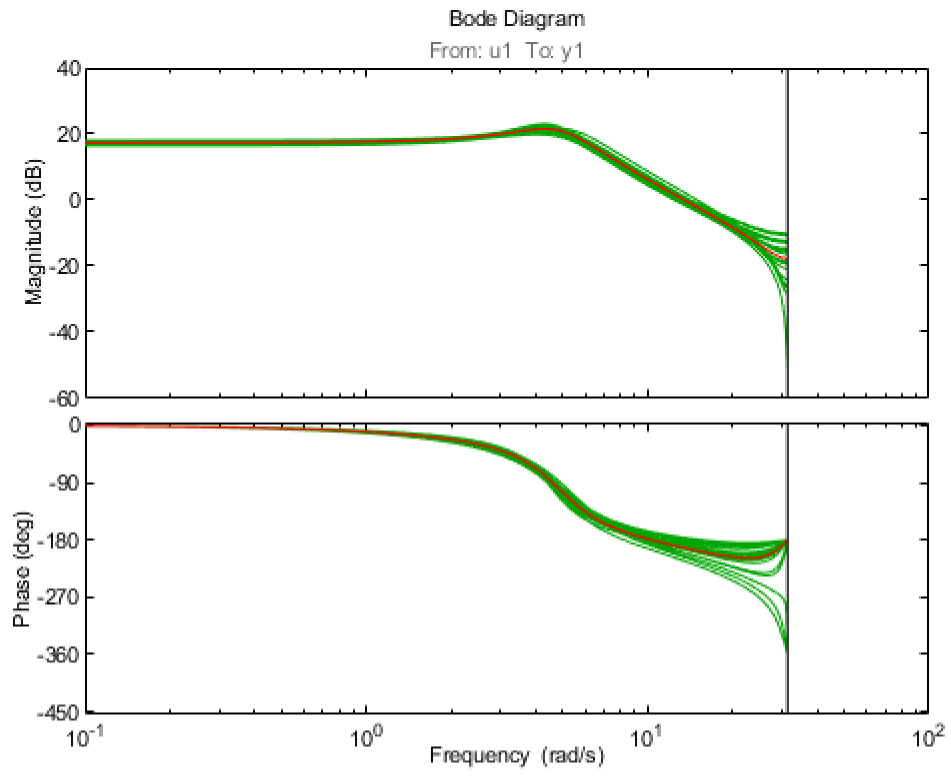
```
load iddata1 z1
sys = armax(z1,[2 2 2 1]);
```

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**.

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

`simsd` | `init` | `noisecnv` | `noise2meas` | `iopzmap` | `bode` | `step` | `translatecov`

Introduced in R2012a

segment

Segment data and estimate models for each segment

Syntax

```
segm = segment(z,nn)
```

```
[segm,V,thm,R2e] = segment(z,nn,R2,q,R1,M,th0,P0,ll,mu)
```

Description

`segment` builds models of AR, ARX, or ARMAX/ARMA type,

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

assuming that the model parameters are piecewise constant over time. It results in a model that has split the data record into segments over which the model remains constant. The function models signals and systems that might undergo abrupt changes.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. If the system has several inputs, `u` has the corresponding number of columns.

The argument `nn` defines the model order. For the ARMAX model

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

where `na`, `nb`, and `nc` are the orders of the corresponding polynomials. See “What Are Polynomial Models?”. Moreover, `nk` is the delay. If the model has several inputs, `nb` and `nk` are row vectors, giving the orders and delays for each input.

For an ARX model (`nc = 0`) enter

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

For an ARMA model of a time series

```
z = y;
nn = [na nc];
```

and for an AR model

```
nn = na;
```

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.

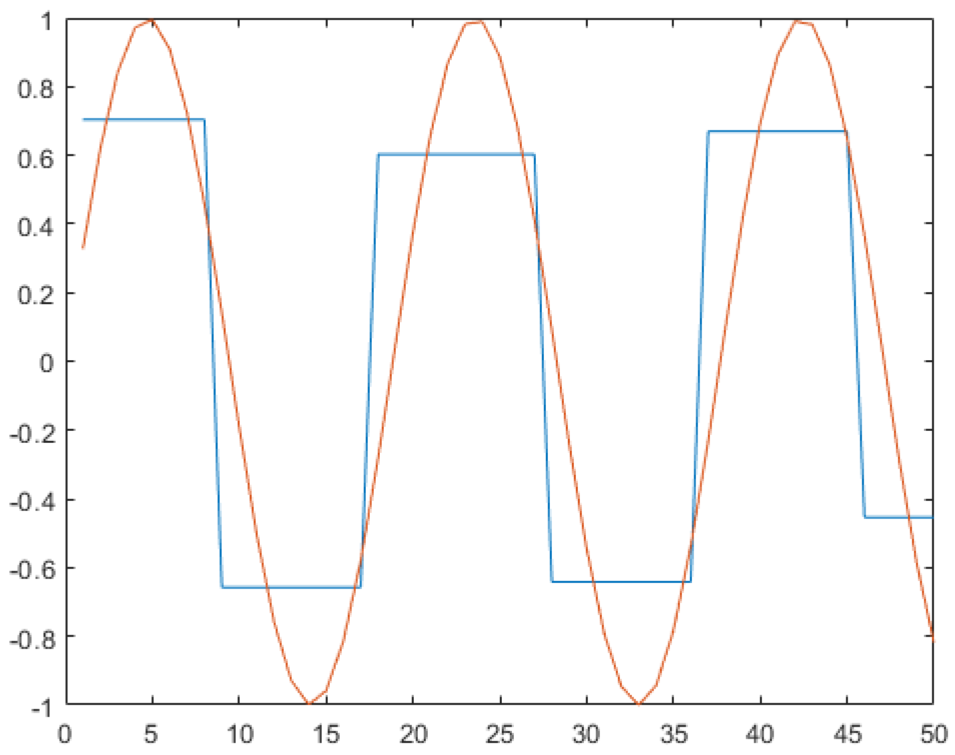
```
R2 = 0.1;
```

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])
```

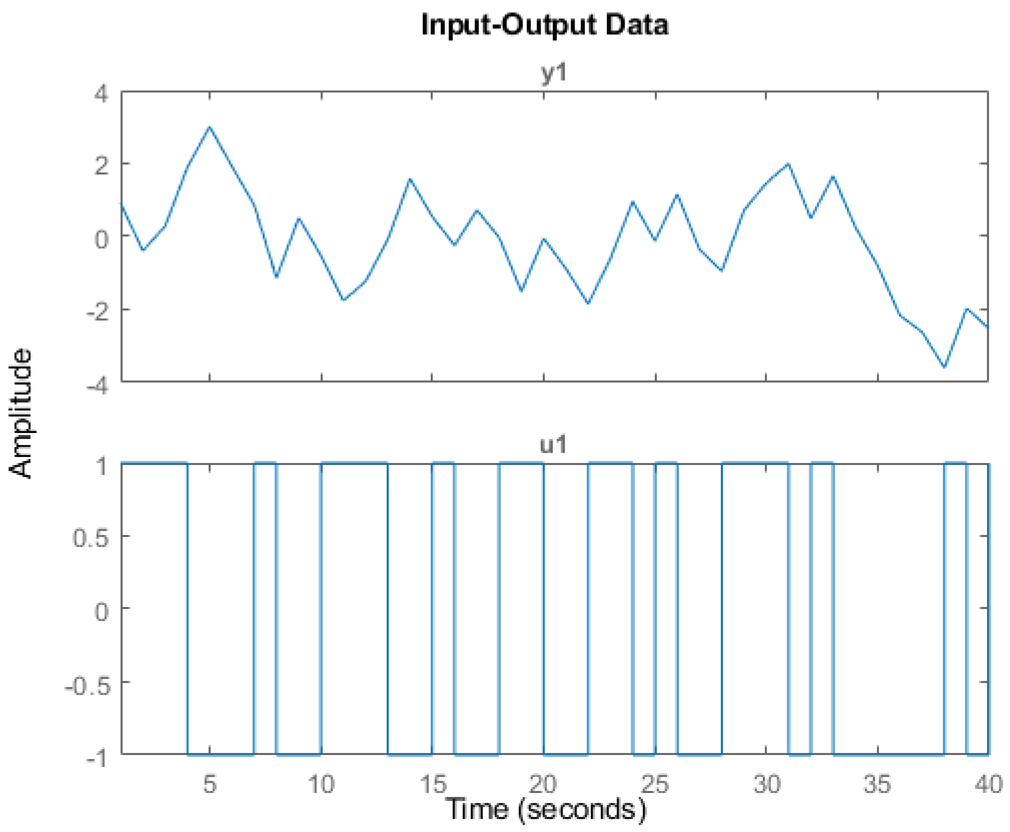


Vary the value of R2 to change the estimated noise variance. Decreasing R2 increases the number of segments produced for this model.

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:

$$y(t) + ay(t-1) = b_1u(t-1) + b_2u(t-2)$$

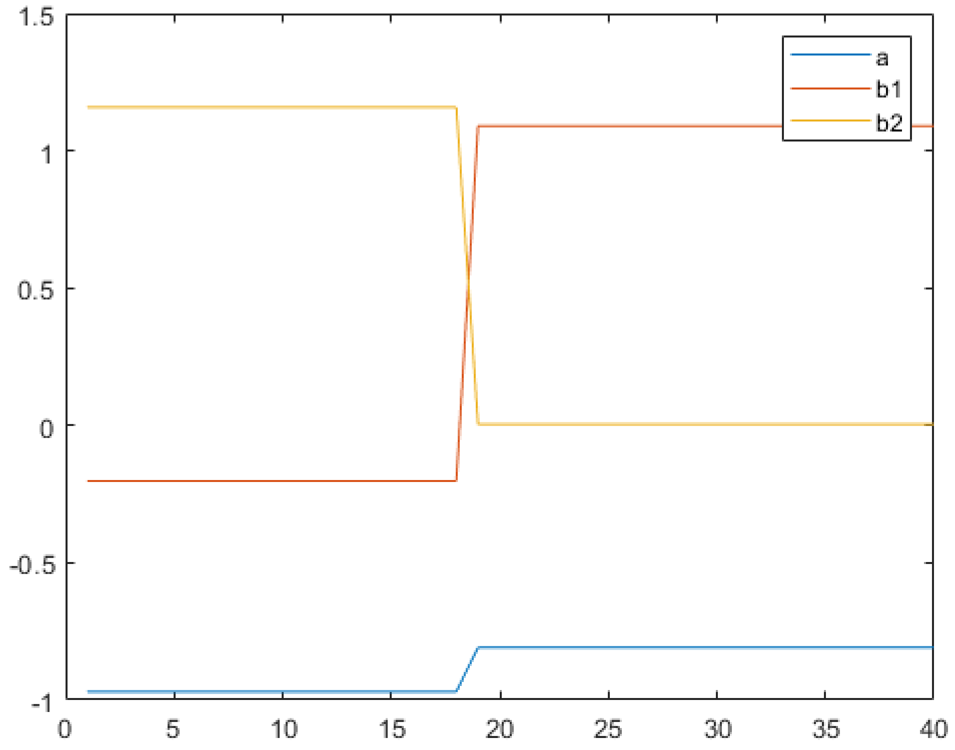
```
nn = [1 2 1];
```

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.

Limitations

`segment` is not compatible with MATLAB Coder or MATLAB Compiler.

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.

See Also

Topics
"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

$$
\begin{aligned}
V_{\mathrm{mod}} &= \log\left(V\left(1 + \frac{2d}{N}\right)\right) \\
&= \log(V) + \frac{2d}{N}, N \gg d
\end{aligned}
$$

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. $\log(V) + \dfrac{2d}{N}$ 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.

$$
V_{\mathrm{mod}} = V\left(1 + \frac{d\log(N)}{N}\right)
$$

When `c` equals a numerical value, the structure that minimizes $V_{\mathrm{mod}} = V\left(1 + \dfrac{cd}{N}\right)$

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

Specify Model Properties

Create a SISO state-space model with matrices *A*, *B*, *C*, and *D* equal to 1, 2, 3, and 4, respectively.

```
sys = ss(1,2,3,4);
```

Modify the properties of the model. Add an input delay of 0.1 second, label the input as `torque`, and set the *D* matrix to 0.

```
set(sys,'InputDelay',0.1,'InputName','torque','D',0);
```

View the model properties, and verify the changes.

```
get(sys)
```

```
                A: 1
                B: 2
                C: 3
```

```
               D: 0
               E: []
          Scaled: 0
       StateName: {''}
       StatePath: {''}
       StateUnit: {''}
   InternalDelay: [0x1 double]
      InputDelay: 0.1000
     OutputDelay: 0
              Ts: 0
        TimeUnit: 'seconds'
       InputName: {'torque'}
       InputUnit: {''}
      InputGroup: [1x1 struct]
      OutputName: {''}
      OutputUnit: {''}
     OutputGroup: [1x1 struct]
           Notes: [0x1 string]
        UserData: []
            Name: ''
    SamplingGrid: [1x1 struct]
```

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:

$$\bar{h}\left(z^{-1}\right) = \frac{1+2z^{-1}}{1+3z^{-1}+4z^{-2}} = zh(z)$$

Note Because the resulting transfer functions are different, make sure to use the convention consistent with your choice of variable.

See Also

get | frd | ss | tf | zpk | idfrd | idss | idtf | idgrey | idproc | idpoly | idnlarx | idnlhw | idnlgrey

Introduced before R2006a

setcov

Set parameter covariance data in identified model

Syntax

```
sys = setcov(sys0,cov)
```

Description

`sys = setcov(sys0,cov)` sets the parameter covariance of identified model `sys0` as `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.

Identified model, specified as an `idtf`, `idss`, `idgrey`, `idpoly`, `idproc`, or `idnlgrey` model. You cannot set the covariance for nonlinear black-box models (`idnlarx` and `idnlhw`).

`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` | `sim` | `setpvec`

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 handle or plot options property

Syntax

```
setoptions(h,p)
setoptions(h,'property1','value1',...,'propertyN','valueN')
setoptions(h,p,'property1','value1',...,'propertyN','valueN')
```

Description

You can use `setoptions` to set the plot handle options or properties list and use it to customize the plot, such as modify the axes labels, limits and units. For a list of the properties and values available for each plot type, see “Properties and Values Reference” (Control System Toolbox). To customize an existing plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox).

`setoptions(h,p)` sets preferences for response plot using the plot handle `h` is the plot handle and plot options handle `p` that contains information about plot options.

`setoptions(h,'property1','value1',...,'propertyN','valueN')` assigns values to property-value pairs instead of using the plot options handle `p`. For a list of the properties and values available for each plot type, see “Properties and Values Reference” (Control System Toolbox).

`setoptions(h,p,'property1','value1',...,'propertyN','valueN')` first assigns properties using the plot options handle `p`, and then overrides any properties governed by the specified property-value pairs.. For a list of the properties and values available for each plot type, see “Properties and Values Reference” (Control System Toolbox).

Examples

Impulse Plot with Specified Grid Color

For this example, consider a MIMO state-space model with 3 inputs, 3 outputs and 3 states. Create a impulse plot with red colored grid lines.

Create the MIMO state-space model `sys_mimo`.

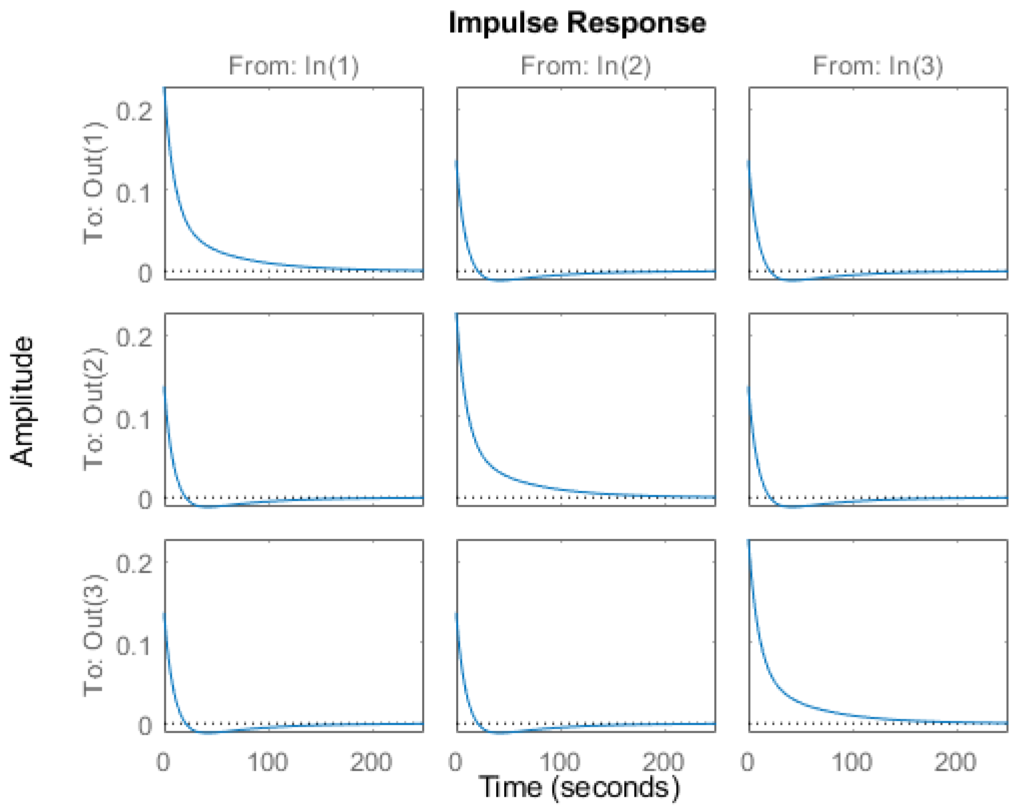
```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
```

```
D = 0;
sys_mimo = ss(A,B,C,D);
size(sys_mimo)
```

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

Create an impulse plot with plot handle `h` and use `getoptions` for a list of the options available.

```
h = impulseplot(sys_mimo)
```



```
h =

    resppack.timeplot
```

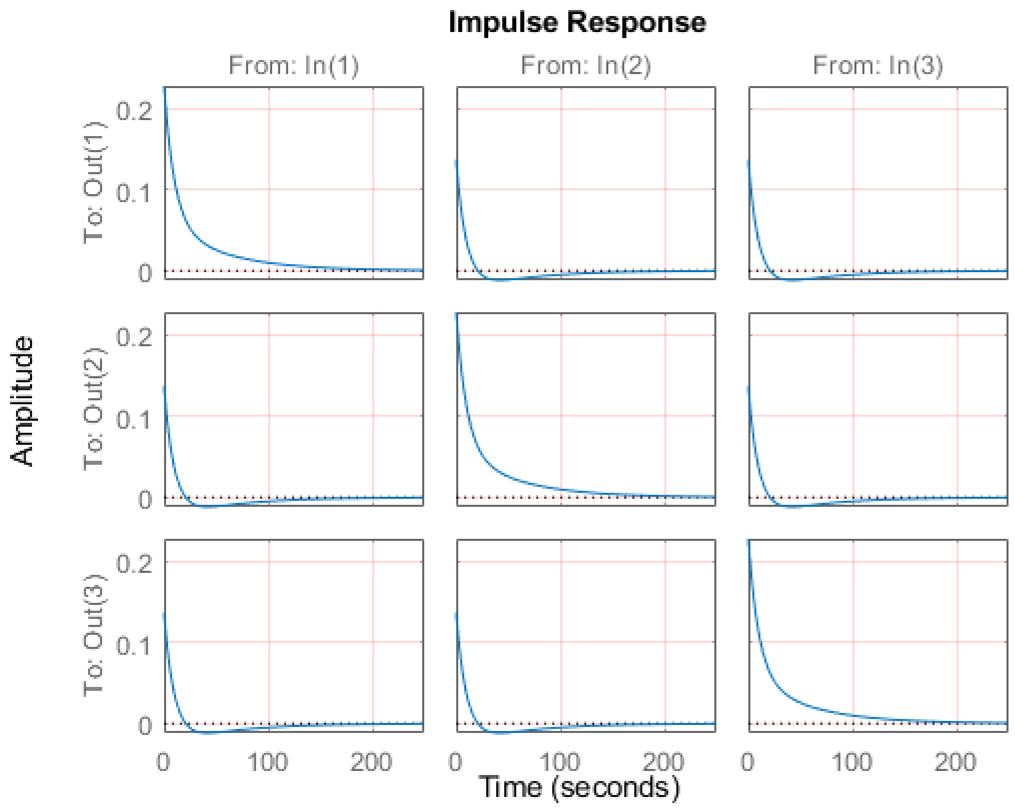
```
p = getoptions(h)
```

```
p =

                   Normalize: 'off'
          SettleTimeThreshold: 0.0200
               RiseTimeLimits: [0.1000 0.9000]
                    TimeUnits: 'seconds'
    ConfidenceRegionNumberSD: 1
                   IOGrouping: 'none'
                  InputLabels: [1x1 struct]
                 OutputLabels: [1x1 struct]
                 InputVisible: {3x1 cell}
                OutputVisible: {3x1 cell}
```

```
        Title: [1x1 struct]
       XLabel: [1x1 struct]
       YLabel: [1x1 struct]
    TickLabel: [1x1 struct]
         Grid: 'off'
    GridColor: [0.1500 0.1500 0.1500]
         XLim: {3x1 cell}
         YLim: {3x1 cell}
     XLimMode: {3x1 cell}
     YLimMode: {3x1 cell}
```

Use `setoptions` to update the plot with the requires customization.

```
setoptions(h,'Grid','on','GridColor',[1 0 0]);
```



The impulse plot automatically updates when you call `setoptions`. For MIMO models, `impulseplot` produces a grid of plots, each plot displaying the impulse response of one I/O pair.

Bode Plot with Specified Frequency Scale and Units

For this example, consider a MIMO state-space model with 3 inputs, 3 outputs and 3 states. Create a Bode plot with linear frequency scale, specify frequency units in Hz and turn the grid on.

Create the MIMO state-space model `sys_mimo`.

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
size(sys_mimo)
```

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

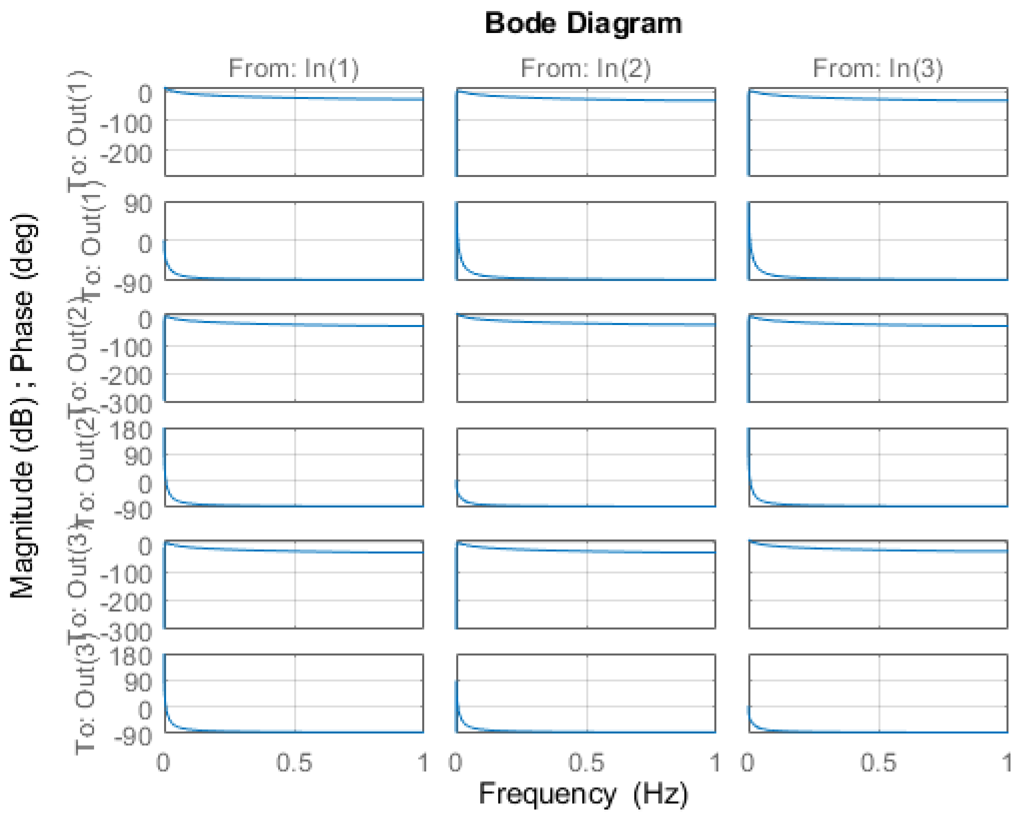
Create a Bode plot with plot handle `h` and use `getoptions` for a list of the options available.

```
h = bodeplot(sys_mimo);
p = getoptions(h)
```

```
p = 

                   FreqUnits: 'rad/s'
                   FreqScale: 'log'
                    MagUnits: 'dB'
                    MagScale: 'linear'
                  MagVisible: 'on'
             MagLowerLimMode: 'auto'
                  PhaseUnits: 'deg'
                PhaseVisible: 'on'
               PhaseWrapping: 'off'
               PhaseMatching: 'off'
           PhaseMatchingFreq: 0
    ConfidenceRegionNumberSD: 1
                 MagLowerLim: 0
          PhaseMatchingValue: 0
         PhaseWrappingBranch: -180
                  IOGrouping: 'none'
                 InputLabels: [1x1 struct]
                OutputLabels: [1x1 struct]
                InputVisible: {3x1 cell}
               OutputVisible: {3x1 cell}
                       Title: [1x1 struct]
                      XLabel: [1x1 struct]
                      YLabel: [1x1 struct]
                   TickLabel: [1x1 struct]
                        Grid: 'off'
                   GridColor: [0.1500 0.1500 0.1500]
                        XLim: {3x1 cell}
                        YLim: {6x1 cell}
                    XLimMode: {3x1 cell}
                    YLimMode: {6x1 cell}
```

Use `setoptions` to update the plot with the requires customization.

```
setoptions(h,'FreqScale','linear','FreqUnits','Hz','Grid','on');
```



The Bode plot automatically updates when you call `setoptions`. For MIMO models, `bodeplot` produces an array of Bode plots, each plot displaying the frequency response of one I/O pair.

Change Frequency Units in Response Plot

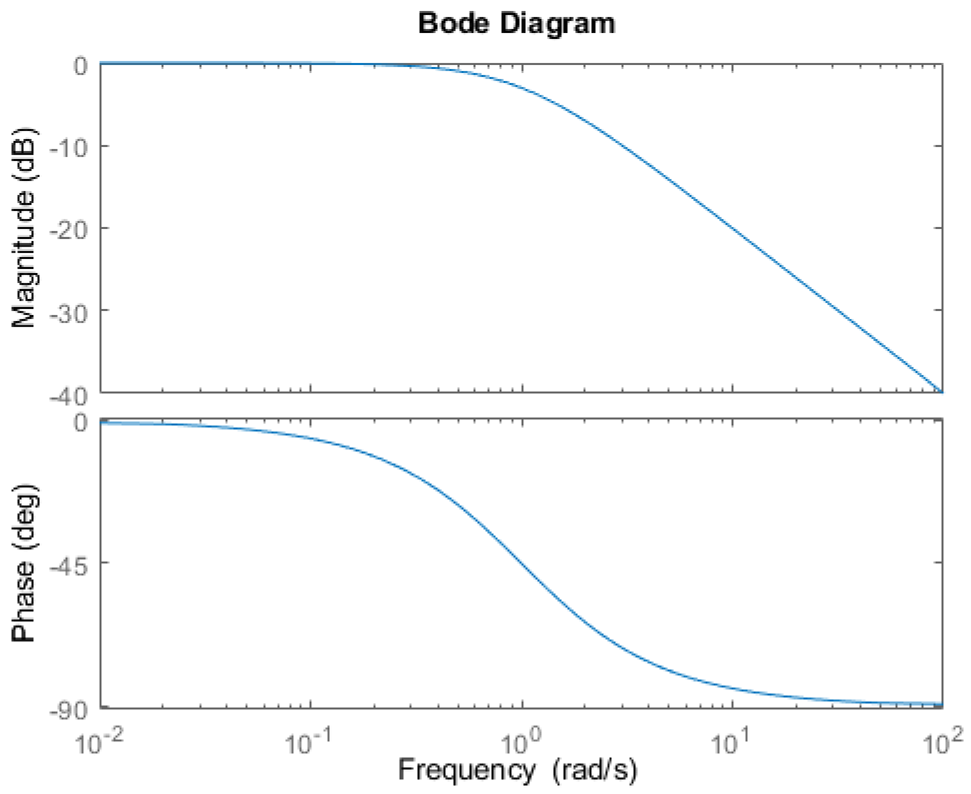
Create the following continuous-time transfer function:

$$H(s) = \frac{1}{s+1}$$

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

Create a Bode plot with plot handle `h`.

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



Create a plot options handle `p`.

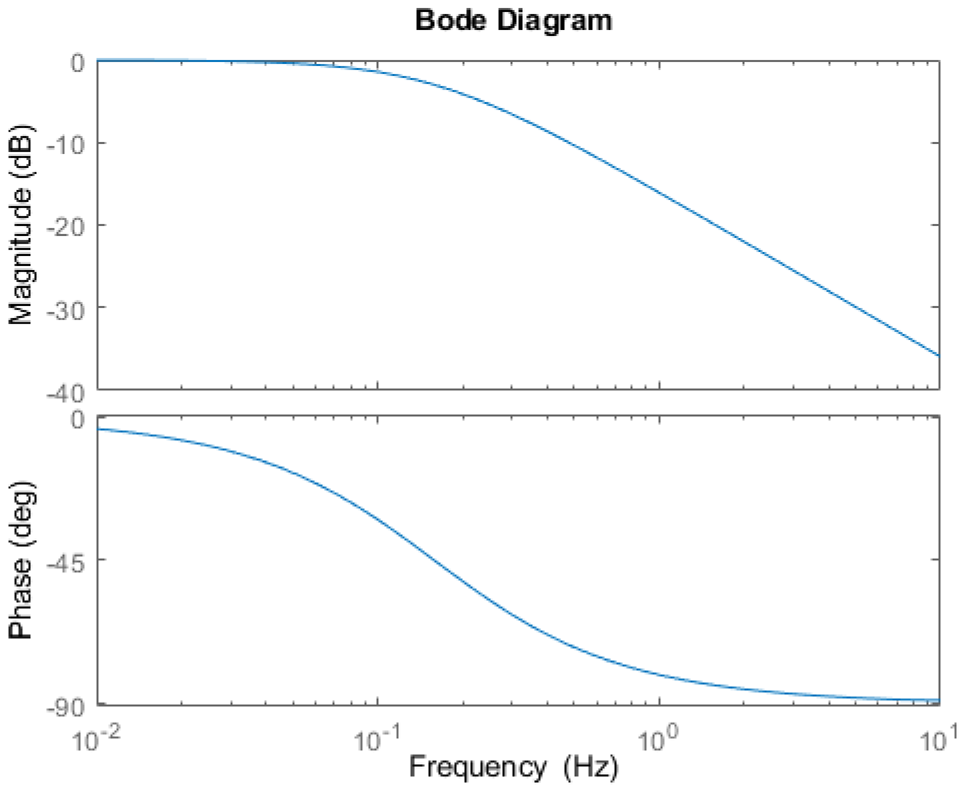
```
p = getoptions(h);
```

Change frequency units of the plot to `Hz`.

```
p.FreqUnits = 'Hz';
```

Apply the plot options to the Bode plot.

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



Alternatively, use `setoptions(h,'FrequencyUnits','Hz')`.

Input Arguments

h — Plot handle
plot handle object

Plot handle, specified as a plot handle object. For example, `h` is a `mpzplot` object for a pole-zero or I/O pole-zero plot.

p — Plot options handle
plot options handle object

Plot options handle, specified as a plot options handle object. For example, `p` is a `PZMapOptions` object for a pole-zero or I/O pole-zero plot.

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 plot

- `hsvoptions` — Hankel singular values plot
- `nicholsoptions` — Nichols plot
- `nyquistoptions` — Nyquist plot
- `pzoptions` — Pole-zero plot
- `sigmaoptions` — Sigma plot
- `timeoptions` — Time plots (step, initial, impulse, etc.)

For example,

```
p = bodeoptions
```

returns a plot options handle for Bode plot.

See Also
`getoptions`

Topics
“Properties and Values Reference” (Control System Toolbox)
“Customizing Response Plots from the Command Line” (Control System Toolbox)

Introduced in R2012a

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 = 5x1 cell
    {'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 | setpvec | setcov

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,:));
```

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;
```

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

`idpoly` | `get` | `set` | `polydata` | `tfdata`

Topics

"Extracting Numerical Model Data"

Introduced in R2010a

setpvec

Modify values 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`. For `idnlarx` and `idnlhw` models, all parameters are treated as free.

Input Arguments

sys0

Identified model, specified as an `idtf`, `idss`, `idgrey`, `idpoly`, `idproc`, `idnlarx`, `idnlhw`, or `idnlgrey` object.

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` | `setcov` | `nparams`

Introduced in R2012a

sgrid

Generate s-plane grid of constant damping factors and natural frequencies

Syntax

```
sgrid
sgrid(zeta,wn)
sgrid( ___ ,'new')
sgrid(AX, ___ )
```

Description

`sgrid` generates a grid of constant damping factors from 0 to 1 in steps of 0.1 and natural frequencies from 0 to 10 rad/sec in steps of one rad/sec for pole-zero and root locus plots. `sgrid` then plots the grid over the current axis. `sgrid` creates the grid over the plot if the current axis contains a continuous s-plane root locus diagram or pole-zero map.

`sgrid(zeta,wn)` plots a grid of constant damping factor and natural frequency lines for the damping factors and natural frequencies in the vectors `zeta` and `wn`, respectively. `sgrid(zeta,wn)` creates the grid over the plot if the current axis contains a continuous s-plane root locus diagram or pole-zero map.

Alternatively, you can select **Grid** from the context menu to generate the same s-plane grid.

`sgrid( ___ ,'new')` clears the current axes first and sets `hold on`.

`sgrid(AX, ___ )` plots the s-plane grid on the `Axes` or `UIAxes` object in the current figure with the handle `AX`. Use this syntax when creating apps with `sgrid` in the App Designer.

Examples

Generate S-Plane Grid on Root Locus Plot

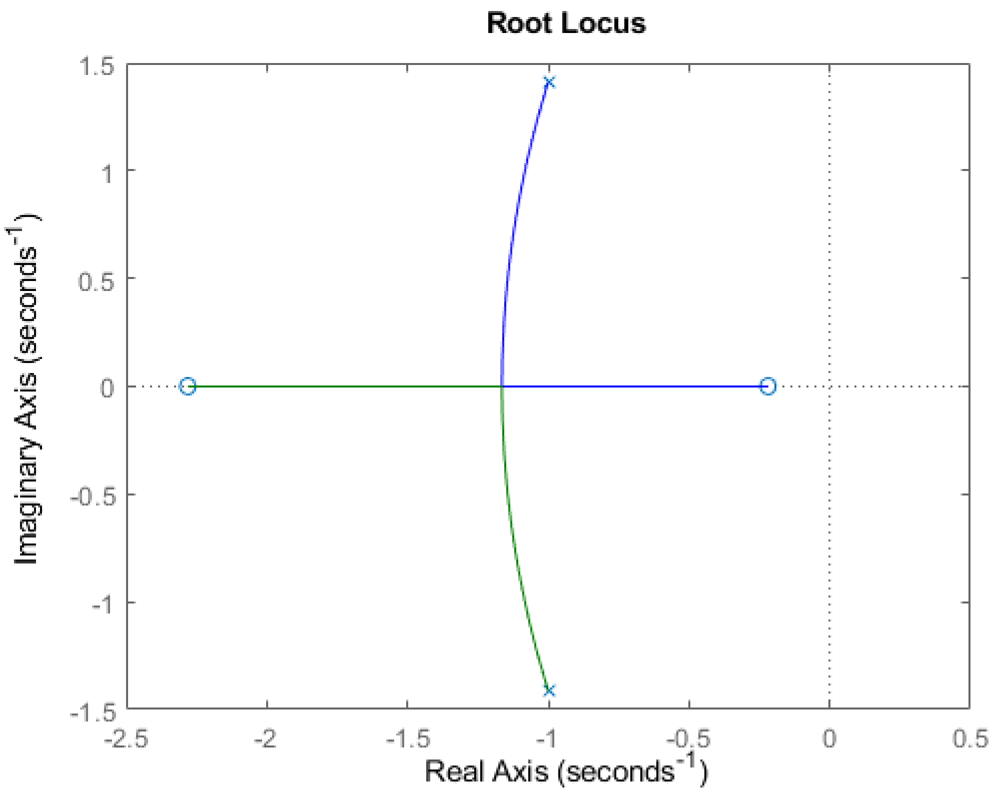
Create the following continuous-time transfer function:

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

```
H = tf([2 5 1],[1 2 3]);
```

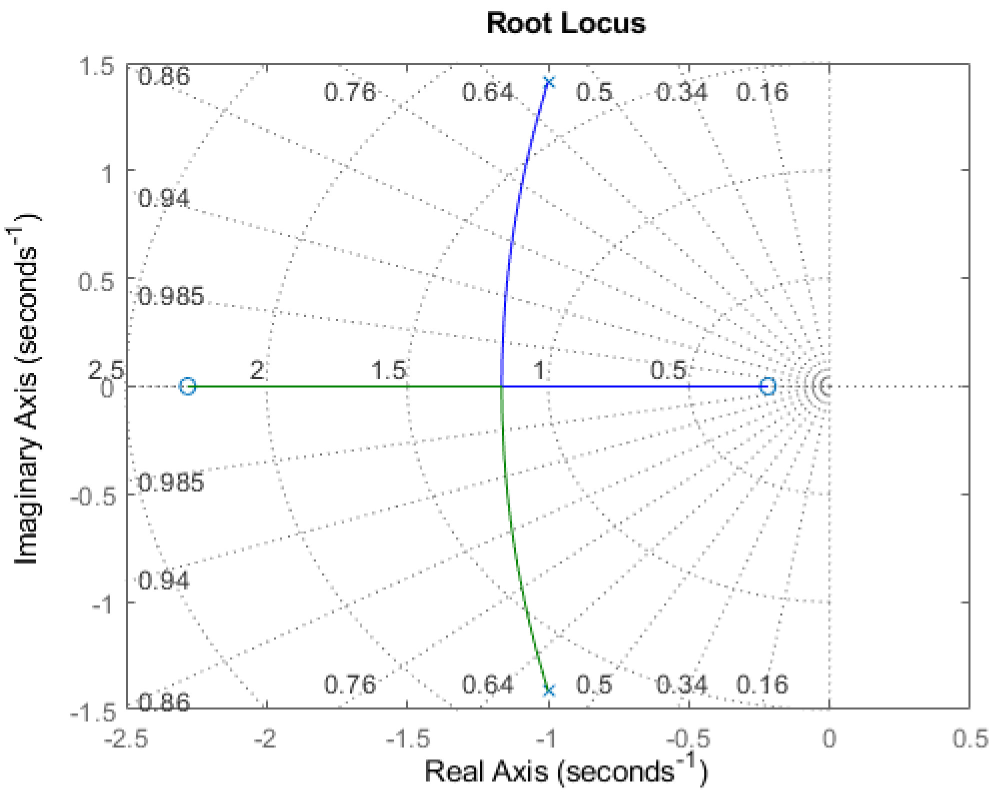
Plot the root locus of the transfer function.

```
rlocus(H)
```



Plot *s*-plane grid lines on the root locus.

```
sgrid
```



Input Arguments

`zeta` — Damping ratio
vector

Damping ratio, specified as a vector in the same order as `wn`.

`wn` — Normalized natural frequency
vector

Normalized natural frequency, specified as a vector.

`AX` — Object handle
`Axes` object | `UIAxes` object

Object handle, specified as an `Axes` or `UIAxes` object. Use `AX` to create apps with `sgrid` in the App Designer.

See Also

`pzmap` | `rlocus` | `zgrid`

Introduced before R2006a

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')`

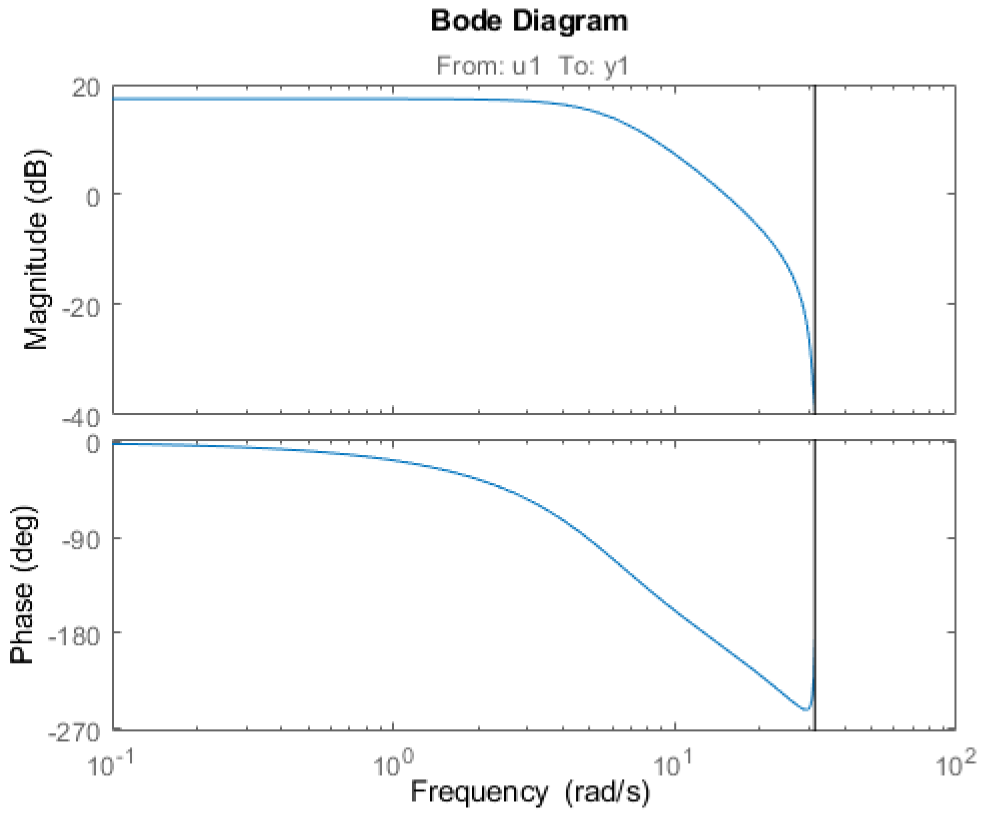
Examples

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.

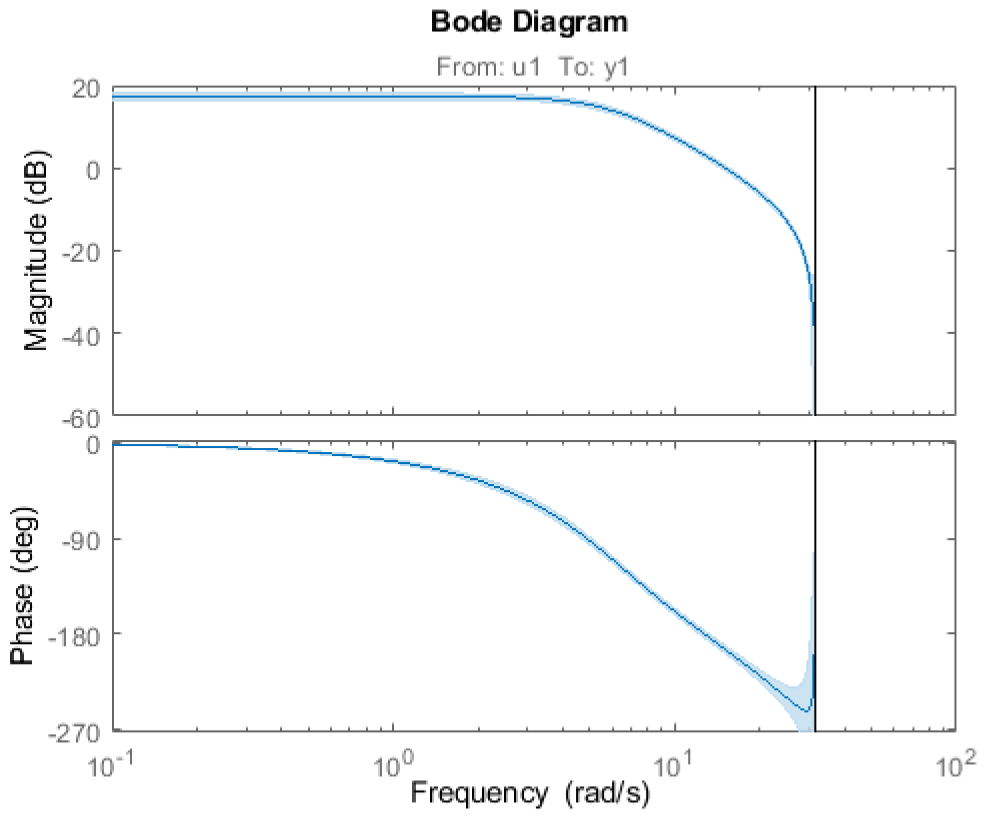
```
load iddata1 z1
sys = arx(z1, [2 2 1]);
h = bodeplot(sys);
```



`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);
```



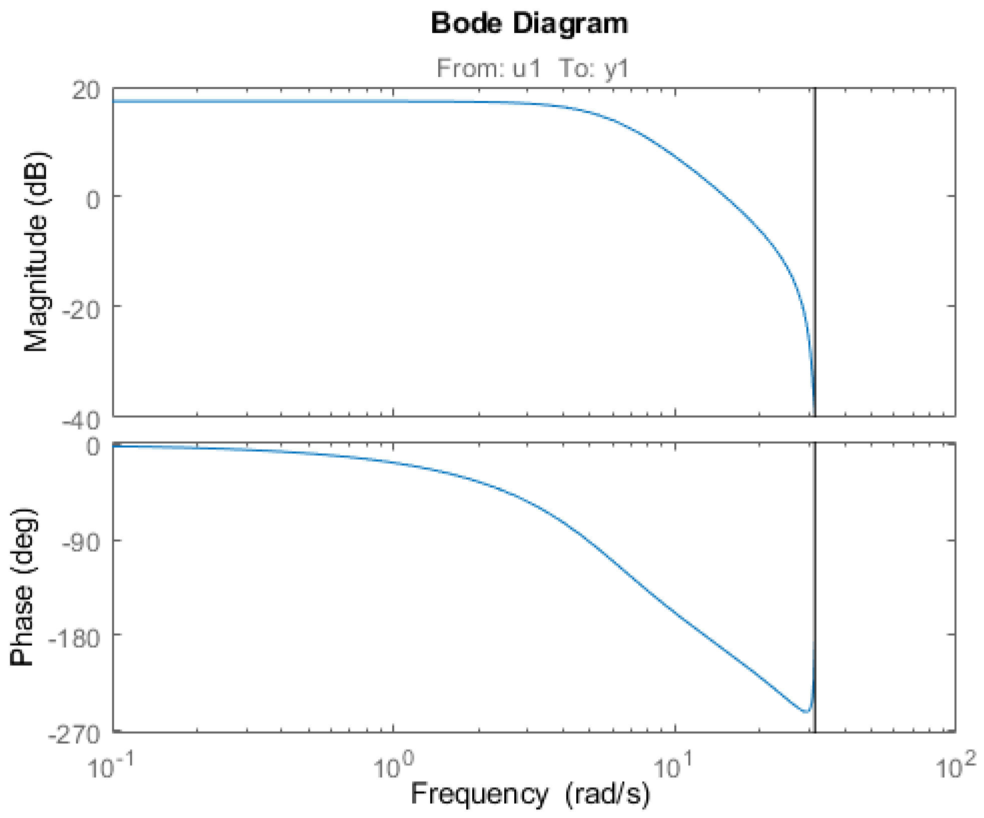
This plot depicts the confidence region for 1 standard deviation.

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.

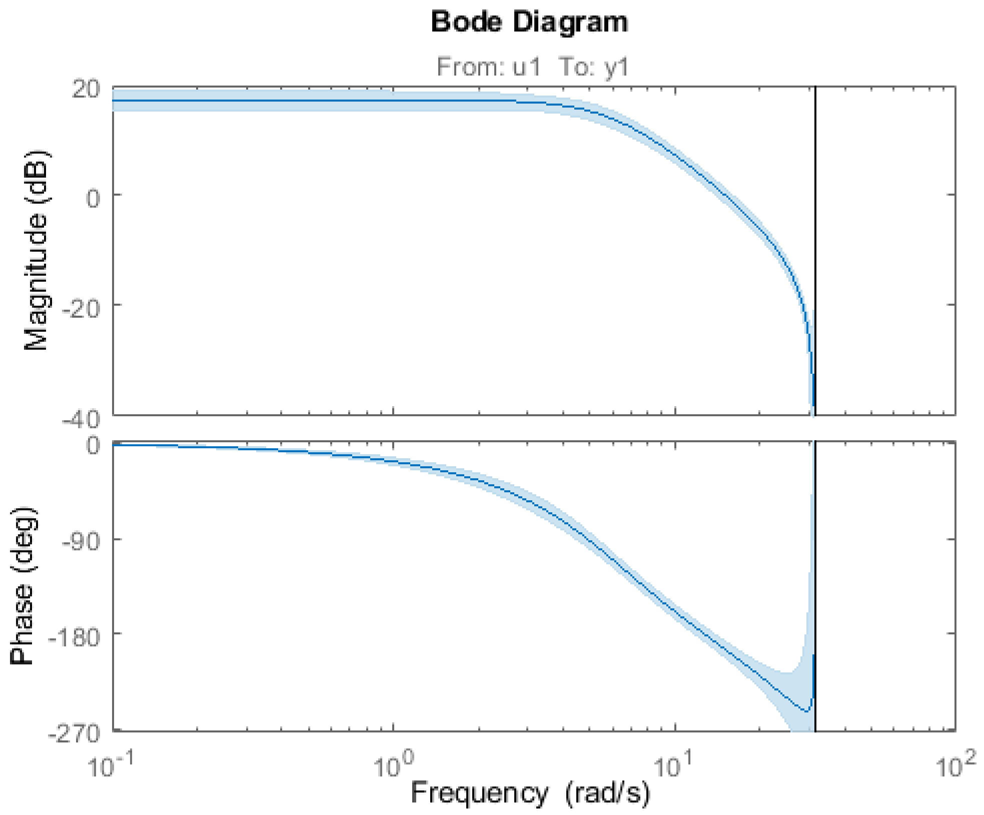
```
load iddata1 z1
sys = arx(z1, [2 2 1]);
h = bodeplot(sys);
```



`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 | stepplot | impulseplot | nyquistplot | iopzplot

Introduced in R2012a

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-space 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

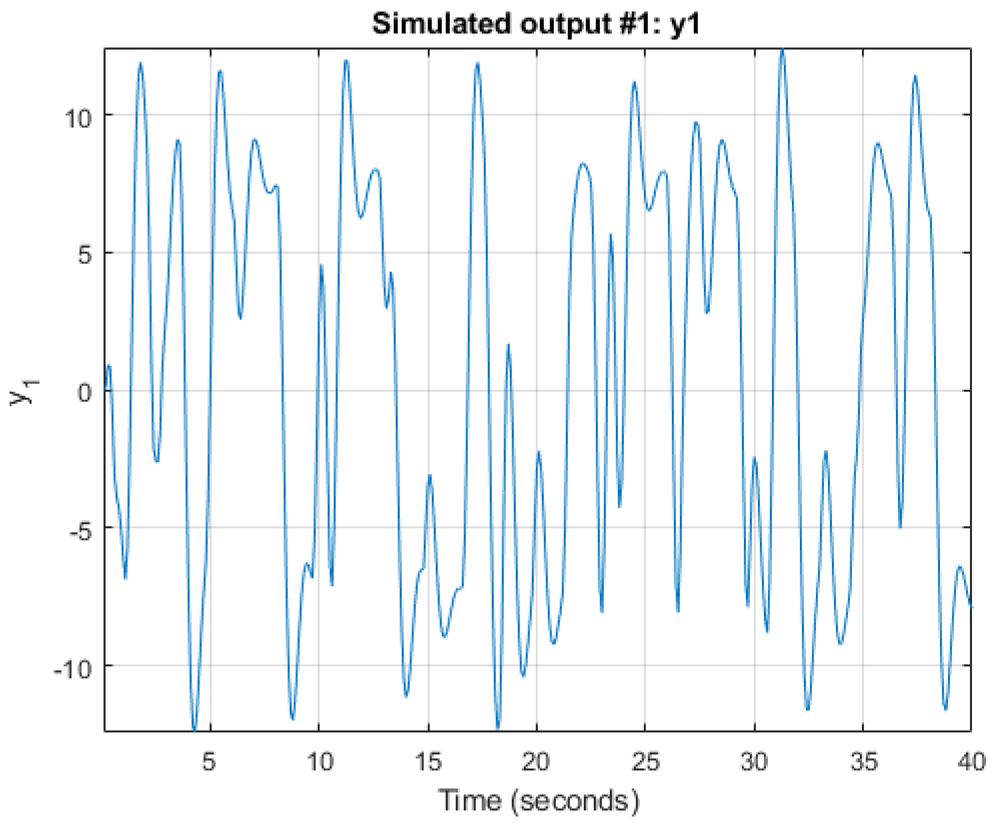
Obtain the identified model.

```
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.

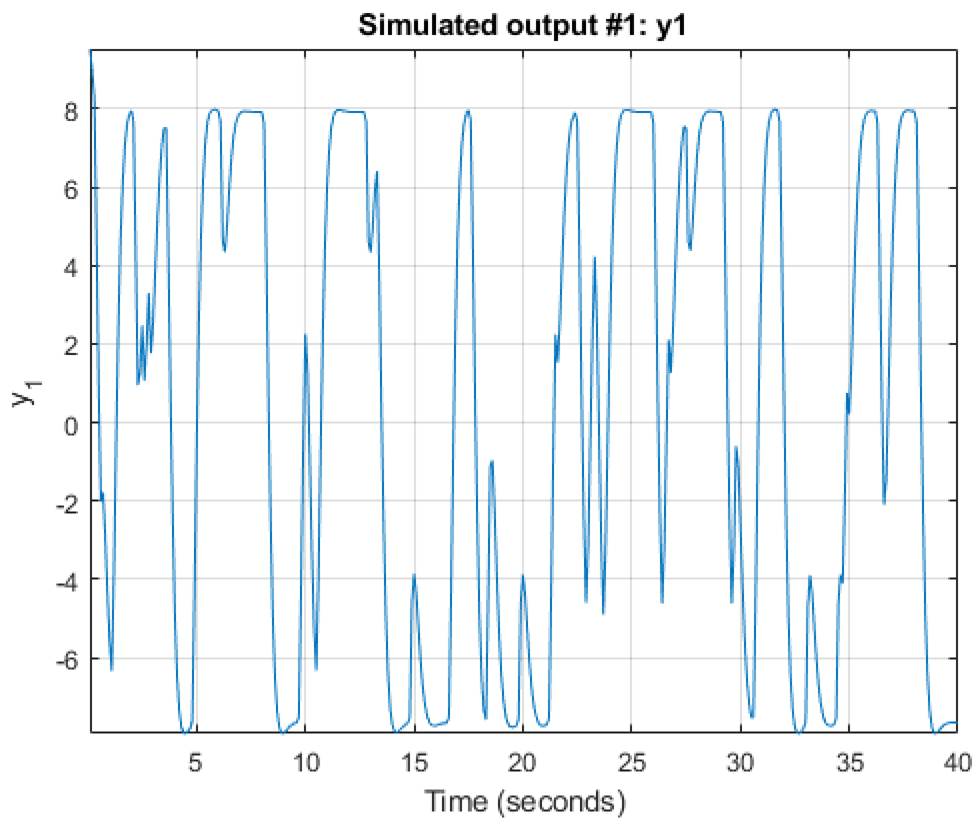
```
M = nlarx(z2,[2 2 1],'idTreePartition');
```

Estimate current states of model based on past data. Specify as many past samples as there are lags in the input and output variables (2 here).

```
x0 = data2state(M,struct('Input',ones(2,1),'Output',10*ones(2,1)));
```

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],idTreePartition);
```

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.

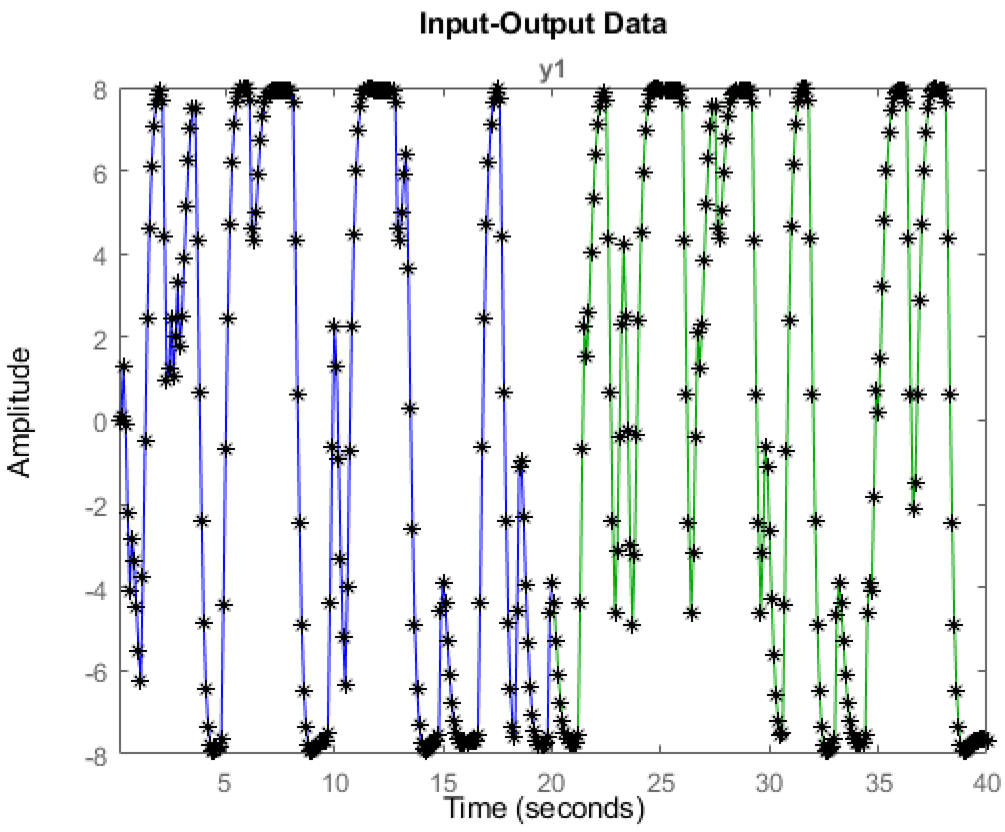
```
firstSimData = [ys1,u1];
opt2 = simOptions('InitialCondition',firstSimData);
ys2 = sim(M,u2,opt2);
```

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],idWaveletNetwork('NumberOfUnits',20));
```

Estimate the initial states of `M` to best fit `z2.y` in the simulated response.

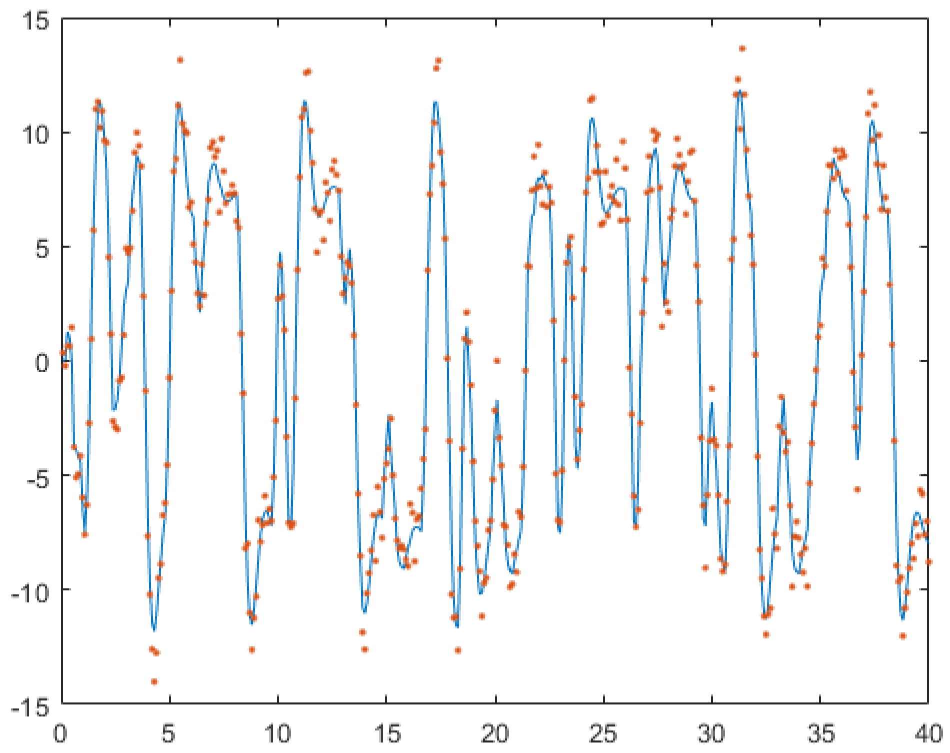
```
x0 = findstates(M,z2,Inf);
```

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.

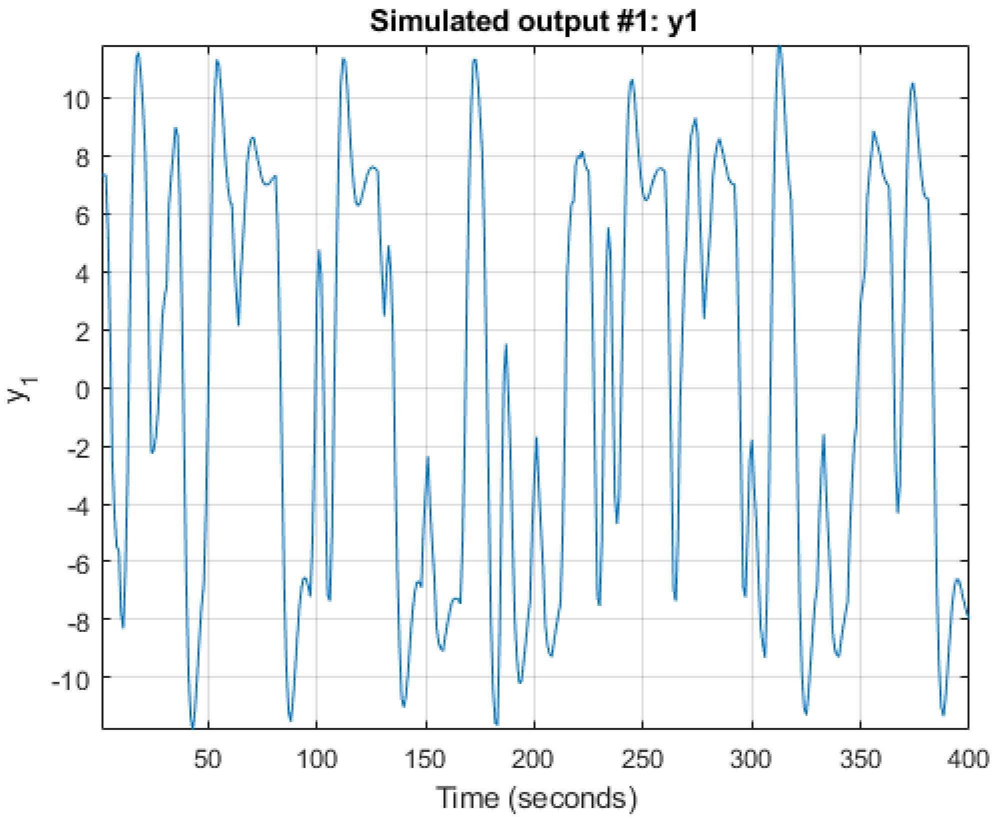
```
M = nlarx(z2,[4 3 2],idWaveletNetwork);
```

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.

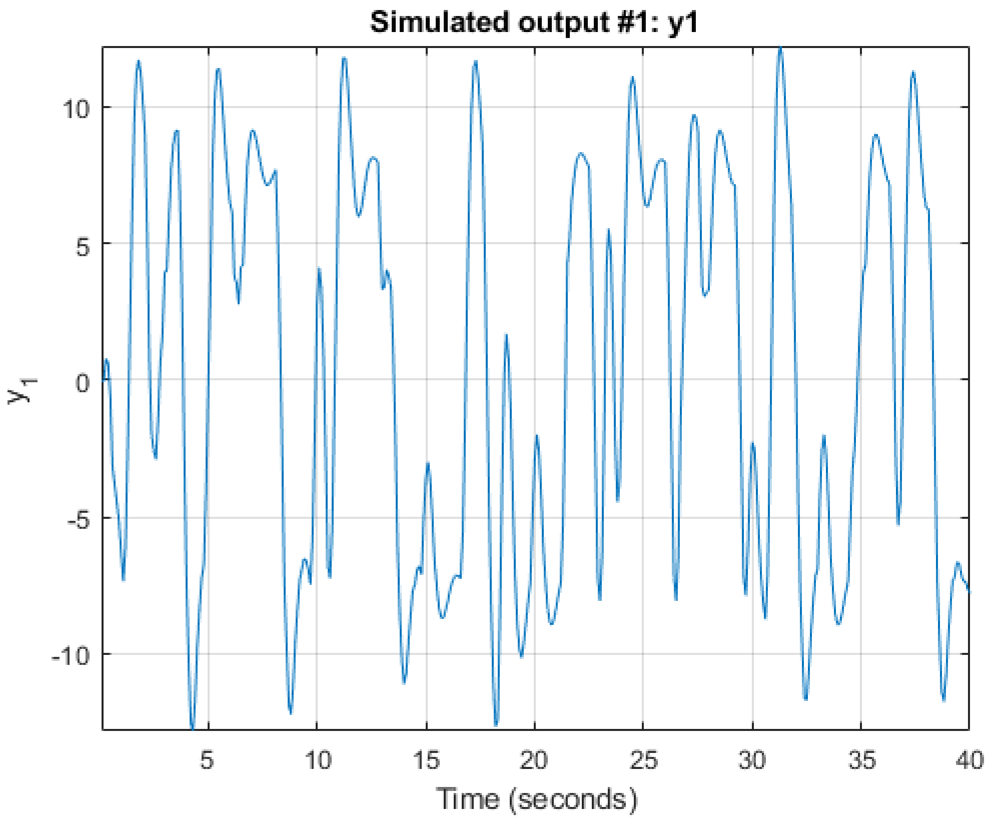
```
M = nlhw(z2,[4 3 2],[],idPiecewiseLinear);
```

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.

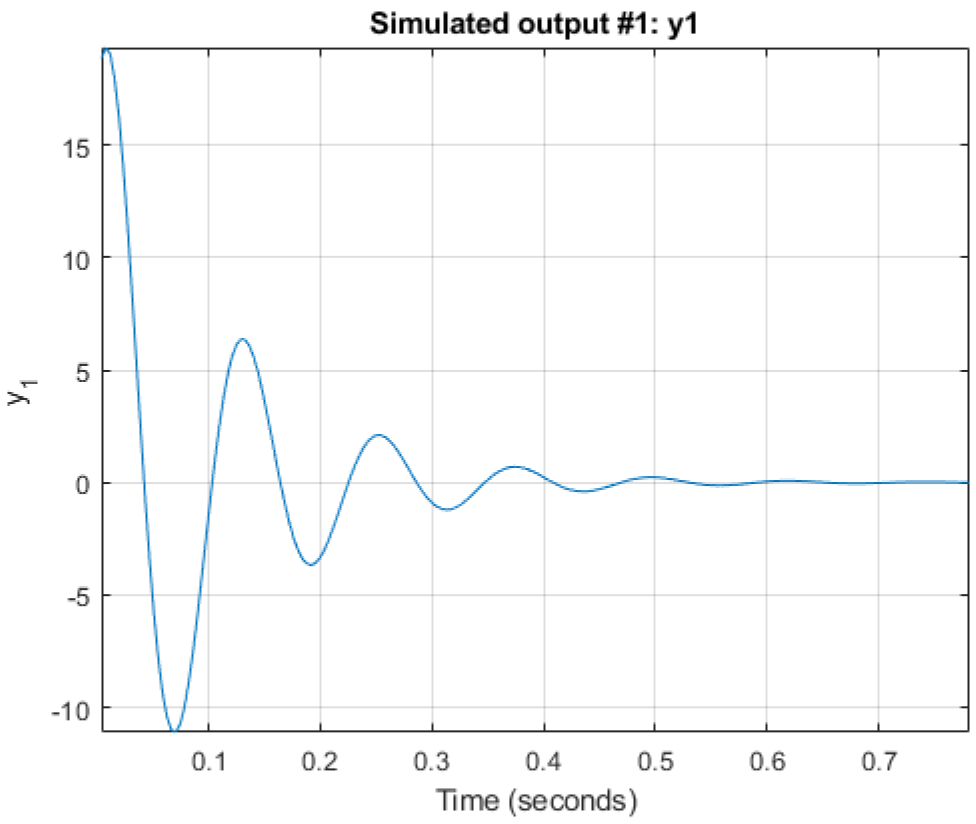
```
data = iddata([],zeros(200,0),z9.Ts);
```

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)
```



Understand Use of Historical Data for Model Simulation

Use historical input-output data as a proxy for initial conditions when simulating your model. You first simulate using the `sim` command and specify the historical data using the `simOptions` option set. You then reproduce the simulated output by manually mapping the historical data to initial states.

Load a two-input, one-output data set.

```
load iddata7 z7
```

Identify a fifth-order state-space model using the data.

```
sys = n4sid(z7,5);
```

Split the data set into two parts.

```
zA = z7(1:15);
zB = z7(16:end);
```

Simulate the model using the input signal in `zB`.

```
uSim = zB;
```

Simulation requires initial conditions. The signal values in `zA` are the historical data, that is, they are the input and output values for the time immediately preceding data in `zB`. Use `zA` as a proxy for the required initial conditions.

```
IO = struct('Input',zA.InputData,'Output',zA.OutputData);
opt = simOptions('InitialCondition',IO);
```

Simulate the model.

```
ysim = sim(sys,uSim,opt);
```

Now reproduce the output by manually mapping the historical data to initial states of `sys`. To do so, use the `data2state` command.

```
xf = data2state(sys,zA);
```

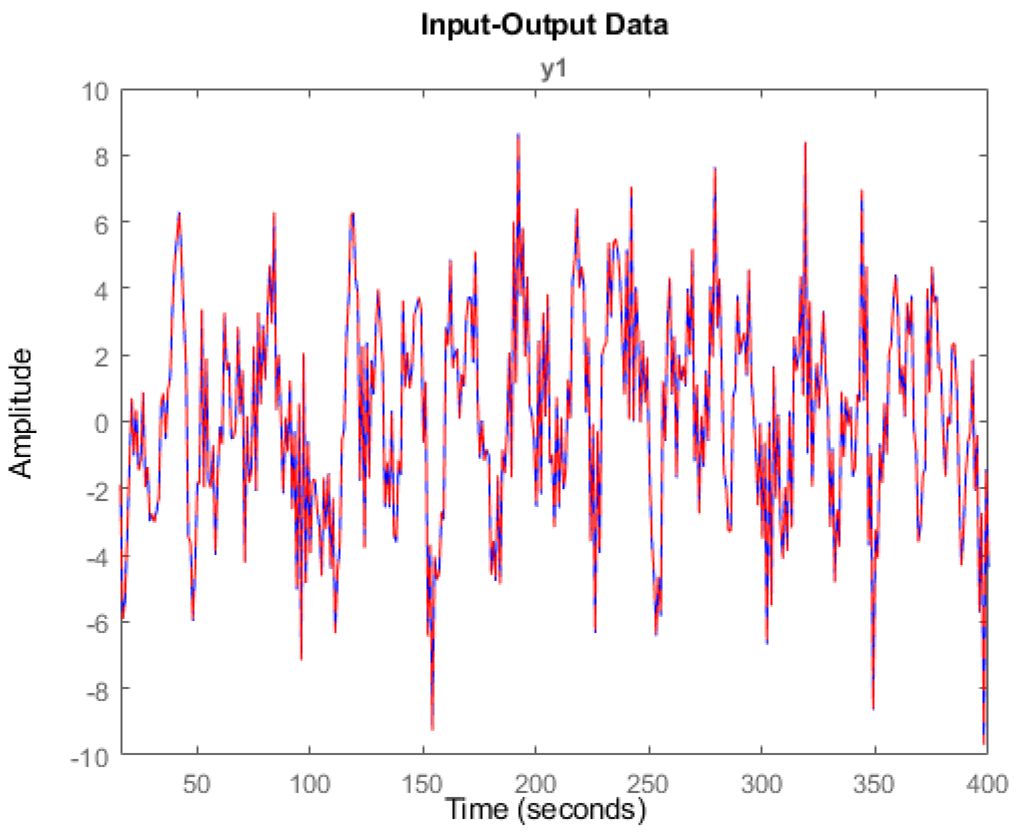
`xf` contains the state values of `sys` at the time instant immediately after the most recent data sample in `zA`.

Simulate the system using `xf` as the initial states.

```
opt2 = simOptions('InitialCondition',xf);
ysim2 = sim(sys,uSim,opt2);
```

Plot the output of the `sim` command `ysim` and the manually computed results `ysim2`.

```
plot(ysim,'b',ysim2,'--r')
```



`ysim2` is the same as `ysim`.

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 | matrix

Simulated response for `sys`, returned as an `iddata` object or matrix, depending on how you specify `udata`. For example, if `udata` is an `iddata` object, then so is `y`.

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
double matrix

Estimated standard deviation of the simulated response for linear models or nonlinear grey-box models, returned as an *Ns*-by-*Ny* matrix, where *Ns* is the number of samples and *Ny* is the number of outputs. The software computes the standard deviation by taking into account the model parameter covariance, initial state covariance, and additive noise covariance. The additive noise covariance is stored in the `NoiseVariance` property of the model.

`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 an `idss`, `idgrey`, or `idnlgrey` model. 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. The software computes the standard deviation by taking into account the model parameter covariance, initial state covariance, and additive noise covariance. The additive noise covariance is stored in the `NoiseVariance` property of the model.

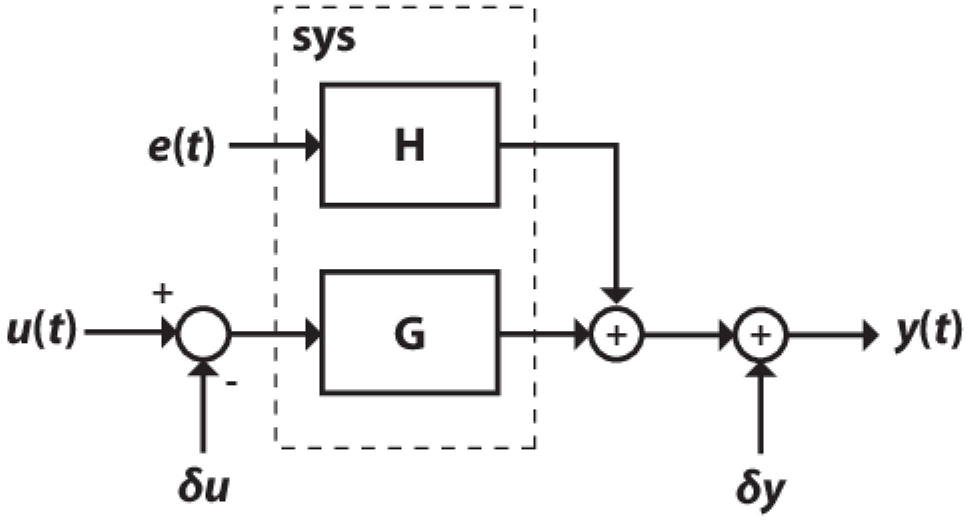
`x_sd` is only relevant if `sys` is an `idss`, `idgrey`, or `idnlgrey` model. If `sys` is not a state-space model, `x_sd` is returned as `[]`.

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 1-1512.

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.

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`.

See Also
`simOptions` | `simsd` | `lsim` | `step` | `compare` | `predict` | `forecast` | `idinput` | `findstates`

Topics
"Simulate and Predict Identified Model Output"
"Simulation and Prediction at the Command Line"

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;
```

Use Historical Data to Specify Initial Conditions for Model Simulation

Use historical input-output data as a proxy for initial conditions when simulating your model.

Load a two-input, one-output data set.

```
load iddata7 z7
```

Identify a fifth-order state-space model using the data.

```
sys = n4sid(z7, 5);
```

Split the data set into two parts.

```
zA = z7(1:15);
zB = z7(16:end);
```

Simulate the model using the input signal in `zB`.

```
uSim = zB;
```

Simulation requires initial conditions. The signal values in `zA` are the historical data, that is, they are the input and output values for the time immediately preceding data in `zB`. Use `zA` as a proxy for the required initial conditions.

```
IO = struct('Input',zA.InputData,'Output',zA.OutputData);
opt = simOptions('InitialCondition',IO);
```

Simulate the model.

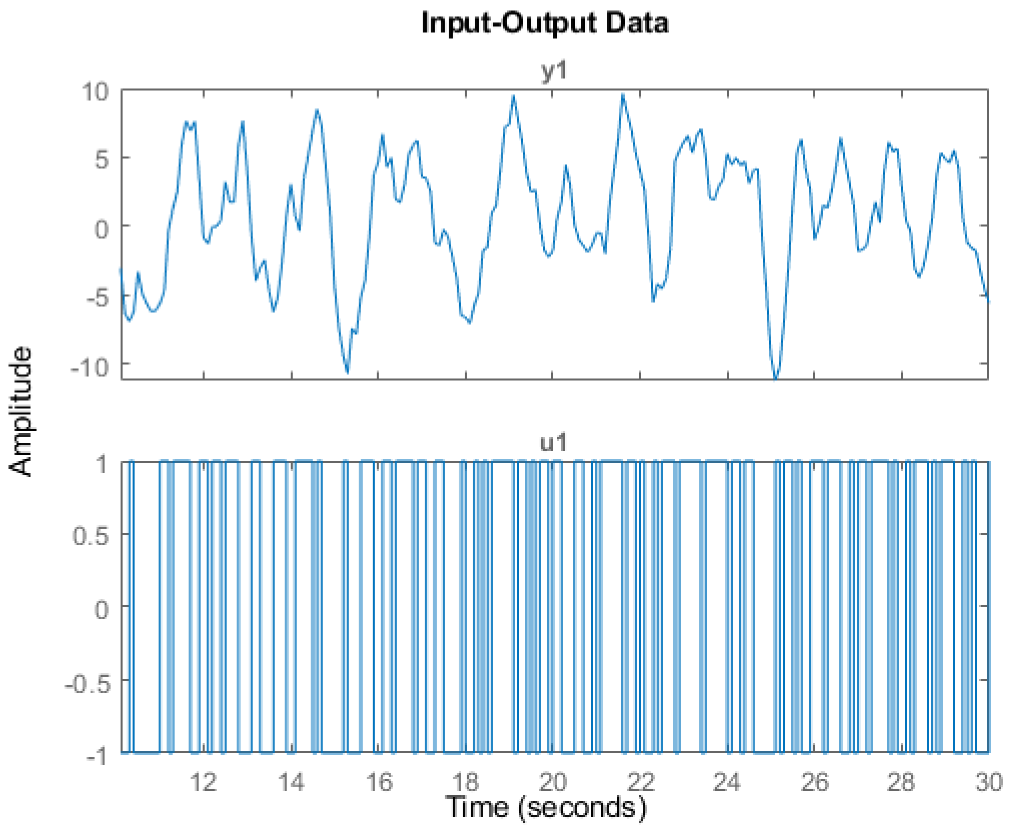
```
ysim = sim(sys,uSim,opt);
```

To understand how the past data is mapped to the initial states of the model, see "Understand Use of Historical Data for Model Simulation" on page 1-1516.

Obtain and Apply Estimated Initial Conditions

Load and plot the data.

```
load iddata1ic z1i
plot(z1i)
```



Examine the initial value of the output data `y(1)`.

```
ystart = z1i.y(1)
```

```
ystart = -3.0491
```

The measured output does not start at 0.

Estimate a second-order transfer function `sys` and return the estimated initial condition `ic`.

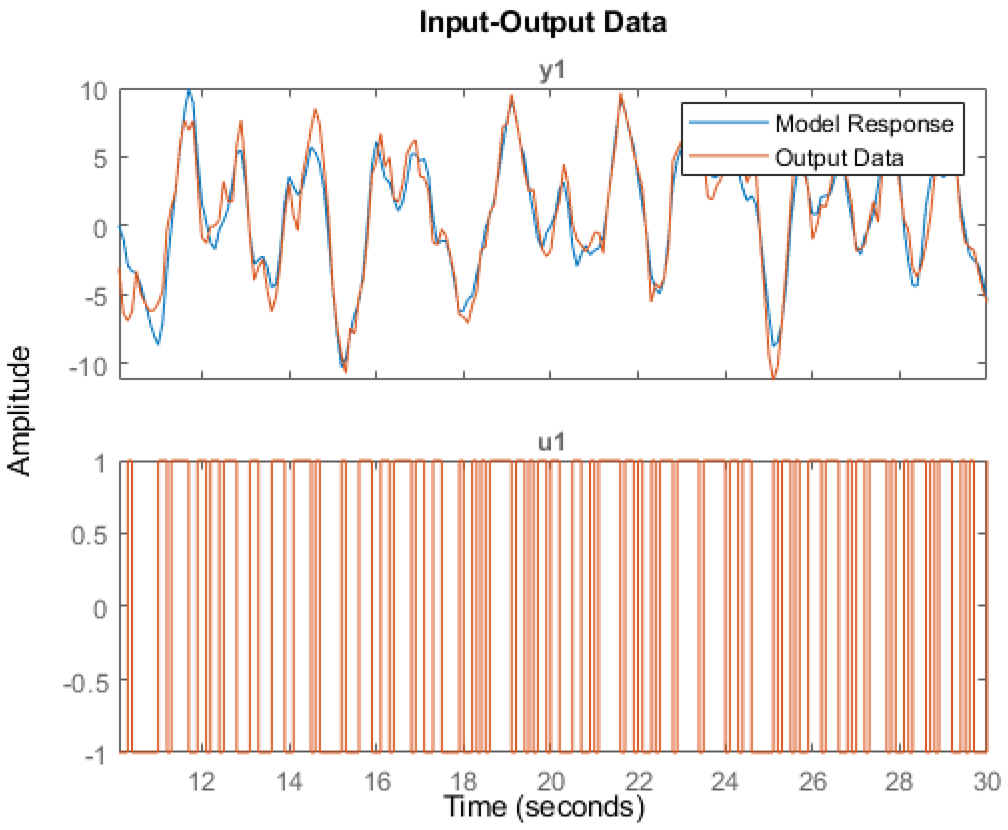
```
[sys,ic] = tfest(z1i,2,1);
ic
```

```
ic = 
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [0.2957 5.2441]
    Ts: 0
```

`ic` is an `initialCondition` object that encapsulates the free response of `sys`, in state-space form, to the initial state vector in `X0`.

Simulate `sys` using the estimation data but without incorporating the initial conditions. Plot the simulated output with the measured output.

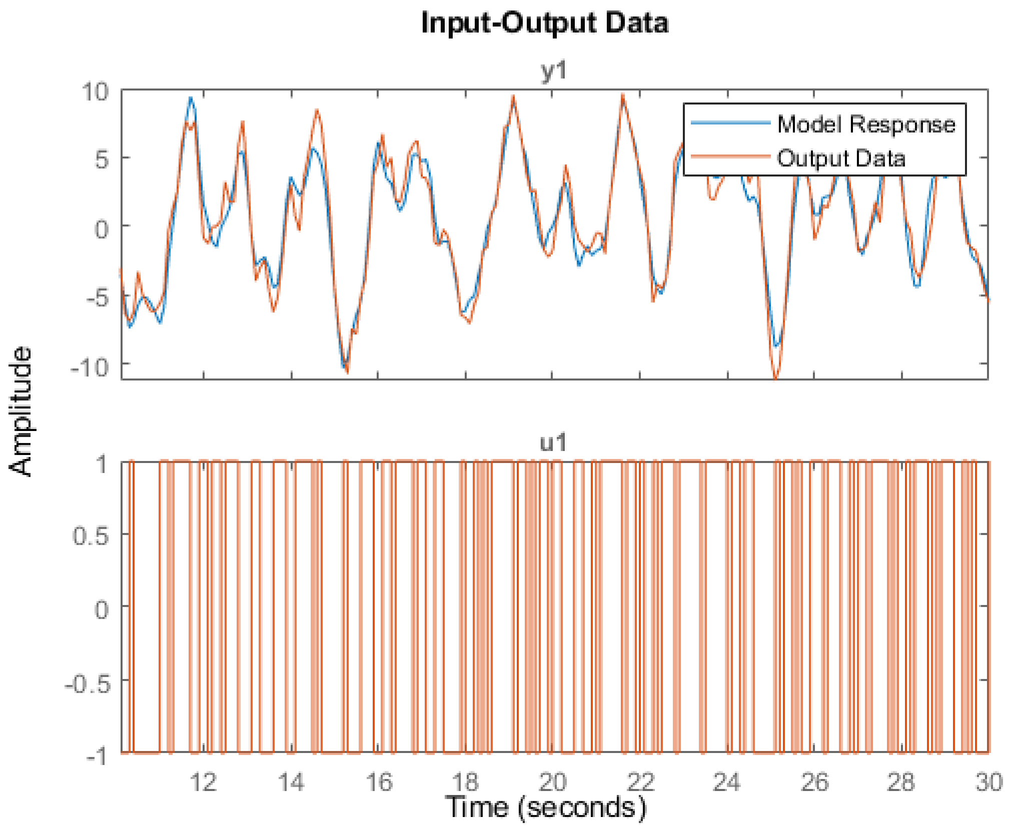
```
y_no_ic = sim(sys,z1i);
plot(y_no_ic,z1i)
legend('Model Response','Output Data')
```



The measured and simulated outputs do not agree at the beginning of the simulation.

Incorporate the initial condition into the `simOptions` option set.

```
opt = simOptions('InitialCondition',ic);
y_ic = sim(sys,z1i,opt);
plot(y_ic,z1i)
legend('Model Response','Output Data')
```



The simulation combines the model response to the input signal with the free response to the initial condition. The measured and simulated outputs now have better agreement at the beginning of the simulation. This initial condition is valid only for the estimation data `z1i`.

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 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

`[]` (default) | column vector | matrix | `initialCondition` object | object array | 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 multiexperiment 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.

- `initialCondition` object — `initialCondition` object that represents a model of the free response of the system to initial conditions. For multiexperiment data, specify a 1-by-N_e array of objects, where N_e is the number of experiments.

  Use this option for linear models only. For an example, see "Obtain and Apply Estimated Initial Conditions" on page 1-1523.
- 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 `[]`. The number of rows must be greater than or equal to the model order.
`Output`	Output history, specified as a matrix with *Ny* columns, where *Ny* is the number of output channels. The number of rows must be greater than or equal to the model order.

  For an example, see "Use Historical Data to Specify Initial Conditions for Model Simulation" on page 1-1523.

  For multiexperiment data, configure the initial conditions separately for each experiment by specifying `InitialCondition` as a structure array with *Ne* elements. To specify the same initial conditions for all experiments, use a single structure.

  The software uses `data2state` to map the historical data to states. If your model is not `idss`, `idgrey`, `idnlgrey`, or `idnlarx`, the software first converts the model to its state-space representation and then maps the data to states. If conversion of your model to `idss` is not possible, the estimated states are returned empty.
- `'model'` — Use this option for `idnlgrey` models only. The software sets the initial states to the values specified in the `sys.InitialStates` property of the model `sys`.
- `[]` — Corresponds to zero initial conditions for all models except `idnlgrey`. For `idnlgrey` models, the software treats `[]` as `'model'` and specifies the initial states as `sys.InitialStates`.

`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 multiexperiment 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,udata)
simsd(sys,udata,N)
simsd(sys,udata,N,opt)

y = simsd( ___ )
[y,y_sd] = simsd( ___ )
```

Description

`simsd` simulates linear models using the Monte Carlo method. The command performs multiple simulations using different values of the uncertain parameters of the model, and different realizations of additive noise and simulation initial conditions. `simsd` uses Monte Carlo techniques to generate response uncertainty, whereas `sim` generates the uncertainty using the Gauss Approximation Formula.

`simsd(sys,udata)` simulates and plots the response of 10 perturbed realizations of the identified model `sys`. Simulation input data `udata` is used to compute the simulated response.

The parameters of the perturbed realizations of `sys` are consistent with the parameter covariance of the original model, `sys`. If `sys` does not contain parameter covariance information, the 10 simulated responses are identical. For information about how the parameter covariance information is used to generate the perturbed models, see “Generating Perturbations of Identified Model” on page 1-1537.

`simsd(sys,udata,N)` simulates and plots the response of `N` perturbed realizations of the identified model `sys`.

`simsd(sys,udata,N,opt)` simulates the system response using the simulation behavior specified in the option set `opt`. Use `opt` to specify uncertainties in the initial conditions and include the effect of additive disturbances.

The simulated responses are all identical if `sys` does not contain parameter covariance information, and you do not specify additive noise or covariance values for initial states. You specify these values in the `AddNoise` and `X0Covariance` options of `opt`.

`y = simsd( ___ )` returns the `N` simulation results in `y` as a cell array. No simulated response plot is produced. Use with any of the input argument combinations in the previous syntaxes.

`[y,y_sd] = simsd( ___ )` also returns the estimated standard deviation `y_sd` for the simulated response.

Examples

Simulate State-Space Model Using Monte Carlo Method

Load the estimation data.

```
load iddata1 z1
```

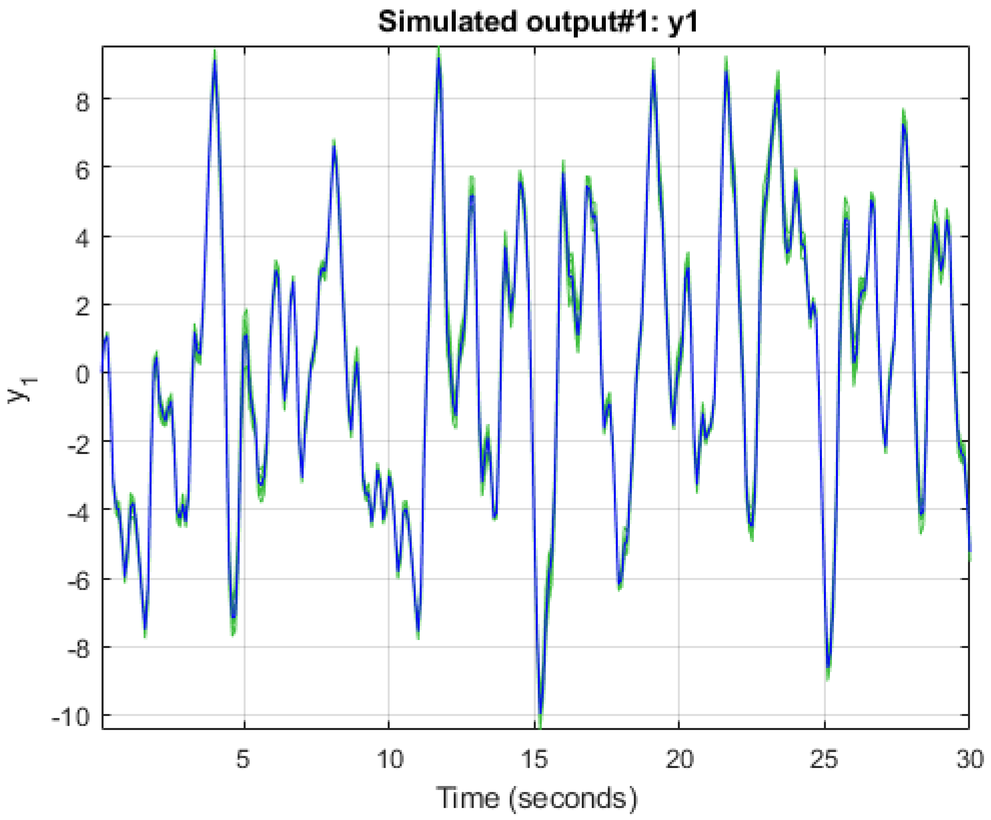
`z1` is an `iddata` object that stores the input-output estimation data.

Estimate a third-order state-space model.

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

Simulate the response of the estimated model using the Monte Carlo method and input estimation data, and plot the response.

```
simsd(sys,z1);
```



The blue line plots the simulated response of the original nominal model `sys`. The green lines plot the simulated response of 10 perturbed realizations of `sys`.

Simulate Estimated Model Using Monte Carlo Method

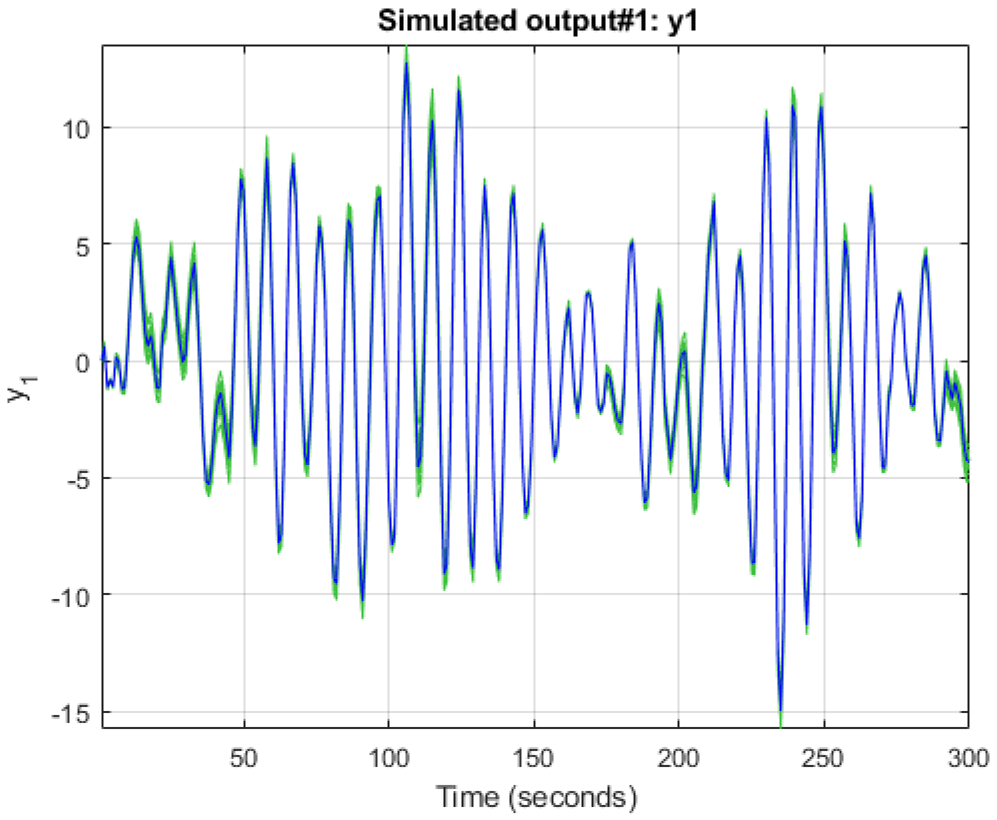
Simulate an estimated model using the Monte Carlo method for a specified number of model perturbations.

Estimate a second-order state-space model using estimation data. Obtain `sys` in the observability canonical form.

```
load iddata3 z3
sys = ssest(z3,2,'Form','canonical');
```

Compute the simulated response of the estimated model using the Monte Carlo method, and plot the responses. Specify the number of random model perturbations as 20.

```
N = 20;
simsd(sys,z3,N)
```



The blue line plots the simulated response of the original nominal model `sys`. The green lines plot the simulated response of the 20 perturbed realizations of `sys`.

You can also obtain the simulated response for each perturbation of `sys`. No plot is generated when you use this syntax.

```
y = simsd(sys,z3,N);
```

`y` is the simulated response, returned as a cell array of `N+1` elements. `y{1}` contains the nominal response for `sys`. The remaining elements contain the simulated response for the `N` perturbed realizations.

Simulate Time Series Model Using Monte Carlo Method

Load time series data.

```
load iddata9 z9
```

`z9` is an `iddata` object with 200 output data samples and no inputs.

Estimate a sixth-order AR model using the least-squares algorithm.

```
sys = ar(z9,6,'ls');
```

For time series data, specify the desired simulation length, `Ns` = 200 using an `Ns`-by-0 input data set.

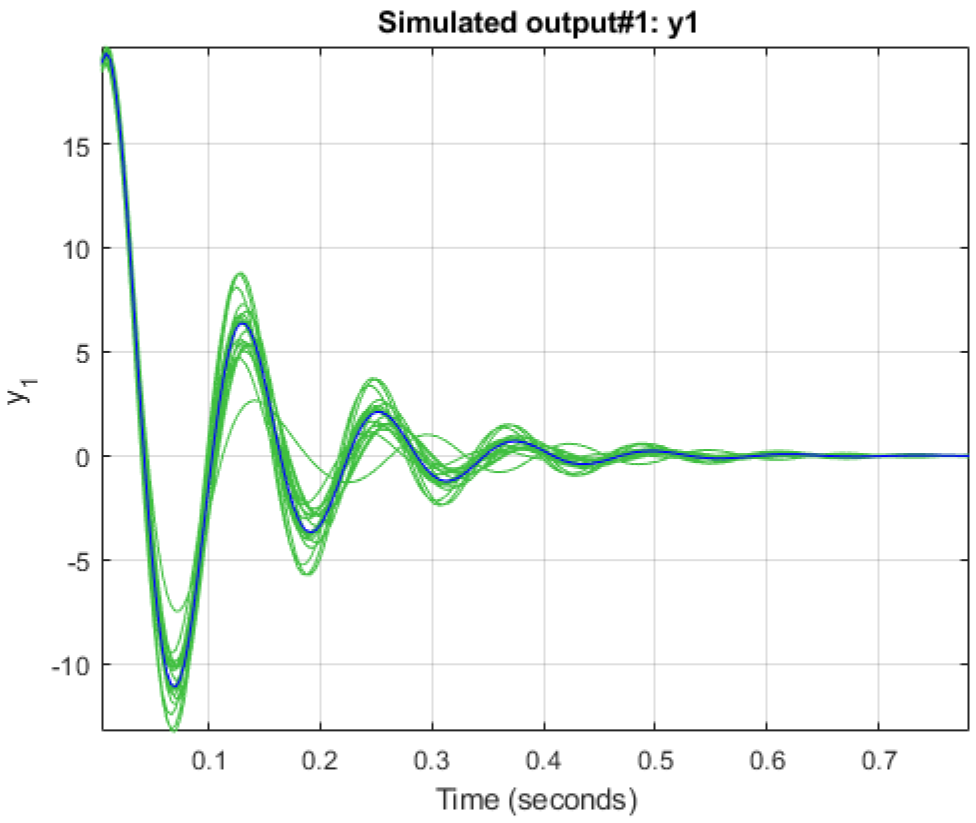
```
data = iddata([],zeros(200,0),z9.Ts);
```

Set the initial conditions to use the initial samples of the time series as historical output samples. The past data is mapped to the initial states of each perturbed system individually.

```
IC = struct('Input',[],'Output',z9.y(1:6));
opt = simsdOptions('InitialCondition',IC);
```

Simulate the model using Monte Carlo method and specified initial conditions. Specify the number of random model perturbations as 20.

```
simsd(sys,data,20,opt)
```



The blue line plots the simulated response of the original nominal model `sys`. The green lines plot the simulated response of the 20 perturbed realizations of `sys`.

Study Effect of Initial Condition Uncertainty on Model Response

Load data, and split it into estimation and simulation data.

```
load iddata3
ze = z3(1:200);
zsim = z3(201:256);
```

Estimate a second-order state-space model `sys` using estimation data. Specify that no parameter covariance data is generated. Obtain `sys` in the observability canonical form.

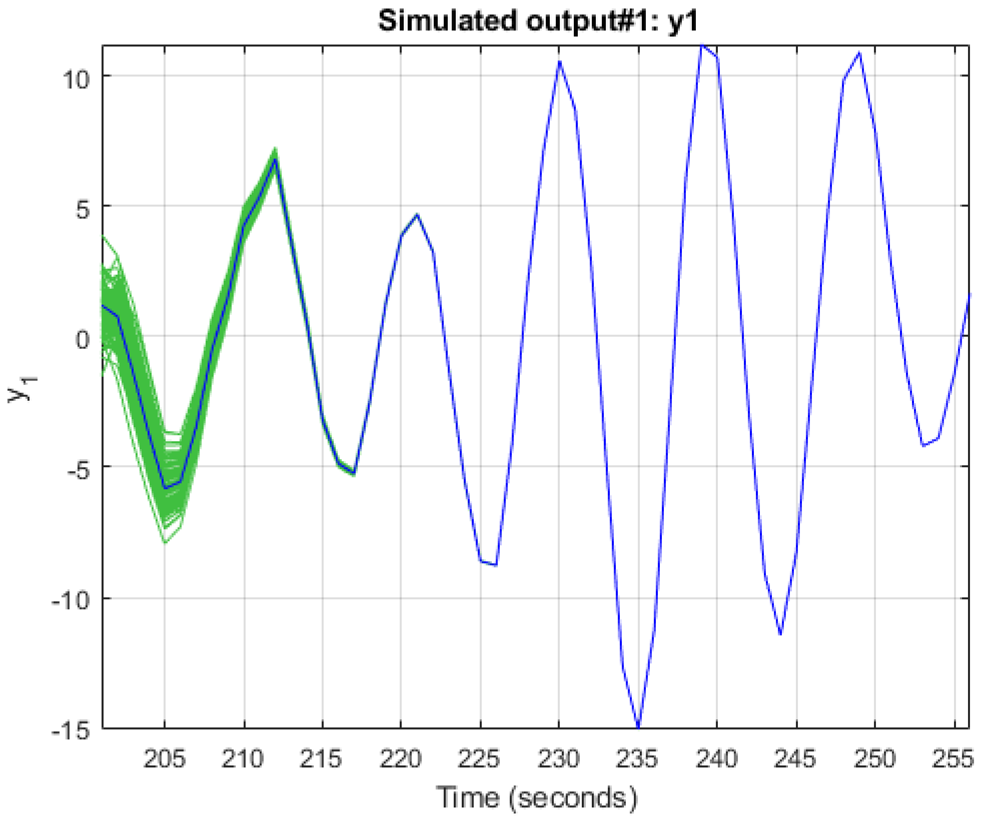
```
opt = ssestOptions('EstimateCovariance',false);
sys = ssest(ze,2,'Form','canonical',opt);
```

Set the initial conditions for simulating the estimated model. Specify initial state values `x0` for the two states and also the covariance of initial state values `x0Cov`. The covariance is specified as a 2-by-2 matrix because there are two states.

```
x0 = [1.2; -2.4];
x0Cov = [0.86 -0.39; -0.39 1.42];
opt = simsdOptions('InitialCondition',x0,'X0Covariance',x0Cov);
```

Simulate the model using Monte Carlo method and specified initial conditions. Specify the number of random model perturbations as 100.

```
simsd(sys,zsim,100,opt)
```



The blue line plots the simulated response of the original nominal model `sys`. The green lines plot the simulated response of the 100 perturbed realizations of `sys`. The software uses a different realization of the initial states to simulate each perturbed model. Initial states are drawn from a Gaussian distribution with mean `InitialCondition` and covariance `X0Covariance`.

Study Effect of Additive Disturbance on Response Uncertainty

Load the estimation data.

```
load iddata1 z1
```

`z1` is an `iddata` object that stores 300 input-output estimation data samples.

Estimate a second-order state-space model using the estimation data.

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

Create a default option set for `simsd`, and modify the option set to add noise.

```
opt = simsdOptions;
opt.AddNoise = true;
```

Compute the simulated response of the estimated model using the Monte Carlo method. Specify the number of random model perturbations as 20, and simulate the model using the specified option set.

```
[y,y_sd] = simsd(sys,z1,20,opt);
```

`y` is the simulated response, returned as a cell array of 21 elements. `y{1}` contains the nominal, noise-free response for `sys`. The remaining elements contain the simulated response for the 20 perturbed realizations of `sys` with additive disturbances added to each response.

`y_sd` is the estimated standard deviation of simulated response, returned as an `iddata` object with no inputs. The standard deviations are computed from the 21 simulated outputs. To access the standard deviation, use `y_sd.OutputData`.

Input Arguments

sys — Model to be simulated
parametric linear identified model

Model to be simulated, specified as one of the following parametric linear identified models: `idtf`, `idproc`, `idpoly`, `idss`, or `idgrey`.

To generate the set of simulated responses, the software perturbs the parameters of `sys` in a way that is consistent with the parameter covariance information. Use `getcov` to examine the parameter uncertainty for `sys`. For information about how the perturbed models are generated from `sys`, see `rsample`.

The simulated responses are all identical if `sys` does not contain parameter covariance information and you do not specify additive noise or covariance values for initial states. You specify these values in the `AddNoise` and `X0Covariance` options of `opt`.

udata — Simulation input data
`iddata` object | matrix

Simulation input data, specified as one of the following:

- `iddata` object — Input data can be either time-domain or frequency-domain. The software uses only the input channels of the `iddata` object.

  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 each of the `N` perturbed realizations of `sys`. For example, to simulate 100 output samples, specify `udata` as follows.

  ```
  udata = iddata([],zeros(100,0),Ts);
  ```

  For an example, see “Simulate Time Series Model Using Monte Carlo Method” on page 1-1532.
- matrix — For simulation of discrete-time systems using time-domain data only. Columns of the matrix correspond to each input channel.

If you do not have data from an experiment, use idinput to generate signals with various characteristics.

N — Number of perturbed realizations
`10` (default) | positive integer

Number of perturbed realizations of `sys` to be simulated, specified as a positive integer.

opt — Simulation options
simsdOptions option set

Simulation options for simulating models using Monte Carlo methods, specified as a `simsdOptions` option set. You can use this option set to specify:

- Input and output signal offsets — Specify an offset to remove from the input signal and an offset to add to the response of `sys`.
- Initial condition handling — Specify initial conditions for simulation and their covariance. For state-space and linear grey-box models (`idss` and `idgrey`), if you want to simulate the effect of uncertainty in initial states, set the `InitialCondition` option to a double vector, and specify its covariance using the `X0Covariance` option. For an example, see "Study Effect of Initial Condition Uncertainty on Model Response" on page 1-1534.
- Addition of noise to simulated data — If you want to include the influence of additive disturbances, specify the `AddNoise` option as `true`. For an example, see "Study Effect of Additive Disturbance on Response Uncertainty" on page 1-1535.

Output Arguments

y — Simulated response
cell array

Simulated response, returned as a cell array of `N+1` elements. `y{1}` contains the nominal response for `sys`. The remaining elements contain the simulated response for the `N` perturbed realizations.

The command performs multiple simulations using different values of the uncertain parameters of the model, and different realizations of additive noise and simulation initial conditions. Thus, the simulated responses are all identical if `sys` does not contain parameter covariance information and you do not specify additive noise and covariance values for initial states in `opt`.

y_sd — Estimated standard deviation of simulated response
iddata object

Estimated standard deviation of simulated response, returned as an `iddata` object. The standard deviation is computed as the sample standard deviation of the *y* ensemble:

$$y\_sd = \sqrt{\frac{1}{N}\sum_{i=2}^{N+1}(y\{1\} - y\{i\})^2}$$

Here `y{1}` is the nominal response for `sys`, and `y{i}` (`i = 2:N+1`) are the simulated responses for the `N` perturbed realizations of `sys`.

More About

Generating Perturbations of Identified Model

The software generates `N` perturbations of the identified model `sys` and then simulates the response of each of these perturbations. The parameters of the perturbed realizations of `sys` are consistent with the parameter covariance of the original model `sys`. The parameter covariance of `sys` gives information about the distribution of the parameters. However, for some parameter values, the resulting perturbed systems can be unstable. To reduce the probability of generation of unrealistic systems, the software prescales the parameter covariance.

If *Δp* is the parameter covariance for the parameters *p* of `sys`, then the simulated output *f(p+Δp)* of a perturbed model as a first-order approximation is:

$$f(p + \Delta p) = f(p) + \frac{\partial f}{\partial p}\Delta p$$

The `simsd` command first scales *Δp* by a scaling factor *s* (approximately 0.1%) to generate perturbed systems with parameters *(p+sΔp)*. The command then computes *f(p+sΔp)*, the simulated response of these perturbed systems. Where,

$$f(p + s\Delta p) = f(p) + s\frac{\partial f}{\partial p}\Delta p$$

The command then computes the simulated response *f(p+Δp)* as:

$$f(p + \Delta p) = f(p) + \frac{1}{s}(f(p + s\Delta p) - f(p))$$

Note This scaling is not applied to the free delays of `idproc` or `idtf` models.

If you specify the `AddNoise` option of `simsdOptions` as `true`, the software adds different realizations of the noise sequence to the noise-free responses of the perturbed system. The realizations of the noise sequence are consistent with the noise component of the model.

For state-space models, if you specify the covariance of initial state values in `X0Covariance` option of `simsdOptions`, different realizations of the initial states are used to simulate each perturbed model. Initial states are drawn from a Gaussian distribution with mean `InitialCondition` and covariance `X0Covariance`.

See Also

`simsdOptions` | `getcov` | `sim` | `rsample` | `showConfidence`

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 a default option set.

```
opt = simsdOptions;
```

Modify the option set to add noise to the data.

```
opt.AddNoise = true;
```

When you use this option set and `simsd` command to simulate the response of a model `sys`. The command returns the perturbed realizations of `sys` with additive disturbances added to each response.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `opt = simsdOptions('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 `X0` 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. You can also specify the covariance of the initial state vector in `X0Covariance`.
- 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 `[]`. The number of rows must be greater than or equal to the model order.
`Output`	Output history, specified as a matrix with *Ny* columns, where *Ny* is the number of output channels. The number of rows must be greater than or equal to the model order.

  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.

  The software uses `data2state` to map the historical data to states. If your model is not `idss` or `idgrey`, the software first converts the model to its state-space representation and then maps the data to states. If conversion of your model to `idss` is not possible, the estimated states are returned empty.

`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. For the k^{th} experiment, `X0Covariance(:,:,k)` specifies the covariance of initial states `X0(:,k)`.
- `[]` — No uncertainty in the initial states.

Use this option for state-space models (`idss` and `idgrey`) when `'InitialCondition'` is specified as a numerical column vector `X0`. When you specify this option, the software uses a different realization of the initial states to simulate each perturbed model. Initial states are drawn from a Gaussian distribution with mean `InitialCondition` and covariance `X0Covariance`.

`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. Set `NoiseModel` to `true` to study the effect of additive disturbances on the response. A different realization of the noise sequence, consistent with the noise component of the perturbed system, is added to the noise-free response of that system.

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

Query Dimensions of Model Array

Create a 3-by-1 model array of random state-space models with 3 outputs, 2 inputs, and 5 states.

```
sys = rss(5,3,2,3);
```

Verify the size of the model array.

```
size(sys)
```

```
3x1 array of state-space models.
Each model has 3 outputs, 2 inputs, and 5 states.
```

Query Dimensions of Identified Model

Create a 2-input 2-output continuous-time process model with identifiable parameters.

```
type = {'p1d','p2';'p3uz','p0'};
sys = idproc(type);
```

Each element of the `type` cell array describes the model structure for the corresponding input-output pair.

Query the input-output dimensions and number of free parameters in the model.

```
size(sys)
```

```
Process model with 2 outputs, 2 inputs and 12 free parameters.
```

See Also

`isempty` | `issiso` | `ndims` | `nparams`

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 the frequency response, along with uncertainty, and the noise spectrum from time- or frequency-domain data `data`. If `data` is a time series, `spa(data)` returns the output power spectrum along with uncertainty. `spa` computes the spectra at 128 equally spaced frequency values between 0 (excluded) and π, using a Hann window.

`G = spa(data,winSize,freq)` estimates the frequency response at the frequencies contained in `freq`, using a Hann window with size `winSize`.

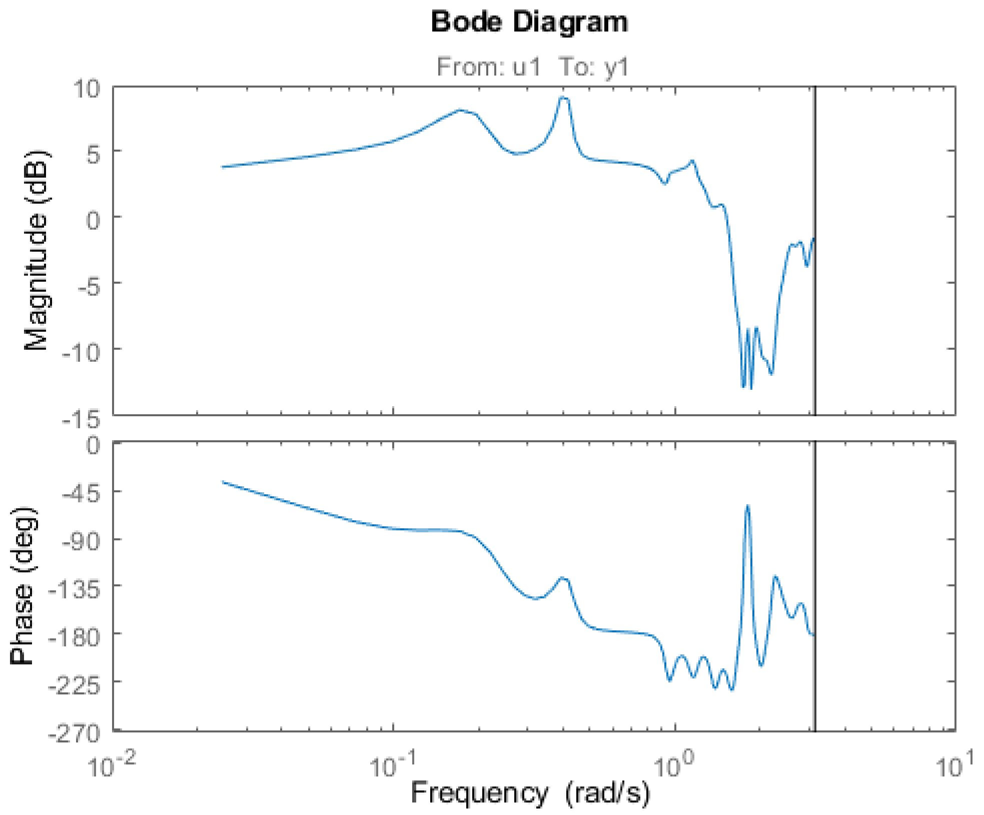
`G = spa(data,winSize,freq,maxSize)` splits the input/output data into segments, each segment containing fewer than `maxSize` elements. Use this syntax to improve computational performance.

Examples

Estimate Frequency Response

Estimate the frequency response for the input/output data in the `iddata` object `z3`. Use the default fixed resolution of 128 equally spaced logarithmic frequency values between 0 (excluded) and π.

```
load iddata3 z3;
g = spa(z3);
bode(g)
```



Estimate Frequency Response at Specified Frequencies

Generate the logarithmically spaced vector `f`.

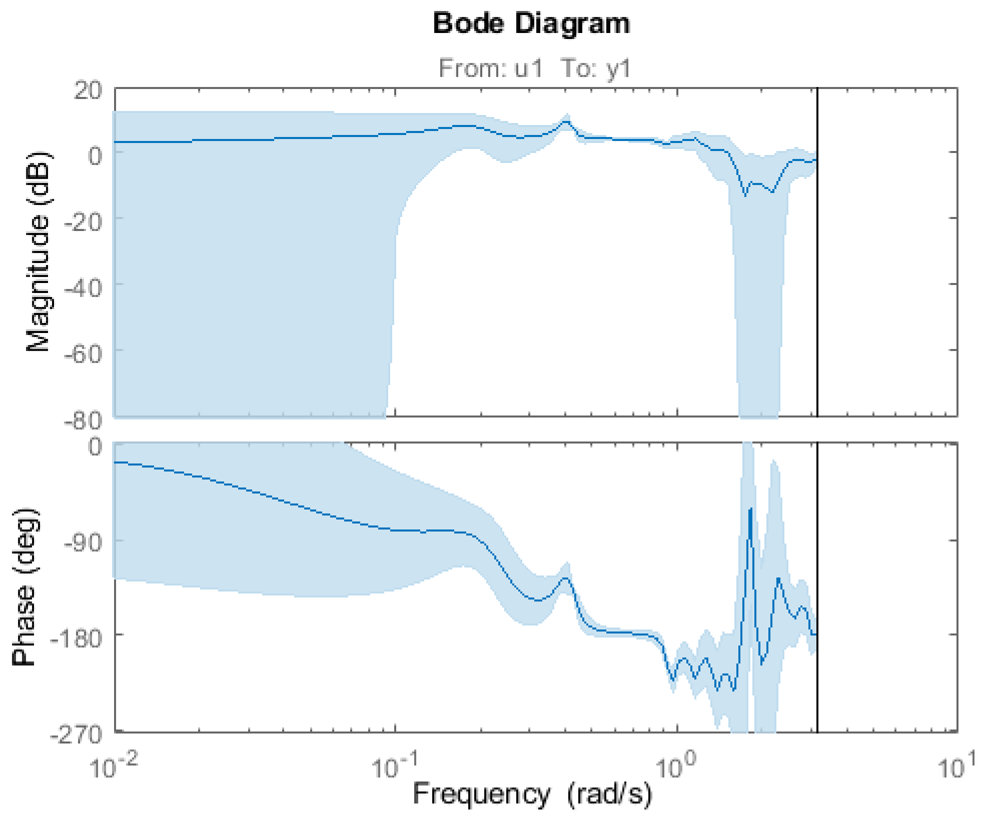
```
f = logspace(-2,pi,128);
```

Estimate the frequency response for the input/output data `z3`. Specify the window size as `[]` to obtain the default lag window size.

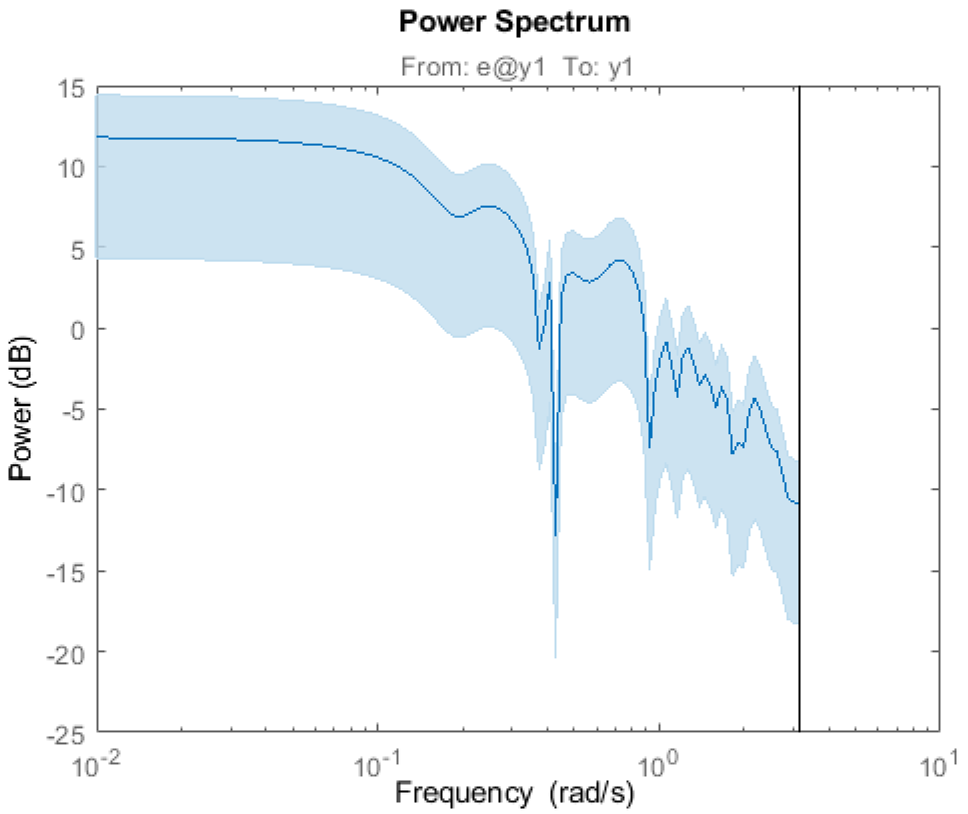
```
load iddata3 z3;
g = spa(z3,[],f);
```

Plot the Bode response and disturbance spectrum with a confidence interval of 3 standard deviations.

```
h = bodeplot(g);
showConfidence(h,3)
```



```
figure
h = spectrumplot(g);
showConfidence(h,3)
```



Input Arguments

data — Input/output data
iddata object | idfrd object

Input/output data, specified as an iddata object or an idfrd object that can contain complex values. data can also contain time series data with only output.

winSize — Window size
[] (default) | scalar integer

Hann window size, also known as lag size, specified as a scalar integer. By default, the function sets the window size to min(length(data)/10,30).

freq — Frequencies
row vector

Frequencies at which to estimate spectral response, specified as a row vector in units of rad/TimeUnit, where TimeUnit refers to the TimeUnit property of data. By default, the function sets freq to a vector of 128 values in the range (0,π], evenly spaced logarithmically. For discrete-time models, set freq within the Nyquist frequency bound.

maxSize — Maximum segment size
250e3 (default) | positive integer

Maximum size of segments within `data`, specified as a positive integer. If you omit this argument, the function performs estimation using the full data set in `data` rather than segmenting the data.

Output Arguments

`G` — Frequency response and noise spectrum

`idfrd` object

Frequency response with uncertainty and noise spectrum, specified 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 `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.
`windowSize`	Size of the Hann window.
`DataUsed`	Attributes of the data used for estimation. Structure with the following fields.

Field	Description
`Name`	Name of the data set.
`Type`	Data type.
`Length`	Number of data samples.
`Ts`	Sample time. This is equivalent to `data.Ts`.
`InterSample`	Input intersample behavior. One of the following values: • `'zoh'` — 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. The value of `InterSample` has no effect on estimation results for discrete-time models.
`InputOffset`	Offset removed from time-domain input data during estimation.
`OutputOffset`	Offset removed from time-domain output data during estimation.

More About

Frequency Response Function

A 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:

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

Here, $u(t)$ 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* when it is evaluated on the unit circle, $G(q{=}e^{i\omega})$.

Together, $G(q{=}e^{i\omega})$ and the output noise spectrum $\widehat{\Phi}_v(\omega)$ compose the frequency-domain description of the system.

The frequency-response function estimated using the Blackman-Tukey approach is given by the following equation:

$$\widehat{G}_N\left(e^{i\omega}\right) = \frac{\widehat{\Phi}_{yu}(\omega)}{\widehat{\Phi}_u(\omega)}$$

In this case, ^ represents approximate quantities. For a derivation of this equation, see the chapter on nonparametric time- and frequency-domain methods in [1].

Output Noise Spectrum

The output noise spectrum (spectrum of $v(t)$) is given by the following equation:

$$\widehat{\Phi}_v(\omega) = \widehat{\Phi}_y(\omega) - \frac{\left|\widehat{\Phi}_{yu}(\omega)\right|^2}{\widehat{\Phi}_u(\omega)}$$

This equation for the noise spectrum is derived by assuming that the linear relationship $y(t) = G(q)u(t) + v(t)$ holds, that $u(t)$ is independent of $v(t)$, and that the following relationships between the spectra hold:

$$\Phi_y(\omega) = \left|G\left(e^{i\omega}\right)\right|^2 \Phi_u(\omega) + \Phi_v(\omega)$$

$$\Phi_{yu}(\omega) = G\left(e^{i\omega}\right)\Phi_u(\omega)$$

Here, the noise spectrum is given by the following equation:

$$\Phi_v(\omega) \equiv \sum_{\tau = -\infty}^{\infty} R_v(\tau)e^{-iw\tau}$$

$\widehat{\Phi}_{yu}(\omega)$ is the output-input cross-spectrum and $\widehat{\Phi}_u(\omega)$ is the input spectrum.

Alternatively, the disturbance $v(t)$ can be described as filtered white noise:

$$v(t) = H(q)e(t)$$

Here, $e(t)$ 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 Compute the covariances and cross-covariance from $u(t)$ and $y(t)$:

$$\widehat{R}_y(\tau) = \frac{1}{N}\sum_{t=1}^{N} y(t+\tau)y(t)$$

$$\widehat{R}_u(\tau) = \frac{1}{N}\sum_{t=1}^{N} u(t+\tau)u(t)$$

$$\widehat{R}_{yu}(\tau) = \frac{1}{N}\sum_{t=1}^{N} y(t+\tau)u(t)$$

2 Compute the Fourier transforms of the covariances and the cross-covariance:

$$\widehat{\Phi}_y(\omega) = \sum_{\tau=-M}^{M} \widehat{R}_y(\tau)W_M(\tau)e^{-i\omega\tau}$$

$$\widehat{\Phi}_u(\omega) = \sum_{\tau=-M}^{M} \widehat{R}_u(\tau)W_M(\tau)e^{-i\omega\tau}$$

$$\widehat{\Phi}_{yu}(\omega) = \sum_{\tau=-M}^{M} \widehat{R}_{yu}(\tau)W_M(\tau)e^{-i\omega\tau}$$

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 to 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, the operation uses fast Fourier transforms (FFT), which are more efficient than for user-defined frequencies.

Note `M` = γ is in Table 6.1 of [1]. Standard deviations are on pages 184 and 188 in [1].

3 Compute the frequency-response function $\widehat{G}_N\left(e^{i\omega}\right)$ and the output noise spectrum $\widehat{\Phi}_v(\omega)$.

$$\widehat{G}_N\left(e^{i\omega}\right) = \frac{\widehat{\Phi}_{yu}(\omega)}{\widehat{\Phi}_u(\omega)}$$

$$\Phi_v(\omega) \equiv \sum_{\tau = -\infty}^{\infty} R_v(\tau)e^{-iw\tau}$$

`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`.

$$S = \sum_{m = -M}^{M} Ez(t+m)z(t)'W_M(T_s)\exp(-i\omega m)$$

Here, `'` is a complex-conjugate transpose.

References

[1] Ljung, Lennart. *System Identification: Theory for the User*. 2nd ed. Prentice Hall Information and System Sciences Series. Upper Saddle River, NJ: Prentice Hall PTR, 1999.

See Also

`etfe` | `freqresp` | `idfrd` | `spafdr` | `bode` | `spectrum`

Topics

"What is a Frequency-Response Model?"
"Estimate Frequency-Response Models at the Command Line"
"Selecting the Method for Computing Spectral Models"

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 input-to-output frequency response G(ω) and noise spectrum Φ_υ of the general linear model

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

where $\Phi_\upsilon(\omega)$ is the spectrum of $\upsilon(t)$. `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 $\Phi_\upsilon(\omega)$ 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 in rad/`TimeUnit`, where `TimeUnit` refers to the `TimeUnit` property of data, 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, the default range goes from $\frac{2\pi}{NT_s}$ to $\frac{\pi}{T_s}$, where *Ts* is the sample time of data and *N* is the number of data points.

Resolution

The argument `Resol` defines the frequency resolution of the estimates. The resolution (measured in rad/`TimeUnit`) 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/`TimeUnit`), 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`.

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

Plot or return output power spectrum of time series model or disturbance spectrum of linear input-output model

Syntax

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

ps = spectrum(sys,w)
ps = spectrum(sys,w)
[ps,wout] = spectrum(sys)
[ps,wout,sdps] = spectrum(sys)
```

Description

Plot Results

`spectrum(sys)` plots the output power spectrum of an identified time series model `sys` or the disturbance spectrum of an identified input-output model `sys`. The function chooses the frequency range and number of points automatically.

- If `sys` is a time series model, then `sys` represents the system:

  $y(t) = He(t)$

  Here, $e(t)$ is a Gaussian white noise and $y(t)$ is the observed output.

  `spectrum` plots $|H'H|$, scaled by the variance of $e(t)$ and the sample time.
- If `sys` is an input-output model, `sys` represents the system:

  $y(t) = Gu(t) + He(t)$

  Here, $u(t)$ is the measured input, $e(t)$ is Gaussian white noise and $y(t)$ is the observed output.

  In this case, `spectrum` plots the spectrum of the disturbance component $He(t)$.

For discrete-time models with sample time T_s, `spectrum` uses the transformation $z = e^{j\omega T_s}$ to map the unit circle to the real frequency axis. The function plots the spectrum only for frequencies smaller than the Nyquist frequency π/T_s, and uses the default value of 1 time unit when Ts is unspecified.

`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')` uses red for `sys1`, yellow dash markers for `sys2`, and green `x` markers for `sys3`.

Return Results

`ps = spectrum(sys,w)` returns the power spectrum amplitude of `sys` for the specified frequencies, `w`. There is no plot display.

`ps = spectrum(sys,w)` returns the power spectrum amplitude of `sys` for the specified frequencies, `w`.

`[ps,wout] = spectrum(sys)` returns the frequency vector, `wout`, for which the output power spectrum is computed.

`[ps,wout,sdps] = spectrum(sys)` returns the estimated standard deviations of the power spectrum.

Examples

Plot Output Spectrum of Time-Series Model

Load the time-series estimation data.

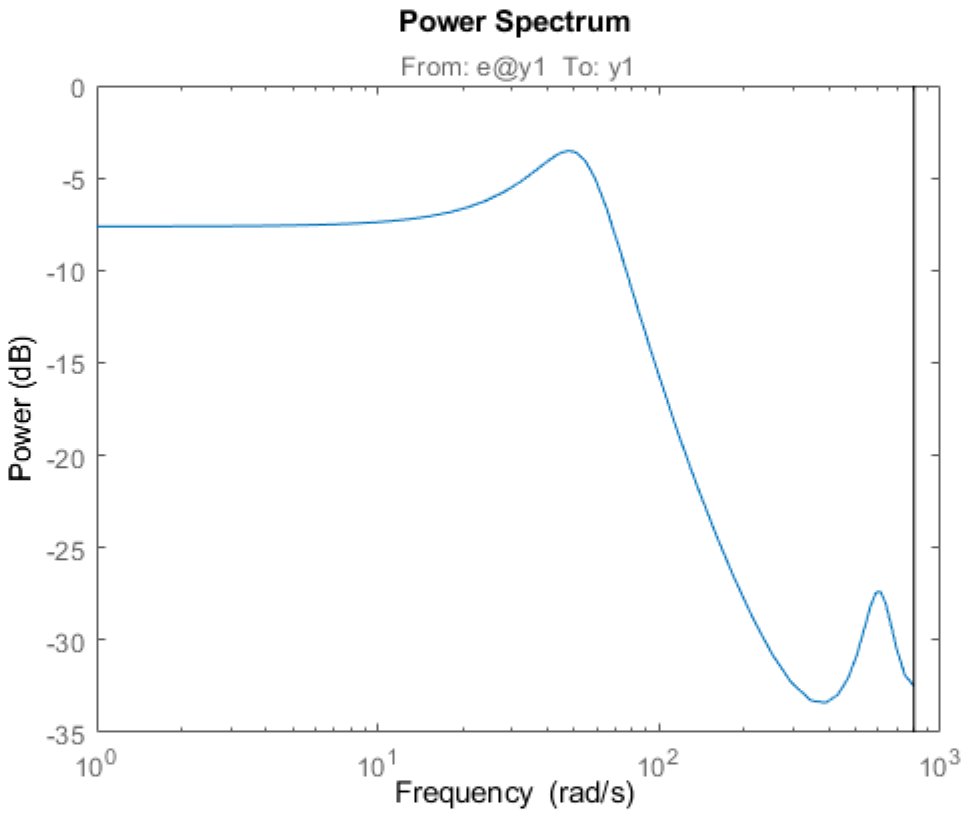
```
load iddata9 z9
```

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);
```



Plot Noise Spectrum of SISO Linear Identified Model

Load the estimation data.

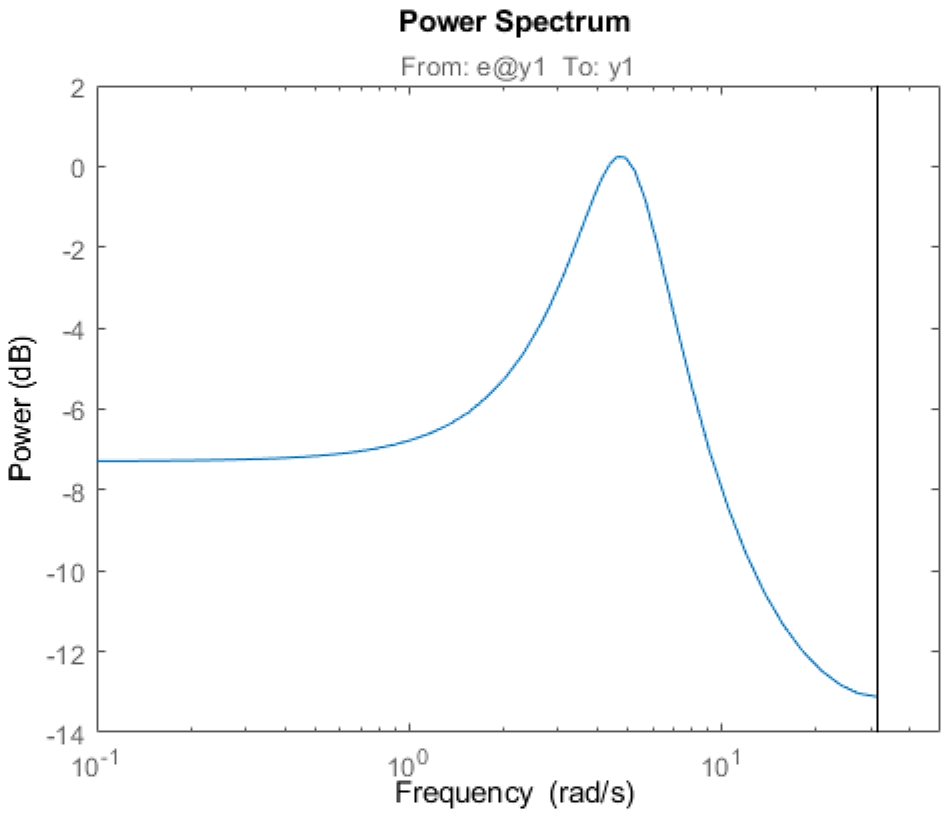
```
load iddata1 z1;
```

Estimate a single-input single-output state-space model.

```
sys = n4sid(z1,2);
```

Plot the noise spectrum for the model. Specify a frequency range from 0.1 to 50 rad/sec.

```
spectrum(sys,{0.1,50});
```



The function plots the spectrum, but limits the frequency range to the Nyquist frequency of approximately 31.4 rad/sec.

Compare Spectrum of Sinusoid Signal with Square

Create an input consisting of the sum of five sinusoids, each spread over the full frequency range. Compare the spectrum of this signal with that of its square.

Create a sum-of-sinusoids input that extends for 20 periods, with each period containing 100 samples. Specify that the signal combine 5 sinusoids of random phase, using 10 trials to find the set with the lowest signal spread. For more information on this step, see `idinput`.

```
u = idinput([100 1 20],'sine',[],[],[5 10 1]);
```

Create an input-only `iddata` object `u` that contains the input u and has a period of 100.

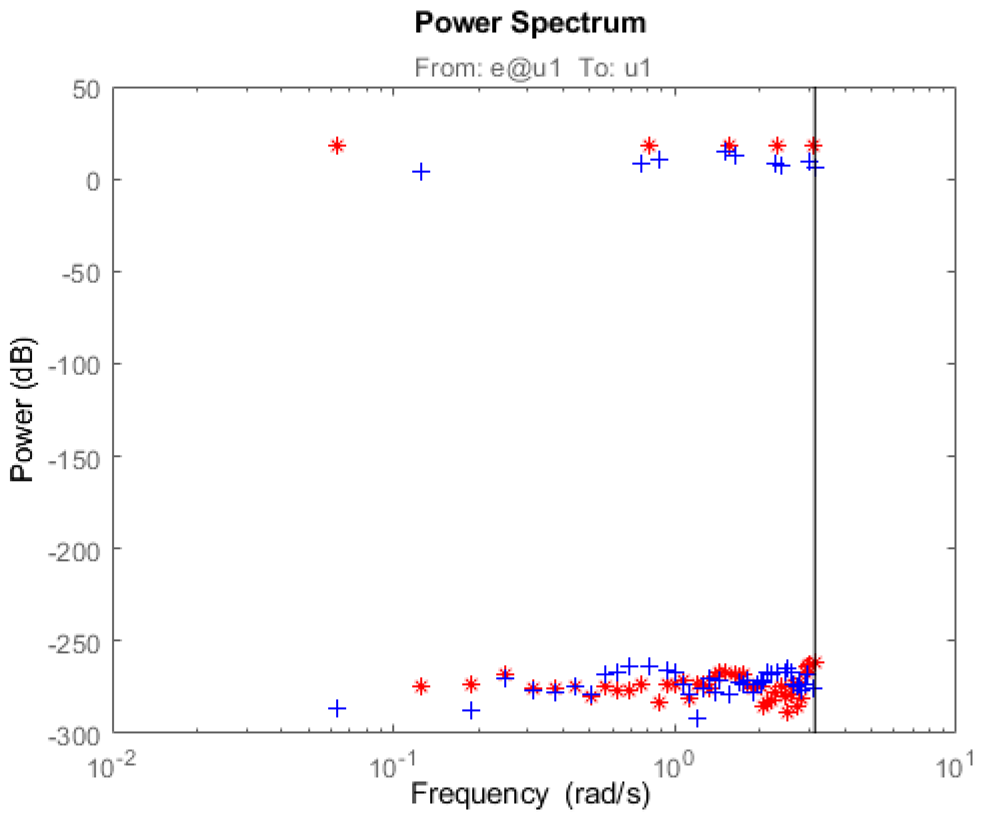
```
u = iddata([],u,1,'per',100);
```

Square the input values and store them in new `iddata` object `u2`.

```
u2 = u.u.^2;
u2 = iddata([],u2,1,'per',100);
```

Use `etfe` to estimate empirical transfer function models from `u` and `u2`. Plot the power spectra of these models together. Use different marker colors and types to distinguish the spectrum sources.

```
spectrum(etfe(u),'r*',etfe(u2),'+')
```



The plot shows some frequency splitting where the `u2`-based spectrum does not line up with the `u1`-based spectrum, but instead, contains two spectral points that flank certain `u1`-based points. This indicates the nonlinearity of the squared system.

Input Arguments

sys — Identified model

`idpoly` object | `idproc` object | `idss` object | `idtf` object

Identified model, specified as an `idpoly` object, an `idproc` object, an `idss` object, or an `idtf` object.

- If `sys` is a time series model, then `sys` represents the system:

  $y(t) = He(t)$

  Here, `e(t)` is a Gaussian white noise and `y(t)` is the observed output.
- If `sys` is an input-output model, then `sys` represents the system:

  $y(t) = Gu(t) + He(t)$

  Here, `u(t)` is the measured input, `e(t)` is a Gaussian white noise and `y(t)` is the observed output.

`wmin` — Minimum frequency
positive number

Minimum frequency of the frequency range for which to plot the spectrum, specified as a positive number.

Specify `wmin` in `rad/TimeUnit`, where `TimeUnit` is `sys.TimeUnit`.

Example: “Plot Noise Spectrum of SISO Linear Identified Model” on page 1-1557

`wmax` — Maximum frequency
positive number

Maximum frequency of the frequency range for which to plot the spectrum, specified as a positive number. By default, the function uses the Nyquist frequency of `sys` as `wmax`.

Specify `wmax` in `rad/TimeUnit`, where `TimeUnit` is `sys.TimeUnit`. If you specify `wmax` to be greater than the Nyquist frequency, then `spectrum` will use the Nyquist frequency instead.

Example: “Plot Noise Spectrum of SISO Linear Identified Model” on page 1-1557

`w` — Frequencies
positive numeric vector

Frequencies for which to plot the spectrum, specified as a vector of positive numbers.

Specify `w` in `rad/TimeUnit`, where `TimeUnit` is `sys.TimeUnit`.

Output Arguments

`ps` — Power spectrum amplitude
numeric array

Power spectrum amplitude returned as a numeric vector or a numeric array.

- For single-output models, `ps` is a 1-by-1- N_w array, where N_w is the length of the frequency vector.
- For multiple-output models, `ps` is an N_y-byN_y-byN_w array, where N_y is the number of outputs. `ps(:,:,k)` corresponds to the power spectrum for the frequency at `w(k)`.

For amplitude values in dB, type `psdb = 10*log10(ps)`.

`wout` — Frequencies
numeric vector

Frequencies for which the spectrum is plotted, returned as a numeric vector. If you supply `w` as an input argument, the function returns the identical vector in `wout`.

`sdps` — Standard deviation
numeric array

Estimated standard deviation of the power spectrum, returned as an array with the same dimensions as `ps`.

See Also

`bode` | `freqresp` | `ar` | `arx` | `armax` | `forecast` | `etfe` | `idinput`

Topics
“Frequency Response Plots for Model Validation”
“Noise Spectrum Plots”
“Plot the Noise Spectrum Using the System Identification App”
“Plot the Noise Spectrum at the Command Line”
“Plot Bode Plots Using the System Identification App”
“Plot Bode and Nyquist Plots at the Command Line”

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.

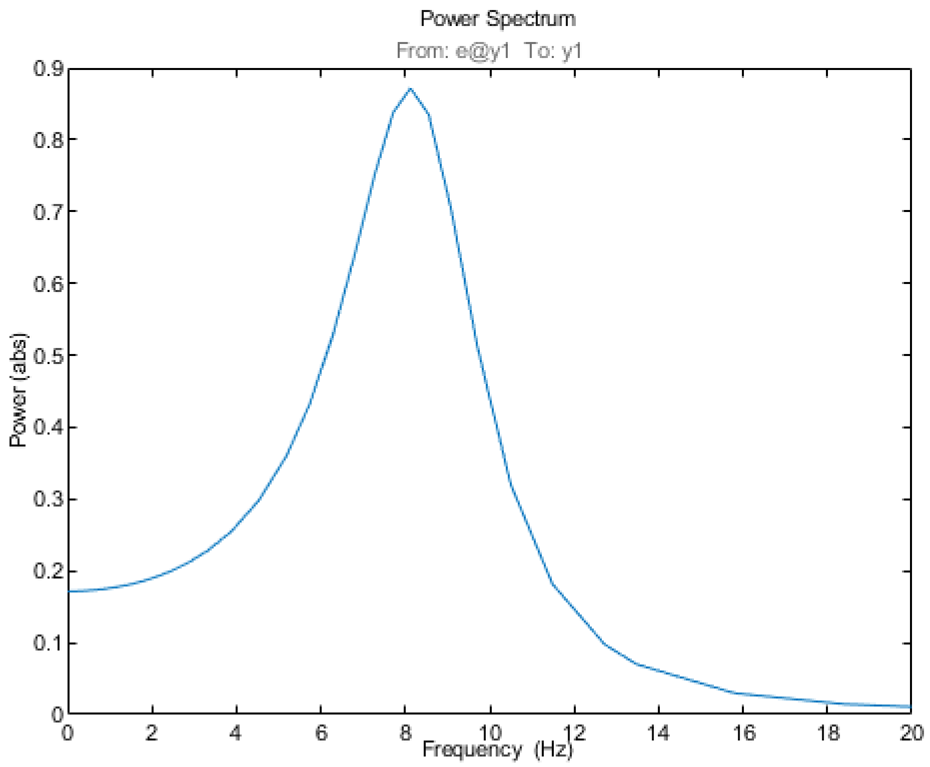
```
plot_options = spectrumoptions;
plot_options.FreqUnits = 'Hz';
plot_options.FreqScale = 'linear';
plot_options.Xlim = {[0 20]};
plot_options.MagUnits = 'abs';
```

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]
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 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'`

Field	Description
`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

`spectrumplot` | `identpref` | `getoptions` | `setoptions`

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 .

`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

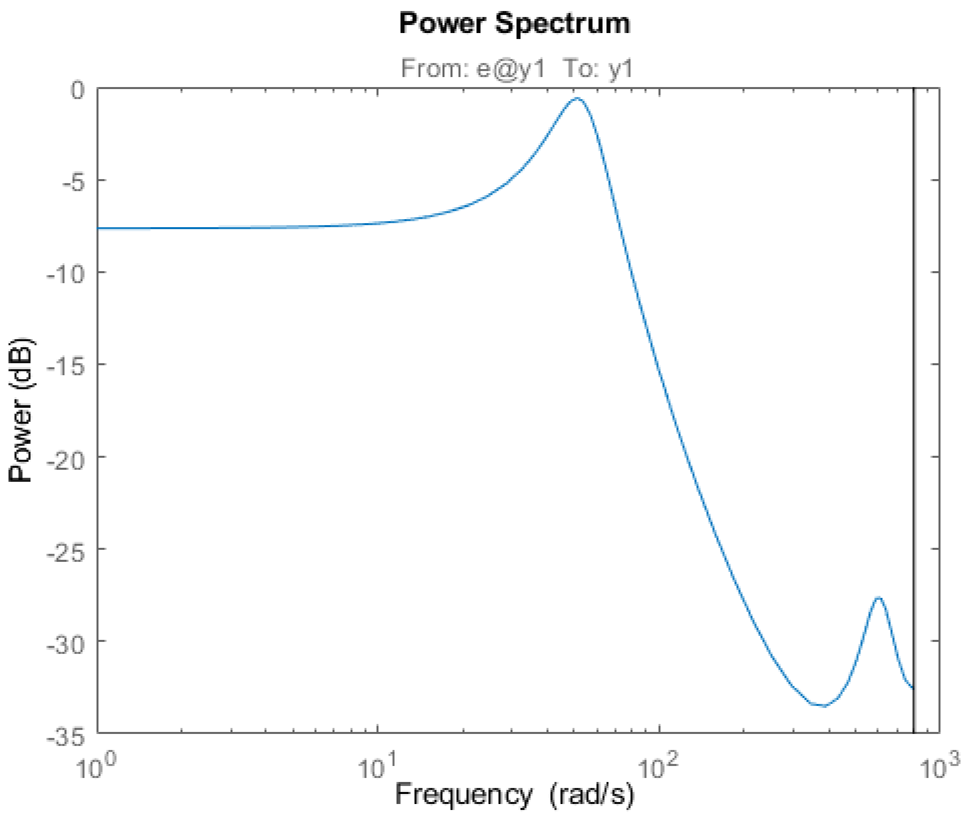
Plot Model Output Spectrum for Identified Model

Obtain the identified model.

```
load iddata9 z9
sys = ar(z9,4);
```

Plot the output spectrum for the model.

```
spectrumplot(sys);
```



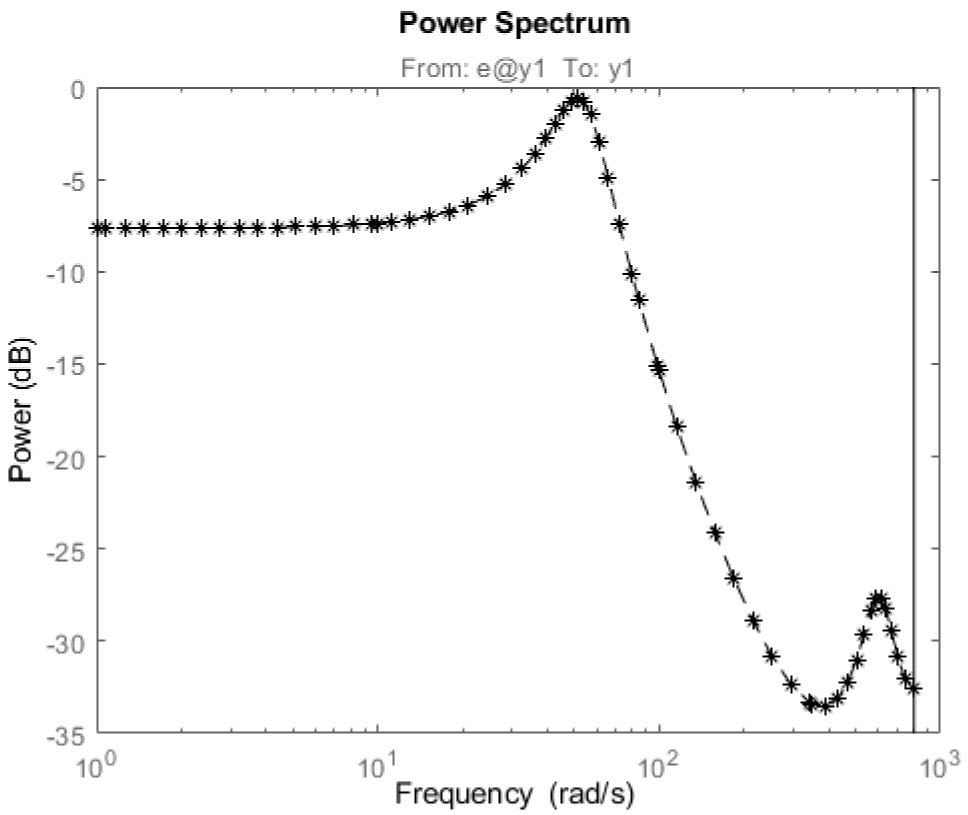
Specify Line Width and Marker Style on Spectrum Plot

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 (`*`).

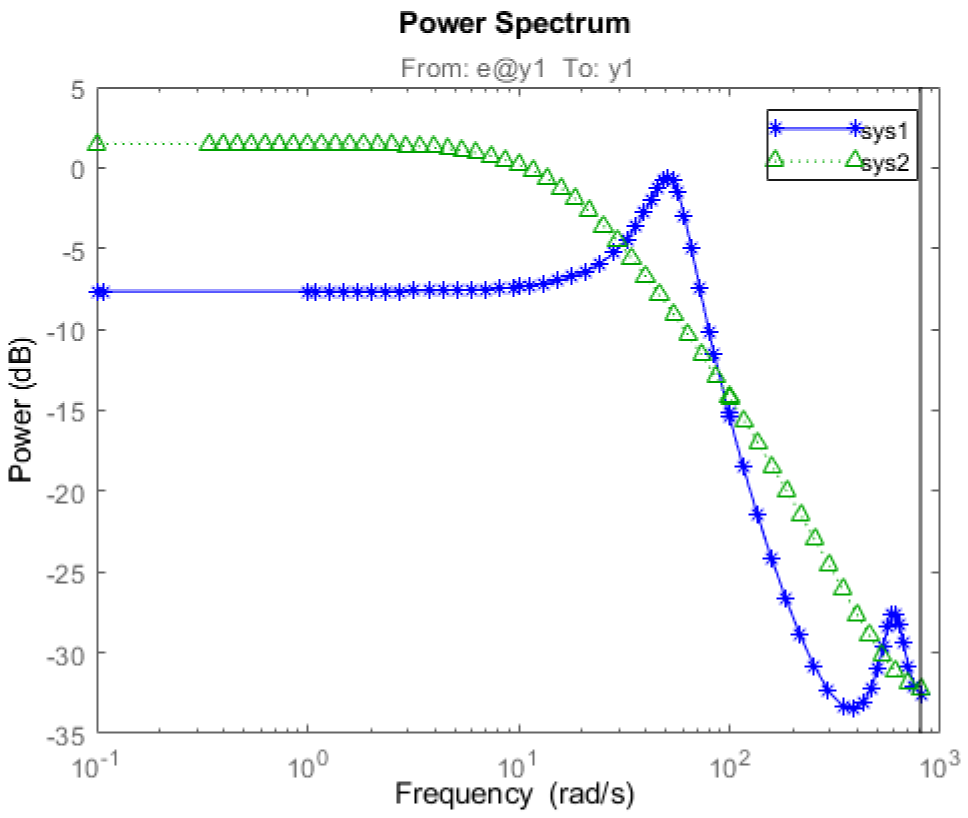
Plot Multiple Models on the Same Axes

Obtain multiple identified models.

```
load iddata9 z9
sys1 = ar(z9,4);
sys2 = ar(z9,2);
```

Plot the output spectrum for both models.

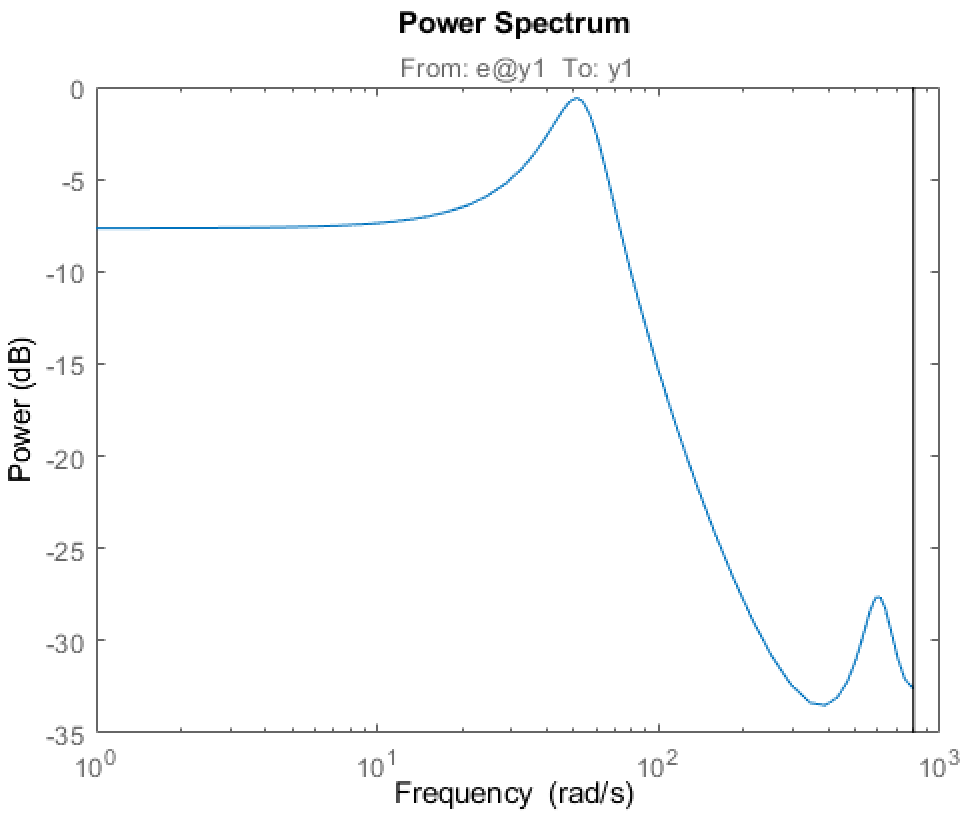
```
spectrumplot(sys1,'b*-',sys2,'g^:');
legend('sys1','sys2');
```



Specify Plot Axes for Spectrum Plot

Obtain the axes handle for a plot.

```
load iddata9 z9
sys1 = ar(z9,4);
spectrumplot(sys1);
```

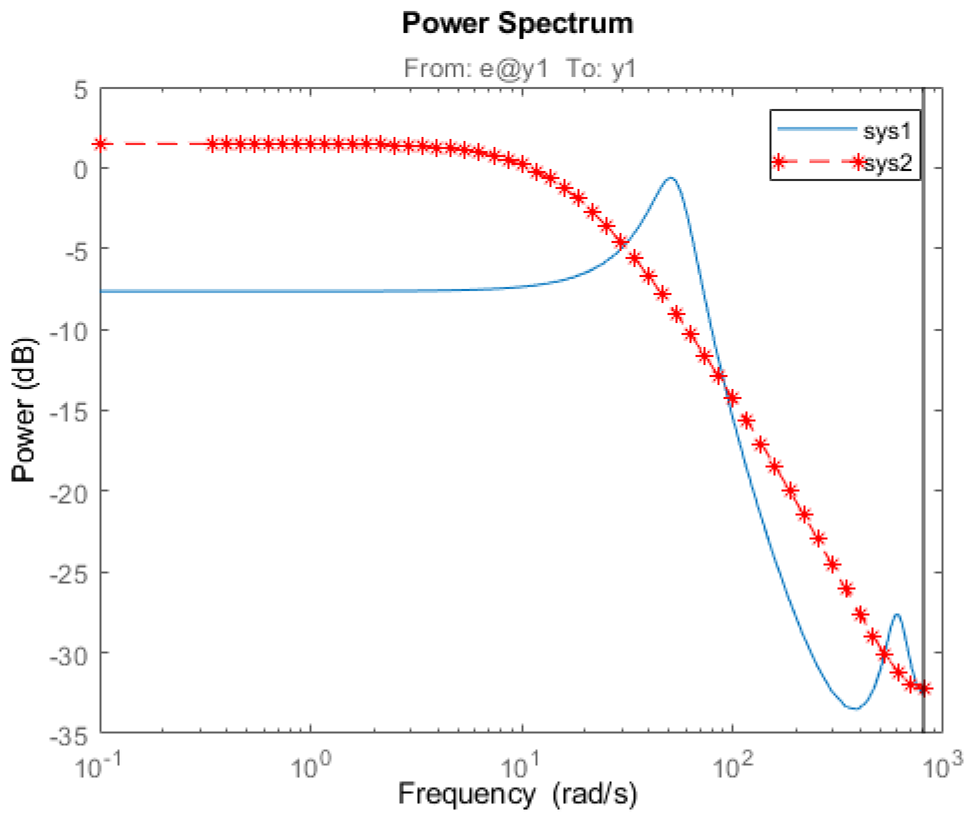


```
ax = gca;
```

`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.

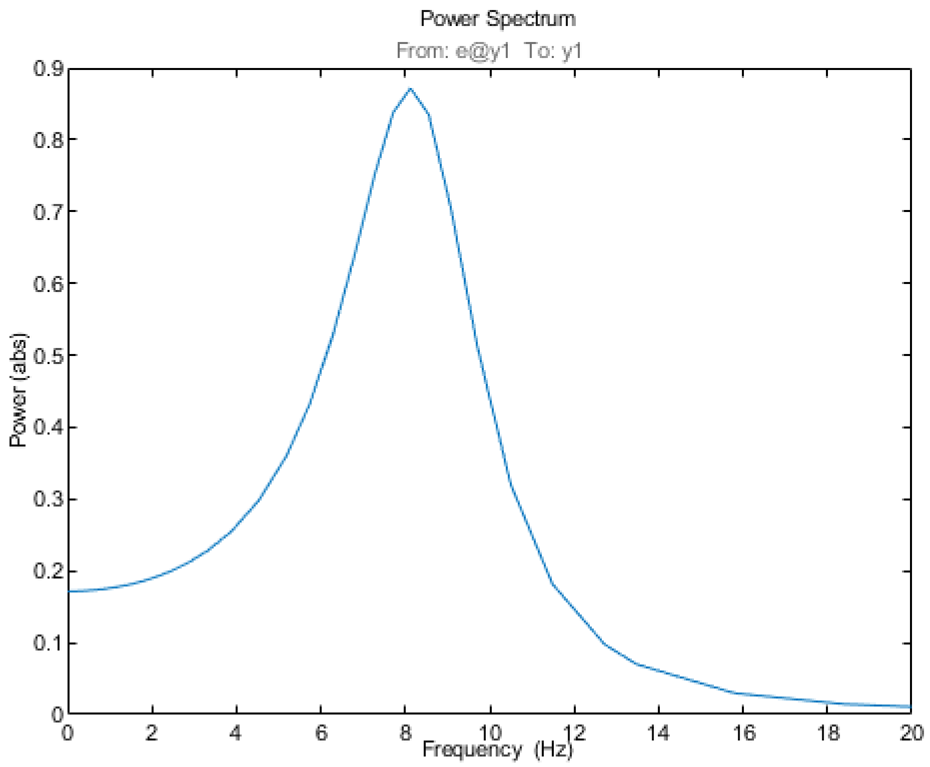
```
plot_options = spectrumoptions;
plot_options.FreqUnits = 'Hz';
plot_options.FreqScale = 'linear';
plot_options.Xlim = {[0 20]};
plot_options.MagUnits = 'abs';
```

Estimate an AR model.

```
load iddata9 z9
sys = ar(z9,4);
```

Plot the output spectrum for the model.

```
spectrumplot(sys,plot_options);
```



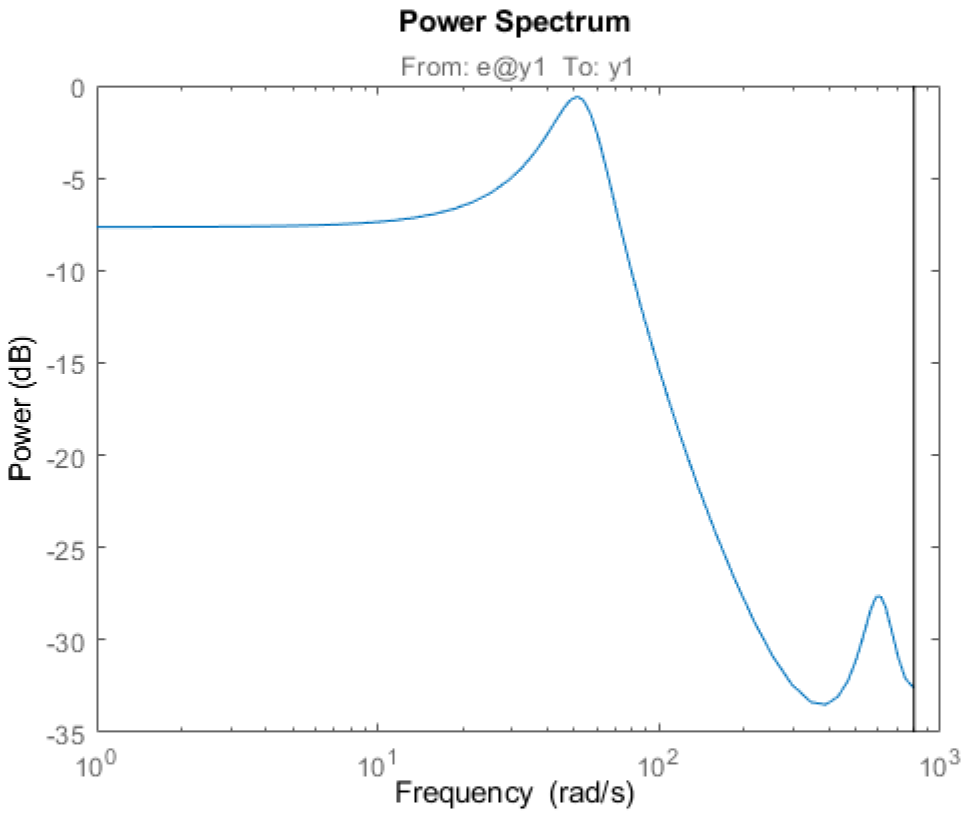
Specify Spectrum Plot Frequency Range

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.

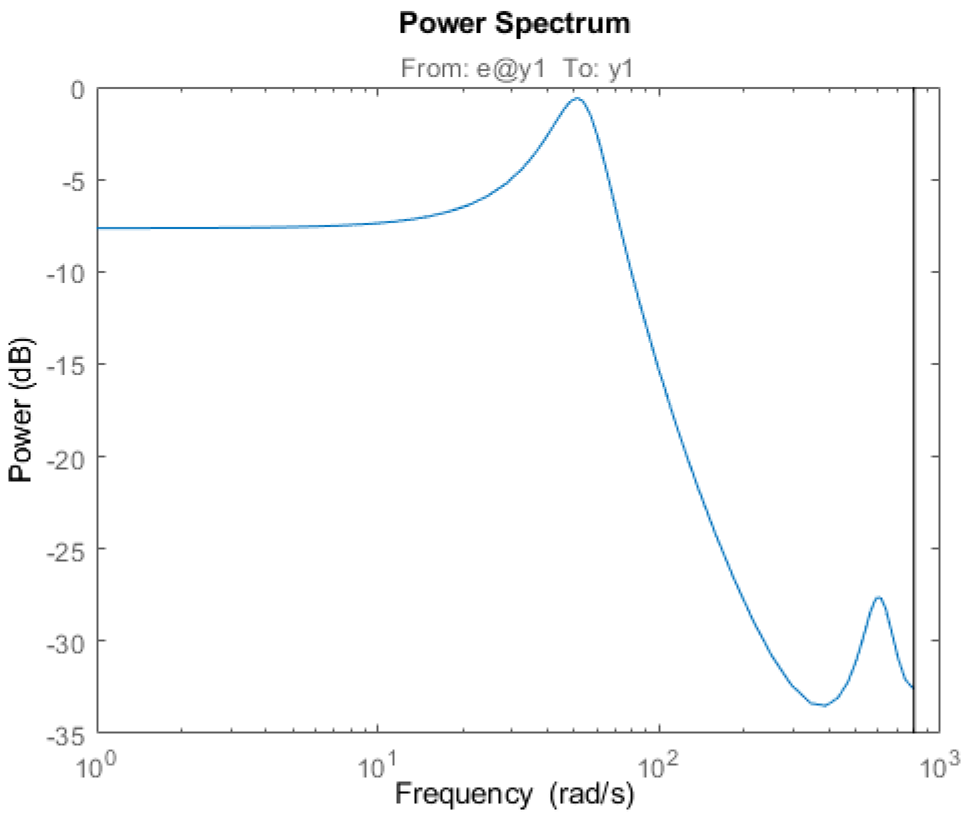
Get Plot Handle for Spectrum Plot Customization

Obtain the identified model.

```
load iddata9 z9
sys = ar(z9,4);
```

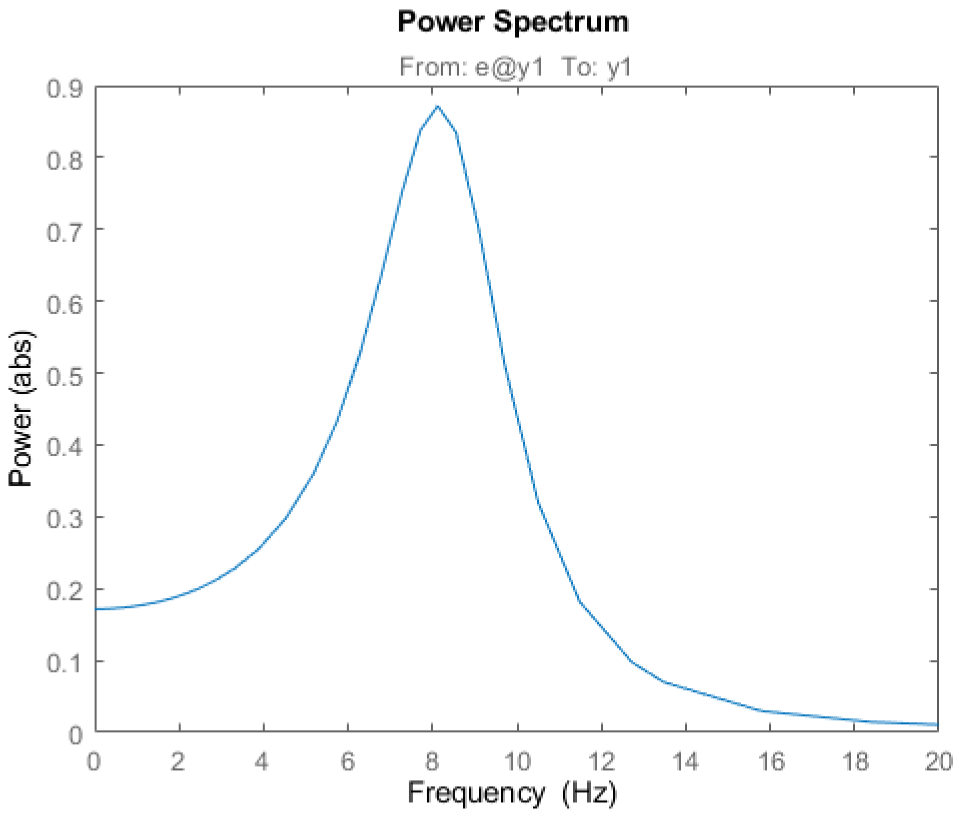
Get the plot handle for the model spectrum plot.

```
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

`spectrum` | `spectrumoptions` | `getoptions` | `setoptions` | `showConfidence` | Axes | Chart Line

Introduced in R2012b

ss2ss

State coordinate transformation for state-space model

Syntax

```
sysT = ss2ss(sys,T)
```

Description

`ss2ss` performs the similarity transformation $z = Tx$ on the state vector x of a state-space model. For more information, see “Algorithms” on page 1-1585.

`sysT = ss2ss(sys,T)` performs the state-coordinate transformation of `sys` using the specified transformation matrix `T`. The matrix `T` must be invertible.

Examples

Similarity Transformation for State-Space Model

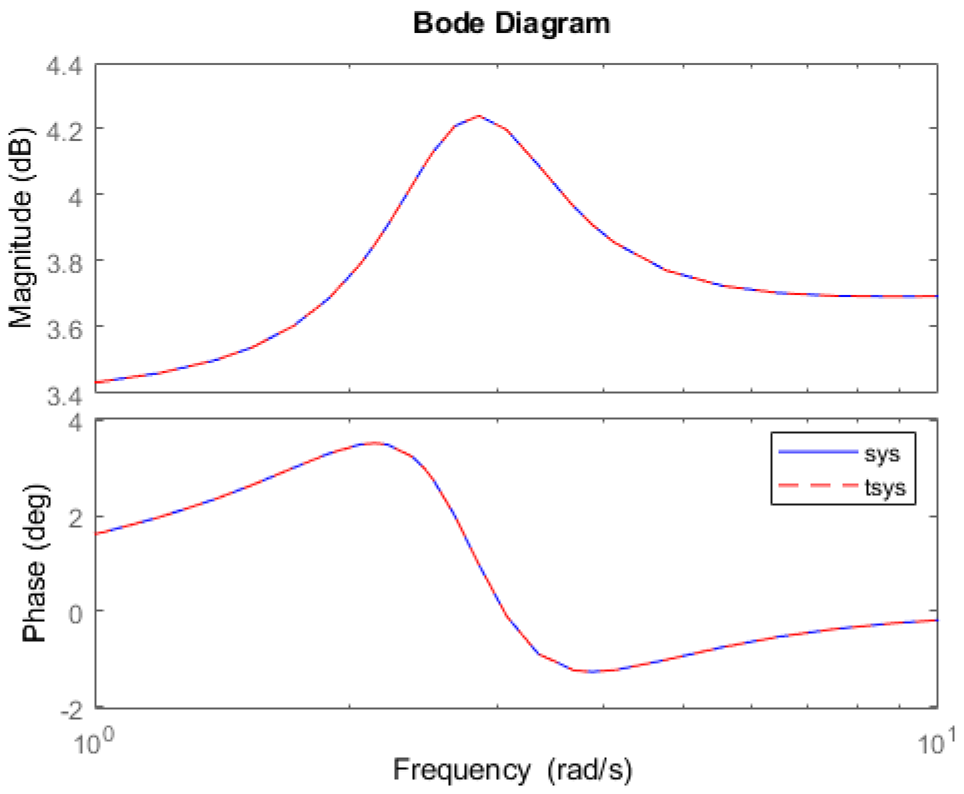
Perform a similarity transform for a state space model.

Generate a random state-space model and a transformation matrix.

```
rng(0)
sys = rss(5);
t = randn(5);
```

Perform the transformation and plot the frequency response of both models.

```
tsys = ss2ss(sys,t);
bode(sys,'b',tsys,'r--')
legend
```



The responses of both models match closely.

Similarity Transformation for Generalized State-Space Models

`ss2ss` applies state transformation only to the state vectors of the numeric portion of the generalized model.

Create a `genss` model.

```
sys = rss(2,2,2) * tunableSS('a',2,2,3) + tunableGain('b',2,3)
```

```
sys =

  Generalized continuous-time state-space model with 2 outputs, 3 inputs, 4 states, and the follo
    a: Tunable 2x3 state-space model, 2 states, 1 occurrences.
    b: Tunable 2x3 gain, 1 occurrences.

Type "ss(sys)" to see the current value, "get(sys)" to see all properties, and "sys.Blocks" to in
```

Specify a transformation matrix and obtain the transformation.

```
T = [1 -2;3 5];
tsys = ss2ss(sys,T)
```

```
tsys =
  Generalized continuous-time state-space model with 2 outputs, 3 inputs, 4 states, and the follo
    a: Tunable 2x3 state-space model, 2 states, 1 occurrences.
    b: Tunable 2x3 gain, 1 occurrences.

Type "ss(tsys)" to see the current value, "get(tsys)" to see all properties, and "tsys.Blocks" to
```

Decompose both models.

```
[H,B,~,~] = getLFTModel(sys);
[H1,B1,~,~] = getLFTModel(tsys);
```

Obtain the transformation separately on the model from decomposed `sys`.

```
H2 = ss2ss(H,T);
```

Compare this transformed model with the model from decomposed `tsys`.

```
isequal(H1,H2)
```

```
ans = logical
   1
```

Both models are equal.

Similarity Transformation for Identified State-Space Models

The file `icEngine.mat` contains one data set with 1500 input-output samples collected at the a sampling rate of 0.04 seconds. The input `u(t)` is the voltage (V) controlling the By-Pass Idle Air Valve (BPAV), and the output `y(t)` is the engine speed (RPM/100).

Use the data in `icEngine.mat` to create a state-space model with identifiable parameters.

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

Specify a random transformation matrix.

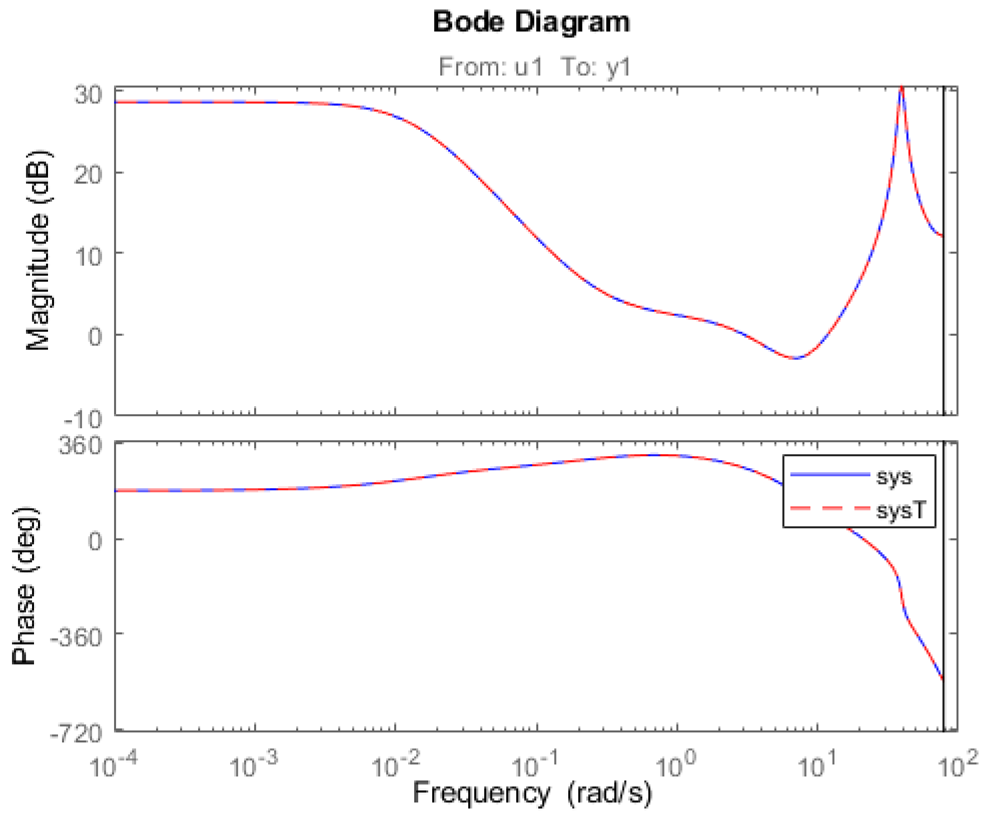
```
T = randn(4);
```

Obtain the transformation.

```
sysT = ss2ss(sys,T);
```

Compare the frequency responses.

```
bode(sys,'b',sysT,'r--')
legend
```



The responses match closely.

Transformation for Models with Complex Coefficients

`ss2ss` also lets you perform similarity transformation for models with complex coefficients.

For this example, generate a random state-space model with complex coefficients.

```
rng(0)
sys = ss(randn(5)+1i*randn(5),randn(5,3),randn(2,5)+1i*randn(2,5),0,.1);
```

Specify a transformation matrix containing complex data.

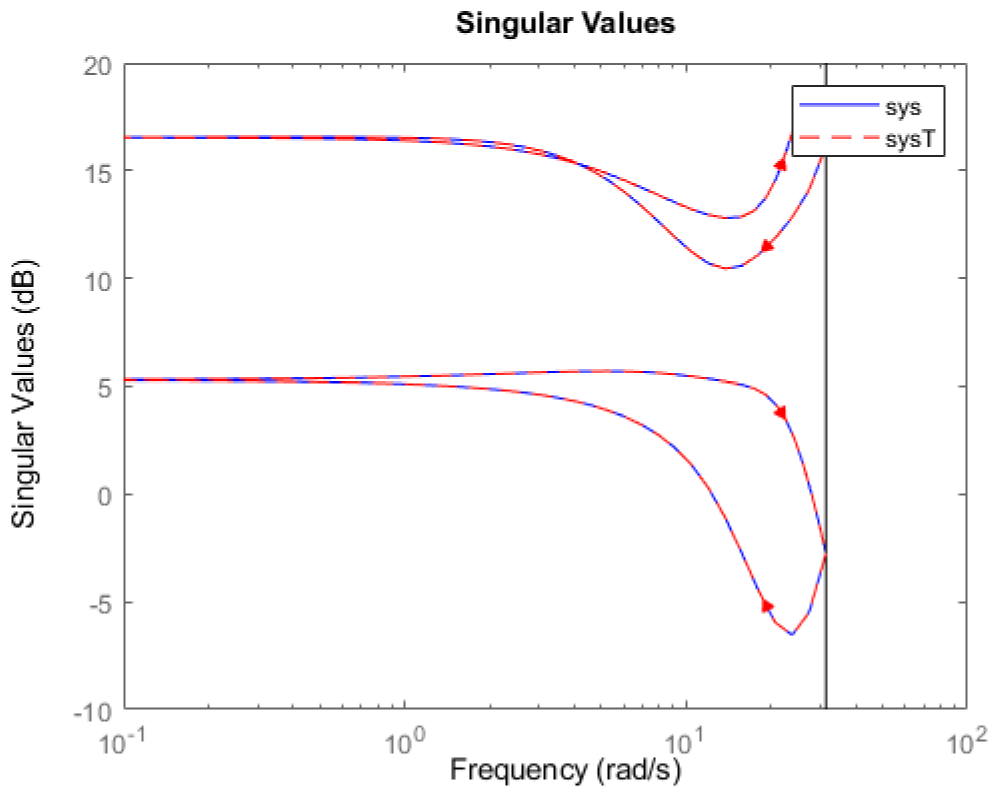
```
T = randn(5)+1i*randn(5);
```

Obtain the transformation.

```
sysT = ss2ss(sys,T);
```

Compare the singular values of the frequency response.

```
sigma(sys,'b',sysT,'r--')
legend
```



The responses match closely for both branches.

Input Arguments

sys — Dynamic system

dynamic system model

Dynamic system, specified as a SISO, or MIMO dynamic system model. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `ss` or `dss` models.
- Generalized or uncertain LTI models, such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)

  For such models, the state transformation is applied only to the state vectors of the numeric portion of the model. For more information about decomposition of these models, see `getLFTModel` and “Internal Structure of Generalized Models” (Control System Toolbox).
- Identified state-space `idss` models.

If `sys` is an array of state-space models, `ss2ss` applies the transformation `T` to each individual model in the array.

T — Transformation matrix

matrix

Transformation matrix, specified as an *n*-by-*n* matrix, where *n* is the number of states. `T` is the transformation between the state vector of the state-space model `sys` and the state vector of the transformed model `sysT`. (See “Algorithms” on page 1-1585.)

Output Arguments

`sysT` — Transformed model
dynamic system model

Transformed state-space model, returned as a dynamic system model of the same type as `sys`.

Algorithms

`ss2ss` performs the similarity transformation $\bar{x} = Tx$ on the state vector x of a state-space model.

This table summarizes the transformations returned by `ss2ss` for each model form.

Input Model	Transformed Model
Explicit state-space models of the form: $\dot{x} = Ax + Bu$, $y = Cx + Du$	$\dot{\bar{x}} = TAT^{-1}\bar{x} + TBu$, $y = CT^{-1}\bar{x} + Du$
Descriptor (implicit) state-space models for the form: $E\dot{x} = Ax + Bu$, $y = Cx + Du$	$ET^{-1}\dot{\bar{x}} = AT^{-1}\bar{x} + Bu$, $y = CT^{-1}\bar{x} + Du$
Identified state-space (`idss`) models of the form: $\frac{dx}{dt} = Ax + Bu + Ke$, $y = Cx + Du + e$	$\dot{\bar{x}} = TAT^{-}1\bar{x} + TBu + TKe$, $y = CT^{-}1\bar{x} + Du + e$

Compatibility Considerations

`ss2ss` returns different transformation results for descriptor state-space models
Behavior changed in R2021b

For a descriptor state-space model

$$E\dot{x} = Ax + Bu$$
$$y = Cx + Du,$$

`ss2ss` now returns

$$ET^{-1}\dot{\bar{x}} = AT^{-1}\bar{x} + Bu$$
$$y = CT^{-1}\bar{x} + Du\,.$$

Previously, the function returned the following transformation.

$$TET^{-1}\dot{\bar{x}} = TAT^{-1}\bar{x} + TBu$$
$$y = CT^{-1}\bar{x} + Du$$

Similarity transformation is no longer supported for `mechss` models
Errors starting in R2021b

`ss2ss` no longer supports sparse second-order (`mechss`) models. Performing similarity transformations on `mechss` models destroys symmetry and has no obvious general form.

See Also

`balreal` | `canon` | `balance`

Topics

"Scaling State-Space Models to Maximize Accuracy" (Control System Toolbox)

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.

For generalized state-space (`genss`) models, `ssdata` returns the state-space models evaluated at the current, nominal value of all control design blocks. To access the dependency of a `genss` model on its static control design blocks, use the `A`, `B`, `C`, and `D` properties of the model.

`[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-domain 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,opt)

[sys,x0] = ssest( ___ )
```

Description

Estimate a State-Space Model

`sys = ssest(data,nx)` estimates a continuous-time state-space model `sys` of order `nx`, using data `data` that can be in the time domain or the frequency domain. `sys` is a model of the following form:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$

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 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)` incorporates additional options specified by one or more name-value pair arguments. For example, estimate a discrete-time model by specifying the sample time `'Ts'` name-value pair argument. 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)` incorporates an option set `opt` that specifies options such as estimation objective, handling of initial conditions, regularization, and numerical search method used for estimation. You can specify `opt` after any of the previous input-argument combinations.

Configure Initial Parameters

`sys = ssest(data,init_sys)` uses the state-space model `init_sys` to configure the initial parameterization.

`sys = ssest(data,init_sys,opt)` estimates the model using an option set `opt`.

Return Estimated Initial States

`[sys,x0] = ssest( ___ )` returns the value of initial states computed during estimation. You can use this syntax with any of the previous input-argument combinations.

Examples

State-Space Model

Estimate a state-space model and compare its response with the measured output.

Load the input-output data, which is stored in an `iddata` object.

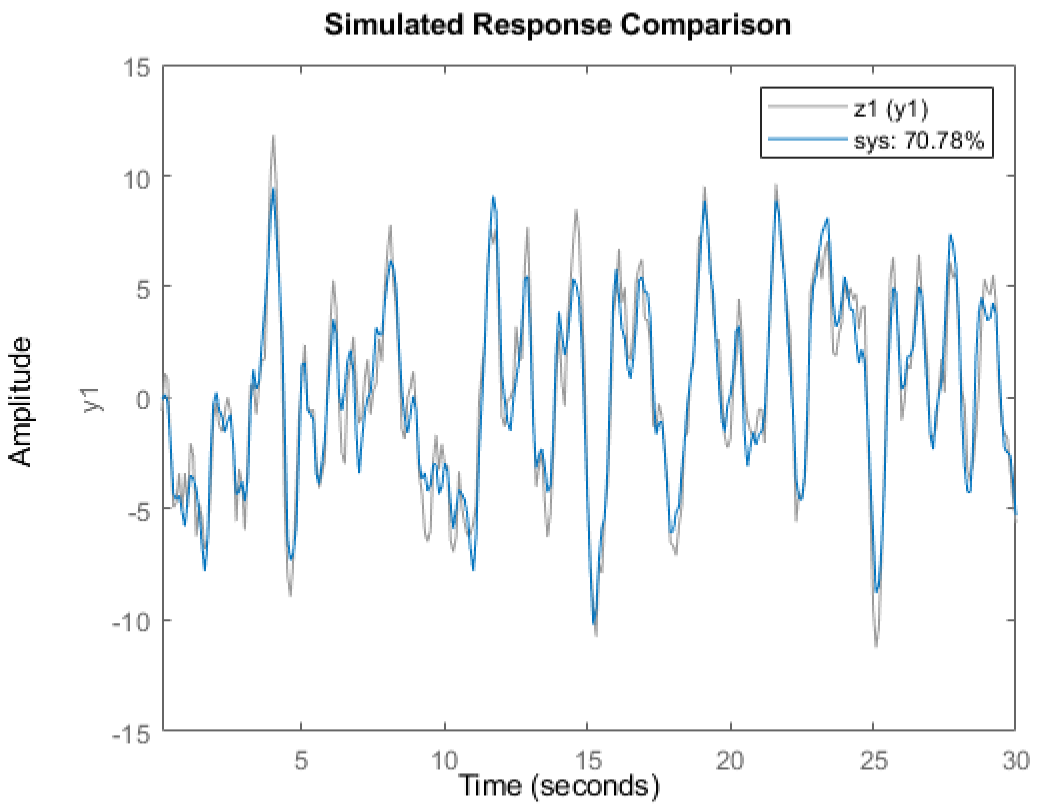
```
load iddata1 z1
```

Estimate a fourth-order state-space model.

```
nx = 4;
sys = ssest(z1,nx);
```

Compare the simulated model response with the measured output.

```
compare(z1,sys)
```



The plot shows that the fit percentage between the simulated model and the estimation data is greater than 70%.

You can view more information about the estimation by exploring the `idss` property `sys.Report`.

```
sys.Report
```

```
ans =
          Status: 'Estimated using SSEST with prediction focus'
          Method: 'SSEST'
    InitialState: 'zero'
        N4Weight: 'CVA'
       N4Horizon: [6 10 10]
             Fit: [1x1 struct]
      Parameters: [1x1 struct]
     OptionsUsed: [1x1 idoptions.ssest]
       RandState: []
        DataUsed: [1x1 struct]
     Termination: [1x1 struct]
```

For example, find out more information about the termination conditions.

```
sys.Report.Termination
```

```
ans = struct with fields:
                 WhyStop: 'No improvement along the search direction with line search.'
              Iterations: 7
    FirstOrderOptimality: 85.9759
                FcnCount: 123
              UpdateNorm: 8.1714
         LastImprovement: 0
```

The report includes information on the number of iterations and the reason the estimation stopped iterating.

Determine Optimal Estimated Model Order

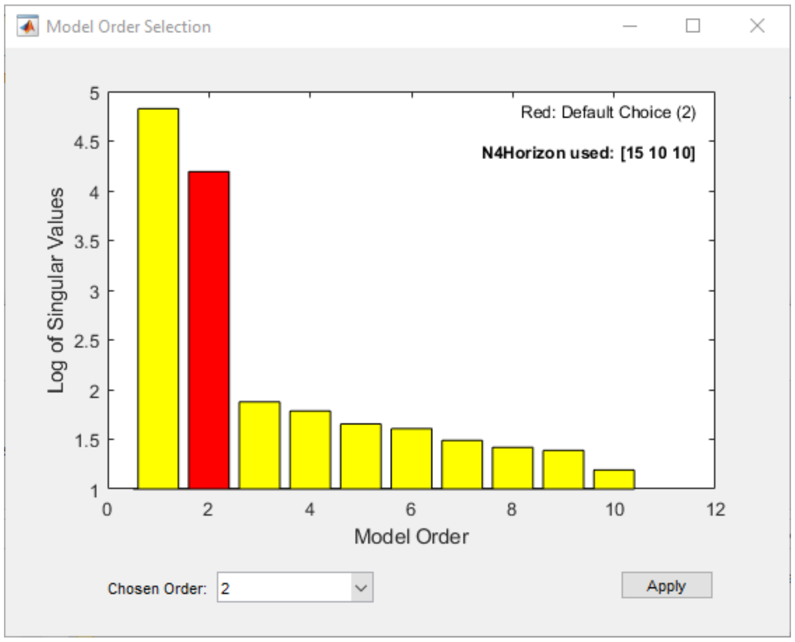
Load the input-output data `z1`, which is stored in an `iddata` object. This is the same data used to estimate a fourth-order model in “State-Space Model” on page 1-1589.

```
load iddata1 z1
```

Determine the optimal model order by specifying argument `nx` as a range from `1:10`.

```
nx = 1:10;
sys = ssest(z1,nx);
```

An automatically generated plot shows the Hankel singular values for models of the orders specified by `nx`.



States with relatively small Hankel singular values can be safely discarded. The suggested default order choice is 2.

Select the model order in the **Chosen Order** 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]);
```

Modify Form, Feedthrough, and Disturbance-Model Matrices

Modify the canonical form of the A, B, and C matrices, include a feedthrough term in the D matrix, and eliminate disturbance-model estimation in the K matrix.

Load input-output data and estimate a fourth-order system using the `ssest` default options.

```
load iddata1 z1
sys1 = ssest(z1,4);
```

Specify the companion form and compare the A matrix with the default A matrix.

```
sys2 = ssest(z1,4,'Form','companion');
A1 = sys1.A

A1 = 4×4

   -0.5155   -3.8483    0.6657   -0.2666
    5.8665   -2.7285    1.0649   -1.4694
   -0.4487    0.9308   -0.6235   18.8148
   -0.4192    0.5595  -16.0688    0.5399


A2 = sys2.A

A2 = 4×4
10³ ×

         0         0         0   -7.1122
    0.0010         0         0   -0.9547
         0    0.0010         0   -0.3263
         0         0    0.0010   -0.0033
```

Include a feedthrough term and compare D matrices.

```
sys3 = ssest(z1,4,'Feedthrough',1);
D1 = sys1.D

D1 = 0

D3 = sys3.D

D3 = 0.0339
```

Eliminate disturbance modeling and compare K matrices.

```
sys4 = ssest(z1,4,'DisturbanceModel','none');
K1 = sys1.K

K1 = 4×1

    0.0520
    0.0973
    0.0151
    0.0270


K4 = sys4.K

K4 = 4×1

     0
     0
     0
     0
```

Estimate Initial States as Independent Parameters

Specify ssest estimate initial states as independent estimation parameters.

ssest can handle initial states using one of several methods. By default, ssest chooses the method automatically based on your estimation data. You can choose the method yourself by modifying the option set using ssestOptions.

Load the input-output data z1 and estimate a second-order state-space model sys using the default options. Use the syntax that returns initial states x0.

```
load iddata1 z1
[sys,x0] = ssest(z1,2);
x0
```

```
x0 = 2×1

     0
     0
```

By default, the estimation is performed using the 'auto' setting for InitialState. Find out which method ssest applied by reviewing the value of InitialState in sys.Report.

```
sys.Report.InitialState
```

```
ans = 
'zero'
```

The software applied the 'zero' method, meaning that the software set the initial states to zero instead of estimating them. This selection is consistent with the 0 values returned for x0.

Specify that ssest estimate the initial states instead as independent parameters using the 'estimate' setting. Use ssestOptions to create a modified option set and specify that option set to estimate a new model.

```
opt = ssestOptions('InitialState','estimate');
[sys1,x0] = ssest(z1,2,opt);
x0
```

```
x0 = 2×1

    0.0068
    0.0052
```

x0 now has estimated parameters with nonzero values.

Estimate State-Space Model Using Regularization

Obtain a regularized fifth-order state-space model for a second-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.

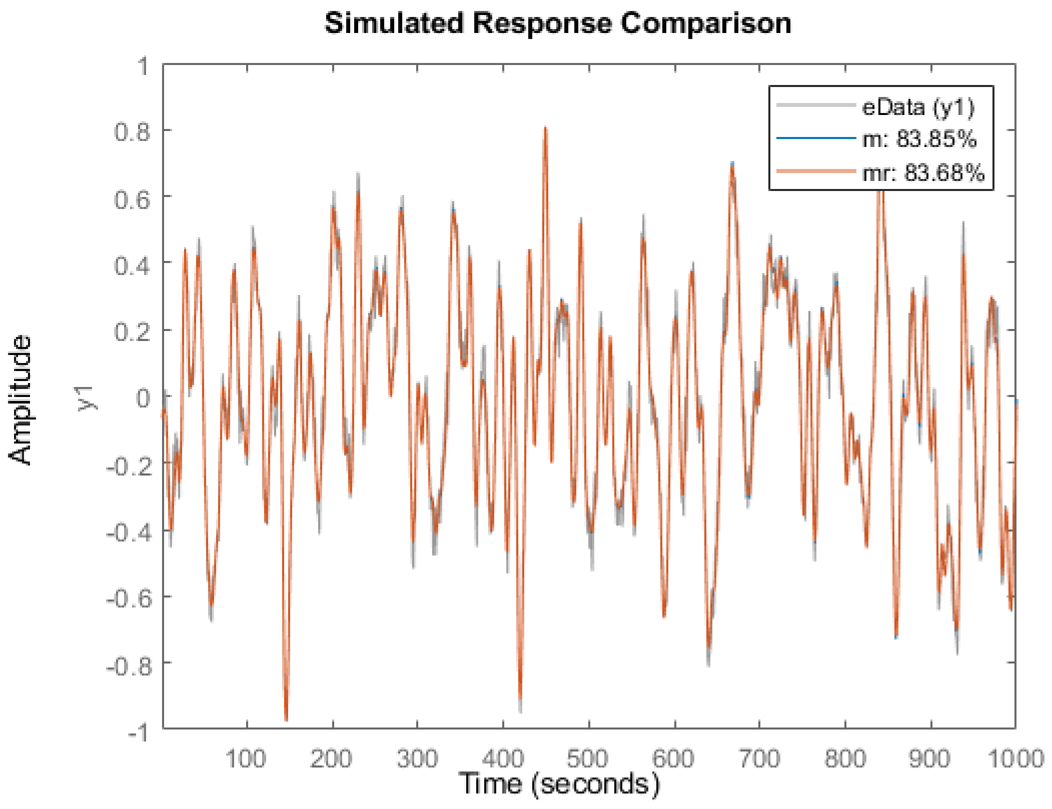
```
opt = ssestOptions('SearchMethod','lm');
m = ssest(eData,5,'form','modal','DisturbanceModel','none','Ts',eData.Ts,opt);
```

Estimate a regularized state-space model.

```
opt.Regularization.Lambda = 10;
mr = ssest(eData,5,'form','modal','DisturbanceModel','none','Ts',eData.Ts,opt);
```

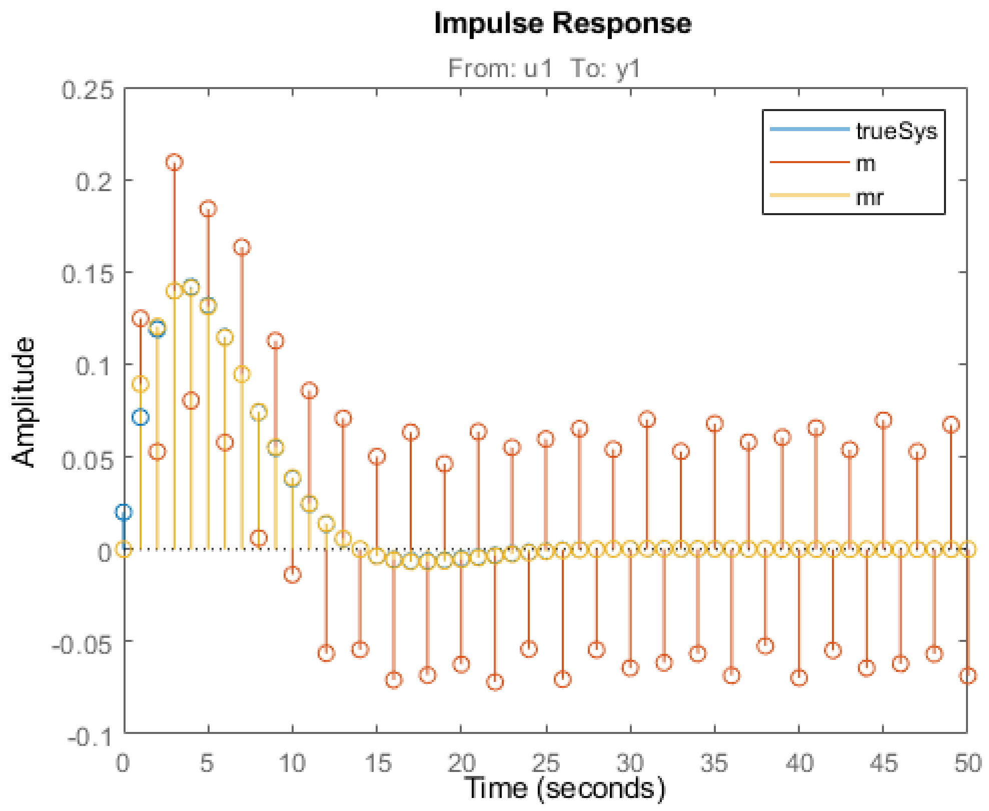
Compare the model outputs with the estimation data.

```
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`.

Input Arguments

data — Estimation data

`iddata` object | `frd` object | `idfrd` object

Estimation data, specified as an `iddata` object, an `frd` object, or an `idfrd` 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 properties specified as follows.
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — `'Frequency'`

Estimation data must be uniformly sampled. By default, the software sets the sample time of the model to the sample time of the estimation data.

For multiexperiment data, the sample times and intersample behavior of all the experiments must match.

The domain of your data determines the type of model you can estimate.

- Time-domain or discrete-time frequency-domain data — Continuous-time and discrete-time models
- Continuous-time frequency-domain data — Continuous-time models only

nx — Order of estimated model

`1:10` (default) | positive integer scalar | positive integer vector | `0`

Order of the estimated model, specified as a nonnegative integer or as a vector containing a range of positive integers.

- If you already know what order you want your estimated model to have, specify `nx` as a scalar.
- If you want to compare a range of potential orders to choose the most effective order for your estimated model, specify the range in `nx`. `ssest` creates a Hankel singular-value plot that shows the relative energy contributions of each state in the system. States with relatively small Hankel singular values contribute little to the accuracy of the model and can be discarded with little

impact. The index of the highest state you retain is the model order. The plot window includes a suggestion for the order to use. You can accept this suggestion or enter a different order. For an example, see “Determine Optimal Estimated Model Order” on page 1-1590.

If you do not specify `nx`, or if you specify `nx` as `best`, the software automatically chooses `nx` from the range `1:10`.

- If you are identifying a static system, set `nx` to `0`.

`opt` — Estimation options
`ssestOptions` option set

Estimation options, specified as an `ssestOptions` option set. Options specified by `opt` include:

- Estimation objective
- Handling of initial conditions
- Regularization
- Numerical search method used for estimation

For examples showing how to use `opt`, see “Estimate Initial States as Independent Parameters” on page 1-1592 and “Estimate State-Space Model Using Regularization” on page 1-1593.

`init_sys` — Linear system that configures initial parameterization of sys
`idss` model | linear model | structure

Linear system that configures the initial parameterization of `sys`, specified as an `idss` model or as a structure. 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”. `ssest` honors constraints on the parameters of `init_sys`, such as fixed coefficients and minimum/maximum bounds.

Use the `Structure` property of `init_sys` to configure initial parameter values 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

  To set more complex constraints, such as interdependence of coefficients, use grey-box estimation using `greyest` and `idgrey`.

You must assign finite initial values for all matrix parameters.

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 you do not specify `opt` and `init_sys` was obtained by estimation, then the software uses estimation options from `init_sys.Report.OptionsUsed`.

For an example, see "Estimate Partially Known State-Space Model Using Structured Estimation" on page 1-1595.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sys = ssest(data,nx,'Ts',0.1)`

`Ts` — Sample time of estimated model
`0` (continuous time) (default) | sample time of data (`data.Ts`) | positive scalar

Sample time of the estimated model, specified as the comma-separated pair consisting of `'Ts'` and either `0` or a positive scalar.

- For continuous-time models, specify `'Ts'` as `0`.
- For discrete-time models, specify `'Ts'` as the data sample time in the units stored in the `TimeUnit` property.

`InputDelay` — Input delays
`0` (default) | scalar | vector

Input delay for each input channel, specified as the comma-separated pair consisting of `'InputDelay'` and a numeric vector.

- For continuous-time models, specify `'InputDelay'` in the time units stored in the `TimeUnit` property.
- For discrete-time models, specify `'InputDelay'` in integer multiples of the sample time `Ts`. For example, setting `'InputDelay'` to `3` specifies a delay of three sampling periods.

For a system with N_u inputs, set `InputDelay` to an N_u-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel. For an example, see "Identify State-Space Model with Input Delay" on page 1-1591.

To apply the same delay to all channels, specify `InputDelay` as a scalar.

`Form` — Type of canonical form
`'free'` (default) | `'modal'` | `'companion'` | `'canonical'`

Type of canonical form of `sys`, specified as the comma-separated pair consisting of `'Form'` and one of the following values:

- `'free'` — All entries of the matrices A, B, C, D, and K are treated as free.
- `'modal'` — Obtain `sys` in modal form.
- `'companion'` — Obtain `sys` in companion form.
- `'canonical'` — Obtain `sys` in the observability canonical form.

For definitions of the canonical forms, see "Canonical State-Space Realizations".

For more information, see “Estimate State-Space Models with Canonical Parameterization”. For an example, see “Modify Form, Feedthrough, and Disturbance-Model Matrices” on page 1-1591.

`Feedthrough` — Direct feedthrough from input to output
`0` (default) | `1` | logical vector

Direct feedthrough from input to output, specified as the comma-separated pair consisting of `'Feedthrough'` and a logical vector of length N_u, where N_u is the number of inputs. If you specify `Feedthrough` as a logical scalar, that value is applied to all the inputs. For static systems, the software always assumes `'Feedthrough'` is `1`.

For an example, see “Modify Form, Feedthrough, and Disturbance-Model Matrices” on page 1-1591.

`DisturbanceModel` — Option to estimate time-domain noise component parameters
`'estimate'` (default) | `'none'`

Option to estimate time-domain noise component parameters in the K matrix, specified as the comma-separated pair consisting of `'DisturbanceModel'` and one of the following values:

- `'estimate'` — Estimate the noise component. The *K* matrix is treated as a free parameter.
- `'none'` — Do not estimate the noise component. The elements of the *K* matrix are fixed at zero.

For frequency-domain data, the software assumes that `'DisturbanceModel'` is `'none'`.

For an example, see “Modify Form, Feedthrough, and Disturbance-Model Matrices” on page 1-1591.

Output Arguments

`sys` — Identified state-space model
`idss` model

Identified state-space model, returned as an `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
InitialState	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 Variate 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'.
N4Horizon	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.
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:

Field	Description
FitPercent	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
LossFcn	Value of the loss function when the estimation completes.
MSE	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
FPE	Final prediction error for the model.
AIC	Raw Akaike Information Criteria (AIC) measure of model quality.
AICc	Small-sample-size corrected AIC.
nAIC	Normalized AIC.
BIC	Bayesian Information Criteria (BIC).

Report Field	Description
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `ssestOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.
`DataUsed`	Attributes of the data used for estimation. Structure with the following fields:

Field	Description
`Name`	Name of the data set.
`Type`	Data type.
`Length`	Number of data samples.
`Ts`	Sample time. This is equivalent to `Data.Ts`.
`InterSample`	Input intersample behavior. One of the following values: • `'zoh'` — 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. The value of `Intersample` has no effect on estimation results for discrete-time models.
`InputOffset`	Offset removed from time-domain input data during estimation.
`OutputOffset`	Offset removed from time-domain output data during estimation.

Report Field	Description
`Termination`	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields: **Field** — **Description** `WhyStop` — Reason for terminating the numerical search. `Iterations` — Number of search iterations performed by the estimation algorithm. `FirstOrderOptimality` — ∞-norm of the gradient search vector when the search algorithm terminates. `FcnCount` — Number of times the objective function was called. `UpdateNorm` — Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `LastImprovement` — Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`. `Algorithm` — Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used. 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 estimation
column vector | matrix

Initial states computed during the estimation, returned as an array containing a column vector corresponding to each experiment.

This array is also stored in the `Parameters` field of the model `Report` property.

For an example, see “Estimate Initial States as Independent Parameters” on page 1-1592.

Algorithms

`ssest` initializes the parameter estimates using either a noniterative subspace approach or an iterative rational function estimation approach. It then refines the parameter values using the prediction error minimization approach. For more information, see `pem` and `ssestOptions`.

References

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

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `ssestOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = ssestOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

Functions

`ssestOptions` | `ssregest` | `idss` | `n4sid` | `tfest` | `procest` | `polyest` | `iddata` | `idfrd` | `canon` | `idgrey` | `pem`

Live Editor Tasks

Estimate State Space Model

Topics

“Estimate State-Space Models at the Command Line”
“Estimate State-Space Models with Free-Parameterization”
“Estimate State-Space Models with Canonical Parameterization”
“Estimate State-Space Models with Structured Parameterization”
“Use State-Space Estimation to Reduce Model Order”
“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”

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`InitializeMethod` — Algorithm used to initialize the state-space parameters
`'auto'` (default) | `'n4sid'` | `'lsrf'`

Algorithm used to initialize the state-space parameter values for `ssest`, specified as one of the following values:

- `'auto'` — `ssest` selects automatically:
  - `lsrf`, if the system is non-MIMO, the data is frequency-domain, and the state-space parameters are real-valued.
  - `n4sid` otherwise (time-domain, MIMO, or with complex-valued state-space parameters).
- `'n4sid'` — Subspace state-space estimation approach — can be used with all systems (see `n4sid`).
- `'lsrf'` — Least-squares rational function estimation-based approach [7] (see “Continuous-Time Transfer Function Estimation Using Continuous-Time Frequency-Domain Data” on page 1-1732) — can provide higher-accuracy results for non-MIMO frequency-domain systems with real-valued state-space parameters, but cannot be used for any other systems (time-domain, MIMO, or with complex-valued state-space parameters).

`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 Variate 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 | `'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 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.
- `'invsqrt'` — Applicable for frequency-domain data only, with `InitializeMethod` set to `'lsrf'` only. Uses $1/\sqrt{|G(\omega)|}$ as the weighting filter, where $G(\omega)$ is the complex frequency-response data. Use this option for capturing relatively low amplitude dynamics in data.
- `'inv'` — Applicable for frequency-domain data only, with `InitializeMethod` set to `'lsrf'` only. Uses $1/|G(\omega)|$ as the weighting filter. Similarly to `'invsqrt'`, this option captures relatively low-amplitude dynamics in data. Use it when `'invsqrt'` weighting produces an estimate that is missing dynamics in the low-amplitude regions. `'inv'` is more sensitive to noise than `'invsqrt'`.

`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`

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

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

If `EstimateCovariance` 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'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

SearchOptions Structure When SearchMethod is Specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description	Default
Tolerance	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
MaxIterations	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIterations = 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

Field Name	Description	Default
Advanced	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
GnPinvConstant	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than GnPinvConstant*max(size(J)*norm(J)*eps) are discarded when computing the search direction. Applicable when SearchMethod is 'gn'.	10000
InitialGnaTolerance	Initial value of *gamma*, specified as a positive scalar. Applicable when SearchMethod is 'gna'.	0.0001
LMStartValue	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
MaxBisections	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	25
MaxFunctionEvaluations	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.	Inf
MinParameterChange	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	0
RelativeImprovement	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	0
StepReduction	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor StepReduction after each try. This reduction continues until MaxBisections tries are completed or a lower value of the criterion function is obtained. StepReduction is not applicable for SearchMethod 'lm' (Levenberg-Marquardt method).	2

SearchOptions Structure When SearchMethod is Specified as 'lsqnonlin'

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of FunctionTolerance is the same as that of opt.SearchOptions.Advanced.TolFun.	1e-5
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of StepTolerance is the same as that of opt.SearchOptions.Advanced.TolX.	1e-6
MaxIterations	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance. The value of MaxIterations is the same as that of opt.SearchOptions.Advanced.MaxIter.	20
Advanced	Advanced search settings, specified as an option set for lsqnonlin. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use optimset('lsqnonlin') to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see "Large-Scale vs. Medium-Scale Algorithms" (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see "Constrained Nonlinear Optimization Algorithms" (Optimization Toolbox) and "Choosing the Algorithm" (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	1e-6
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar.	1e-6
MaxIterations	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance.	100

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

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

  - 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';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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.

[7] Ozdemir, A. A., and S. Gumossoy. "Transfer Function Estimation in System identification Toolbox via Vector Fitting." *Proceedings of the 20th World Congress of the International Federation of Automatic Control.* Toulouse, France, July 2017.

See Also

`ssest`

Topics
“Loss Function and Model Quality Metrics”

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);
```

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.

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` | `ssest` | `n4sid`

Topics

"Estimate State-Space Models at the Command Line"

"Supported State-Space Parameterizations"

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('RegularizationKernel','DC');
m0 = impulseest(eData,70,opt);
```

Convert the model into a state-space model and reduce the model order.

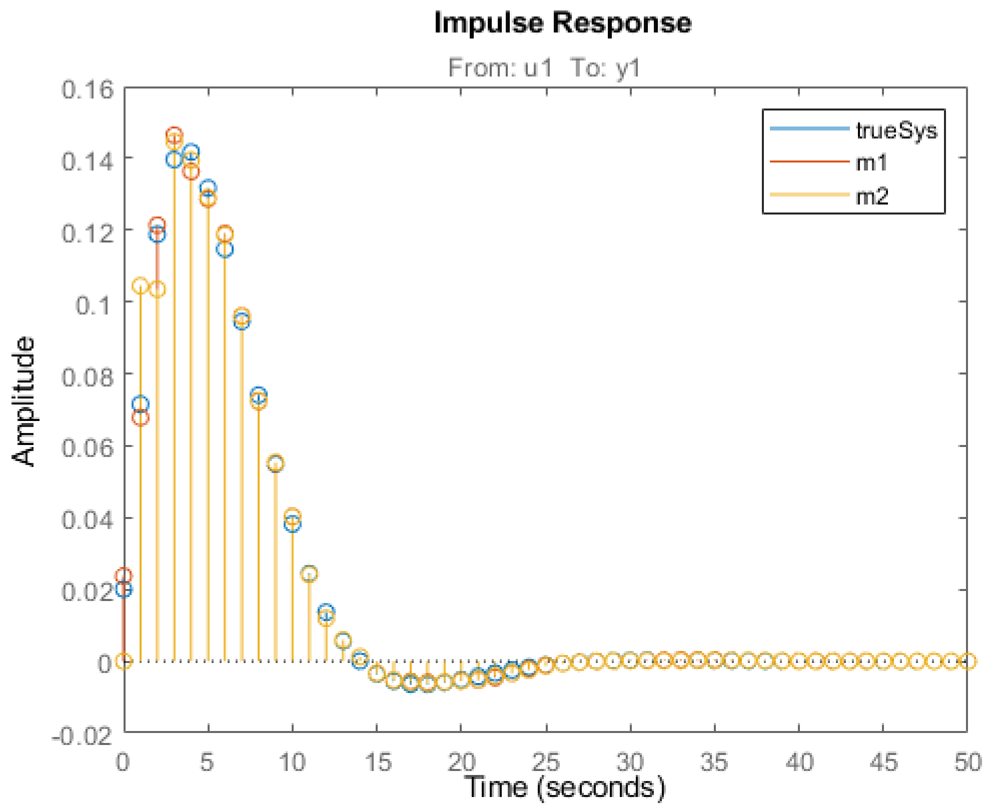
```
m1 = balred(idss(m0),15);
```

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 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 on page 1-1627.
- 'companion' — Obtain sys in companion form on page 1-1628.

- `'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.

Report Field	Description
`Method`	Estimation command used.
`InitialState`	Handling of initial states during estimation, returned as one of the following values: • `'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'`.
`ARXOrder`	ARX model orders, returned as a matrix of nonnegative integers `[na nb nk]`.
`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:

Field	Description
`FitPercent`	Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE).
`LossFcn`	Value of the loss function when the estimation completes.
`MSE`	Mean squared error (MSE) measure of how well the response of the model fits the estimation data.
`FPE`	Final prediction error for the model.
`AIC`	Raw Akaike Information Criteria (AIC) measure of model quality.
`AICc`	Small-sample-size corrected AIC.
`nAIC`	Normalized AIC.
`BIC`	Bayesian Information Criteria (BIC).

`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `ssregestOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description
DataUsed	Attributes of the data used for estimation, returned as a structure with the following fields. **Field** / **Description** Name — Name of the data set. Type — Data type. Length — Number of data samples. Ts — Sample time. InterSample — Input intersample behavior, returned as one of the following values: • 'zoh' — 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. InputOffset — Offset removed from time-domain input data during estimation. For nonlinear models, it is []. OutputOffset — Offset removed from time-domain output data during estimation. For nonlinear models, it is [].

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 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

$$A = \begin{bmatrix} 0 & 0 & 0 & \ldots & 0 & -\alpha_n \\ 1 & 0 & 0 & \ldots & 0 & -\alpha_{n-1} \\ 0 & 1 & 0 & \ldots & 0 & -\alpha_{n-2} \\ 0 & 0 & 1 & \ldots & 0 & -\alpha_{n-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & -\alpha_1 \end{bmatrix}$$

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.

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

`ssregestOptions` | `arxRegul` | `arx` | `balred` | `ssest` | `n4sid`

Topics

"Regularized Estimates of Model Parameters"

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 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`.

`RegularizationKernel` — 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.

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

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

If `EstimateCovariance` 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.

  Default: `'fmincon'`

Output Arguments

`options` — Option set for `ssregest`
`ssregestOptions` options set

Estimation options for `ssregest`, returned as an `ssregestoptions` option set.

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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`

Topics

“Loss Function and Model Quality Metrics”

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” (Control System Toolbox).

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,LineSpec1,...,sysN,LineSpecN,___)
step(___,opts)

y = step(sys,t)
[y,tOut] = step(sys)
[y,tOut] = step(sys,tFinal)
[y,t,x] = step(sys)
[y,t,x,ysd] = step(sys)
[___] = step(___,opts)
```

Description

Step Response Plots

`step(sys)` plots the response of a dynamic system model to a step input of unit amplitude. The model `sys` can be continuous- or discrete-time, and SISO or MIMO. For MIMO systems, the plot displays the step responses for each I/O channel. `step` automatically determines the time steps and duration of the simulation based on the system dynamics.

`step(sys,tFinal)` simulates the step response from `t = 0` to the final time `t = tFinal`. The function uses system dynamics to determine the intervening time steps.

`step(sys,t)` plots the step response at the times that you specify in the vector `t`.

`step(sys1,sys2,...,sysN,___)` plots the step response of multiple dynamic systems on the same plot. All systems must have the same number of inputs and outputs. You can use multiple dynamic systems with any of the previous input-argument combinations.

`step(sys1,LineSpec1,...,sysN,LineSpecN,___)` specifies a color, line style, and marker for each system in the plot. You can use `LineSpec` with any of the previous input-argument combinations. When you need additional plot customization options, use `stepplot` instead.

`step(___,opts)` specifies additional options for computing the step response, such as the step amplitude or input offset. Use `stepDataOptions` to create the option set `opts`. You can use `opts` with any of the previous input-argument and output-argument combinations.

Step Response Data

`y = step(sys,t)` returns the step response of a dynamic system model `sys` at the times specified in the vector `t`. This syntax does not draw a plot.

`[y,tOut] = step(sys)` also returns a vector of times `tOut` corresponding to the responses in `y`. If you do not provide an input vector `t` of times, `step` chooses the length and time step of `tOut` based on the system dynamics.

`[y,tOut] = step(sys,tFinal)` computes the step response up to the end time `tFinal`. `step` chooses the time step of `tOut` based on the system dynamics.

`[y,t,x] = step(sys)` also returns the state trajectories `x`, when `sys` is a state-space model such as an `ss` or `idss` model.

`[y,t,x,ysd] = step(sys)` also computes the standard deviation `ysd` of the step response `y`, when `sys` is an identified model such as an `idss`, `idtf`, or `idnlarx` model.

`[ ___ ] = step( ___ ,opts)` specifies additional options for computing the step response, such as the step amplitude or input offset. Use `stepDataOptions` to create the option set `opts`. You can use `opts` with any of the previous input-argument and output-argument combinations.

Examples

Step Response of Dynamic System

Plot the step response of a continuous-time system represented by the following transfer function.

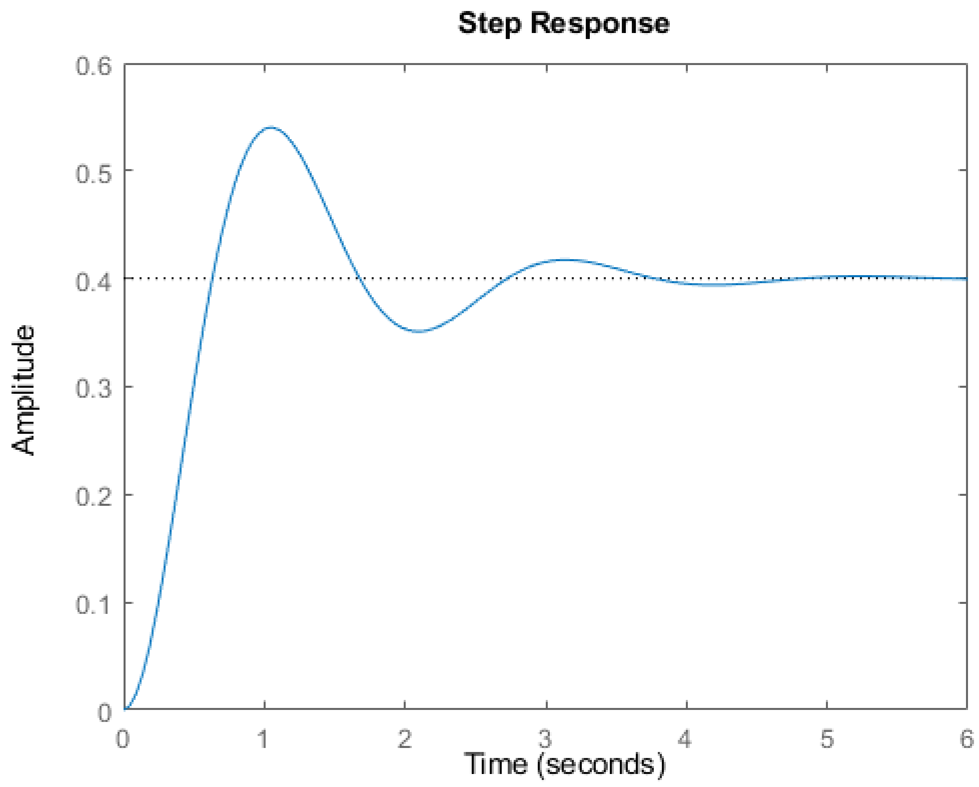
$$\mathrm{sys}(s) = \frac{4}{s^2 + 2s + 10}.$$

For this example, create a `tf` model that represents the transfer function. You can similarly plot the step response of other dynamic system model types, such as zero-pole gain (`zpk`) or state-space (`ss`) models.

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

Plot the step response.

```
step(sys)
```



The `step` plot automatically includes a dotted horizontal line indicating the steady-state response. In a MATLAB® figure window, you can right-click on the plot to view other step-response characteristics such as peak response and settling time. For more information about these characteristics, see `stepinfo` (Control System Toolbox).

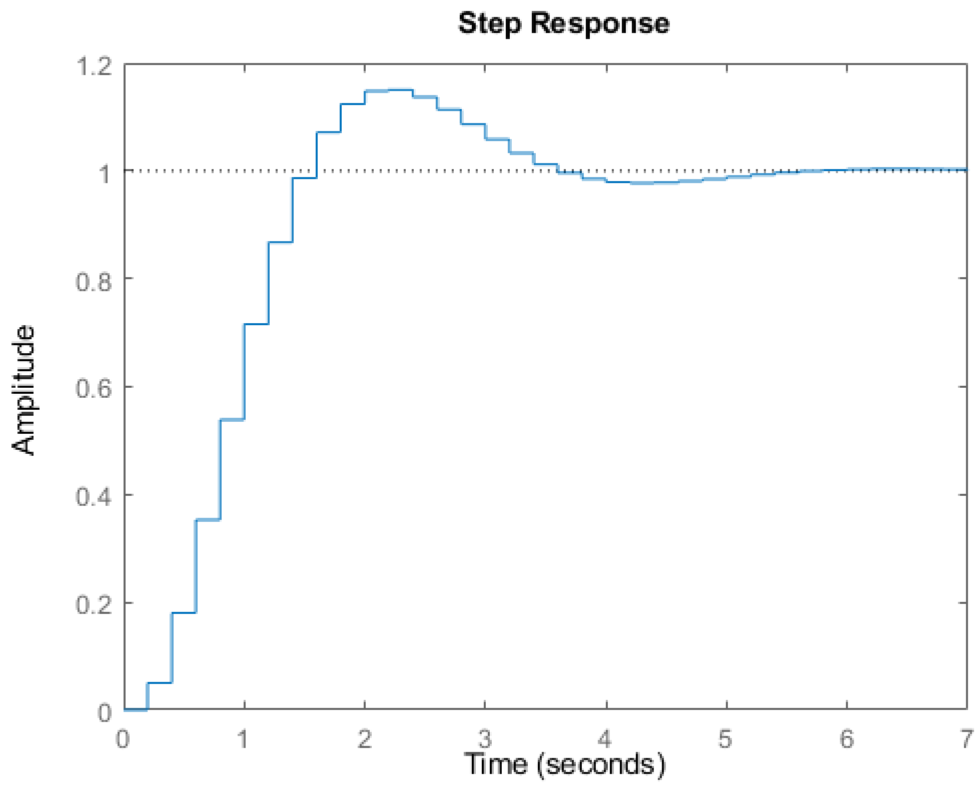
Step Response of Discrete-Time System

Plot the step response of a discrete-time system. The system has a sample time of 0.2 s and is represented by the following state-space matrices.

```
A = [1.6 -0.7;
      1  0];
B = [0.5; 0];
C = [0.1 0.1];
D = 0;
```

Create the state-space model and plot its step response.

```
sys = ss(A,B,C,D,0.2);
step(sys)
```



The step response reflects the discretization of the model, showing the response computed every 0.2 seconds.

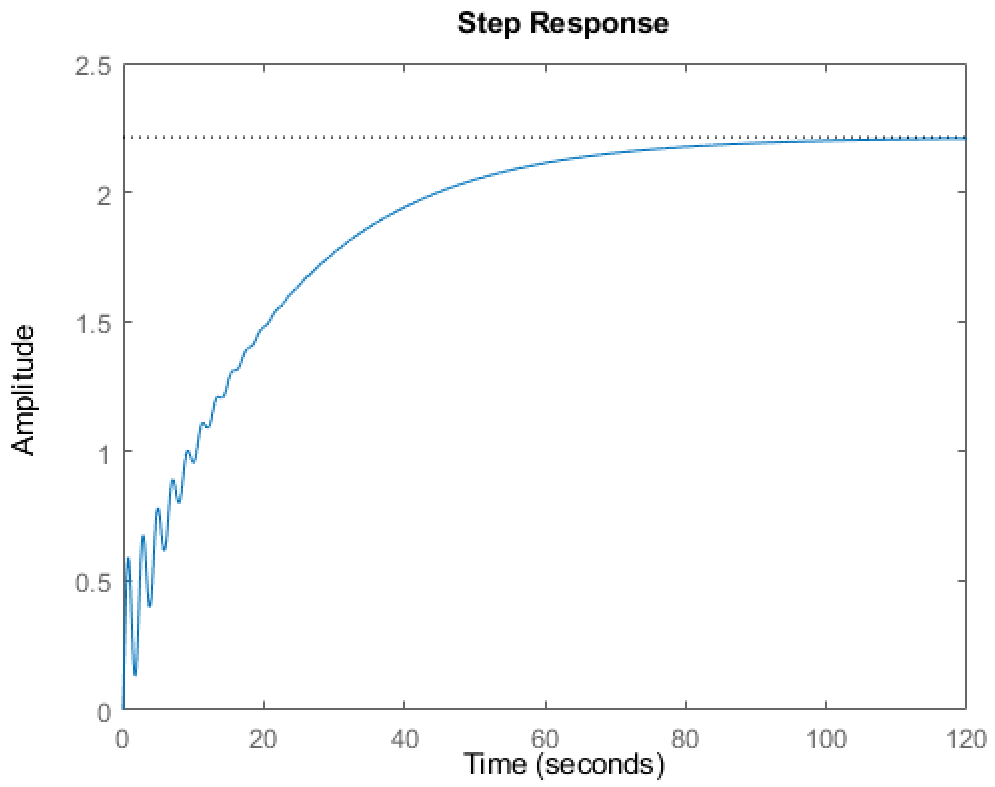
Step Response at Specified Times

Examine the step response of the following transfer function.

```
sys = zpk(-1,[-0.2+3j,-0.2-3j],1) * tf([1 1],[1 0.05])
```

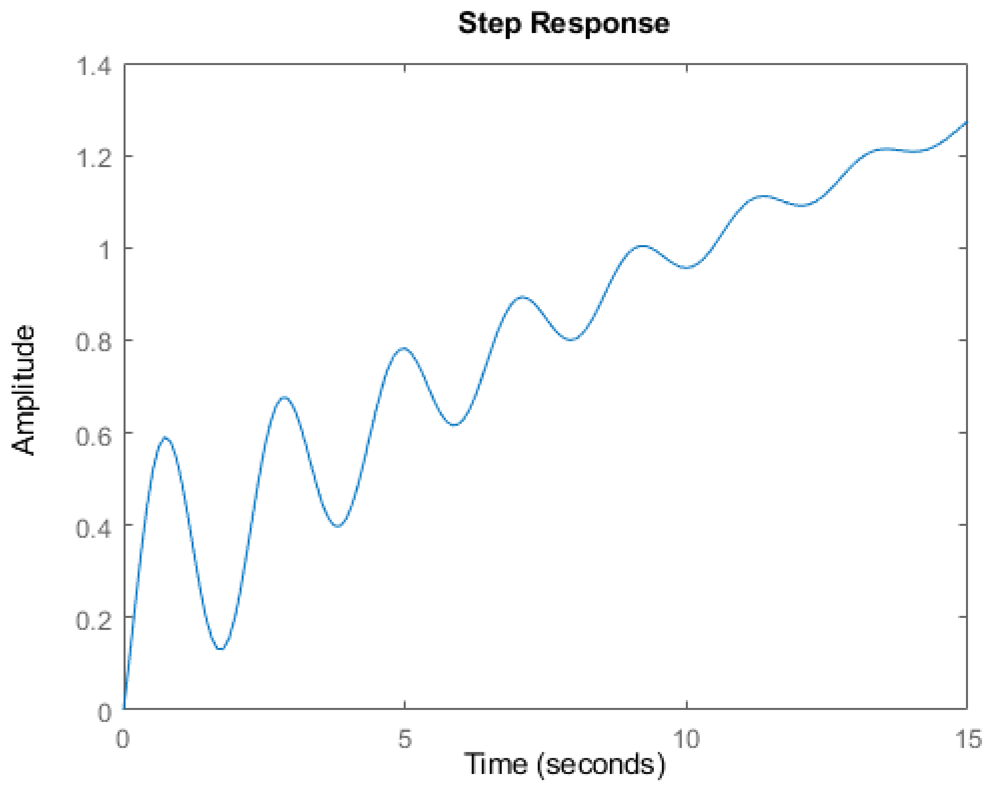
```
sys =
 
            (s+1)^2
  ----------------------------
  (s+0.05) (s^2 + 0.4s + 9.04)
 
Continuous-time zero/pole/gain model.
```

```
step(sys)
```



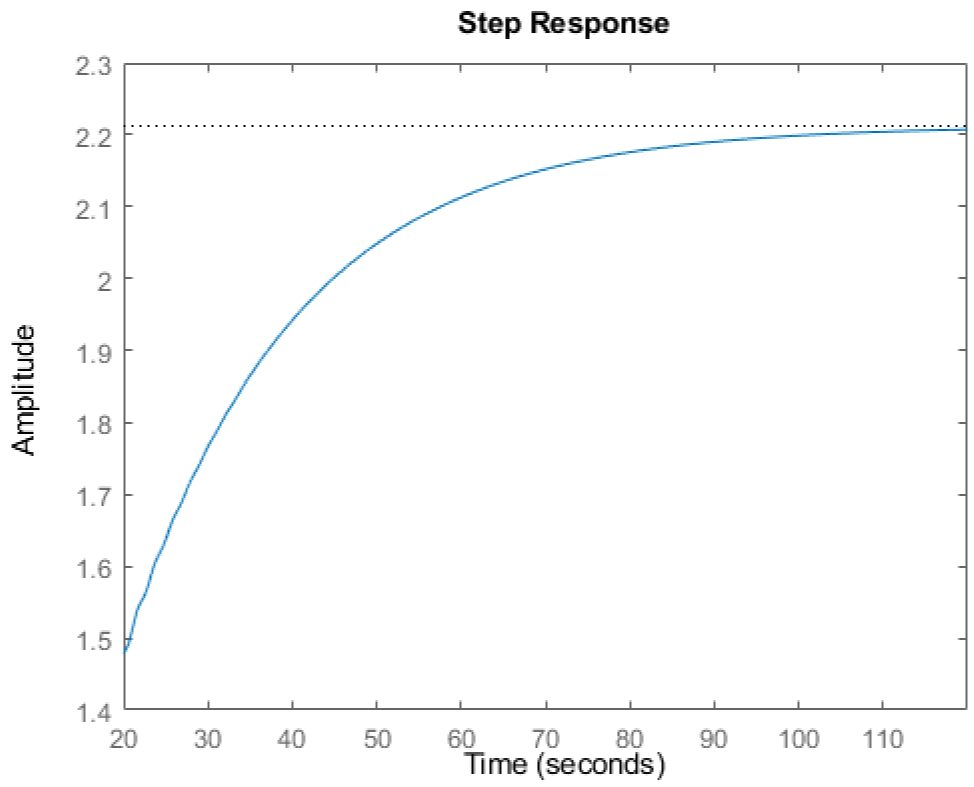
By default, `step` chooses an end time that shows the steady state that the response is trending toward. This system has fast transients, however, which are obscured on this time scale. To get a closer look at the transient response, limit the step plot to `t` = 15 s.

```
step(sys,15)
```



Alternatively, you can specify the exact times at which you want to examine the step response, provided they are separated by a constant interval. For instance, examine the response from the end of the transient until the system reaches steady state.

```
t = 20:0.2:120;
step(sys,t)
```



Even though this plot begins at `t` = 20, `step` always applies the step input at `t` = 0.

Step Response Plot of MIMO Systems

Consider the following second-order state-space model:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}-0.5572 & -0.7814\\ 0.7814 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + \begin{bmatrix}1 & -1\\ 0 & 2\end{bmatrix}\begin{bmatrix}u_1\\ u_2\end{bmatrix}$$

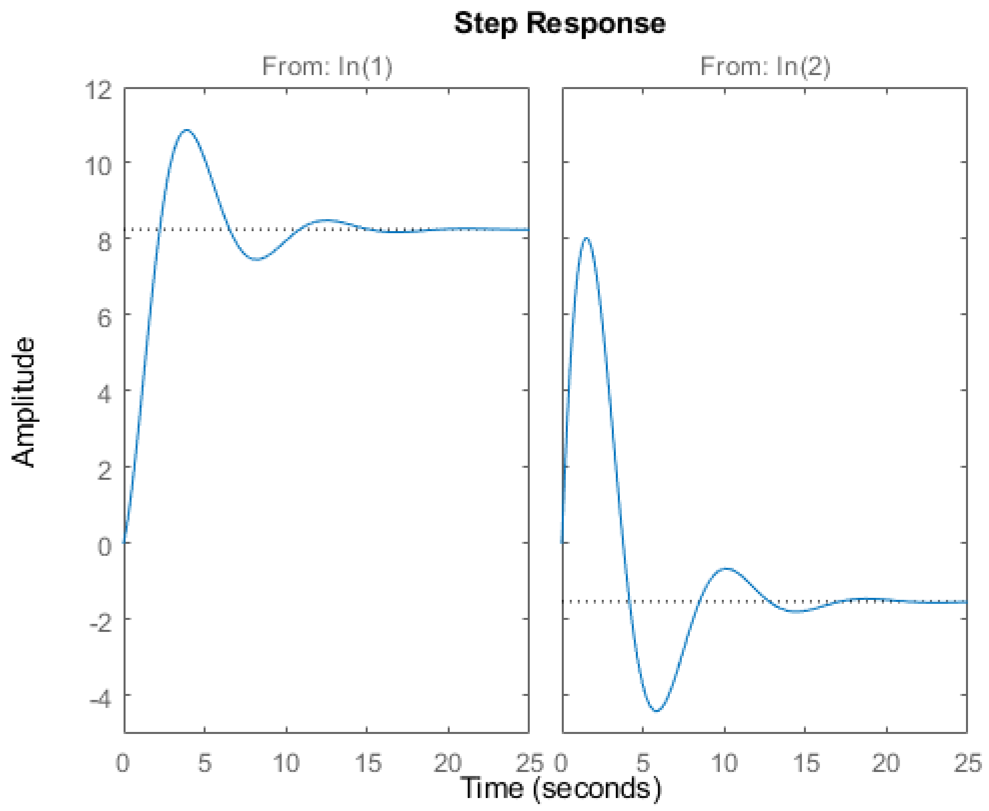
$$y = \begin{bmatrix}1.9691 & 6.4493\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}$$

```
A = [-0.5572,-0.7814;0.7814,0];
B = [1,-1;0,2];
C = [1.9691,6.4493];
sys = ss(A,B,C,0);
```

This model has two inputs and one output, so it has two channels: from the first input to the output, and from the second input to the output. Each channel has its own step response.

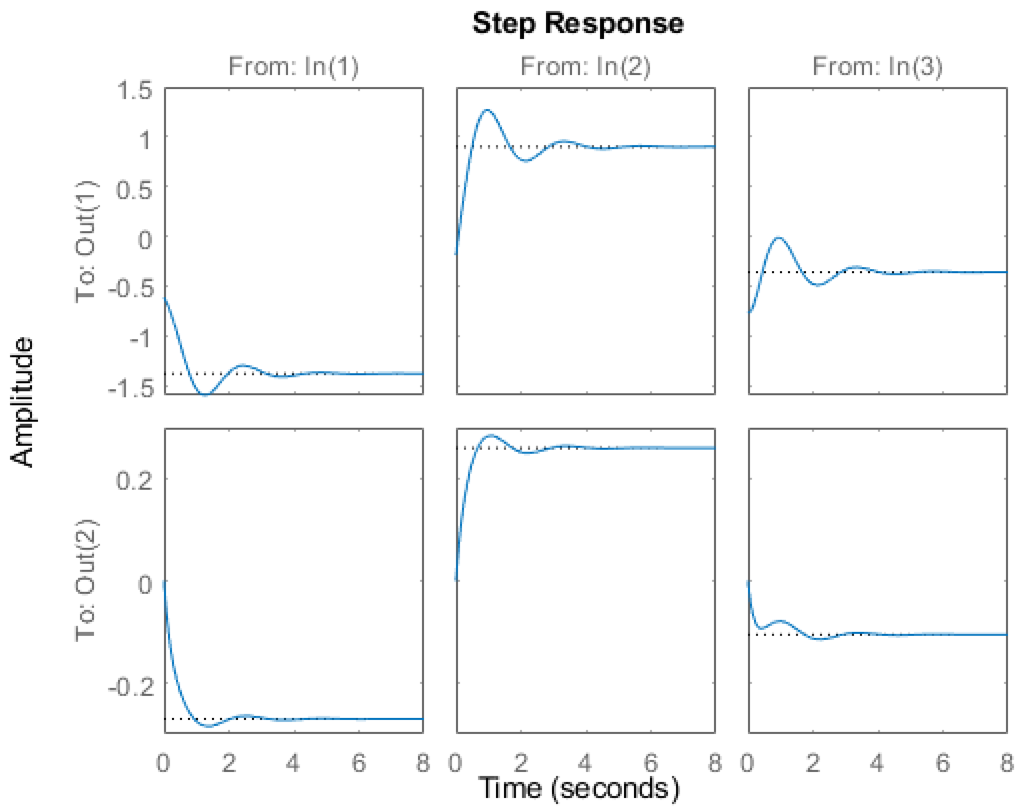
When you use `step`, it computes the responses of all channels.

```
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. Whenever you use `step` to plot the responses of a MIMO model, it generates an array of plots representing all the I/O channels of the model. For instance, create a random state-space model with five states, three inputs, and two outputs, and plot its step response.

```
sys = rss(5,2,3);
step(sys)
```



In a MATLAB figure window, you can restrict the plot to a subset of channels by right-clicking on the plot and selecting **I/O Selector**.

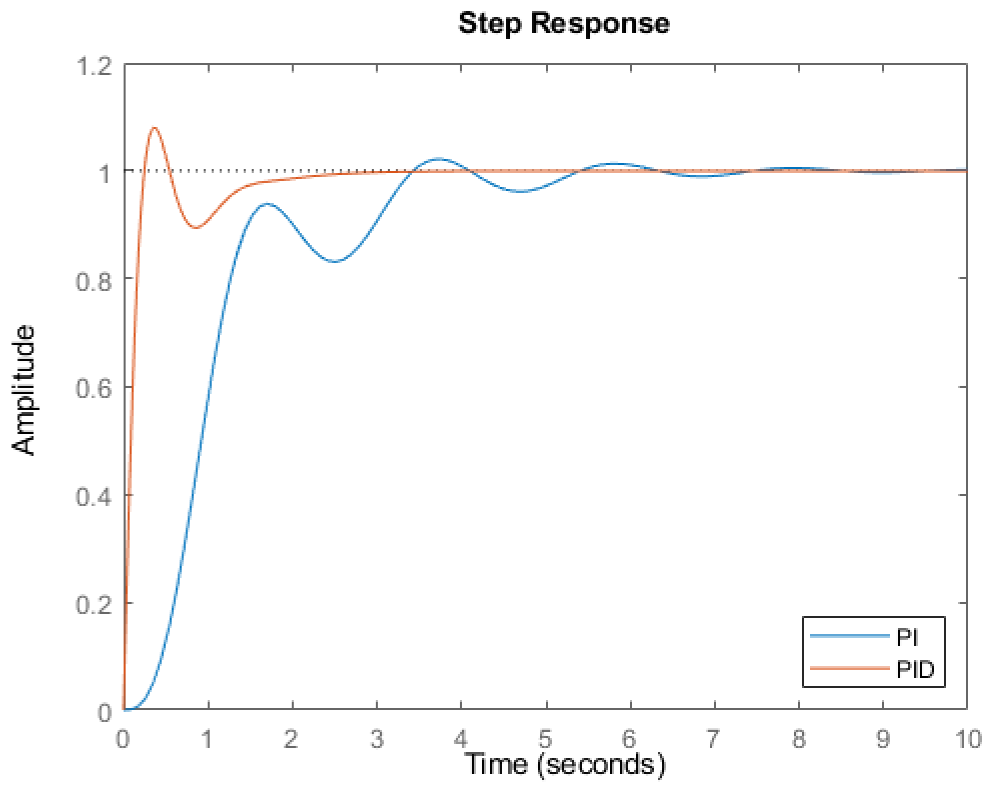
Compare Responses of Multiple Systems

`step` allows you to plot the responses of multiple dynamic systems on the same axis. For instance, compare the closed-loop response of a system with a PI controller and a PID controller. Create a transfer function of the system and tune the controllers.

```
H = tf(4,[1 2 10]);
C1 = pidtune(H,'PI');
C2 = pidtune(H,'PID');
```

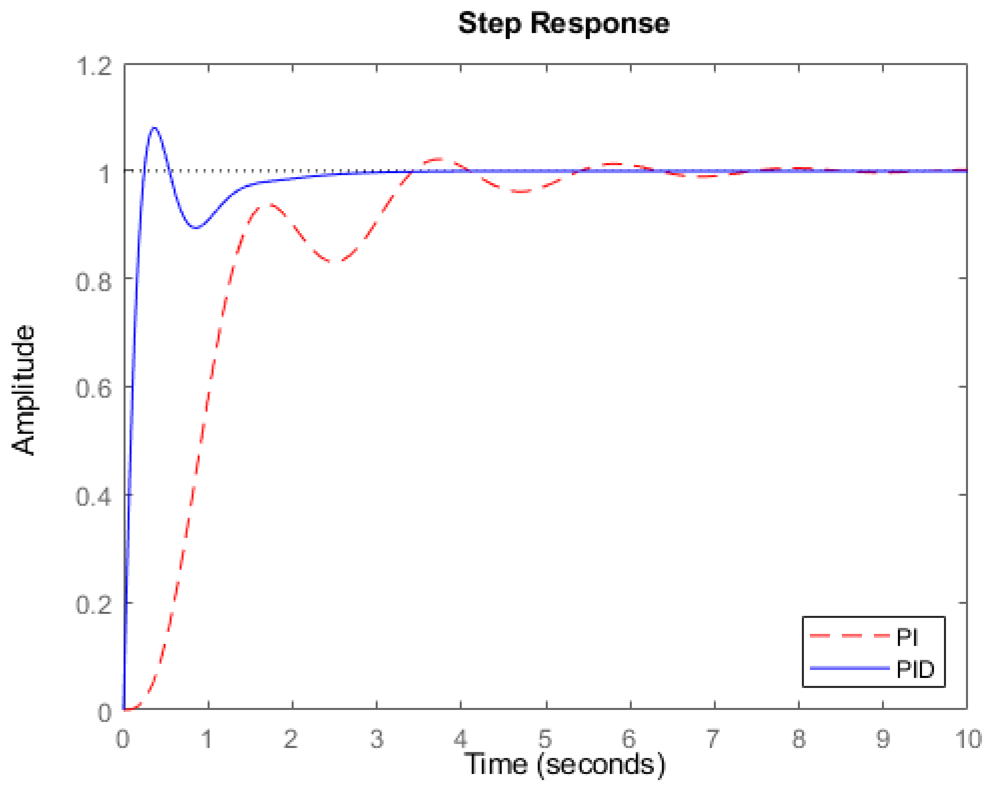
Form the closed-loop systems and plot their step responses.

```
sys1 = feedback(H*C1,1);
sys2 = feedback(H*C2,1);
step(sys1,sys2)
legend('PI','PID','Location','SouthEast')
```



By default, `step` chooses distinct colors for each system that you plot. You can specify colors and line styles using the `LineSpec` input argument.

```
step(sys1,'r--',sys2,'b')
legend('PI','PID','Location','SouthEast')
```



The first LineSpec `'r--'` specifies a dashed red line for the response with the PI controller. The second LineSpec `'b'` specifies a solid blue line for the response with the PID controller. The legend reflects the specified colors and linestyles. For more plot customization options, use `stepplot`.

Step Response of Systems in a Model Array

The example Compare Responses of Multiple Systems shows how to plot responses of several individual systems on a single axis. When you have multiple dynamic systems arranged in a model array, `step` plots all their responses at once.

Create a model array. For this example, use a one-dimensional array of second-order transfer functions having different natural frequencies. First, preallocate memory for the model array. The following command creates a 1-by-5 row of zero-gain SISO transfer functions. The first two dimensions represent the model outputs and inputs. The remaining dimensions are the array dimensions.

```
 sys = tf(zeros(1,1,1,5));
```

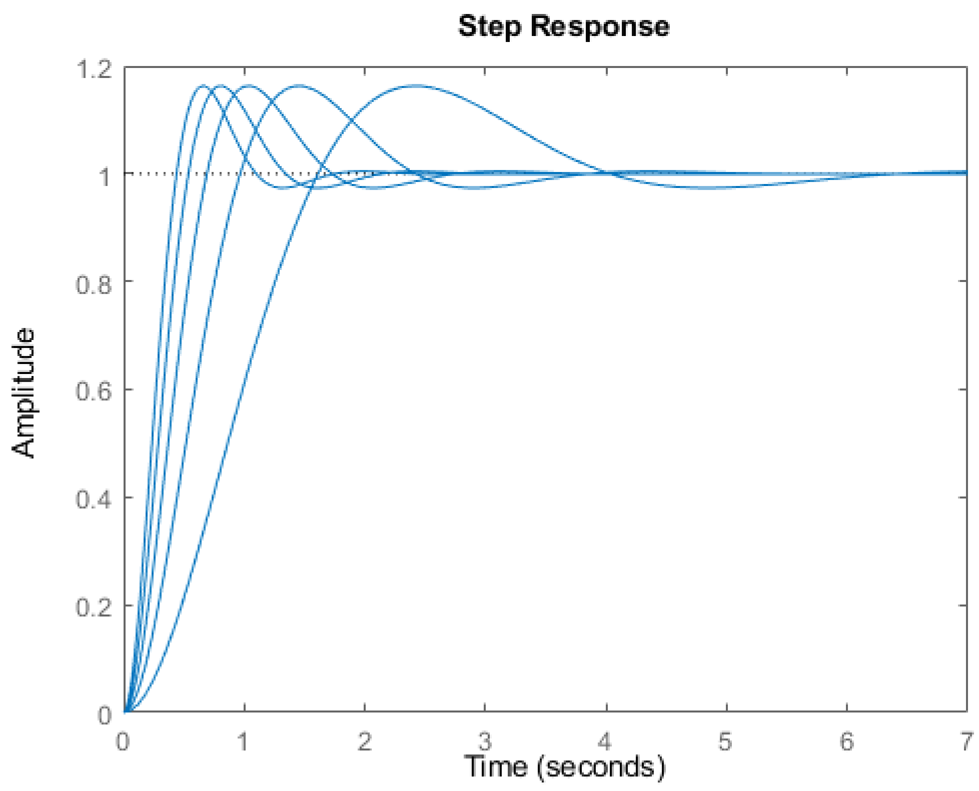
Populate the array.

```
w0 = 1.5:1:5.5;    % natural frequencies
zeta = 0.5;        % damping constant
for i = 1:length(w0)
```

```
   sys(:,:,1,i) = tf(w0(i)^2,[1 2*zeta*w0(i) w0(i)^2]);
end
```

(For more information about model arrays and how to create them, see “Model Arrays” (Control System Toolbox).) Plot the step responses of all models in the array.

```
step(sys)
```



`step` uses the same linestyle for the responses of all entries in the array. One way to distinguish among entries is to use the `SamplingGrid` property of dynamic system models to associate each entry in the array with the corresponding `w0` value.

```
sys.SamplingGrid = struct('frequency',w0);
```

Now, when you plot the responses in a MATLAB figure window, you can click a trace to see which frequency value it corresponds to.

Step Response Data

When you give it an output argument, `step` returns an array of response data. For a SISO system, the response data is returned as a column vector of length equal to the number of time points at which the response is sampled. You can provide the vector t of time points, or allow `step` to select time points for you based on system dynamics. For instance, extract the step response of a SISO system at 101 time points between $t = 0$ and $t = 5$ s.

```
sys = tf(4,[1 2 10]);
t = 0:0.05:5;
y = step(sys,t);
size(y)
```

```
ans = 1×2

   101     1
```

For a MIMO system, the response data is returned in an array of dimensions *N*-by-*Ny*-by-*Nu*, where *Ny* and *Nu* are the number of outputs and inputs of the dynamic system. For instance, consider the following state-space model, representing a two-input, one-output system.

```
A = [-0.5572,-0.7814;0.7814,0];
B = [1,-1;0,2];
C = [1.9691,6.4493];
sys = ss(A,B,C,0);
```

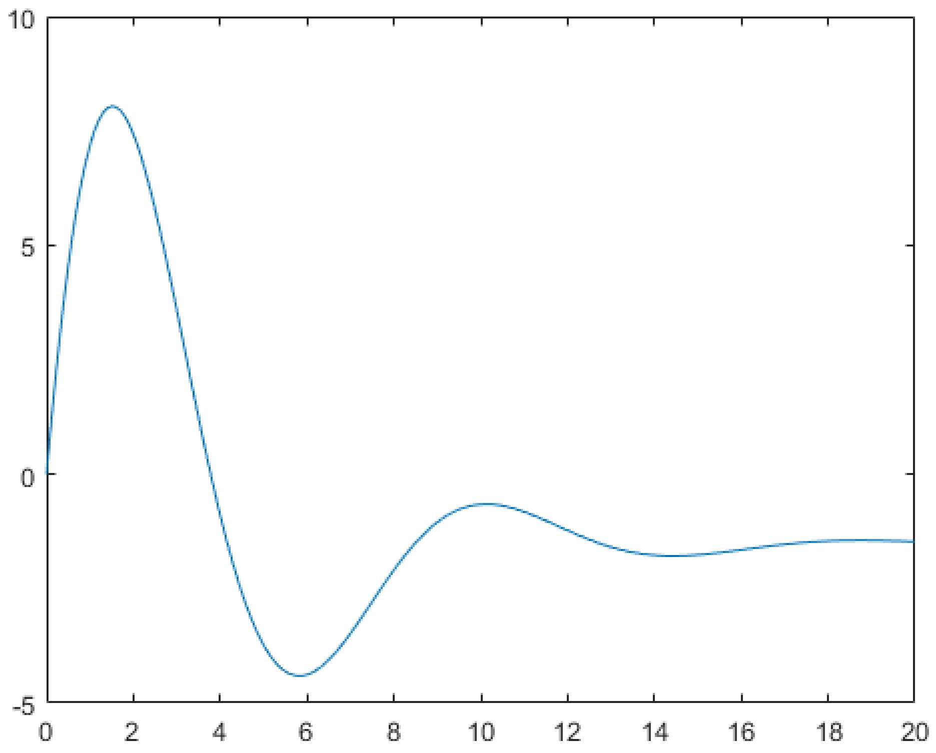
Extract the step response of this system at 200 time points between $t = 0$ and $t = 20$ s.

```
t = linspace(0,20,200);
y = step(sys,t);
size(y)
```

```
ans = 1×3

   200     1     2
```

`y(:,i,j)` is a column vector containing the step response from the *j*th input to the *i*th output at the times *t*. For instance, extract the step response from the second input to the output.

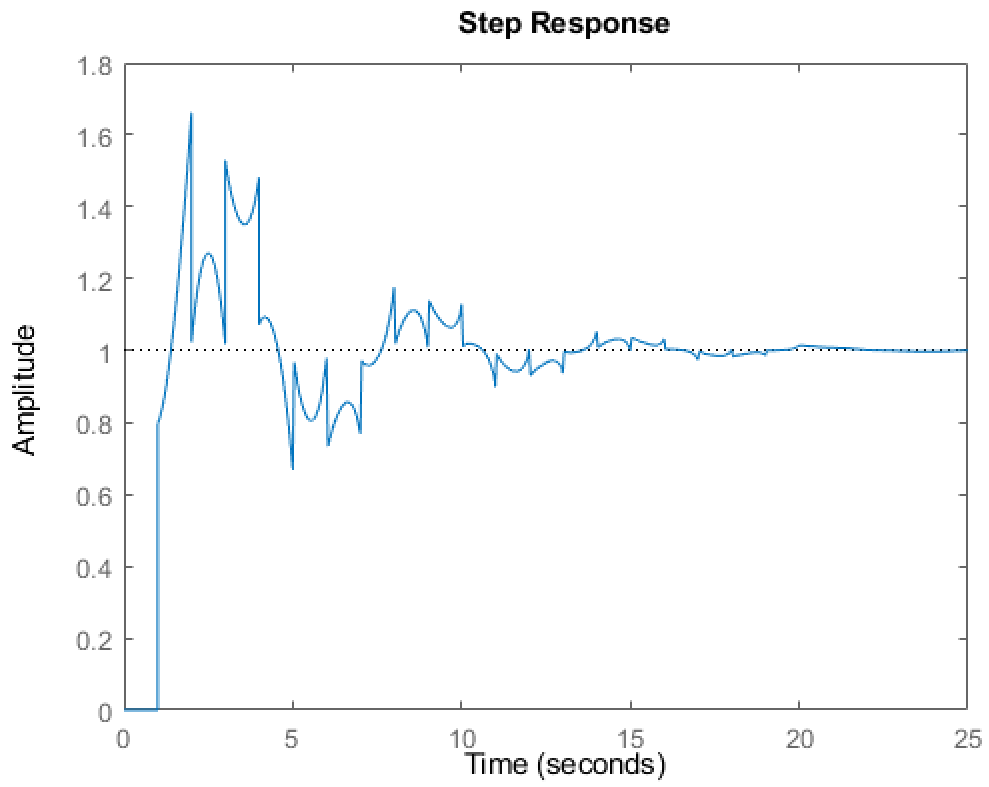
```
y12 = y(:,1,2);
plot(t,y12)
```



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);
sys = feedback(ss(G),1);
step(sys)
```

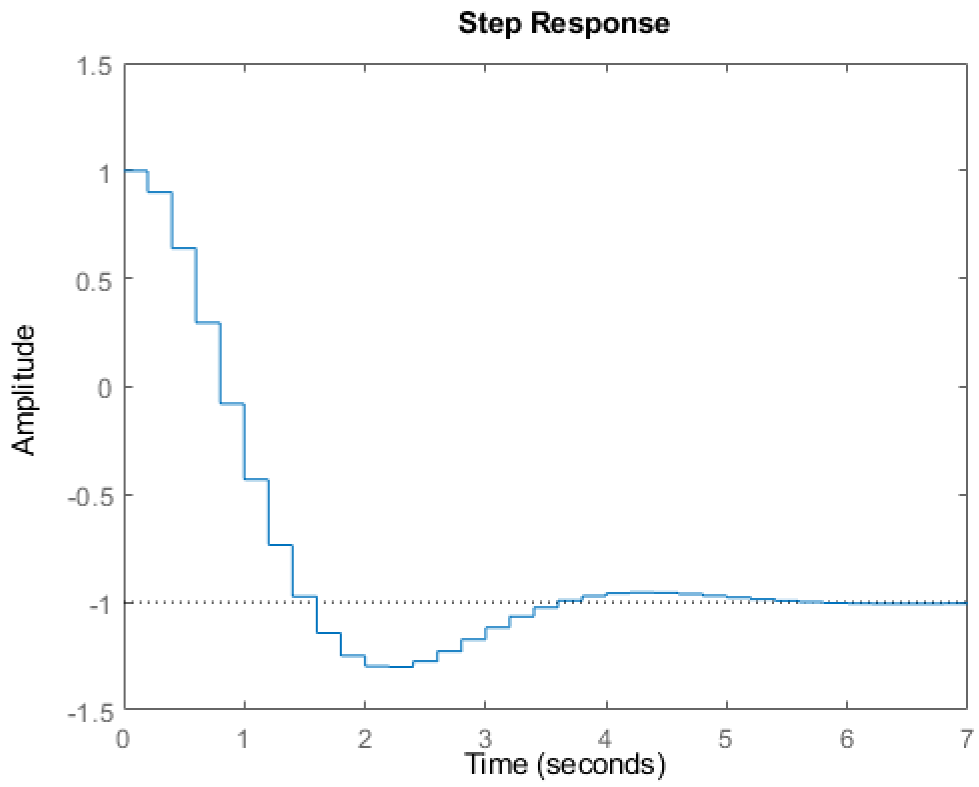


The system step response displayed is chaotic. The step response of systems with internal delays may exhibit odd behavior, such as recurring jumps. Such behavior is a feature of the system and not software anomalies.

Response to Custom Step Input

By default, `step` applies an input signal that changes from 0 to 1 at $t = 0$. To customize the amplitude and offset, use `stepDataOptions`. For instance, compute the response of a SISO state-space model to a signal that changes from 1 to –1 to at $t = 0$.

```
A = [1.6 -0.7;
      1  0];
B = [0.5; 0];
C = [0.1 0.1];
D = 0;
sys = ss(A,B,C,D,0.2);

opt = stepDataOptions;
opt.InputOffset = 1;
opt.StepAmplitude = -2;

step(sys,opt)
```



For responses to arbitrary input signals, use `lsim` (Control System Toolbox).

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
```

Estimate a parametric model.

```
sys1 = ssest(z1,4);
```

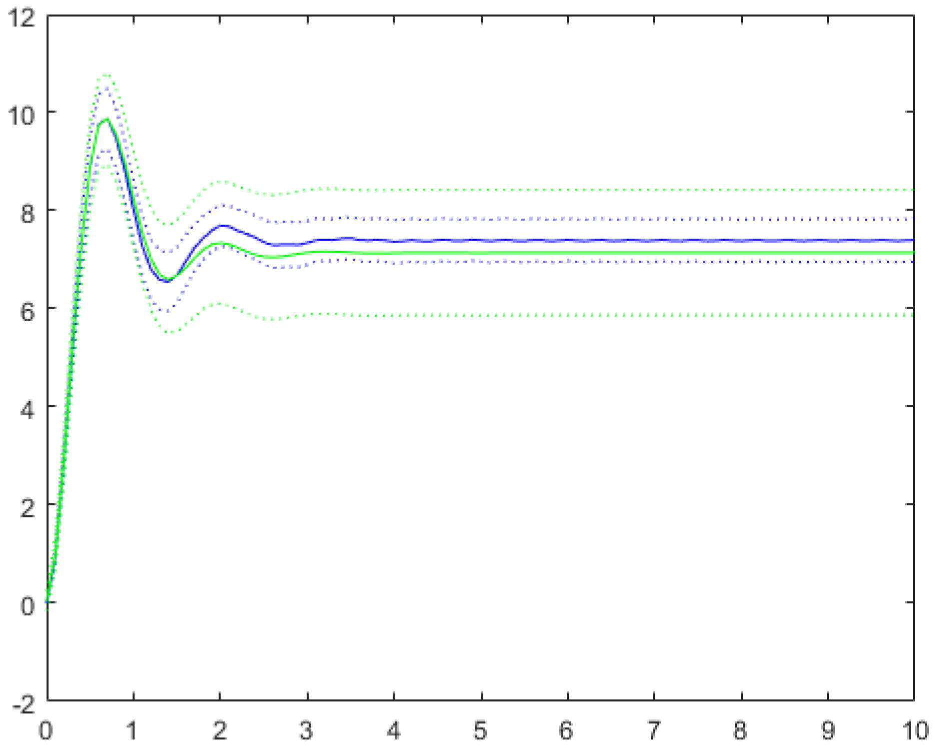
Estimate a non-parametric model.

```
sys2 = impulseest(z1);
```

Plot the step responses for comparison.

```
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:')
```



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;
```

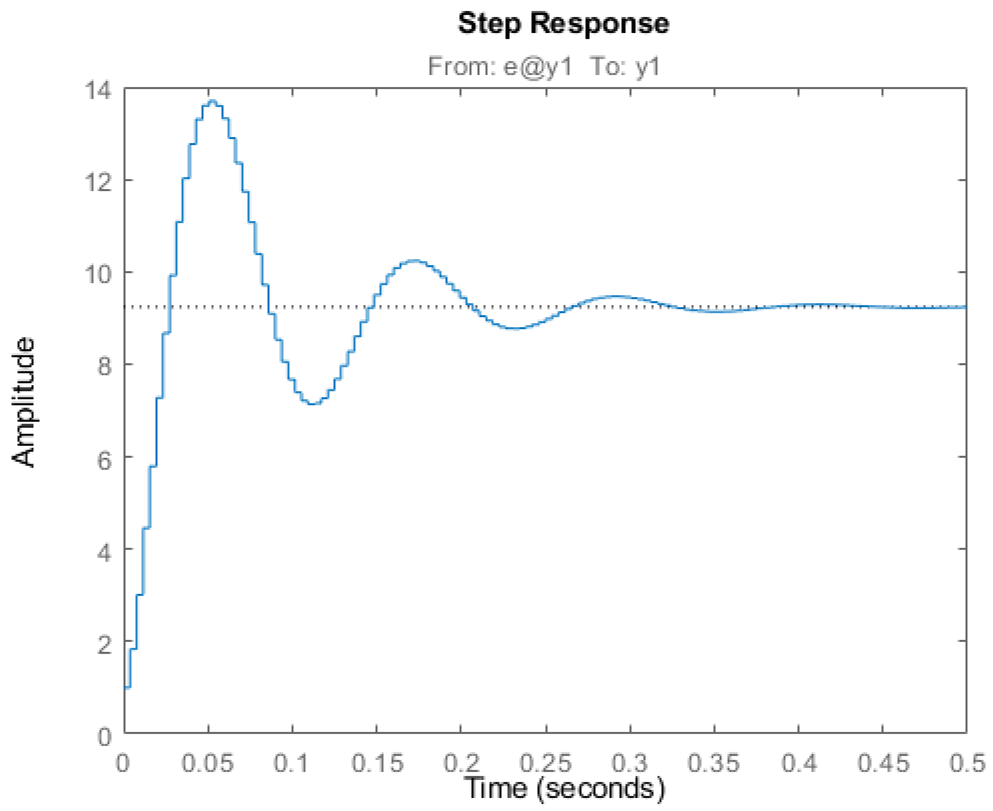
Estimate a time-series model.

```
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)
```



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],idTreePartition,'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);
```

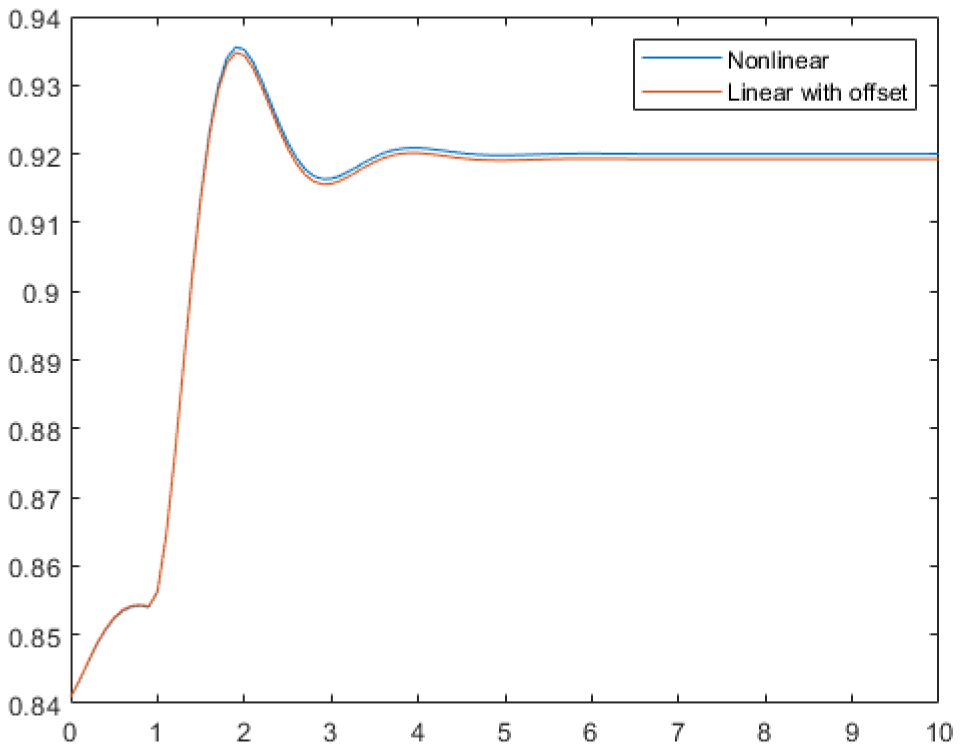
The linear system `sys` expresses the relationship between the perturbations in input to the corresponding perturbation in output. It is unaware of the nonlinear system's equilibrium values.

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')
```



Input Arguments

sys — Dynamic system

dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value for both plotting and returning response data.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model. When you use output arguments, the function returns response data for the nominal model only.
- Sparse state-space models such as `sparss` and `mechss` models.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models. For such models, the function can also plot confidence intervals and return standard deviations of the frequency response. See “Step Responses of Identified Models with Confidence Regions” on page 1-1651.

`step` does not support frequency-response data models such as `frd`, `genfrd`, or `idfrd` models.

If `sys` is an array of models, the function plots the responses of all models in the array on the same axes. See “Step Response of Systems in a Model Array” on page 1-1646.

`tFinal` — End time for step response

positive scalar

End time for the step response, specified as a positive scalar value. `step` simulates the step response from `t = 0` to `t = tFinal`.

- For continuous-time systems, the function determines the step size and number of points automatically from system dynamics. Express `tFinal` in the system time units, specified in the `TimeUnit` property of `sys`.
- For discrete-time systems, the function uses the sample time of `sys` as the step size. 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.

t — Time vector

vector

Time vector at which to compute the step response, specified as a vector of positive scalar values. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`.

- For continuous-time models, specify `t` in the form `Ti:dt:Tf`. To obtain the response at each time step, the function uses `dt` as the sample time of a discrete approximation to the continuous system (see “Algorithms” on page 1-1657).
- For discrete-time models, specify `t` in the form `Ti:Ts:Tf`, where `Ts` is the sample time of `sys`.

`step` always applies the step input at `t = 0`, regardless of `Ti`.

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a string or vector of one, two, or three characters. The characters can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line. For more information about configuring this argument, see the `LineSpec` input argument of the `plot` function.

Example: `'r--'` specifies a red dashed line

Example: `'*b'` specifies blue asterisk markers

Example: `'y'` specifies a yellow line

`opts` — Input offset and amplitude
`stepDataOptions` options set

Input offset and amplitude of the applied step signal, specified as a `stepDataOptions` option set. By default, `step` applies an input that goes from 0 to 1 at time `t = 0`. Use this input argument to change the initial and final values of the step input. See “Response to Custom Step Input” on page 1-1650 for an example.

Output Arguments

`y` — Step response data
array

Step response data, returned as an array.

- For SISO systems, `y` is a column vector of the same length as `t` (if provided) or `tOut` (if you do not provide `t`).
- For single-input, multi-output systems, `y` is a matrix with as many rows as there are time samples and as many columns as there are outputs. Thus, the *j*th column of `y`, or `y(:,j)`, contains the step response of from the input to the *j*th output.
- For MIMO systems, the step responses of each input channel are stacked up along the third dimension of `y`. The dimensions of `y` are then *N*-by-*Ny*-by-*Nu*, where:

  - *N* is the number of time samples.
  - *Ny* is the number of system outputs.
  - *Nu* is the number of system inputs.

  Thus, `y(:,i,j)` is a column vector containing the step response from the *j*th input to the *i*th output at the times specified in `t` or `tOut`.

`tOut` — Times at which step response is computed
vector

Times at which step response is computed, returned as a vector. When you do not provide a specific vector `t` of times, `step` chooses this time vector based on the system dynamics. The times are expressed in the time units of `sys`.

`x` — State trajectories
array

State trajectories, returned as an array. When `sys` is a state-space model, `x` contains the evolution of the states of `sys` at each time in `t` or `tOut`. The dimensions of `x` are *N*-by-*Nx*-by-*Nu*, where:

- *N* is the number of time samples.
- *Nx* is the number of states.
- *Nu* is the number of system inputs.

Thus, the evolution of the states in response to a step injected at the *k*th input is given by the array `x(:,:,k)`. The row vector `x(i,:,k)` contains the state values at the *i*th time step.

ysd — Standard deviation of step response
array

Standard deviation of the step response of an identified model, returned as an array of the same dimensions as `y`. If `sys` does not contain parameter covariance information, then `ysd` is empty.

Tips

- When you need additional plot customization options, use `stepplot` instead.
- To simulate system responses to arbitrary input signals, use `lsim`.

Algorithms

To obtain samples of continuous-time models without internal delays, `step` converts such models to state-space models and discretizes them using a zero-order hold on the inputs. `step` chooses the sampling time for this discretization automatically based on the system dynamics, except when you supply the input time vector `t` in the form `t = 0:dt:Tf`. In that case, `step` uses `dt` as the sampling time. The resulting simulation time steps `tOut` are equisampled with spacing `dt`.

For systems with internal delays, Control System Toolbox software uses variable step solvers. As a result, the time steps `tOut` 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

Functions
`impulse` | `stepDataOptions` | `initial` | `lsim` | `stepplot`

Apps
Linear System Analyzer

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 = 1×2

    1.0000   -0.8298
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans = 3×3

    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 the recursive least squares algorithm.

```
obj = recursiveLS(2);
```

Load the estimation data, which for this example is a static data set.

```
load iddata3
input = z3.u;
output = z3.y;
```

Create a variable to store `u(t-1)`. This variable is updated at each time step.

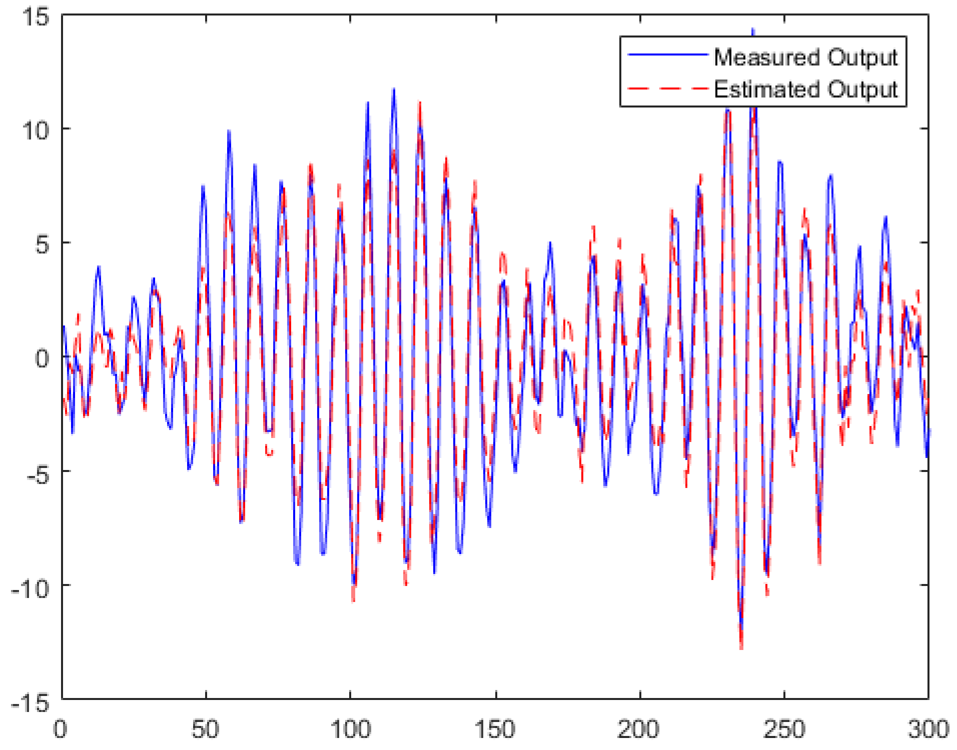
```
oldInput = 0;
```

Estimate the parameters and output using `step` and input-output data, maintaining the current regressor pair in `H`. Invoke the `step` function implicitly by calling the `obj` system object with input arguments.

```
for i = 1:numel(input)
    H = [input(i) oldInput];
    [theta, EstimatedOutput] = obj(output(i),H);
    estimatedOut(i)= EstimatedOutput;
    theta_est(i,:) = theta;
    oldInput = input(i);
end
```

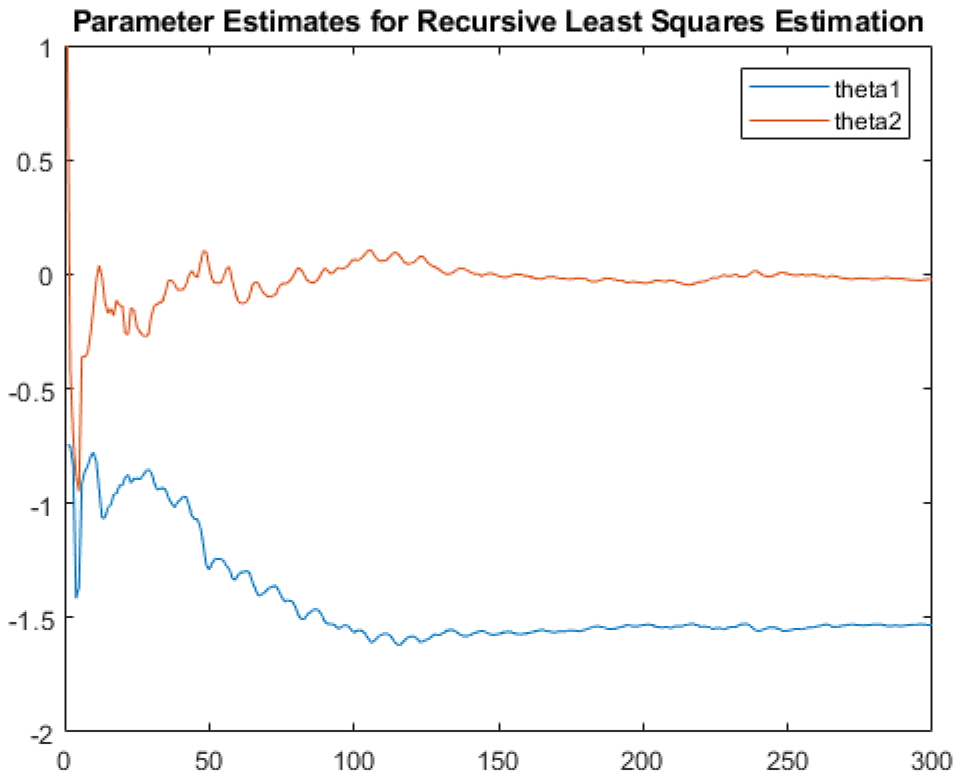
Plot the measured and estimated output data.

```
numSample = 1:numel(input);
plot(numSample,output,'b',numSample,estimatedOut,'r--');
legend('Measured Output','Estimated Output');
```



Plot the parameters.

```
plot(numSample,theta_est(:,1),numSample,theta_est(:,2))
title('Parameter Estimates for Recursive Least Squares Estimation')
legend("theta1","theta2")
```



View the final estimates.

```
theta_final = theta
```

```
theta_final = 2×1

   -1.5322
   -0.0235
```

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`.

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.

See Also

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

Topics
"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"
"What Is Online Estimation?"

Introduced in R2015b

stepDataOptions

Options for the `step` command

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.

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);
```

Input Arguments

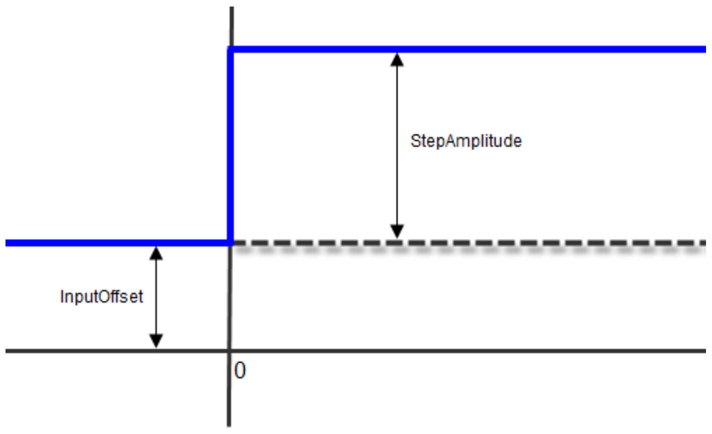
Name-Value Pair Arguments

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

Example: `...,'InputOffset',2`

`InputOffset` — Input signal offset level
0 (default) | scalar

Input signal offset level, specified as the comma-separated pair consisting of '`InputOffset`' and a scalar, for all time t < 0, as shown in the following figure.



`StepAmplitude` — Change of input signal offset
1 (default) | scalar

Change of input signal offset level, specified as the comma-separated pair consisting of '`StepAmplitude`' and a scalar, which occurs at time t = 0, as shown in the previous figure.

Output Arguments

`opt` — Options for the `step` command
`step` options set

Options for the `step` command, returned as a `step` options set.

See Also
`step`

Introduced in R2012a

stepinfo

Rise time, settling time, and other step-response characteristics

Syntax

```
S = stepinfo(sys)

S = stepinfo(y,t)
S = stepinfo(y,t,yfinal)
S = stepinfo(y,t,yfinal,yinit)

S = stepinfo( ___ ,'SettlingTimeThreshold',ST)
S = stepinfo( ___ ,'RiseTimeLimits',RT)
```

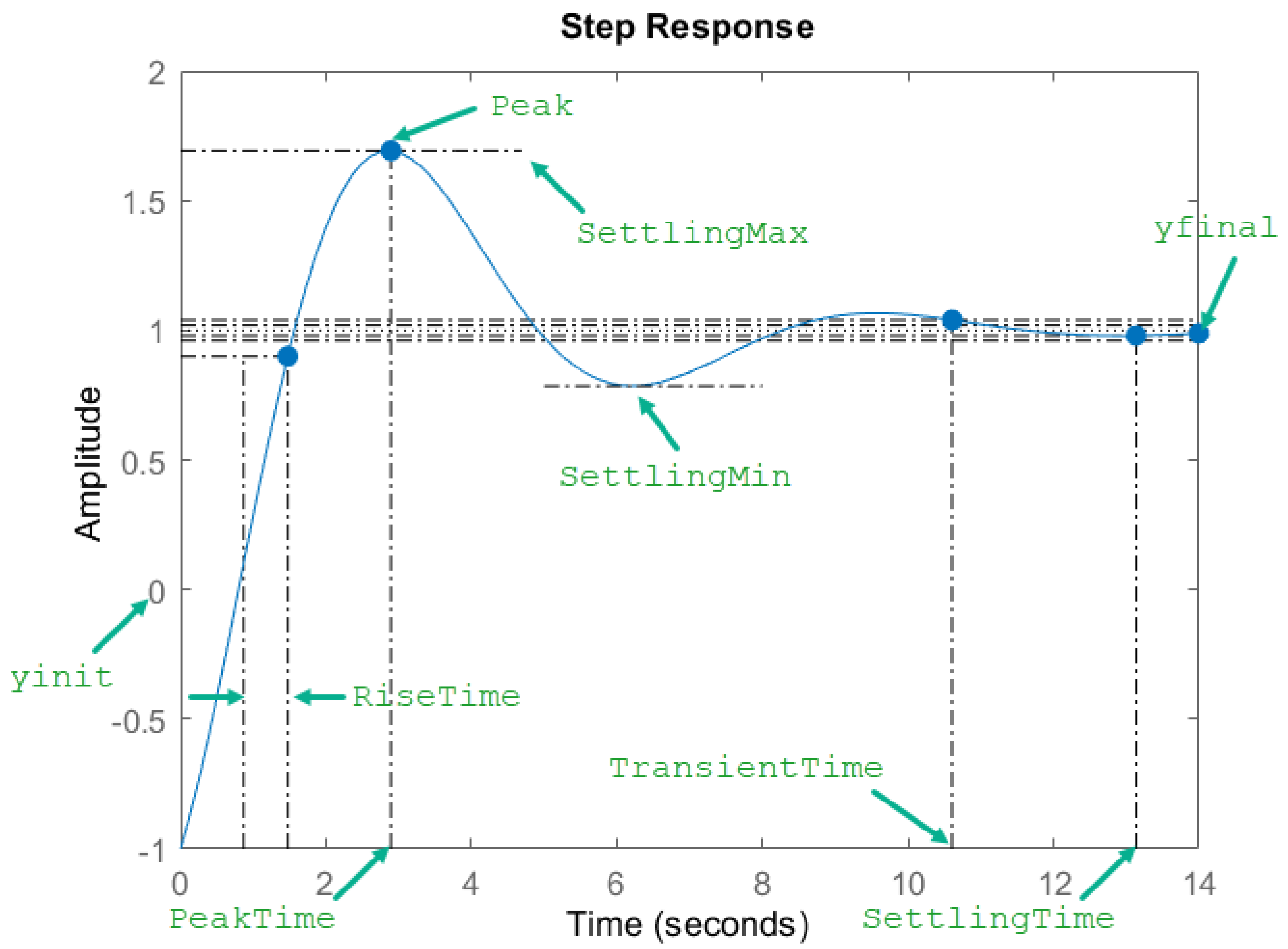
Description

`stepinfo` lets you compute step-response characteristics for a dynamic system model or for an array of step-response data. For a step response $y(t)$, `stepinfo` computes characteristics relative to y_{init} and y_{final}, where y_{init} is the initial offset, that is, the value before the step is applied, and y_{final} is the steady-state value of the response. These values depend on the syntax you use.

- For a dynamic system model `sys`, `stepinfo` uses $y_{init} = 0$ and y_{final} = steady-state value.
- For an array of step-response data `[y,t]`, `stepinfo` uses $y_{init} = 0$ and y_{final} = last sample value of `y`, unless you explicitly specify these values.

For more information on how `stepinfo` computes the step-response characteristics, see "Algorithms" on page 1-1680.

The following figure illustrates some of the characteristics `stepinfo` computes for a step response. For this response, assume that $y(t) = 0$ for $t < 0$, so $y_{init} = 0$.



`S = stepinfo(sys)` computes the step-response characteristics for a dynamic system model `sys`. This syntax uses y_{init} = 0 and y_{final} = steady-state value for computing the characteristics that depend on these values.

Using this syntax requires a Control System Toolbox license.

`S = stepinfo(y,t)` computes step-response characteristics from an array of step-response data `y` and a corresponding time vector `t`. For SISO system responses, `y` is a vector with the same number of entries as `t`. For MIMO response data, `y` is an array containing the responses of each I/O channel. This syntax uses y_{init} = 0 and the last value in `y` (or the last value in each channel's corresponding response data) as y_{final}.

`S = stepinfo(y,t,yfinal)` computes step-response characteristics relative to the steady-state value `yfinal`. This syntax is useful when you know that the expected steady-state system response differs from the last value in `y` for reasons such as measurement noise. This syntax uses y_{init} = 0.

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 response characteristics corresponding to each I/O pair.

`S = stepinfo(y,t,yfinal,yinit)` computes step-response characteristics relative to the response initial value `yinit`. This syntax is useful when your `y` data has an initial offset; that is, `y` is nonzero before the step occurs.

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 and `yinit` as an `NY`-by-`NU` array. `stepinfo` then returns a `NY`-by-`NU` structure array `S` of response characteristics corresponding to each I/O pair.

`S = stepinfo( ___ ,'SettlingTimeThreshold',ST)` lets you specify the threshold `ST` used in the definition of settling and transient times. The default value is `ST = 0.02` (2%). You can use this syntax with any of the previous input-argument combinations.

`S = stepinfo( ___ ,'RiseTimeLimits',RT)` lets you specify the lower and upper thresholds used in the definition of rise time. By default, the rise time is the time the response takes to rise from 10% to 90% of the way from the initial value to the steady-state value (`RT = [0.1 0.9]`). The upper threshold `RT(2)` is also used to calculate `SettlingMin` and `SettlingMax`. These values are the minimum and maximum values of the response occurring after the response reaches the upper threshold. You can use this syntax with any of the previous input-argument combinations.

Examples

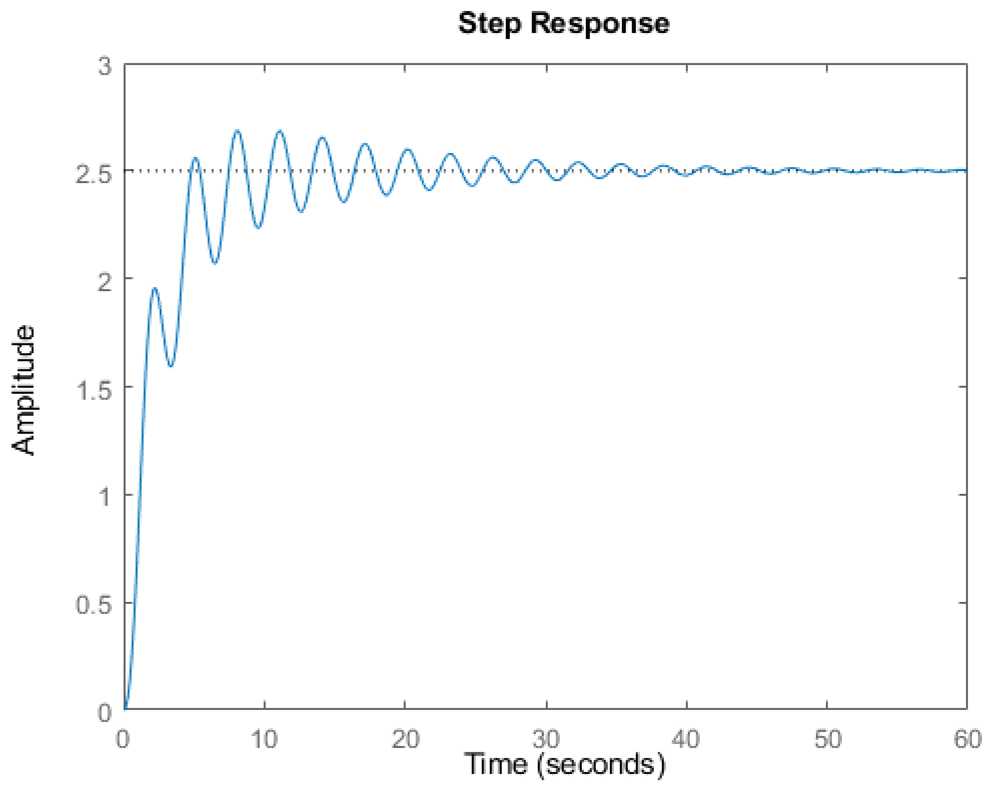
Step-Response Characteristics of Dynamic System

Compute step-response characteristics, such as rise time, settling time, and overshoot, for a dynamic system model. For this example, use a continuous-time transfer function:

$$sys = \frac{s^2 + 5s + 5}{s^4 + 1.65s^3 + 5s^2 + 6.5s + 2}$$

Create the transfer function and examine its step response.

```
sys = tf([1 5 5],[1 1.65 5 6.5 2]);
step(sys)
```



The plot shows that the response rises in a few seconds, and then rings down to a steady-state value of about 2.5. Compute the characteristics of this response using `stepinfo`.

```
S = stepinfo(sys)
```

```
S = struct with fields:
         RiseTime: 3.8456
    TransientTime: 27.9762
     SettlingTime: 27.9762
      SettlingMin: 2.0689
      SettlingMax: 2.6873
        Overshoot: 7.4915
       Undershoot: 0
             Peak: 2.6873
         PeakTime: 8.0530
```

Here, the function uses $y_{\mathrm{init}} = 0$ to compute characteristics for the dynamic system model `sys`.

By default, the settling time is the time it takes for the error to stay below 2% of $|y_{\mathrm{init}} - y_{final}|$. The result `S.SettlingTime` shows that for `sys`, this condition occurs after about 28 seconds. The default definition of rise time is the time it takes for the response to go from 10% to 90% of the way from $y_{\mathrm{init}} = 0$ to y_{final}. `S.RiseTime` shows that for `sys`, this rise occurs in less than 4 seconds. The maximum overshoot is returned in `S.Overshoot`. For this system, the peak value `S.Peak`, which occurs at the time `S.PeakTime`, overshoots by about 7.5% of the steady-state value.

Step-Response Characteristics of MIMO System

For a MIMO system, `stepinfo` returns a structure array in which each entry contains the response characteristics of the corresponding I/O channel of the system. For this example, use a two-output, two-input discrete-time system. Compute the step-response characteristics.

```
A = [0.68 -0.34; 0.34 0.68];
B = [0.18 -0.05; 0.04 0.11];
C = [0 -1.53; -1.12 -1.10];
D = [0 0; 0.06 -0.37];
sys = ss(A,B,C,D,0.2);

S = stepinfo(sys)
```

```
S=2×2 struct array with fields:
    RiseTime
    TransientTime
    SettlingTime
    SettlingMin
    SettlingMax
    Overshoot
    Undershoot
    Peak
    PeakTime
```

Access the response characteristics for a particular I/0 channel by indexing into `S`. For instance, examine the response characteristics for the response from the first input to the second output of `sys`, corresponding to `S(2,1)`.

```
S(2,1)
```

```
ans = struct with fields:
         RiseTime: 0.4000
    TransientTime: 2.8000
     SettlingTime: 3
      SettlingMin: -0.6724
      SettlingMax: -0.5188
        Overshoot: 24.6476
       Undershoot: 11.1224
             Peak: 0.6724
         PeakTime: 1
```

To access a particular value, use dot notation. For instance, extract the rise time of the (2,1) channel.

```
rt21 = S(2,1).RiseTime
```

```
rt21 = 0.4000
```

Specify Percentage for Settling Time or Rise Time

You can use `SettlingTimeThreshold` and `RiseTimeThreshold` to change the default percentage for settling and rise times, respectively, as described in the Algorithms on page 1-1680 section. For this example, use the system given by:

$$sys = \frac{s^2 + 5s + 5}{s^4 + 1.65s^3 + 6.5s + 2}$$

Create the transfer function.

```
sys = tf([1 5 5],[1 1.65 5 6.5 2]);
```

Compute the time it takes for the error in the response of `sys` to stay below 0.5% of the gap $|y_{\text{final}} - y_{\text{init}}|$. To do so, set `SettlingTimeThreshold` to 0.5%, or 0.005.

```
S1 = stepinfo(sys,'SettlingTimeThreshold',0.005);
st1 = S1.SettlingTime
```

```
st1 = 46.1325
```

Compute the time it takes the response of `sys` to rise from 5% to 95% of the way from y_{init} to y_{final}. To do so, set `RiseTimeThreshold` to a vector containing those bounds.

```
S2 = stepinfo(sys,'RiseTimeThreshold',[0.05 0.95]);
rt2 = S2.RiseTime
```

```
rt2 = 4.1690
```

You can define percentages for both settling time and rise time in the same computation.

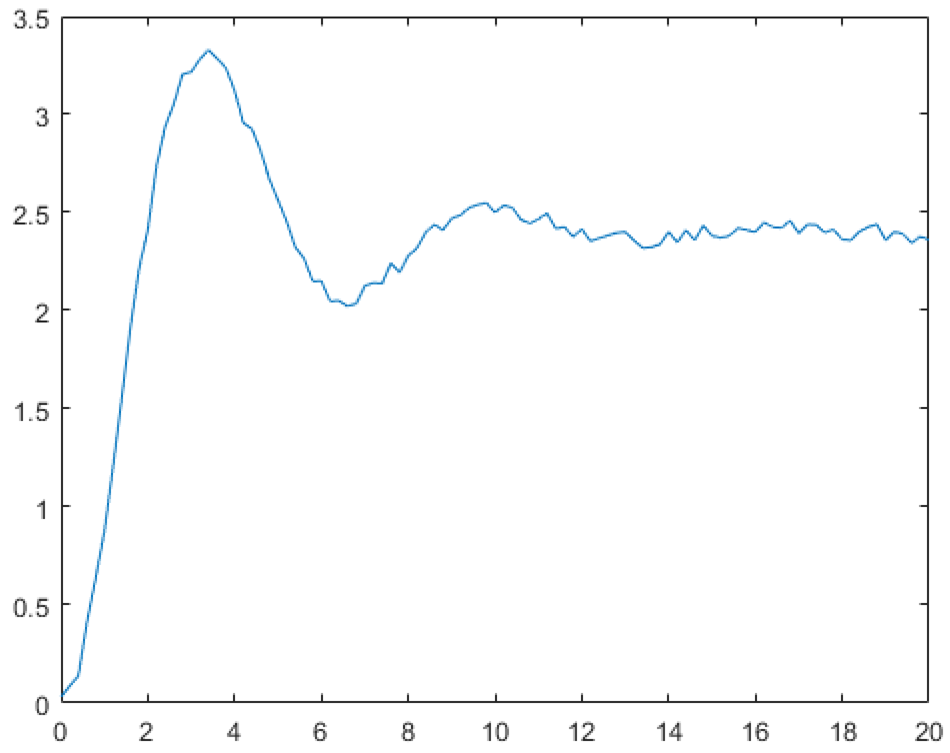
```
S3 = stepinfo(sys,'SettlingTimeThreshold',0.005,'RiseTimeThreshold',[0.05 0.95])
```

```
S3 = struct with fields:
         RiseTime: 4.1690
    TransientTime: 46.1325
     SettlingTime: 46.1325
      SettlingMin: 2.0689
      SettlingMax: 2.6873
        Overshoot: 7.4915
       Undershoot: 0
             Peak: 2.6873
         PeakTime: 8.0530
```

Step-Response Characteristics from Response Data

You can extract step-response characteristics from step-response data even if you do not have a model of your system. For instance, suppose you have measured the response of your system to a step input and saved the resulting response data in a vector `y` of response values at the times stored in another vector `t`. Load the response data and examine it.

```
load StepInfoData t y
plot(t,y)
```



Compute step-response characteristics from this response data using `stepinfo`. If you do not specify the steady-state response value `yfinal`, then `stepinfo` assumes that the last value in the response vector `y` is the steady-state response`.`Because the data has some noise, the last value in `y` is likely not the true steady-state response value. When you know what the steady-state value should be, you can provide it to `stepinfo`. For this example, suppose that the steady-state response is 2.4.

```
S1 = stepinfo(y,t,2.4)
```

```
S1 = struct with fields:
         RiseTime: 1.2897
    TransientTime: 19.6478
     SettlingTime: 19.6439
      SettlingMin: 2.0219
      SettlingMax: 3.3302
        Overshoot: 38.7575
       Undershoot: 0
             Peak: 3.3302
         PeakTime: 3.4000
```

Because of the noise in the data, the default definition of the settling time is too stringent, resulting in an arbitrary value of almost 20 seconds. To allow for the noise, increase the settling-time threshold from the default 2% to 5%.

```
S2 = stepinfo(y,t,2.4,'SettlingTimeThreshold',0.05)
```

```
S2 = struct with fields:
         RiseTime: 1.2897
```

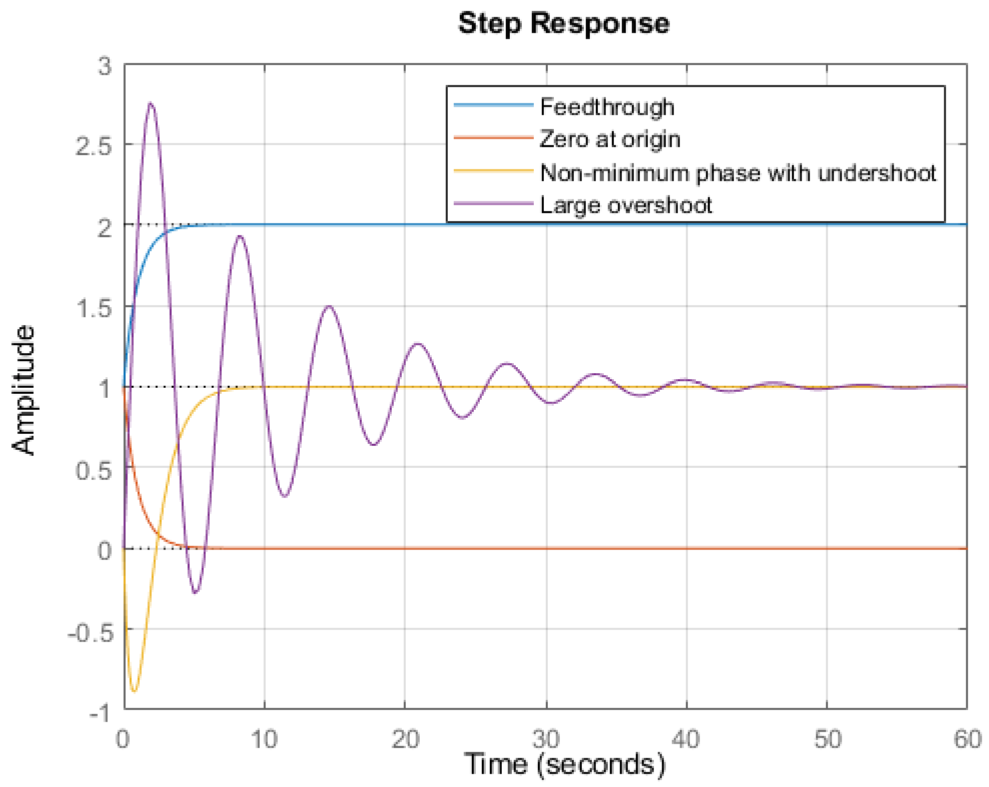
```
    TransientTime: 10.4201
     SettlingTime: 10.4149
      SettlingMin: 2.0219
      SettlingMax: 3.3302
        Overshoot: 38.7575
       Undershoot: 0
             Peak: 3.3302
         PeakTime: 3.4000
```

Difference Between Transient Time and Settling Time for Step Responses

Settling time and transient time are equal when the peak error $e_{\max}$ is equal to the gap $|y_{\mathrm{final}} - y_{\mathrm{init}}|$ (see “Algorithms” (Control System Toolbox)), which is the case for models with no undershoot or feedthrough and with less than 100% overshoot. They tend to differ for models with feedthrough, zeros at the origin, unstable zeros (undershoot), or large overshoot.

Consider the following models.

```
s = tf('s');
sys1 = 1+tf(1,[1 1]);                           % feedthrough
sys2 = tf([1 0],[1 1]);                         % zero at the origin
sys3 = tf([-3 1],[1 2 1]);                      % non-minimum phase with undershoot
sys4 = (s/0.5 + 1)/(s^2 + 0.2*s + 1);           % large overshoot

step(sys1,sys2,sys3,sys4)
grid on
legend('Feedthrough','Zero at origin','Non-minimum phase with undershoot','Large overshoot')
```



Compute the step-response characteristics.

```
S1 = stepinfo(sys1)
```

```
S1 = struct with fields:
         RiseTime: 1.6095
    TransientTime: 3.9121
     SettlingTime: 3.2190
      SettlingMin: 1.8005
      SettlingMax: 2.0000
        Overshoot: 0
       Undershoot: 0
             Peak: 2.0000
         PeakTime: 10.5458
```

```
S2 = stepinfo(sys2)
```

```
S2 = struct with fields:
         RiseTime: 0
    TransientTime: 3.9121
     SettlingTime: NaN
      SettlingMin: 2.6303e-05
      SettlingMax: 1
        Overshoot: Inf
       Undershoot: 0
             Peak: 1
         PeakTime: 0
```

```
S3 = stepinfo(sys3)
```

```
S3 = struct with fields:
         RiseTime: 2.9198
    TransientTime: 6.5839
     SettlingTime: 7.3229
      SettlingMin: 0.9004
      SettlingMax: 0.9991
        Overshoot: 0
       Undershoot: 88.9466
             Peak: 0.9991
         PeakTime: 10.7900
```

```
S4 = stepinfo(sys4)
```

```
S4 = struct with fields:
         RiseTime: 0.3896
    TransientTime: 40.3317
     SettlingTime: 46.5052
      SettlingMin: -0.2796
      SettlingMax: 2.7571
        Overshoot: 175.7137
       Undershoot: 27.9629
             Peak: 2.7571
         PeakTime: 1.8850
```

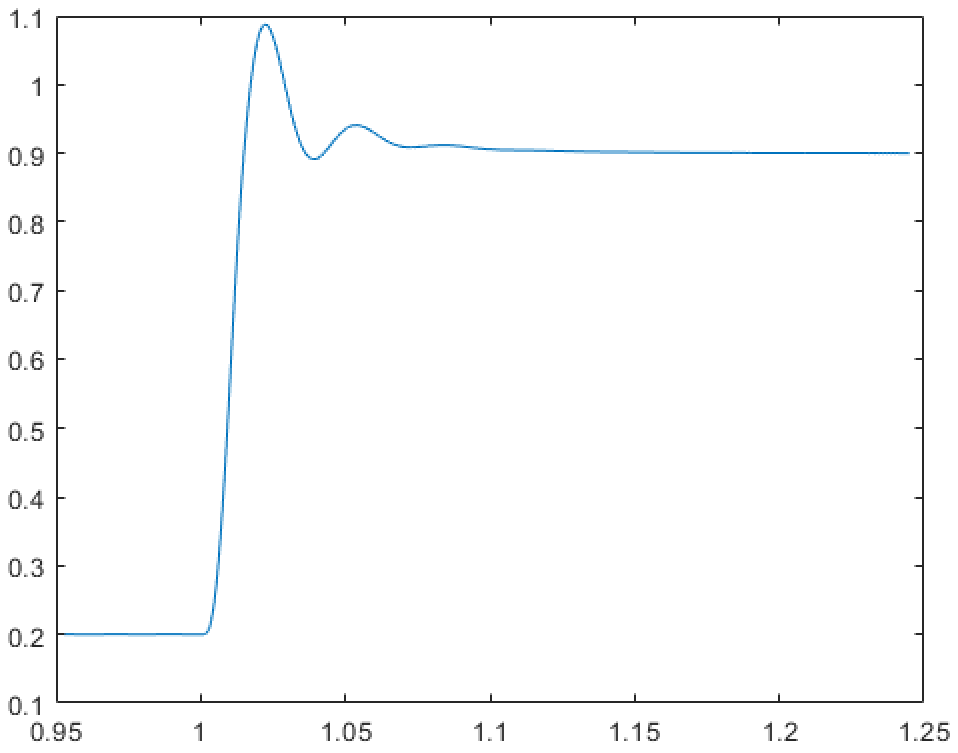
Examine the plots and characteristics. For these models, the settling time and transient time differ because the peak error exceeds the gap between the initial and the final value. For models such as `sys2`, the settling time is returned as `NaN` because the steady-state value is zero.

Step Response Characteristics for Data with Initial Offset

In this example, you compute the step-response characteristics from step-response data that has an initial offset. This means that the value of the response data is nonzero before the step occurs.

Load the step-response data and examine the plot.

```
load stepDataOffset.mat
plot(stepOffset.Time,stepOffset.Data)
```



If you do not specify `yfinal` and `yinit`, then `stepinfo` assumes that `yfinal` is the last value in the response vector `y` and `yinit` is zero. When you know what the steady-state and initial values are, you can provide them to `stepinfo`. Here, the steady state of the response `yfinal` is 0.9 and the initial offset `yinit` is 0.2.

Compute step-response characteristics from this response data.

```
S = stepinfo(stepOffset.Data,stepOffset.Time,0.9,0.2)
```

```
S = struct with fields:
         RiseTime: 0.0084
    TransientTime: 1.0662
     SettlingTime: 1.0662
      SettlingMin: 0.8461
      SettlingMax: 1.0878
        Overshoot: 26.8259
       Undershoot: 0.0429
             Peak: 0.8878
         PeakTime: 1.0225
```

Here, the peak value of this response is 0.8878 because `stepinfo` measures the maximum deviation from `yinit`.

Input Arguments

`sys` — Dynamic system
dynamic system model

Dynamic system, specified as a SISO or MIMO dynamic system model. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.) For generalized models, `stepinfo` computes the step-response characteristics using the current value of tunable blocks and the nominal value of uncertain blocks.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models.

`y` — Step-response data
vector | array

Step-response data, specified as one of the following:

- For SISO response data, a vector of length `Ns`, where `Ns` is the number of samples in the response data
- For MIMO response data, an `Ns`-by-`Ny`-by-`Nu` array, where `Ny` is the number of system outputs and `Nu` is the number of system inputs

`t` — Time vector
vector

Time vector corresponding to the response data in `y`, specified as a vector of length `Ns`.

`yfinal` — Steady-state value
scalar | array

Steady-state value, specified as a scalar or an array.

- For SISO response data, specify a scalar value.
- For MIMO response data, specify an `Ny`-by-`Nu` array, where each entry provides the steady-state response value for the corresponding system channel.

If you do not provide `yfinal`, then `stepinfo` uses the last value in the corresponding channel of `y` as the steady-state response value.

This argument is only supported when you provide step-response data as an input. For a dynamic system model `sys` as an input, `stepinfo` uses y_{final} = steady-state value to compute the characteristics that depend on this value.

`yinit` — Initial value
scalar | array

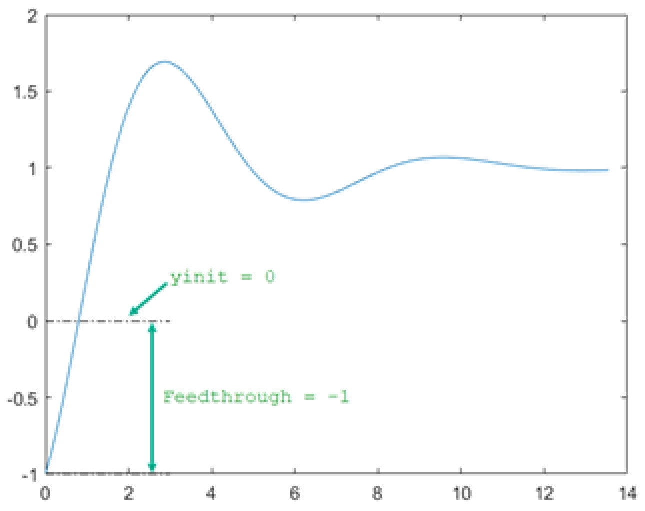
Value of `y` before the step occurs, specified as a scalar or an array.

- For SISO response data, specify a scalar value.
- For MIMO response data, specify an `Ny`-by-`Nu` array, where each entry provides the response initial value for the corresponding system channel.

If you do not provide `yinit`, then `stepinfo` uses zero as the response initial value.

The response $y(0)$ at $t = 0$ is equal to y_{init} for systems without feedthrough. However, the two quantities differ in the presence of feedthrough because of the discontinuity at $t = 0$.

For example, the following figure shows the step response of a system with feedthrough `sys = tf([-1 0.2 1],[1 0.7 1])`.



Here, y_{init} is zero and the feedthrough value is –1.

This argument is only supported when you provide step-response data as an input. For a dynamic system model `sys` as an input, `stepinfo` uses $y_{init} = 0$ to compute the characteristics that depend on this value.

ST — Settling time threshold
0.02 (default) | scalar between 0 and 1

Threshold for defining settling and transient times, specified as a scalar value between 0 and 1. To change the default settling and transient time definitions (see “Algorithms” on page 1-1680), set ST to a different value. For instance, to measure when the error falls below 5%, set ST to 0.05.

RT — Rise time thresholds
`[0.1 0.9]` (default) | 2-element row vector

Threshold for defining rise time, specified as a 2-element row vector of nondescending values between 0 and 1. To change the default rise time definition (see “Algorithms” on page 1-1680), set RT to a different value. For instance, to define the rise time as the time it takes for the response to rise from 5% to 95% from the initial value to the steady-state value, set RT to `[0.05 0.95]`.

Output Arguments

S — Step-response characteristics
structure

Step-response characteristics, returned as a structure containing the fields:

- `RiseTime`
- `TransientTime`
- `SettlingTime`
- `SettlingMin`
- `SettlingMax`
- `Overshoot`
- `Undershoot`
- `Peak`
- `PeakTime`

For more information on how `stepinfo` defines these characteristics, see “Algorithms” on page 1-1680.

For MIMO models or responses data, `S` is a structure array in which each entry contains the step-response characteristics of the corresponding I/O channel. For instance, if you provide a 3-input, 3-output model or an array of response data, then `S(2,3)` contains the characteristics of the response from the third input to the second output. For an example, see “Step-Response Characteristics of MIMO System” on page 1-1671.

If `sys` is unstable, then all step-response characteristics are `NaN`, except for `Peak` and `PeakTime`, which are `Inf`.

Algorithms

For a step response $y(t)$, `stepinfo` computes characteristics relative to y_{init} and y_{final}. For a dynamic system model `sys`, `stepinfo` uses $y_{init} = 0$ and y_{final} = steady-state value.

This table shows how `stepinfo` computes each characteristic.

Step-Response Characteristic	Description
`RiseTime`	Time it takes for the response to rise from 10% to 90% of the way from y_{init} to y_{final}
`TransientTime`	The first time T such that the error $\lvert y(t) - y_{final}\rvert \leq SettlingTimeThreshold \times e_{max}$ for $t \geq T$, where e_{max} is the maximum error $\lvert y(t) - y_{final}\rvert$ for $t \geq 0$. By default, *SettlingTimeThreshold* = 0.02 (2% of the peak error). Transient time measures how quickly the transient dynamics die off.
`SettlingTime`	The first time T such that the error $\lvert y(t) - y_{final}\rvert \leq SettlingTimeThreshold \times \lvert y_{final} - y_{init}\rvert$ for $t \geq T$. By default, `SettlingTime` measures the time it takes for the error to stay below 2% of $\lvert y_{final} - y_{init}\rvert$.
`SettlingMin`	Minimum value of $y(t)$ once the response has risen
`SettlingMax`	Maximum value of $y(t)$ once the response has risen

Step-Response Characteristic	Description
`Overshoot`	Percentage overshoot. Relative to the normalized response $y_{norm}(t) = (y(t) - y_{init})/(y_{final} - y_{init})$, the overshoot is the larger of zero and $100 \times \max(y_{norm}(t) - 1)$.
`Undershoot`	Percentage undershoot. Relative to the normalized response $y_{norm}(t)$, the undershoot is the smaller of zero and $-100 \times \max(y_{norm}(t) - 1)$.
`Peak`	Peak value of $\lvert y(t) - y_{init} \rvert$
`PeakTime`	Time at which the peak value occurs

Compatibility Considerations

Response characteristics computation changes
Behavior changed in R2021b

The computation method of some response characteristics has changed. Additionally, the settling time calculation is now based on how quickly the response stays below a specified threshold of the gap between the initial and the final value.

The following table summarizes the changes to the fields of the structure returned by `stepinfo`.

Before R2021b	R2021b
`RiseTime` — Time it takes for the response to rise from 10% to 90% of the way from $y(1)$ to y_{final}.	`RiseTime` — Time it takes to go from 10% to 90% of the way from y_{init} to y_{final}.
`SettlingTime` — The first time T such that the error $\lvert y(t) - y_{final} \rvert \le SettlingTimeThreshold \times e_{max}$ for $t \ge T$, where e_{max} is the maximum error $\lvert y(t) - y_{final} \rvert$ for $t \ge 0$. By default, *SettlingTimeThreshold* = 0.02 (2% of the peak error). `SettlingTime` measures the time for the error to fall below 2% of the peak value of the error.	`SettlingTime` — The first time T such that the error $\lvert y(t) - y_{final} \rvert \le SettlingTimeThreshold \times \lvert y_{final} - y_{init} \rvert$ for $t \ge T$. By default, `SettlingTime` measures the time it takes for the error to stay below 2% of $\lvert y_{final} - y_{init} \rvert$.
`Peak` — Peak absolute value of $y(t)$.	`Peak` — Peak absolute value of $y(t) - y_{init}$.

Additionally, the output structure `S` now contains a `TransientTime` field. This characteristic contains the peak-error-based settling time calculation used in releases before R2021b. Transient time measures how quickly the transient dynamics die off.

These changes also apply to the characteristics of `step`, `impulse`, and `initial` plots. Additionally:

- For `step` plots, y_{init} is always assumed to be zero and y_{final} is the steady-state value.
- For the step response, transient time and settling time tend to differ for models with feedthrough, zeros at the origin, unstable zeros (undershoot), or large overshoot. They match for models with no undershoot or feedthrough, and with less than 100% overshoot. For an example, see "Difference Between Transient Time and Settling Time for Step Responses" on page 1-1674.
- For the step response of models with feedthrough, the new `RiseTime` value can differ because $y(1)$ is nonzero whereas y_{init} is zero by default. Before R2021b, the rise time computed was the time it takes to go from 10% to 90% of the way from $y(1)$ to y_{final}, instead of y_{init} to y_{final} now.

See Also

`step` | `lsiminfo`

Introduced in R2006a

stepplot

Plot step response with additional plot customization options

Syntax

```
h = stepplot(sys)
h = stepplot(sys1,sys2,...,sysN)
h = stepplot(sys1,LineSpec1,...,sysN,LineSpecN)
h = stepplot( ___ ,tFinal)
h = stepplot( ___ ,t)
h = stepplot(AX, ___ )
h = stepplot( ___ ,plotoptions)
h = stepplot( ___ ,dataoptions)
```

Description

`stepplot` lets you plot dynamic system step responses with a broader range of plot customization options than `step`. You can use `stepplot` to obtain the plot handle and use it to customize the plot, such as modify the axes labels, limits and units. You can also use `stepplot` to draw a step response plot on an existing set of axes represented by an axes handle. To customize an existing step plot using the plot handle:

1. Obtain the plot handle
2. Use `getoptions` to obtain the option set
3. Update the plot using `setoptions` to modify the required options

For more information, see “Customizing Response Plots from the Command Line” (Control System Toolbox). To create step plots with default options or to extract step response data, use `step`.

`h = stepplot(sys)` plots the step response of the dynamic system model `sys` and returns the plot handle `h` to the plot. You can use this handle `h` to customize the plot with the `getoptions` and `setoptions` commands.

`h = stepplot(sys1,sys2,...,sysN)` plots the step response of multiple dynamic systems `sys1,sys2,…,sysN` on the same plot. All systems must have the same number of inputs and outputs to use this syntax.

`h = stepplot(sys1,LineSpec1,...,sysN,LineSpecN)` sets the line style, marker type, and color for the step response of each system. All systems must have the same number of inputs and outputs to use this syntax.

`h = stepplot( ___ ,tFinal)` simulates the step response from `t = 0` to the final time `t = tFinal`. Specify `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.

`h = stepplot( ___ ,t)` simulates the step response using the time vector `t`. Specify `t` in the system time units, specified in the `TimeUnit` property of `sys`.

`h = stepplot(AX, ___ )` plots the step response on the `Axes` object in the current figure with the handle `AX`.

`h = stepplot( ___ ,plotoptions)` plots the step response with the options set specified in `plotoptions`. You can use these options to customize the step plot appearance using the command line. Settings you specify in `plotoptions` overrides the preference settings in the MATLAB session in which you run `stepplot`. Therefore, this syntax is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

`h = stepplot( ___ ,dataoptions)` plots the step response with the options set specified in `dataoptions`. You can use this syntax to specify options such as the step amplitude and input offset using the options set `dataoptions`. This syntax is useful when you want to write a script to generate multiple plots with the same option set. Use `stepDataOptions` to create the options set.

Examples

Customize Step Plot using Plot Handle

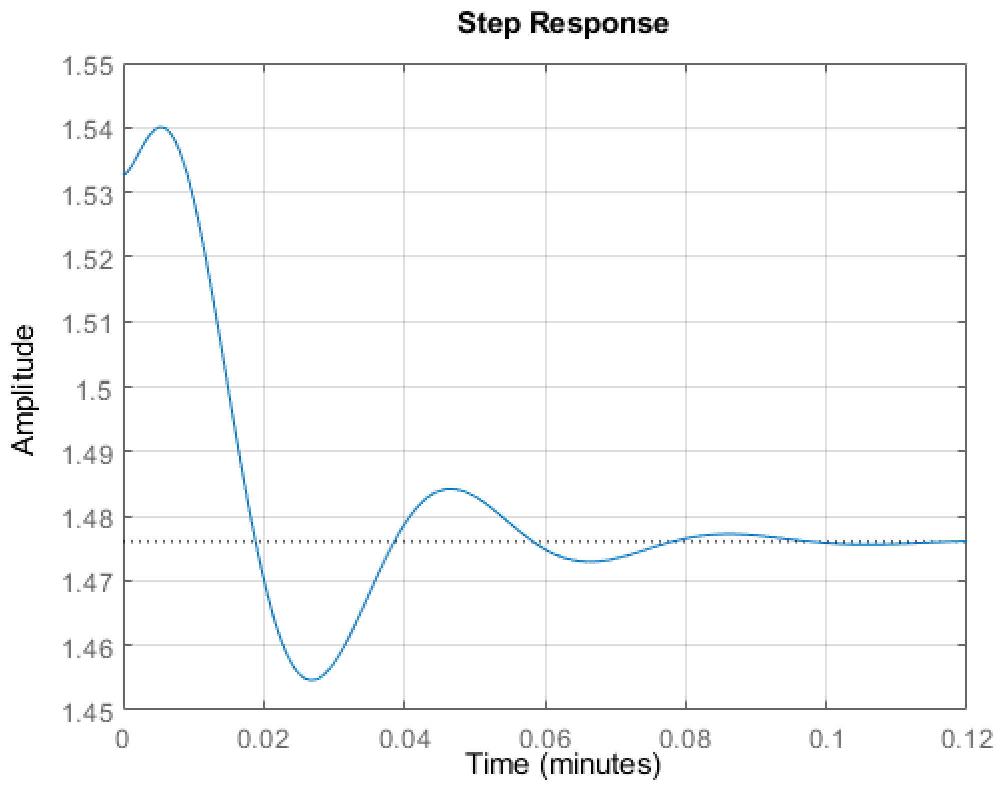
For this example, use the plot handle to change the time units to minutes and turn on the grid.

Generate a random state-space model with 5 states and create the step response plot with plot handle `h`.

```
rng("default")
sys = rss(5);
h = stepplot(sys);
```

Change the time units to minutes and turn on the grid. To do so, edit properties of the plot handle, `h` using `setoptions`.

```
setoptions(h,'TimeUnits','minutes','Grid','on');
```



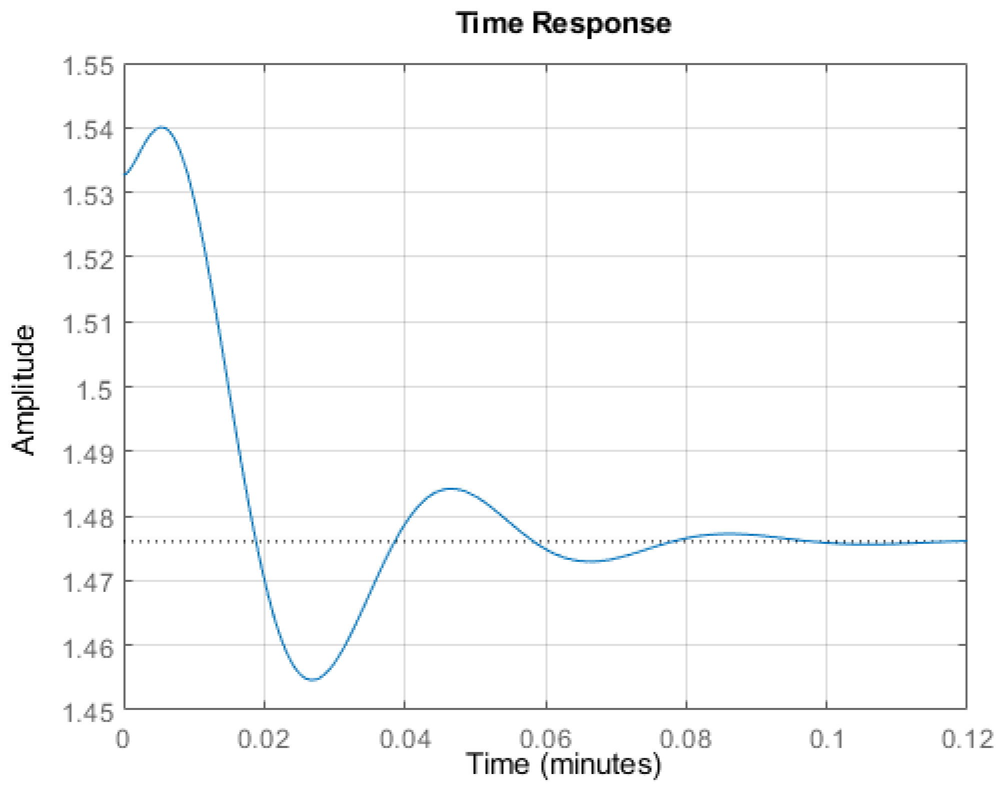
The step plot automatically updates when you call `setoptions`.

Alternatively, you can also use the `timeoptions` command to specify the required plot options. First, create an options set based on the toolbox preferences.

```
plotoptions = timeoptions('cstprefs');
```

Change properties of the options set by setting the time units to minutes and enabling the grid.

```
plotoptions.TimeUnits = 'minutes';
plotoptions.Grid = 'on';
stepplot(sys,plotoptions);
```

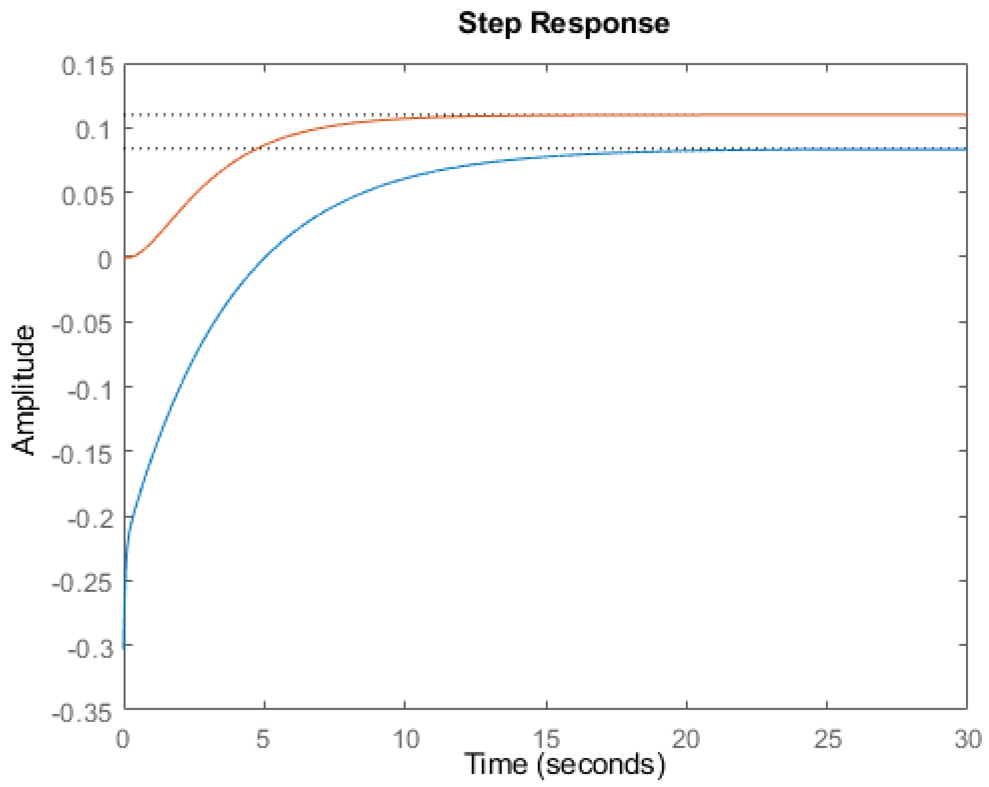


You can use the same option set to create multiple step plots with the same customization. Depending on your own toolbox preferences, the plot you obtain might look different from this plot. Only the properties that you set explicitly, in this example `TimeUnits` and `Grid`, override the toolbox preferences.

Display Normalized Response on Step Plot

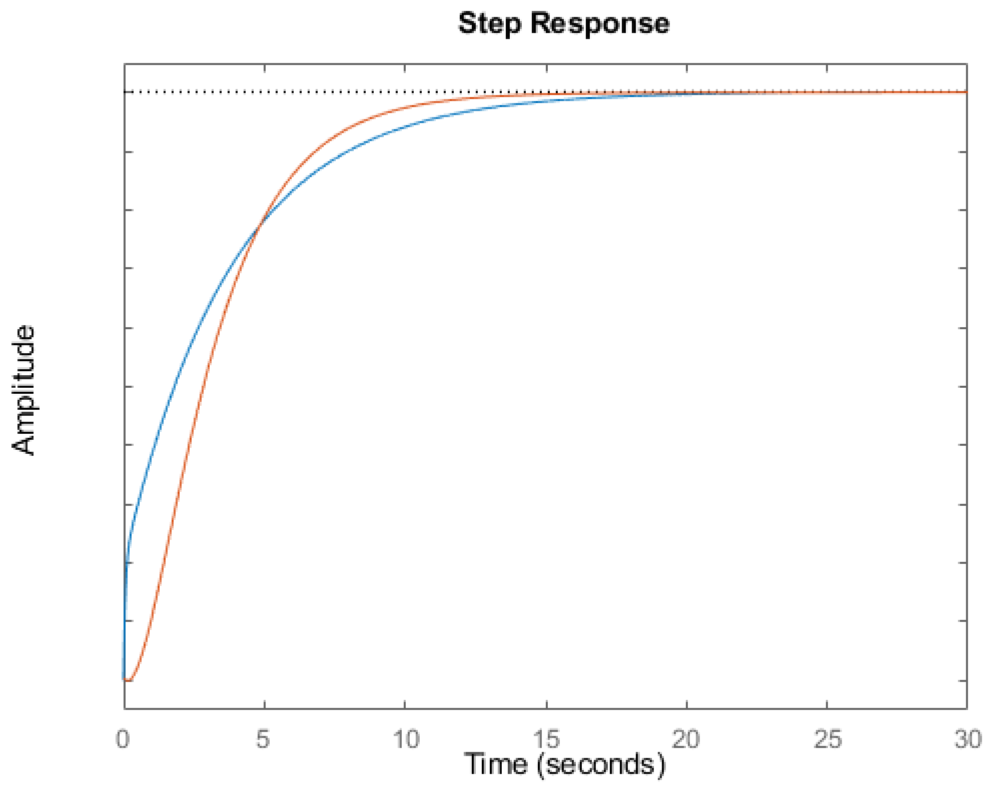
Generate a step response plot for two dynamic systems.

```
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.

Plot Step Responses of Identified Models with Confidence Region

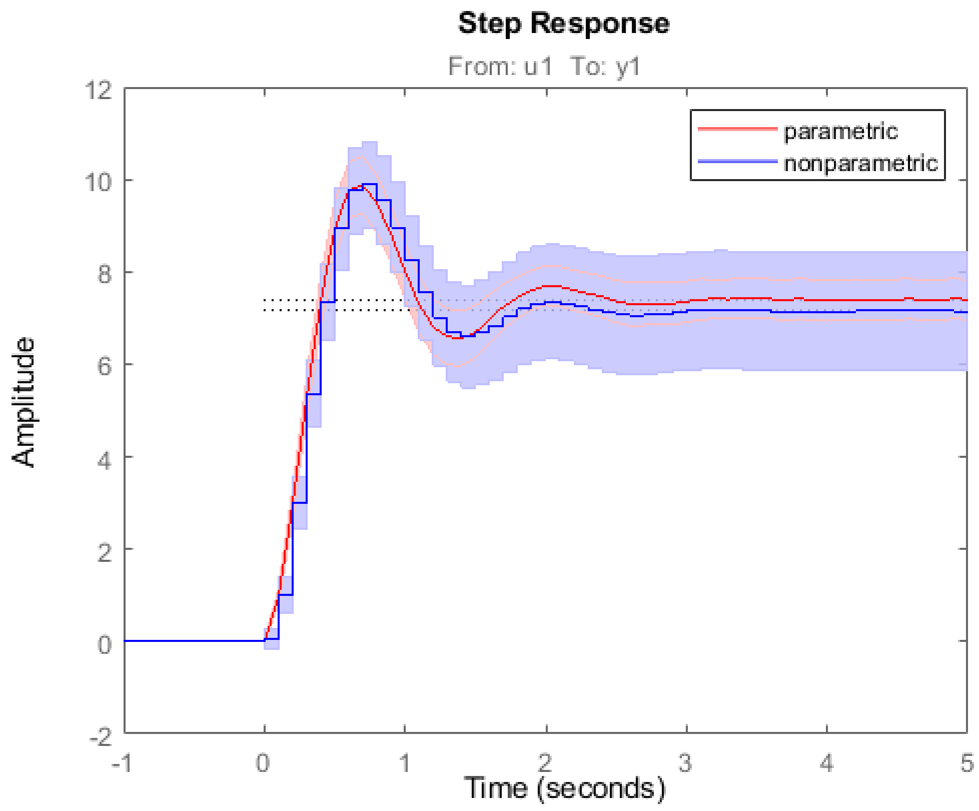
Compare the step response of a parametric identified model to a nonparametric (empirical) model, and view their 3-σ confidence regions. (Identified models require System Identification Toolbox™ software.)

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,'r',sys2,'b',t);
showConfidence(h,3)
legend('parametric','nonparametric')
```



The nonparametric model `sys2` shows higher uncertainty.

Customized Step Response Plot at Specified Time

For this example, examine the step response of the following zero-pole-gain model and limit the step plot to `tFinal` = 15 s. Use 15-point blue text for the title. This plot should look the same, regardless of the preferences of the MATLAB session in which it is generated.

```
sys = zpk(-1,[-0.2+3j,-0.2-3j],1)*tf([1 1],[1 0.05]);
tFinal = 15;
```

First, create a default options set using `timeoptions`.

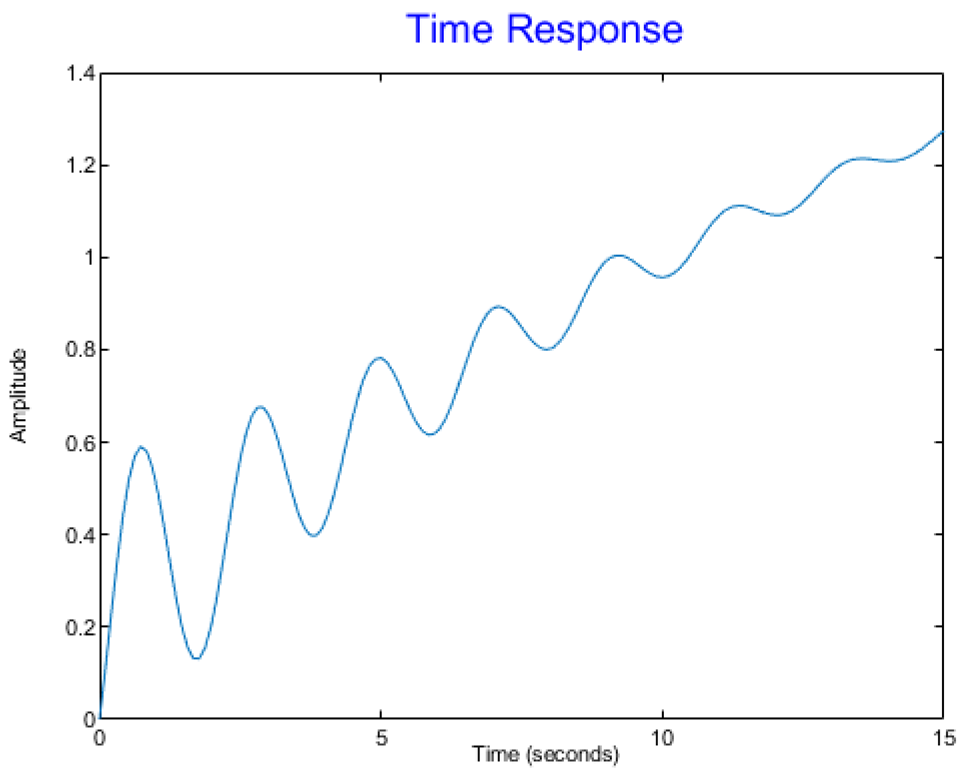
```
plotoptions = timeoptions;
```

Next change the required properties of the options set `plotoptions`.

```
plotoptions.Title.FontSize = 15;
plotoptions.Title.Color = [0 0 1];
```

Now, create the step response plot using the options set `plotoptions`.

```
h = stepplot(sys,tFinal,plotoptions);
```



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

Plot Step Response of Nonlinear Identified Model

Load data for estimating a nonlinear Hammerstein-Wiener model.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','twotankdata'));
z = iddata(y,u,0.2,'Name','Two tank system');
```

`z` is an `iddata` object that stores the input-output estimation data.

Estimate a Hammerstein-Wiener Model of order [1 5 3] using the estimation data. Specify the input nonlinearity as piecewise linear and output nonlinearity as one-dimensional polynomial.

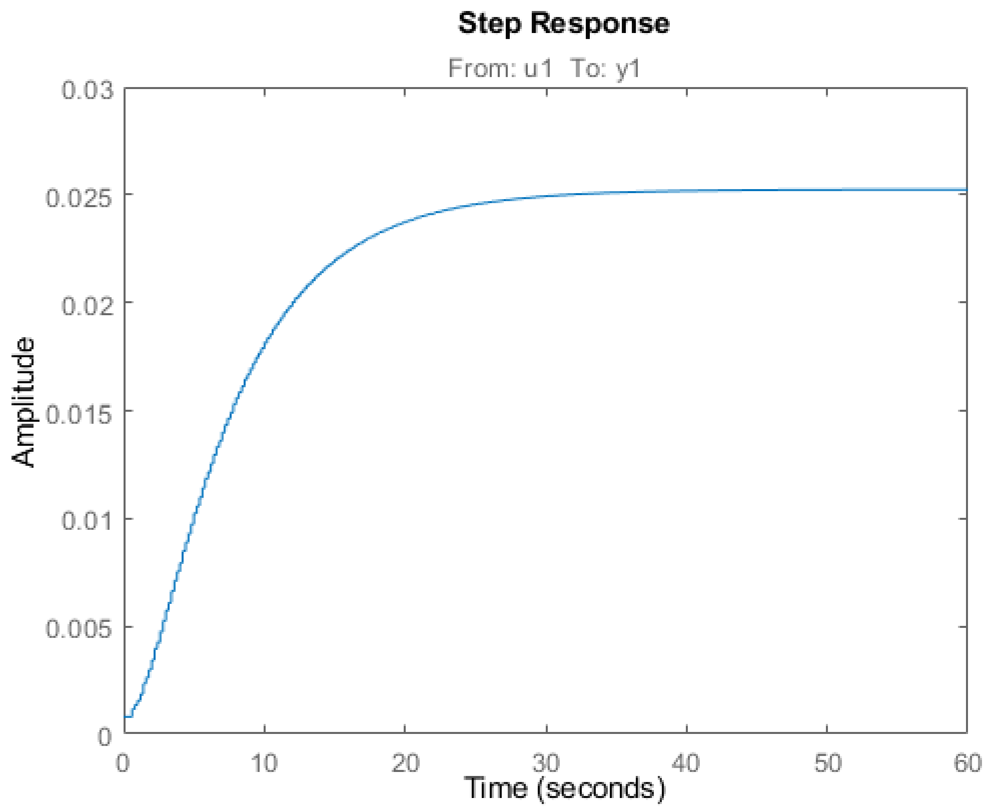
```
sys = nlhw(z,[1 5 3],idPiecewiseLinear,idPolynomial1D);
```

Create an option set to specify input offset and step amplitude level.

```
opt = stepDataOptions('InputOffset',2,'StepAmplitude',0.5);
```

Plot the step response until 60 seconds using the specified options.

```
stepplot(sys,60,opt);
```



Input Arguments

sys — Dynamic system
dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model or array of dynamic system models. Dynamic systems that you can use include:

- Continuous-time or discrete-time numeric LTI models, such as `tf`, `zpk`, or `ss` models.
- Sparse state-space models, such as `sparss` or `mechss` models. Final time `tFinal` must be specified when using sparse models.
- Generalized or uncertain LTI models such as `genss` or `uss` models. (Using uncertain models requires Robust Control Toolbox software.)
  - For tunable control design blocks, the function evaluates the model at its current value to plot the step response data.
  - For uncertain control design blocks, the function plots the nominal value and random samples of the model.
- Identified LTI models, such as `idtf`, `idss`, or `idproc` models.

If `sys` is an array of models, the function plots the step response of all models in the array on the same axes.

LineSpec — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: '--or' is a red dashed line with circle markers

Line Style	Description
-	Solid line
--	Dashed line
:	Dotted line
-.	Dash-dot line

Marker	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'_'	Horizontal line
'|'	Vertical line
's'	Square
'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'p'	Pentagram
'h'	Hexagram

Color	Description
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

`tFinal` — Final time for step response computation
scalar

Final time for step response computation, specified as a scalar. Specify `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.

`t` — Time for step response simulation
vector

Time for step response simulation, specified as a vector. Specify the time vector `t` in the system time units, specified in the `TimeUnit` property of `sys`. The time vector must be real, finite, and must contain monotonically increasing and evenly spaced time samples.

The time vector `t` is:

- $t = T_{initial}:T_{sample}:T_{final}$, for discrete-time systems.
- $t = T_{initial}:dt:T_{final}$, for continuous-time systems. Here, dt is the sample time of a discrete approximation of the continuous-time system.

`AX` — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes and if the current axes are Cartesian axes, then `stepplot` plots on the current axes. Use `AX` to plot into specific axes when creating a step plot.

`plotoptions` — Step plot options set
`TimePlotOptions` object

Step plot options set, specified as a `TimePlotOptions` object. You can use this option set to customize the step plot appearance. Use `timeoptions` to create the option set. Settings you specify in `plotoptions` overrides the preference settings in the MATLAB session in which you run `stepplot`. Therefore, `plotoptions` is useful when you want to write a script to generate multiple plots that look the same regardless of the local preferences.

For the list of available options, see `timeoptions`.

`dataoptions` — Step response data options set
`step` object

Step response data options set, specified as a `step` object. Specify options such as the step amplitude and input offset using the options set `dataoptions`. This is useful when you want to write a script to generate multiple plots with the same step amplitude and input offset values. Use `stepDataOptions` to create the options set.

Output Arguments

`h` — Plot handle
`handle` object

Plot handle, returned as a `handle` object. Use the handle `h` to get and set the properties of the step plot using `getoptions` and `setoptions`. For the list of available options, see the *Properties and*

Values Reference section in “Customizing Response Plots from the Command Line” (Control System Toolbox).

See Also
`getoptions` | `setoptions` | `showConfidence` | `step` | `stepDataOptions` | `timeoptions`

Topics
“Customizing Response Plots from the Command Line” (Control System Toolbox)

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 = 3x1 cell
    {'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);
```

Tips

- Use with `arxstruc` or `ivstruc` to compute loss functions for ARX models, one for each model order combination returned by `struc`.

See Also

`arxstruc` | `ivstruc` | `selstruc`

Topics

"Estimating Model Orders Using an ARX Model Structure"
"Preliminary Step – Estimating Model Orders and Input Delays"

Introduced before R2006a

System Identification

Identify models of dynamic systems from measured data

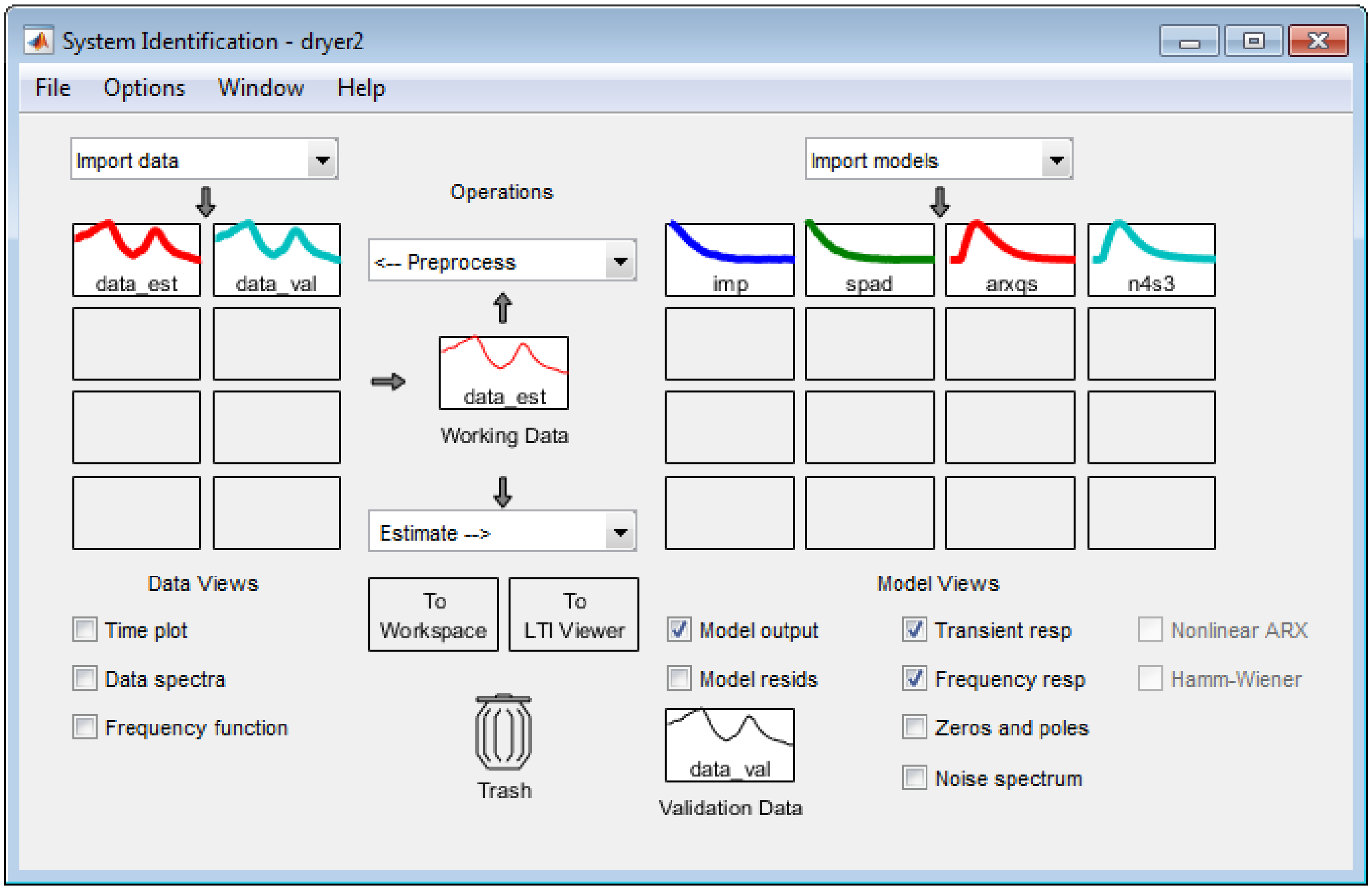
Description

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**.



Data Views: Time plot [ ], Data spectra [ ], Frequency function [ ]

Model Views: Model output [✓], Model resids [ ], Transient resp [✓], Frequency resp [✓], Zeros and poles [ ], Noise spectrum [ ], Nonlinear ARX [ ], Hamm-Wiener [ ]

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`.

Limitations

MATLAB Online™ does not support the **System Identification** app.

See Also

Functions
`iddata` | `tfest` | `ssest` | `procest` | `polyest` | `arx` | `nlarx` | `nlhw` | `midprefs`

Topics
“Working with System Identification App”
“Identify Linear Models Using System Identification App”
“Identify Nonlinear Black-Box Models Using System Identification App”

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)
___ = tfdata(sys,J1,...,JN)
[num,den] = tfdata(sys,'v')
```

Description

`[num,den] = tfdata(sys)` returns the numerator and denominator coefficients of the transfer function for the `tf`, `ss` and `zpk` model objects or the array of model objects represented by `sys`.

The outputs `num` and `den` are two-dimensional cell arrays if `sys` contains a single LTI model. When `sys` is an array of models, `num` and `den` are returned as multidimensional cell arrays.

`[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 `[]`.

`___ = tfdata(sys,J1,...,JN)` extracts the data for the `J1,...,JN` entry in the model array `sys`.

`[num,den] = tfdata(sys,'v')` returns the numerator and denominator coefficients as row vectors rather than cell arrays for a SISO transfer function represented by `sys`.

Examples

Extract Numerator and Denominator Coefficients from Transfer Function

For this example, consider `tfData.mat` which contains a continuous-time SISO transfer function `sys1`.

Load the data and use `tfdata` to extract the numerator and denominator coefficients.

```
load('tfData.mat','sys1');
[num,den] = tfdata(sys1);
```

`num` and `den` are returned as cell arrays. To display data, use `celldisp`.

```
celldisp(num)
```

```
num{1} =
     0     1     5     2
```

```
celldisp(den)
```

```
den{1} =

     7     4     2     1
```

You can also extract the numerator and denominator coefficients as row vectors with the following syntax.

```
[num,den] = tfdata(sys1,'v');
```

Extract Discrete-Time Transfer Function Data

For this example, consider `tfData.mat` which contains a discrete-time SISO transfer function `sys2`.

Load the data and use `tfdata` to extract the numerator and denominator coefficients along with the sample time.

```
load('tfData.mat','sys2');
[num,den,ts] = tfdata(sys2)
```

```
num = 1x1 cell array
    {[0 0 2 0]}

den = 1x1 cell array
    {[4 0 3 -1]}

ts = 0.1000
```

`num` and `den` are returned as cell arrays. To display data, use `celldisp`.

```
celldisp(num)
```

```
num{1} =

     0     0     2     0
```

```
celldisp(den)
```

```
den{1} =

     4     0     3    -1
```

Extract Identified Transfer Function Data

For this example, estimate a transfer function with 2 poles and 1 zero from identified data contained in `iddata7.mat` with an input delay value.

Load the identified data and estimate the transfer function.

```
load('iddata7.mat');
sys = tfest(z7,2,1,'InputDelay',[1 0]);
```

Extract the numerator, denominator and their standard deviations for the 2-input, 1 output identified transfer function.

```
[num,den,~,sdnum,sdden] = tfdata(sys)
```

```
num=1×2 cell array
    {[0 -0.5212 1.1886]}    {[0 0.0552 -0.0013]}


den=1×2 cell array
    {[1 0.3390 0.2353]}    {[1 0.0360 0.0314]}


sdnum=1×2 cell array
    {[0 0.1311 0.0494]}    {[0 0.0246 0.0033]}


sdden=1×2 cell array
    {[0 0.0183 0.0085]}    {[0 0.0278 0.0048]}
```

Extract Data from Specific Model in Transfer Function Array

For this example, extract numerator and denominator coefficients for a specific transfer function contained in the 3x1 array of continuous-time transfer functions `sys`.

Load the data and extract the numerator and denominator coefficients of the second model in the array.

```
load('tfArray.mat','sys');
[num,den] = tfdata(sys,2);
```

Use celldisp to visualize the data in the cell array `num` and `den`.

```
celldisp(num)
```

```
num{1} =

     0     0     2
```

```
celldisp(den)
```

```
den{1} =

     1     1     2
```

Input Arguments

sys — Dynamic system
dynamic system model | model array

Dynamic system, specified as a SISO or MIMO dynamic system model, or an array of SISO or MIMO dynamic system models. Dynamic systems that you can use include continuous-time or discrete-time numeric LTI models such as `tf`, `ss` and `zpk` models.

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, use the following syntax to return the numerator and denominator coefficients directly as row vectors rather than as cell arrays:

```
[num,den] = tfdata(sys,'v')
```

J1,...,JN — Indices of models in array whose data you want to access
positive integer

Indices of models in array whose data you want to access, specified as a positive integer. You can provide as many indices as there are array dimensions in `sys`. For example, if `sys` is a 4-by-5 array of transfer functions, the following command accesses the data for entry (2,3) in the array.

```
[num,den] = tfdata(sys,2,3);
```

Output Arguments

num — Coefficients of the numerator
cell array | row vector

Coefficients of the numerator of the transfer function, returned as a cell array or row vector.

When `sys` contains a single LTI model, the output `num` is returned as a cell array with the following characteristics:

- `num` has as many rows as outputs and as many columns as inputs of `sys`.
- The `(i,j)` entries in `num{i,j}` are row vectors specifying the numerator coefficients of the transfer function from input `j` to output `i`. `tfdata` orders these coefficients in *descending* powers of *s* or *z*.

When `sys` contains an array of LTI models, `num` is returned as a multidimensional cell array of the same size as `sys`.

`den` — Coefficients of the denominator
cell array | row vector

Coefficients of the denominator of the transfer function, returned as a cell array or row vector.

When `sys` contains a single LTI model, the output `den` is returned as a cell array with the following characteristics:

- `den` has as many rows as outputs and as many columns as inputs of `sys`.
- The `(i,j)` entries in `den{i,j}` are row vectors specifying the denominator coefficients of the transfer function from input `j` to output `i`. `tfdata` orders these coefficients in *descending* powers of s or z.

When `sys` contains an array of LTI models, `den` is returned as a multidimensional cell array of the same size as `sys`.

`ts` — Sample time
non-negative scalar

Sample time, returned as a non-negative scalar.

`sdnum` — Standard uncertainty of the numerator coefficients
cell array

Standard uncertainty of the numerator coefficients of the identified system `sys`, returned as a cell array of the same size as `num`. `sdnum{i,j}(k)` is the 1 standard uncertainty in the value `num{i,j}(k)`. If `sys` does not contain uncertainty information, `sdnum` is empty `[]`.

`sdden` — Standard uncertainty the denominator coefficients
cell array

Standard uncertainty of the denominator coefficients of the identified system `sys`, returned as a cell array of the same size as `den`. `sdden{i,j}(k)` is the 1 standard uncertainty in the value `den{i,j}(k)`. If `sys` does not contain uncertainty information, `sdden` is empty `[]`.

See Also
`tf` | `ss` | `zpk` | `get` | `ssdata` | `zpkdata`

Introduced before R2006a

tfest

Estimate transfer function

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)

[sys,ic] = tfest( ___ )
```

Description

Estimate a Transfer Function Model

`sys = tfest(data,np)` estimates a continuous-time transfer function `sys` using the time-domain or frequency-domain data `data` and containing `np` poles. The number of zeros in `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 the input-output pairs in `iodelay`.

`sys = tfest( ___ ,Name,Value)` uses additional options specified by one or more name-value pair arguments. You can use this syntax with any of the previous input-argument combinations.

Configure Initial Parameters

`sys = tfest(data,init_sys)` uses the linear system `init_sys` to configure the initial parameterization of `sys`.

Specify Additional Options

`sys = tfest( ___ ,opt)` specifies the estimation behavior using the option set `opt`. You can use this syntax with any of the previous input-argument combinations.

Return Estimated Initial Conditions

`[sys,ic] = tfest( ___ )` returns the estimated initial conditions as an `initialCondition` object. Use this syntax if you plan to simulate or predict the model response using the same estimation input data and then compare the response with the same estimation output data. Incorporating the initial conditions yields a better match during the first part of the simulation.

Examples

Estimate Transfer Function Model by Specifying Number of Poles

Load the time-domain system-response data `z1`.

```
load iddata1 z1;
```

Set the number of poles `np` to `2` and estimate a transfer function.

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

`sys` is an `idtf` model containing the estimated two-pole transfer function.

View the numerator and denominator coefficients of the resulting estimated model `sys`.

```
sys.Numerator
```

```
ans = 1×2

    2.4554  176.9856

```

```
sys.Denominator
```

```
ans = 1×3

    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 `z2` and use it to estimate a transfer function that contains two poles and one zero.

```
load iddata2 z2;
np = 2;
nz = 1;
sys = tfest(z2,np,nz);
```

`sys` is an `idtf` model containing the estimated transfer function.

Estimate Transfer Function Containing Known Transport Delay

Load the data `z2`, which is an `iddata` object that contains time-domain system response data.

```
load iddata2 z2;
```

Estimate a transfer function model `sys` that contains two poles and one zero, and which includes a known transport delay `iodelay`.

```
np = 2;
nz = 1;
iodelay = 0.2;
sys = tfest(z2,np,nz,iodelay);
```

sys is an idtf model containing the estimated transfer function, with the IODelay property set to 0.2 seconds.

Estimate Transfer Function Containing Unknown Transport Delay

Load time-domain system response data z2 and use it to estimate a two-pole one-zero transfer function for the system. Specify an unknown transport delay for the transfer function by setting the value of iodelay to NaN.

```
load iddata2 z2;
np = 2;
nz = 1;
iodelay = NaN;
sys = tfest(z2,np,nz,iodelay);
```

sys is an idtf model containing the estimated transfer function, whose IODelay property is estimated using the data.

Estimate Discrete-Time Transfer Function

Load time-domain system response data z2.

```
load iddata2 z2
```

Estimate a discrete-time transfer function with two poles and one zero. Specify the sample time Ts as 0.1 seconds and the transport delay iodelay as 2 seconds.

```
np = 2;
nz = 1;
iodelay = 2;
Ts = 0.1;
sysd = tfest(z2,np,nz,iodelay,'Ts',Ts)
```

```
sysd =

  From input "u1" to output "y1":
                     1.8 z^-1
  z^(-2) * ----------------------------
           1 - 1.418 z^-1 + 0.6613 z^-2

Sample time: 0.1 seconds
Discrete-time identified transfer function.

Parameterization:
   Number of poles: 2   Number of zeros: 1
   Number of free coefficients: 3
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.
```

```
Status:
Estimated using TFEST on time domain data "z2".
Fit to estimation data: 80.26%
FPE: 2.095, MSE: 2.063
```

By default, the model has no feedthrough, and the numerator polynomial of the estimated transfer function has a zero leading coefficient `b0`. To estimate `b0`, specify the `Feedthrough` property during estimation.

Estimate Discrete-Time Transfer Function with Feedthrough

Load the estimation data `z5`.

```
load iddata5 z5
```

First, estimate a discrete-time transfer function model with two poles, one zero, and no feedthrough. Get the sample time from the `Ts` property of `z5`.

```
np = 2;
nz = 1;
sys = tfest(z5,np,nz,'Ts',z5.Ts);
```

The estimated transfer function has the following form:

$$H(z^{-1}) = \frac{b1z^{-1} + b2z^{-2}}{1 + a1z^{-1} + a2z^{-2}}$$

By default, the model has no feedthrough, and the numerator polynomial of the estimated transfer function has a zero leading coefficient `b0`. To estimate `b0`, specify the `Feedthrough` property during estimation.

```
sys = tfest(z5,np,nz,'Ts',z5.Ts,'Feedthrough',true);
```

The numerator polynomial of the estimated transfer function now has a nonzero leading coefficient:

$$H(z^{-1}) = \frac{b0 + b1z^{-1} + b2z^{-2}}{1 + a1z^{-1} + a2z^{-2}}$$

Analyze 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 either to a lack of feedthrough or to an actual transport delay. For discrete-time models, absence of feedthrough corresponds to a lag of one sample between the input and output. Estimating a model using `Feedthrough = false` and `iodelay = 0` thus produces a discrete-time system that is equivalent to a system estimated using `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 representations. Therefore, a best practice is to check if the observed delay can be attributed to a transport delay or to a lack of feedthrough.

Estimate a discrete-time model with no feedthrough.

```
load iddata1 z1
np = 2;
nz = 2;
sys1 = tfest(z1,np,nz,'Ts',z1.Ts);
```

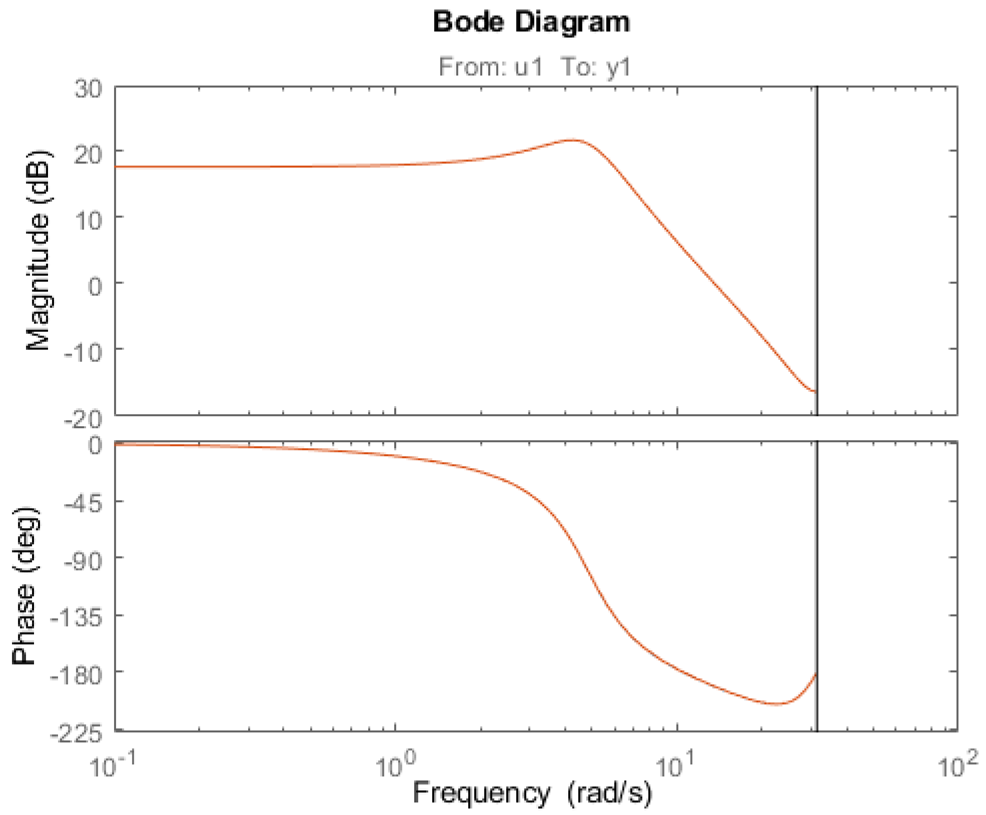
Because `sys1` has no feedthrough and therefore has a numerator polynomial that beings with z^{-1}, `sys1` has a lag of one sample. The `IODelay` property is 0.

Estimate another discrete-time model with feedthrough and with a reduction from two zeros to one, incurring a one-sample input-output delay.

```
sys2 = tfest(z1,np,nz-1,1,'Ts',z1.Ts,'Feedthrough',true);
```

Compare the Bode responses of the models.

```
bode(sys1,sys2);
```

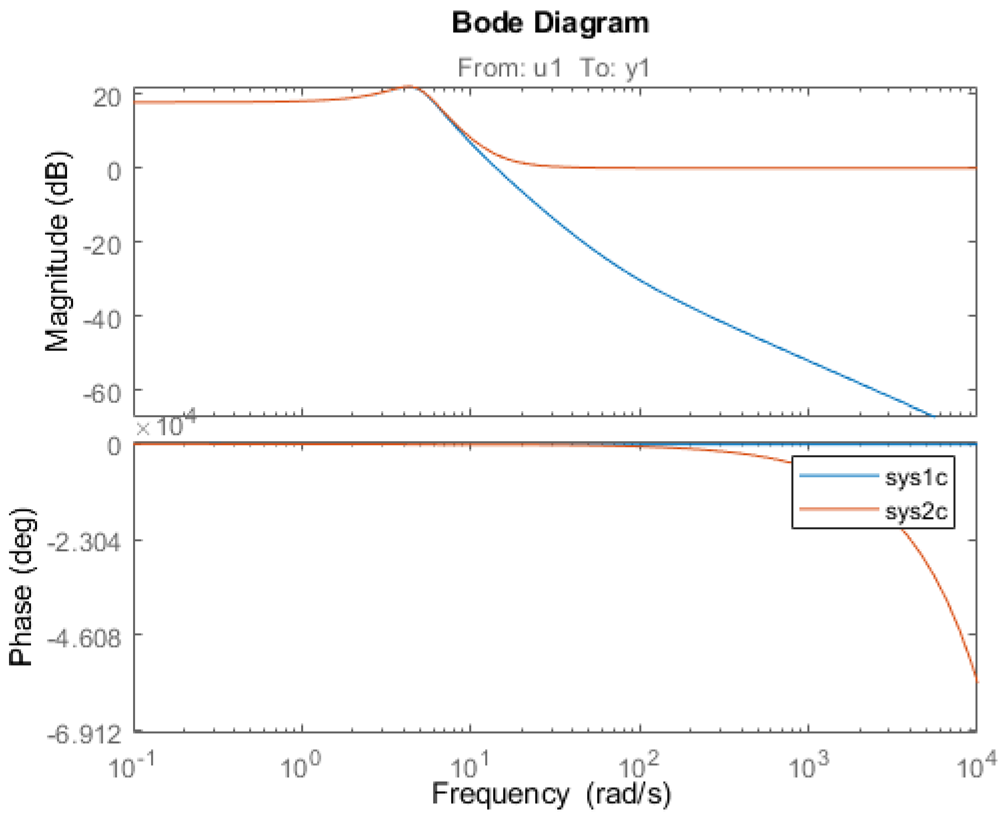


The discrete equations that underlie sys1 and sys2 are equivalent, and so are the Bode responses.

Convert the models to continuous time and compare the Bode responses for these models.

```
sys1c = d2c(sys1);
sys2c = d2c(sys2);
```

```
bode(sys1c,sys2c);
legend
```



As the plot shows, the Bode responses of the two models do not match when you convert them to continuous time. When there is no feedthrough, as with `sys1c`, there must be some lag. When there is feedthrough, as with `sys2c`, there can be no lag. Continuous-time feedthrough maps to discrete-time feedthrough. Continuous-time lag maps to discrete-time delays.

Estimate MISO Discrete-Time Transfer Function with Feedthrough and Delay Specifications for Individual Channels

Estimate a two-input, one-output discrete-time transfer function with a delay of two 2 samples on the first input and zero samples on the second input. Both inputs have no feedthrough.

Load the data and split the data into estimation and validation data sets.

```
load iddata7 z7
ze = z7(1:300);
zv = z7(200:400);
```

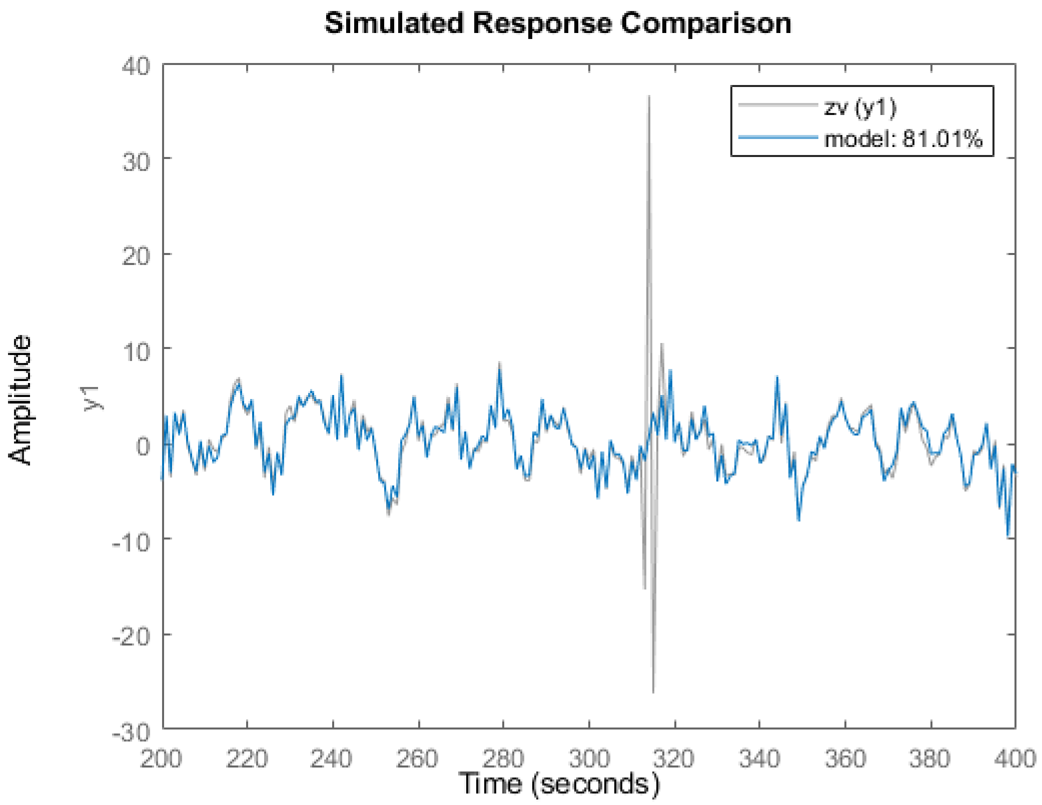
Estimate a two-input, one-output transfer function with two poles and one zero for each input-to-output transfer function.

```
Lag = [2;0];
Ft = [false,false];
model = tfest(ze,2,1,'Ts',z7.Ts,'Feedthrough',Ft,'InputDelay',Lag);
```

The `Feedthrough` value you choose dictates whether the leading numerator coefficient is zero (no feedthrough) or not (nonzero feedthrough). Delays are generally expressed separately using the `InputDelay` or `IODelay` property. This example uses `InputDelay` only 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 the data.

```
load regularizationExampleData m0simdata;
```

Obtain a regularized impulse response (FIR) model.

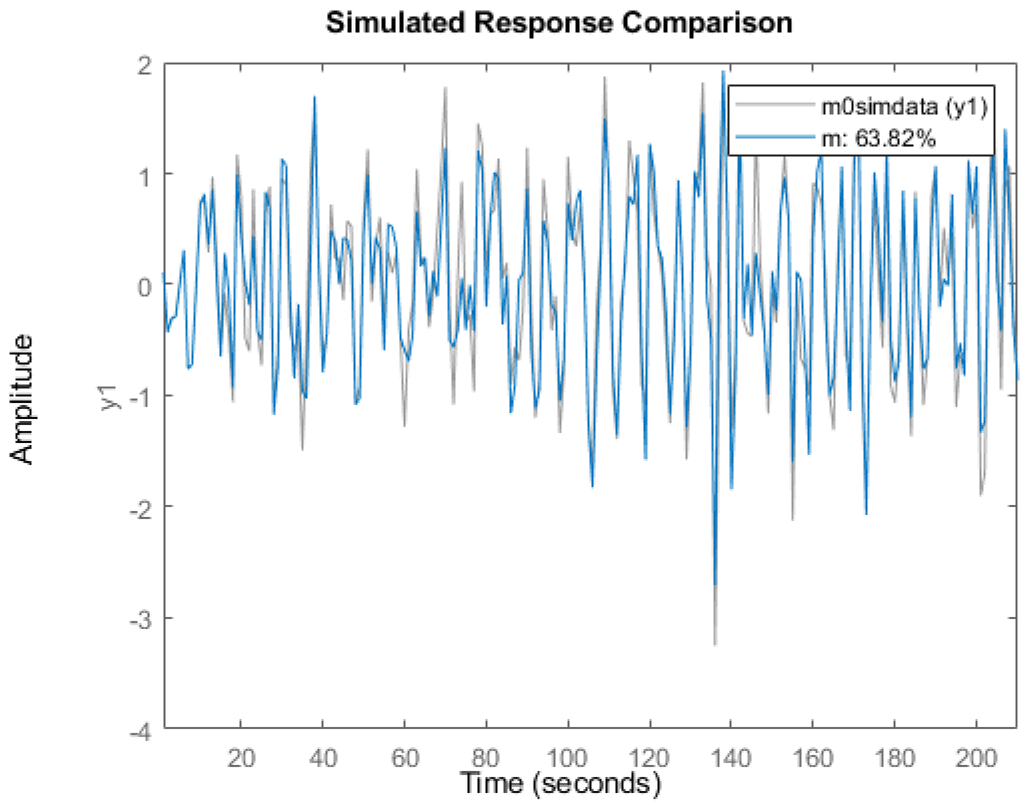
```
opt = impulseestOptions('RegularizationKernel','DC');
m0 = impulseest(m0simdata,70,opt);
```

Convert the model into a transfer function model after reducing the order to 15.

```
m = idtf(balred(idss(m0),15));
```

Compare the model output with the data.

```
compare(m0simdata,m);
```



Estimate Transfer Function Using Estimation Option Set

Create an option set for `tfest` that specifies the initialization and search methods. Also set the display option, which specifies that the loss-function values for each iteration be shown.

```
opt = tfestOptions('InitializeMethod','n4sid','Display','on','SearchMethod','lsqnonlin');
```

Load time-domain system response data `z2` and use it to estimate a transfer function with two poles and one zero. Specify `opt` for the estimation options.

```
load iddata2 z2;
np = 2;
nz = 1;
iodelay = 0.2;
sys = tfest(z2,np,nz,iodelay,opt);
```

sys is an idtf model containing the estimated transfer function.

Specify Model Properties of Estimated Transfer Function

Load the time-domain system response data z2, and use it to estimate a two-pole, one-zero transfer function. Specify an input delay.

```
load iddata2 z2;
np = 2;
nz = 1;
input_delay = 0.2;
sys = tfest(z2,np,nz,'InputDelay',input_delay);
```

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

Use bode to obtain the magnitude and phase response 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 three-pole, one-zero transfer function using data.

```
np = 3;
nz = 1;
sys = tfest(data,np,nz);
```

sys is an idtf model containing the estimated transfer function.

Estimate Transfer Function with Known Transport Delays for Multiple Inputs

Load the time-domain system response data co2data, which contains the data from two experiments, each with two inputs and one output. Convert the data from the first experiment into an iddata object data with a sample time of 0.5 seconds.

```
load co2data;
Ts = 0.5;
data = iddata(Output_exp1,Input_exp1,Ts);
```

Specify estimation options for the search method and the input and output offsets. Also specify the maximum number of search iterations.

```
opt = tfestOptions('SearchMethod','gna');
opt.InputOffset = [170;50];
opt.OutputOffset = mean(data.y(1:75));
opt.SearchOptions.MaxIterations = 50;
```

Estimate a transfer function using the measured data and the estimation option set `opt`. 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 with two input channels and one output channels, and which has a sample rate of 0.5 seconds.

Create an option set `opt`. Specify estimation options for the search method and the input and output offsets. Also specify the maximum number of search iterations.

```
opt = tfestOptions('Display','on','SearchMethod','gna');
opt.InputOffset = [170; 50];
opt.OutputOffset = mean(data.y(1:75));
opt.SearchOptions.MaxIterations = 50;
```

Specify the unknown and known transport delays in `iodelay`, using `2` for a known delay of 2 seconds and `nan` for the unknown delay. Estimate the transfer function using `iodelay` and `opt`.

```
np = 3;
nz = 1;
iodelay = [2 nan];
sys = tfest(data,np,nz,iodelay,opt);
```

`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];
```

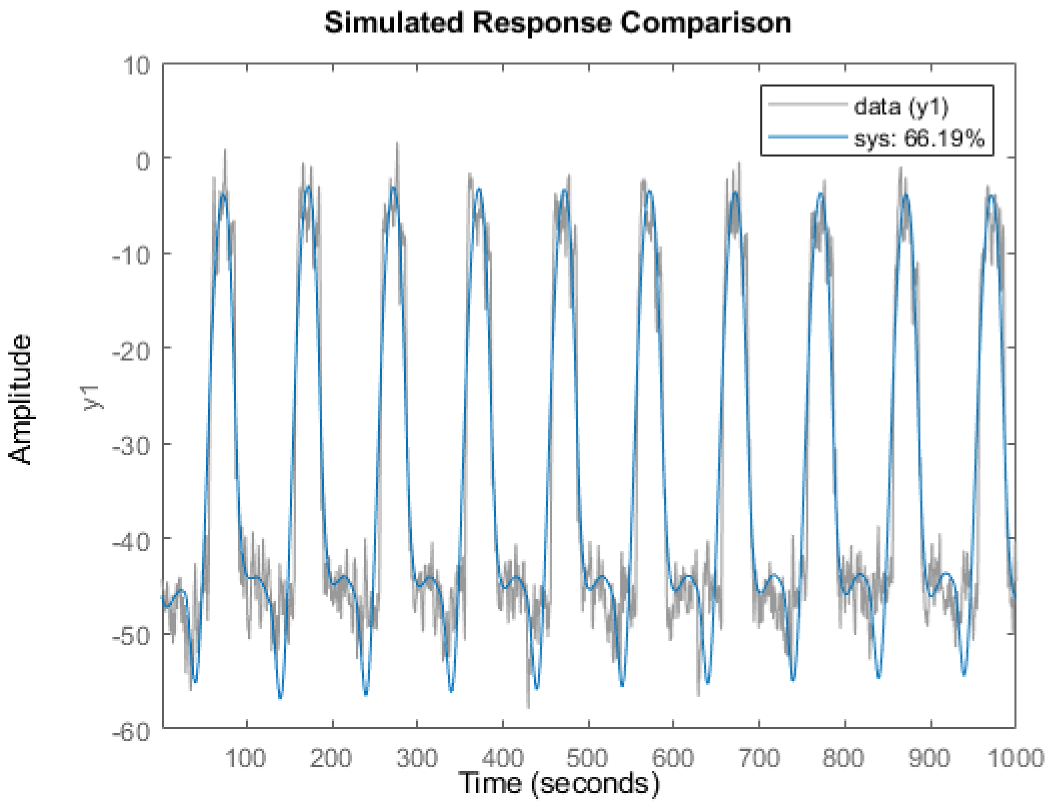
Load time-domain system response data and use it to estimate a transfer function. Specify options in the `tfestOptions` option set `opt`.

```
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.SearchOptions.MaxIterations = 50;
sys = tfest(data,init_sys,opt);
```

`sys` is an `idtf` model containing the estimated transfer function.

Analyze the estimation result by comparison. Create a `compareOptions` option set `opt2` and specify input and output offsets, and then use `compare`.

```
opt2 = compareOptions;
opt2.InputOffset = opt.InputOffset;
opt2.OutputOffset = opt.OutputOffset;
compare(data,sys,opt2)
```



Estimate Transfer Function Containing Different Numbers of Poles for Input-Output Pairs

Estimate a multiple-input, single-output transfer function containing different numbers of poles for input-output pairs for given data.

Obtain frequency-response data.

For example, use `frd` to create a frequency-response data model for the following system:

$$G = \begin{bmatrix} e^{-4s}\dfrac{s+2}{s^3+2s^2+4s+5} \\ e^{-0.6s}\dfrac{5}{s^4+2s^3+s^2+s} \end{bmatrix}$$

Use 100 frequency points, ranging from 0.01 rad/s to 100 rad/s, to obtain the frequency-response data.

```
G = tf({[1 2],[5]},{[1 2 4 5],[1 2 1 1 0]},0,'IODelay',[4 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 three poles. Similarly, the second element of np indicates that the transfer function from the second input to the output contains four 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 one 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 using frequency-response data.

Use idtf to construct a transfer function model G of the following system:

$$G = \begin{bmatrix} \dfrac{s+2}{s^3+2s^2+4s+5} \\ \dfrac{5}{s^4+2s^3+s^2+s+1} \end{bmatrix}$$

```
G = idtf({[1 2], 5},{[1 2 4 5],[1 2 1 1 1]});
```

Use idfrd to obtain a frequency-response data model data for G. Specify 100 frequency points ranging from 0.01 rad/s to 100 rad/s.

```
data = idfrd(G,logspace(-2,2,100));
```

data is an idfrd object.

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 three poles. Similarly, the second element of np indicates that the transfer function from the second input to the output contains four 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 one 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 = 7×1 complex

  -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 the pole display indicates.

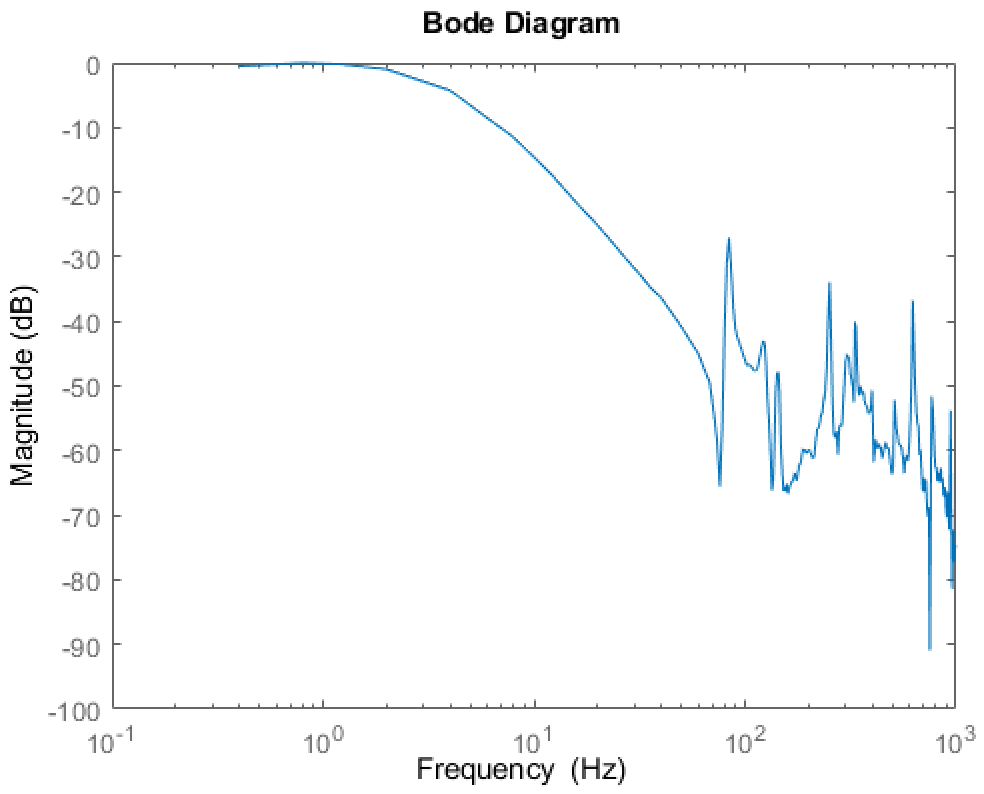
Estimate Transfer Function using High Modal Density Frequency Response Data

Load the high-density frequency-response measurement data. The data corresponds to an unstable process maintained at equilibrium using feedback control.

```
load HighModalDensityData FRF f
```

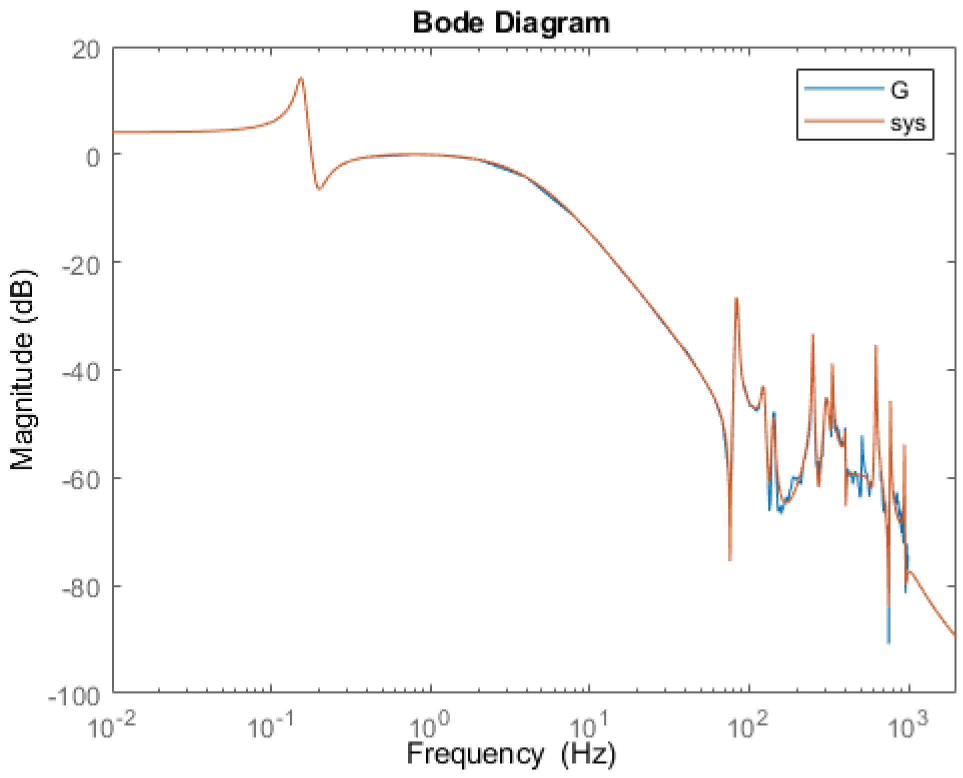
Package the data as an `idfrd` object for identification and find the Bode magnitude response.

```
G = idfrd(permute(FRF,[2 3 1]),f,0,'FrequencyUnit','Hz');
bodemag(G)
```



Estimate a transfer function with 32 poles and 32 zeros, and compare the Bode magnitude response.

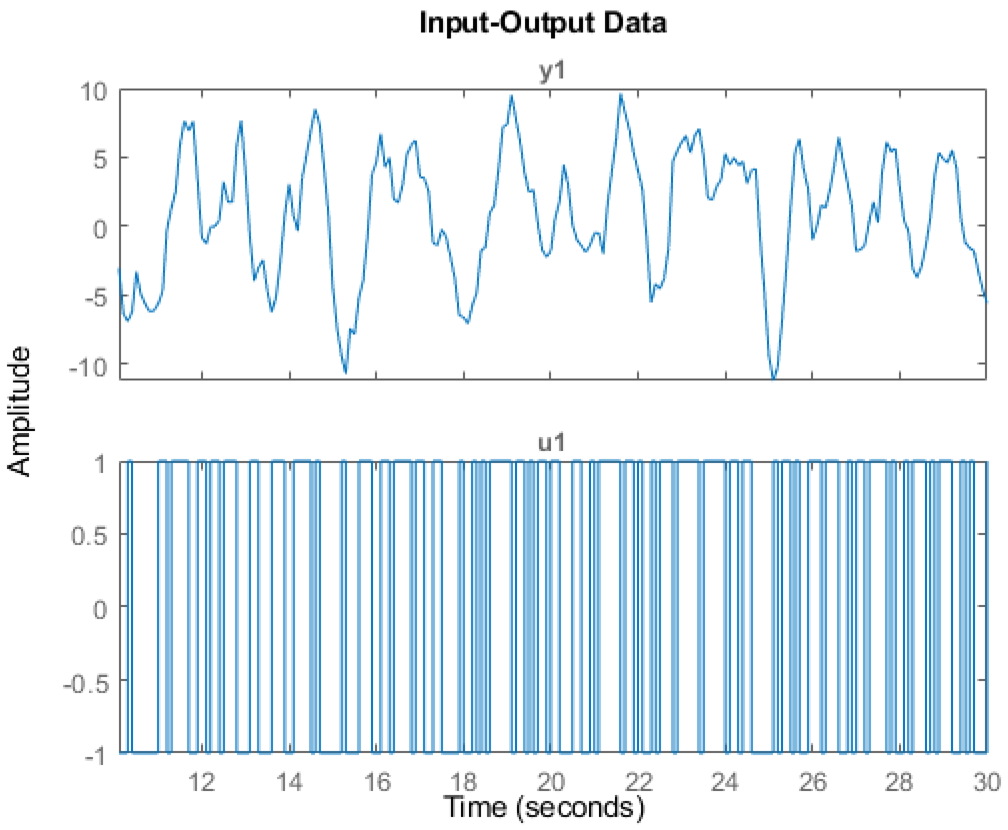
```
sys = tfest(G,32,32);
bodemag(G, sys)
xlim([0.01,2e3])
legend
```



Obtain and Apply Estimated Initial Conditions

Load and plot the data.

```
load iddata1ic z1i
plot(z1i)
```



Examine the initial value of the output data `y(1)`.

```
ystart = z1i.y(1)
```

```
ystart = -3.0491
```

The measured output does not start at 0.

Estimate a second-order transfer function `sys` and return the estimated initial condition `ic`.

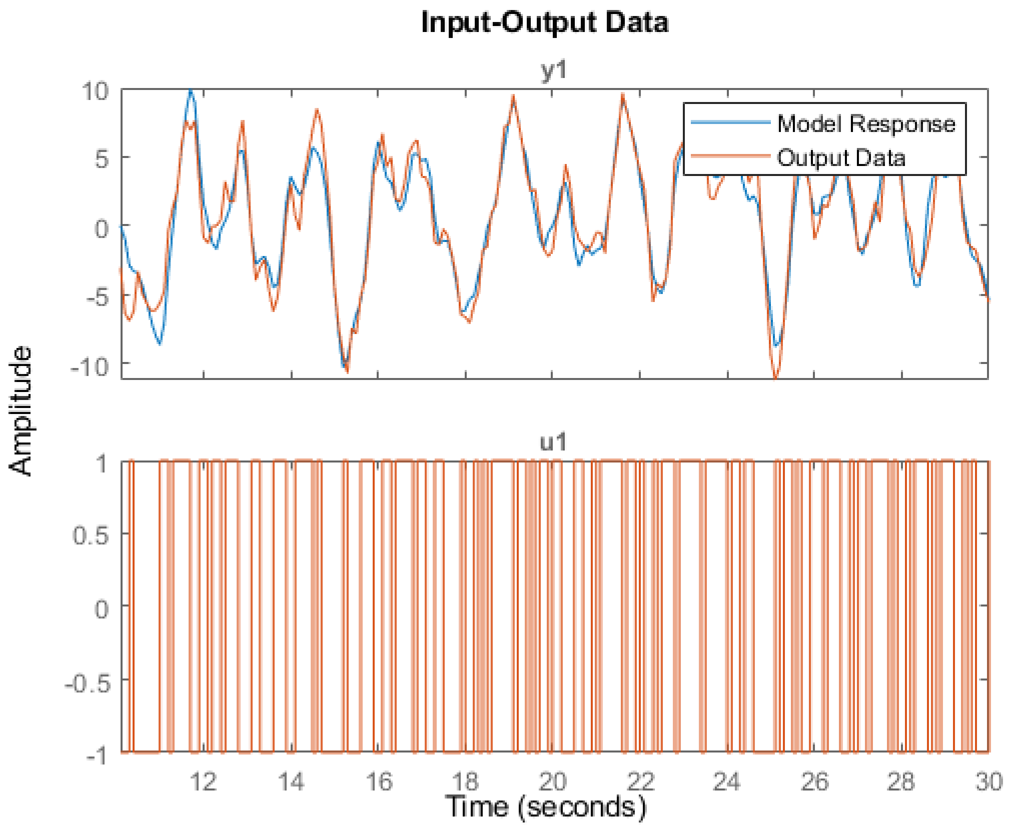
```
[sys,ic] = tfest(z1i,2,1);
ic
```

```
ic = 
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [0.2957 5.2441]
    Ts: 0
```

`ic` is an `initialCondition` object that encapsulates the free response of `sys`, in state-space form, to the initial state vector in `X0`.

Simulate `sys` using the estimation data but without incorporating the initial conditions. Plot the simulated output with the measured output.

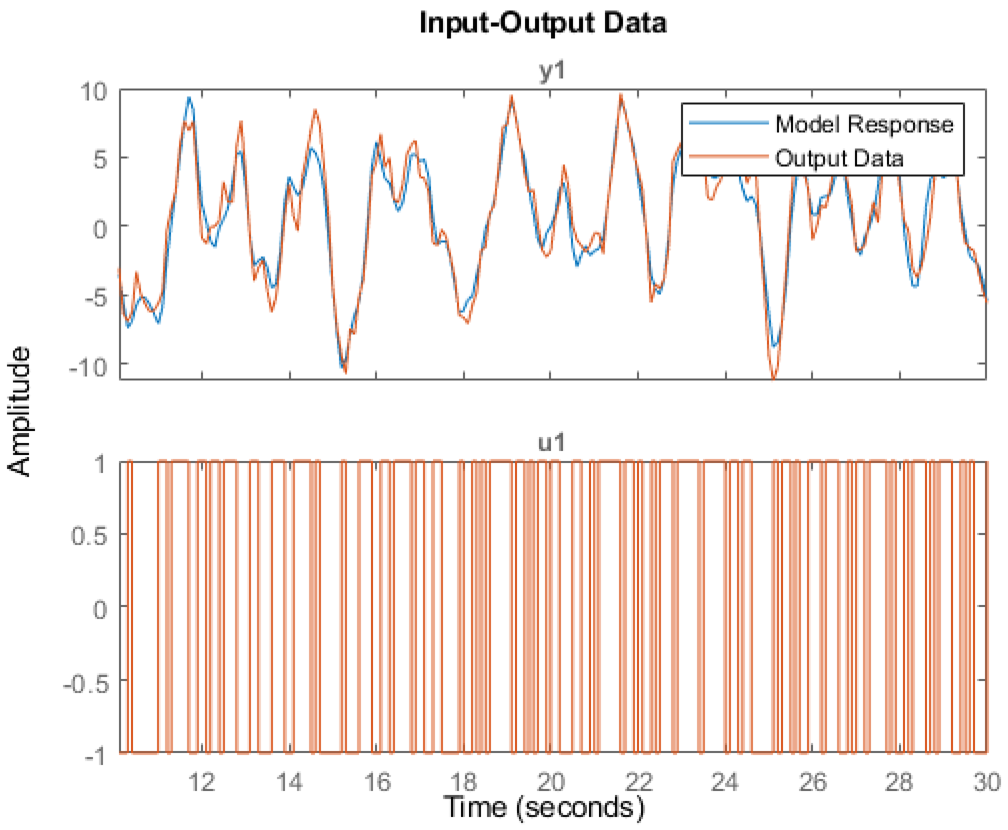
```
y_no_ic = sim(sys,z1i);
plot(y_no_ic,z1i)
legend('Model Response','Output Data')
```



The measured and simulated outputs do not agree at the beginning of the simulation.

Incorporate the initial condition into the `simOptions` option set.

```
opt = simOptions('InitialCondition',ic);
y_ic = sim(sys,z1i,opt);
plot(y_ic,z1i)
legend('Model Response','Output Data')
```



The simulation combines the model response to the input signal with the free response to the initial condition. The measured and simulated outputs now have better agreement at the beginning of the simulation. This initial condition is valid only for the estimation data `z1i`.

Input Arguments

data — Estimation data
`iddata` object | `frd` object | `idfrd` object

Estimation data, specified as an `iddata` object, an `frd` object, or an `idfrd` object.

For time-domain estimation, `data` must be an `iddata` object containing the input and output signal values.

Time-series models, which are models that contain no measured inputs, cannot be estimated using `tfest`. Use `ar`, `arx`, or `armax` for time-series models instead.

For frequency-domain estimation, `data` can be one of the following:

- Recorded frequency response data (`frd` or `idfrd`)
- `iddata` object with properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal

- `Domain` — `'Frequency'`

Estimation data must be uniformly sampled.

For multiexperiment data, the sample times and intersample behavior of all the experiments must match.

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. You can estimate only continuous-time models using continuous-time frequency-domain data.

`np` — Number of poles
nonnegative integer | matrix

Number of poles in the estimated transfer function, specified as a nonnegative integer or a matrix.

For systems that have multiple inputs and/or multiple outputs, you can apply either a global value or individual values of `np` to the input-output pairs, as follows:

- Same number of poles for every pair — Specify `np` as a scalar.
- Individual number of poles for each pair — Specify `np` as an n_y-by-n_u matrix. n_y is the number of outputs and n_u is the number of inputs.

For an example, see “Estimate Transfer Function Model by Specifying Number of Poles” on page 1-1706.

`nz` — Number of zeros
nonnegative integer | matrix

Number of zeros in the estimated transfer function, specified as a nonnegative integer or a matrix.

For systems that have multiple inputs, multiple outputs, or both, you can apply either a global value or individual values of `nz` to the input-output pairs, as follows:

- Same number of poles for every pair — Specify `nz` as a scalar.
- Individual number of poles for each pair — Specify `nz` as an n_y-by-n_u matrix. n_y is the number of outputs and n_u is the number of inputs.

For a continuous-time model estimated using discrete-time data, set `nz <= np`.

For discrete-time model estimation, specify `nz` as the number of zeros of the numerator polynomial of the transfer function. For example, `tfest(data,2,1,'Ts',data.Ts)` estimates a transfer function of the form $b_1z^{-1}/(1 + a_1z^{-1} + b_2z^{-2})$, while `tfest(data,2,2,'Ts',data.Ts)` estimates $(b_1z^{-1} + b_2z^{-2})/(1 + a_1z^{-1} + b_2z^{-2})$. Here, z^{-1} is the Z-transform lag variable. For more information about discrete-time transfer functions, see “Discrete-Time Representation”. For an example, see “Estimate Discrete-Time Transfer Function” on page 1-1708.

`iodelay` — Transport delay
`[ ]` (default) | nonnegative integer | matrix

Transport delay, specified as a nonnegative integer, an `NaN` scalar, or a matrix.

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 delays of a multiple of the sample time `Ts`.

For a MIMO system with N_y outputs and N_u inputs, set `iodelay` to an N_y-by-N_u 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. To denote unknown transport delays, specify `NaN` in the `iodelay` matrix.

Use `[]` or `0` to indicate that there is no transport delay.

For an example, see “Estimate Transfer Function Containing Known Transport Delay” on page 1-1707.

opt — Estimation options
`n4sidOptions` option set

Estimation options, specified as an `tfestOptions` option set. Options specified by `opt` include:

- Estimation objective
- Handling of initial conditions
- Numerical search method to be used in estimation

For an example, see “Estimate Transfer Function Using Estimation Option Set” on page 1-1713.

init_sys — Linear system that configures initial parameterization of sys
`idtf` model | linear model | structure

Linear system that configures the initial parameterization of `sys`, specified as an `idtf` model or as a structure. You obtain `init_sys` either by performing an estimation using measured data or by direct construction.

If `init_sys` is an `idtf` model, `tfest` uses the parameter values of `init_sys` as the initial guess for estimating `sys`.

Use the `Structure` property of `init_sys` to configure initial parameter values and constraints for the numerator, denominator, and transport lag. For instance:

- To specify an initial guess for the A matrix of `init_sys`, set `init_sys.Structure.Numerator.Value` to the initial guess.
- To specify constraints for the B matrix 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.

  For an example, see “Estimate Transfer Function with Unknown, Constrained Transport Delays” on page 1-1715.

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 you do not specify `opt`, and `init_sys` was obtained by estimation rather than construction, then the software uses estimation options from `init_sys.Report.OptionsUsed`.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sys = tfest(data,np,nz,'Ts',0.1)`

`Ts` — Sample time of estimated model

`0` (continuous time) (default) | positive scalar

Sample time of the estimated model, specified as the comma-separated pair consisting of `'Ts'` and either `0` or a positive scalar.

- For continuous-time models, specify `'Ts'` as `0`.
- For discrete-time models, specify `'Ts'` as the data sample time in the units stored in the `TimeUnit` property. In the discrete case, `np` and `nz` refer to the number of roots of z^{-1} for the numerator and denominator polynomials.

For an example, see “Estimate Discrete-Time Transfer Function” on page 1-1708.

`InputDelay` — Input delays

`0` (default) | scalar | vector

Input delay for each input channel, specified as the comma-separated pair consisting of `'InputDelay'` and a numeric vector.

- For continuous-time models, specify `'InputDelay'` in the time units stored in the `TimeUnit` property.
- For discrete-time models, specify `'InputDelay'` in integer multiples of the sample time `Ts`. For example, setting `'InputDelay'` to `3` specifies a delay of three sampling periods.

For a system with N_u inputs, set `InputDelay` to an N_u-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.

To apply the same delay to all channels, specify `InputDelay` as a scalar.

For an example, see “Specify Model Properties of Estimated Transfer Function” on page 1-1714.

`Feedthrough` — Feedthrough for discrete-time transfer function

`0` (default) | `1` | logical matrix

Feedthrough for discrete-time transfer functions, specified as the comma-separated pair consisting of `'Feedthrough'` a logical scalar or an N_y-by-N_u logical matrix. N_y is the number of outputs and N_u is the number of inputs. To use the same feedthrough for all input-output channels, specify `Feedthrough` as a scalar.

Consider a discrete-time model with two poles and three zeros:

$$H(z^{-1}) = \frac{b0 + b1z^{-1} + b2z^{-2} + b3z^{-3}}{1 + a1z^{-1} + a2z^{-2}}$$

When the model has direct feedthrough, *b0* is a free parameter whose value is estimated along with the rest of the model parameters *b1*, *b2*, *b3*, *a1*, and *a2*. When the model has no feedthrough, *b0* is fixed to zero. For an example, see “Estimate Discrete-Time Transfer Function with Feedthrough” on page 1-1709.

Output Arguments

sys — Identified transfer function
`idtf` model

Identified transfer function, returned as an `idtf` 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.
`InitializeMethod`	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 `InitializeMethod` option in the estimation option set is `'all'`.
`N4Weight`	Weighting matrices used in the singular-value decomposition step when `InitializeMethod` is `'n4sid'`, returned as one of the following values: • `'MOESP'` — Use the MOESP algorithm by Verhaegen. • `'CVA'` — Use the canonical variate algorithm (CVA) by Larimore. • `'SSARX'` — Use 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'`.
`N4Horizon`	Forward and backward prediction horizons used when `InitializeMethod` 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.

Report Field	Description
`InitialCondition`	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: **Field** — **Description** `FitPercent` — Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as the percentage *fitpercent* = 100(1-NRMSE). `LossFcn` — Value of the loss function when the estimation completes. `MSE` — Mean squared error (MSE) measure of how well the response of the model fits the estimation data. `FPE` — Final prediction error for the model. `AIC` — Raw Akaike Information Criteria (AIC) measure of model quality. `AICc` — Small-sample-size corrected AIC. `nAIC` — Normalized AIC. `BIC` — Bayesian Information Criteria (BIC).
`Parameters`	Estimated values of model parameters.
`OptionsUsed`	Option set used for estimation. If no custom options were configured, this is a set of default options. See `polyestOptions` for more information.
`RandState`	State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng`.

Report Field	Description	
`DataUsed`	Attributes of the data used for estimation, returned as a structure with the following fields.	
	Field	**Description**
	`Name`	Name of the data set.
	`Type`	Data type.
	`Length`	Number of data samples.
	`Ts`	Sample time.
	`InterSample`	Input intersample behavior, returned as one of the following values: • `'zoh'` — 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.
	`InputOffset`	Offset removed from time-domain input data during estimation. For nonlinear models, it is `[]`.
	`OutputOffset`	Offset removed from time-domain output data during estimation. For nonlinear models, it is `[]`.
`Termination`	Termination conditions for the iterative search used for prediction error minimization, returned as a structure with the following fields:	
	Field	**Description**
	`WhyStop`	Reason for terminating the numerical search.
	`Iterations`	Number of search iterations performed by the estimation algorithm.
	`FirstOrderOptimality`	∞-norm of the gradient search vector when the search algorithm terminates.
	`FcnCount`	Number of times the objective function was called.
	`UpdateNorm`	Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`.
	`LastImprovement`	Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'` or `'fmincon'`.
	`Algorithm`	Algorithm used by `'lsqnonlin'` or `'fmincon'` search method. Omitted when other search methods are used.
	For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.	

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

ic — Initial conditions

`initialCondition` object | object array of `initialCondition` values

Estimated initial conditions, returned as an `initialCondition` object or an object array of `initialCondition` values.

- For a single-experiment data set, `ic` represents, in state-space form, the free response of the transfer function model (A and C matrices) to the estimated initial states (x_0).
- For a multiple-experiment data set with N_e experiments, `ic` is an object array of length N_e that contains one set of `initialCondition` values for each experiment.

If `tfest` returns `ic` values of `0` and the you know that you have non-zero initial conditions, set the `'InitialCondition'` option in `tfestOptions` to `'estimate'` and pass the updated option set to `tfest`. For example:

```
opt = tfestOptions('InitialCondition','estimate')
[sys,ic] = tfest(data,np,nz,opt)
```

The default `'auto'` setting of `'InitialCondition'` uses the `'zero'` method when the initial conditions have a negligible effect on the overall estimation-error minimization process. Specifying `'estimate'` ensures that the software estimates values for `ic`.

For more information, see `initialCondition`. For an example of using this argument, see “Obtain and Apply Estimated Initial Conditions” on page 1-1721.

Algorithms

The details of the estimation algorithms used by `tfest` vary depending on various 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 `InitializeMethod` 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 (`InitializeOptions`) 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 `MaxIterations` times, until the model change is less than `Tolerance`. `MaxIterations` and `Tolerance` are options that you can specify using the `InitializeOptions` structure. The `'n4sid'` initialization option estimates a discrete-time model, using the N4SID estimation algorithm, that it transforms to continuous-time using `d2c`.

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 following 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 ith 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

$$\underset{D_m, N_{i,m}}{\text{minimize}} \sum_{k=1}^{n_f} \left| \frac{W(\omega_k)}{D_{m-1}(\omega_k)}\left(D_m(\omega_k)y(\omega_k) - \sum_{i=1}^{n_u} N_{i,m}(\omega_k) u_i(\omega_k) \right) \right|^2$$

Here, 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 $i = 1,2,\ldots n_u$. 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 have only 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 the 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 can 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 follow.

- Equation for the *j*th denominator parameter:

$$0 = 2\sum_{k=1}^{n_f} \mathrm{Re}\left\{ \frac{|W(\omega_k)|^2 B_j^*(\omega_k) \sum_{i=1}^{n_u} N_{i,m-1}^*(\omega_k) u_i^*(\omega_k)}{D_{m-1}^*(\omega_k) |D_{m-1}(\omega_k)|^2} \left(D_m(\omega_k) y(\omega_k) - \sum_{i=1}^{n_u} N_{i,m}(\omega_k) u_i(\omega_k) \right) \right\}$$

- Equation for the *j*th numerator parameter that corresponds to input *l*:

$$0 = -2\sum_{k=1}^{n_f} \mathrm{Re}\left\{ \frac{|W(\omega_k)|^2 B_j^*(\omega_k) u_l^*(\omega_k)}{|D_{m-1}(\omega_k)|^2} \left(D_m(\omega_k) y(\omega_k) - \sum_{i=1}^{n_u} N_{i,m}(\omega_k) u_i(\omega_k) \right) \right\}$$

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 using 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 1-1734).
- 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 “tfest Estimation Algorithm Update” on page 1-1735) 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 on page 1-1732, except discrete-time data is used.

If you are using the estimation algorithm from R2016a or earlier (see “tfest Estimation Algorithm Update” on page 1-1735), the algorithm is the same as the algorithm for discrete-time transfer function estimation using time-domain data on page 1-1732.

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

The `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 the algorithm for continuous-time transfer function estimation using continuous-time data on page 1-1732.

If you are using the estimation algorithm from R2016a or earlier (see “tfest Estimation Algorithm Update” on page 1-1735), 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. A best practice is to 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”.

Compatibility Considerations

`tfest` Estimation Algorithm Update

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 might 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 the frequency-domain data `data`.

```
sys = tfest(data,np)
```

To use the previous estimation algorithm, use the following syntax.

```
sys = tfest(data,np,'-R2016a')
```

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 76*, no. 13 (January 2003): 1337–57. https://doi.org/10.1080/0020717031000149636.

[2] Ljung, Lennart. “Experiments with Identification of Continuous Time Models.” *IFAC Proceedings Volumes* 42, no. 10 (2009): 1175–80. https://doi.org/10.3182/20090706-3-FR-2004.00195.

[3] Young, Peter, and Anthony Jakeman. “Refined Instrumental Variable Methods of Recursive Time-Series Analysis Part III. Extensions.” *International Journal of Control* 31, no. 4 (April 1980): 741–64. https://doi.org/10.1080/00207178008961080.

[4] Drmač, Z., S. Gugercin, and C. Beattie. “Quadrature-Based Vector Fitting for Discretized H_2 Approximation.” *SIAM Journal on Scientific Computing* 37, no. 2 (January 2015): A625–52. https://doi.org/10.1137/140961511.

[5] Ozdemir, Ahmet Arda, and Suat Gumussoy. “Transfer Function Estimation in System Identification Toolbox via Vector Fitting.” *IFAC-PapersOnLine* 50, no. 1 (July 2017): 6232–37. https://doi.org/10.1016/j.ifacol.2017.08.1026.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Parallel computing support is available for estimation using the `lsqnonlin` search method (requires Optimization Toolbox). To enable parallel computing, use `tfestOptions`, set `SearchMethod` to `'lsqnonlin'`, and set `SearchOptions.Advanced.UseParallel` to `true`.

For example:

```
opt = tfestOptions;
opt.SearchMethod = 'lsqnonlin';
opt.SearchOptions.Advanced.UseParallel = true;
```

See Also

`tfestOptions` | `idtf` | `ssest` | `procest` | `ar` | `arx` | `oe` | `bj` | `polyest` | `greyest`

Topics

"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"
"Apply Initial Conditions when Simulating Identified Linear Models"
"Troubleshoot Frequency-Domain Identification of Transfer Function Models"
"What are Transfer Function Models?"
"Regularized Estimates of Model Parameters"
"Estimating Models Using Frequency-Domain Data"

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`InitializeMethod` — 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.

`InitializeOptions` — 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 `InitializeMethod` is `'n4sid'`.

  `N4Weight` is specified as one of the following values:

    - `'MOESP'` — Uses the MOESP algorithm by Verhaegen.
    - `'CVA'` — Uses the canonical variate 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 `InitializeMethod` 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`
- `MaxIterations` — Maximum number of iterations. Applicable when `InitializeMethod` is `'iv'`.

  Default: `30`
- `Tolerance` — Convergence tolerance. Applicable when `InitializeMethod` 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 $1/|G(\omega)|$ as the weighting filter, where $G(\omega)$ 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 $1/\sqrt{|G(\omega)|}$ 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`

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

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

If `EstimateCovariance` 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'` | `'fmincon'`

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 in sequence at each iteration. The first descent direction leading to a reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConstant*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated as 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 `InitialGnaTolerance` (see `Advanced` in `'SearchOptions'` 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 of `lsqnonlin`. Requires Optimization Toolbox software.
- `'fmincon'` — Constrained nonlinear solvers. You can use the sequential quadratic programming (SQP) and trust-region-reflective algorithms of the `fmincon` solver. If you have Optimization Toolbox software, you can also use the interior-point and active-set algorithms of the `fmincon` solver. Specify the algorithm in the `SearchOptions.Algorithm` option. The `fmincon` algorithms may result in improved estimation results in the following scenarios:

  - Constrained minimization problems when there are bounds imposed on the model parameters.
  - Model structures where the loss function is a nonlinear or non smooth function of the parameters.
  - Multi-output model estimation. A determinant loss function is minimized by default for multi-output model estimation. `fmincon` algorithms are able to minimize such loss functions directly. The other search methods such as `'lm'` and `'gn'` minimize the determinant loss function by alternately estimating the noise variance and reducing the loss value for a given noise variance value. Hence, the `fmincon` algorithms can offer better efficiency and accuracy for multi-output model estimations.

`SearchOptions` — Option set for the search algorithm
search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOptions'` and a search option set with fields that depend on the value of `SearchMethod`.

SearchOptions Structure When SearchMethod is Specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description	Default
Tolerance	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
MaxIterations	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as Tolerance. Setting MaxIterations = 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

Field Name	Description	Default
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvConstant`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConstant*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitialGnaTolerance`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartValue`	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`
`MaxBisections`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`
`MaxFunctionEvaluations`	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.	`Inf`
`MinParameterChange`	Smallest parameter update allowed per iteration, specified as a nonnegative scalar.	`0`
`RelativeImprovement`	Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.	`0`
`StepReduction`	Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until `MaxBisections` tries are completed or a lower value of the criterion function is obtained. `StepReduction` is not applicable for `SearchMethod` `'lm'` (Levenberg-Marquardt method).	`2`

`SearchOptions` Structure When `SearchMethod` is Specified as `'lsqnonlin'`

Field Name	Description	Default
`FunctionTolerance`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `FunctionTolerance` is the same as that of `opt.SearchOptions.Advanced.TolFun`.	`1e-5`
`StepTolerance`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `StepTolerance` is the same as that of `opt.SearchOptions.Advanced.TolX`.	`1e-6`
`MaxIterations`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIterations` is reached or another stopping criterion is satisfied, such as `FunctionTolerance`. The value of `MaxIterations` is the same as that of `opt.SearchOptions.Advanced.MaxIter`.	`20`
`Advanced`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options” (Optimization Toolbox).	Use `optimset('lsqnonlin')` to create a default option set.

`SearchOptions` Structure When `SearchMethod` is Specified as `'fmincon'`

Field Name	Description	Default
`Algorithm`	`fmincon` optimization algorithm, specified as one of the following: • `'sqp'` — Sequential quadratic programming algorithm. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. It is not a large-scale algorithm. For more information, see “Large-Scale vs. Medium-Scale Algorithms” (Optimization Toolbox). • `'trust-region-reflective'` — Subspace trust-region method based on the interior-reflective Newton method. It is a large-scale algorithm. • `'interior-point'` — Large-scale algorithm that requires Optimization Toolbox software. The algorithm satisfies bounds at all iterations, and it can recover from `NaN` or `Inf` results. • `'active-set'` — Requires Optimization Toolbox software. The algorithm can take large steps, which adds speed. It is not a large-scale algorithm. For more information about the algorithms, see “Constrained Nonlinear Optimization Algorithms” (Optimization Toolbox) and “Choosing the Algorithm” (Optimization Toolbox).	`'sqp'`

Field Name	Description	Default
FunctionTolerance	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar.	1e-6
StepTolerance	Termination tolerance on the estimated parameter values, specified as a positive scalar.	1e-6
MaxIterations	Maximum number of iterations during loss function minimization, specified as a positive integer. The iterations stop when MaxIterations is reached or another stopping criterion is satisfied, such as FunctionTolerance.	100

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

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

  - 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('InitializeMethod','n4sid','Display','on');
```

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

```
opt = tfestOptions;
opt.InitializeMethod = 'n4sid';
opt.Display = 'on';
```

Compatibility Considerations

Renaming of Estimation and Analysis Options

The names of some estimation and analysis options were changed in R2018a. Prior names still work. For details, see the R2018a release note “Renaming of Estimation and Analysis Options”.

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`

Topics

“Loss Function and Model Quality Metrics”

Introduced in R2012b

timeoptions

Create list of time plot options

Description

Use the `timeoptions` command to create a `TimePlotOptions` object to customize time plot appearance. You can also use the command to override the plot preference settings in the MATLAB session in which you create the time plots.

Creation

Syntax

```
plotoptions = timeoptions
plotoptions = timeoptions('cstprefs')
```

Description

`plotoptions = timeoptions` returns a list of available options for time plots with default values set. You can use these options to customize the time plot appearance from the command line.

`plotoptions = timeoptions('cstprefs')` initializes the plot options with options you selected in the Control System Toolbox and System Identification Toolbox Preferences Editor. For more information about the editor, see "Toolbox Preferences Editor". 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.

Properties

`Normalize` — Toggle response normalization
'`off`' (default) | '`on`'

Toggle response normalization, specified as either '`on`' or '`off`'.

`SettleTimeThreshold` — Settling time threshold
`0.02` (default) | positive scalar

Settling time threshold, specified as a positive scalar between values `0` and `1`.

`RiseTimeLimits` — Rise time limits
`[0.1,0.9]` (default) | two-element vector of the form `[min,max]`

Rise time limits between the values of `0` and `1`, specified as a two-element vector of the form `[min,max]`.

`TimeUnits` — Time units
'`seconds`' (default)

Time units, specified as one of the following values:

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

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 to plot the confidence region
`1` (default) | scalar

Number of standard deviations to use to plot the confidence region, specified as a scalar. This is applicable to identified models only.

`IOGrouping` — Grouping of input-output pairs
'`none`' (default) | '`inputs`' | '`outputs`' | '`all`'

Grouping of input-output (I/O) pairs, specified as one of the following:

- '`none`' — No input-output grouping.
- '`inputs`' — Group only the inputs.
- '`outputs`' — Group only the outputs.
- '`all`' — Group all the I/O pairs.

`InputLabels` — Input label style
structure (default)

Input label style, specified as a structure with the following fields:

- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is dark grey with the RGB triplet `[0.4,0.4,0.4]`.
- `Interpreter` — Text interpreter, specified as one of these values:

  - 'tex' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - 'latex' — Interpret characters using LaTeX markup.
  - 'none' — Display literal characters.

`OutputLabels` — Output label style
structure (default)

Output label style, specified as a structure with the following fields:

- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is dark grey with the RGB triplet `[0.4,0.4,0.4]`.
- `Interpreter` — Text interpreter, specified as one of these values:
  - 'tex' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - 'latex' — Interpret characters using LaTeX markup.
  - 'none' — Display literal characters.

`InputVisible` — Toggle display of inputs
`{'on'}` (default) | `{'off'}`

Toggle display of inputs, specified as either `{'on'}` or `{'off'}`.

`OutputVisible` — Toggle display of outputs
`{'on'}` (default) | `{'off'}`

Toggle display of outputs, specified as either `{'on'}` or `{'off'}`.

`Title` — Title text and style
structure (default)

Title text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a character vector. By default, the plot is titled `'Time Response'`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:

  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`XLabel` — X-axis label text and style
structure (default)

X-axis label text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a character vector. By default, the axis is titled `'Time'`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:

  - '`tex`' — Interpret characters using a subset of TeX markup. This is the default value of `Interpreter`.
  - '`latex`' — Interpret characters using LaTeX markup.
  - '`none`' — Display literal characters.

`YLabel` — Y-axis label text and style
structure (default)

Y-axis label text and style, specified as a structure with the following fields:

- `String` — Label text, specified as a cell array of character vectors. By default, the axis is titled `'Amplitude'`.
- `FontSize` — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch.
- `FontWeight` — Character thickness, specified as '`Normal`' or '`bold`'. MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- `FontAngle` — Character slant, specified as '`Normal`' or '`italic`'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- `Color` — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet `[0,0,0]`.
- `Interpreter` — Text interpreter, specified as one of these values:

- 'tex' — Interpret characters using a subset of TeX markup. This is the default value of Interpreter.
- 'latex' — Interpret characters using LaTeX markup.
- 'none' — Display literal characters.

TickLabel — Tick label style
structure (default)

Tick label style, specified as a structure with the following fields:

- FontSize — Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch.
- FontWeight — Character thickness, specified as 'Normal' or 'bold'. MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.
- FontAngle — Character slant, specified as 'Normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.
- Color — Text color, specified as an RGB triplet. The default color is black specified by the RGB triplet [0,0,0].

Grid — Toggle grid display
'off' (default) | 'on'

Toggle grid display on the plot, specified as either 'off' or 'on'.

GridColor — Color of the grid lines
[0.15,0.15,0.15] (default) | RGB triplet

Color of the grid lines, specified as an RGB triplet. The default color is light grey specified by the RGB triplet [0.15,0.15,0.15].

XLimMode — X-axis limit selection mode
'auto' (default) | 'manual'

Selection mode for the x-axis limits, specified as one of these values:

- 'auto' — Enable automatic limit selection, which is based on the total span of the plotted data.
- 'manual' — Manually specify the axis limits. To specify the axis limits, set the XLim property.

YLimMode — Y-axis limit selection mode
'auto' (default) | 'manual'

Selection mode for the y-axis limits, specified as one of these values:

- 'auto' — Enable automatic limit selection, which is based on the total span of the plotted data.
- 'manual' — Manually specify the axis limits. To specify the axis limits, set the YLim property.

XLim — X-axis limits
'{[1,10]}' (default) | cell array of two-element vector of the form [min,max]

X-axis limits, specified as a cell array of two-element vector of the form [min,max].

YLim — Y-axis limits
'{[1,10]}' (default) | cell array of two-element vector of the form [min,max]

Y-axis limits, specified as a cell array of two-element vector of the form [min,max].

Object Functions

getoptions	Return plot options handle or plot options property
impulseplot	Plot impulse response with additional plot customization options
initialplot	Plot initial condition response with additional plot customization options
lsimplot	Plot simulated time response of dynamic system to arbitrary inputs with additional plot customization options
setoptions	Set plot options handle or plot options property
stepplot	Plot step response with additional plot customization options

Examples

Plot Normalized Step Response

Create a default time options set.

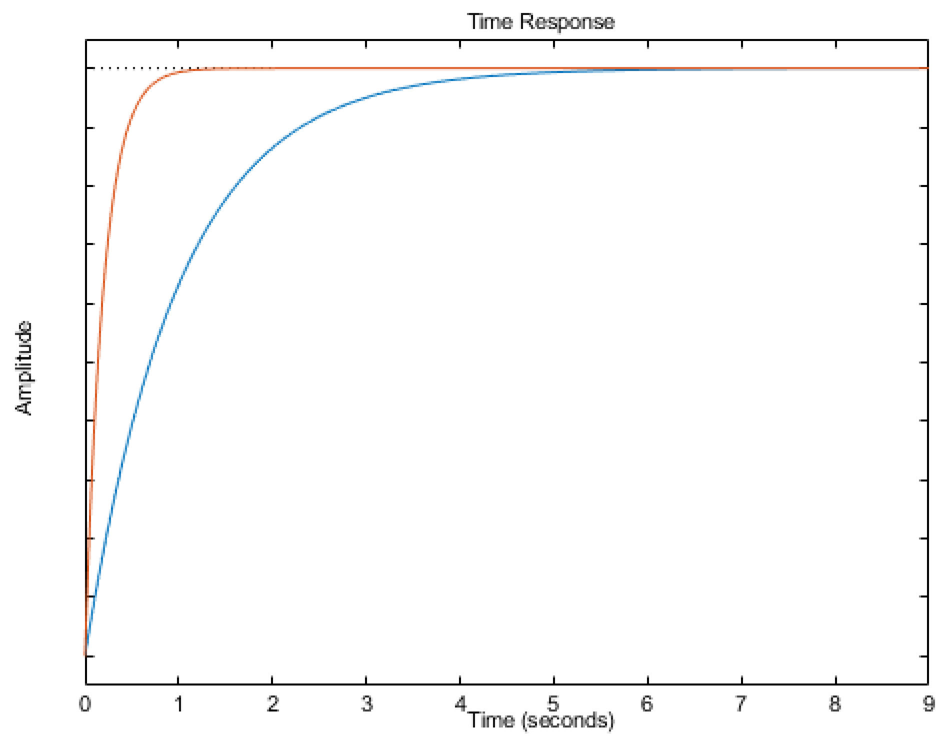
```
opt = timeoptions;
```

Enable plotting of normalized responses.

```
opt.Normalize = 'on';
```

Plot the step response of two transfer function models using the specified options.

```
sys1 = tf(10,[1,1]);
sys2 = tf(5,[1,5]);
stepplot(sys1,sys2,opt);
```



The plot shows the normalized step response for the two transfer function models.

Customize Step Plot using Plot Handle

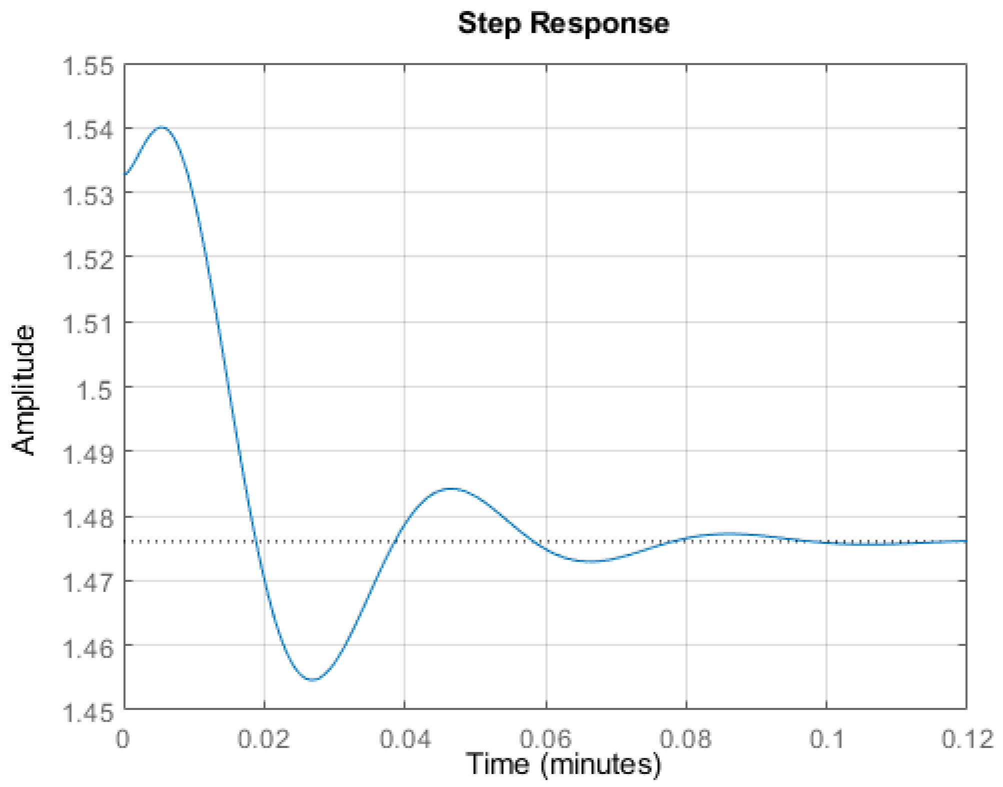
For this example, use the plot handle to change the time units to minutes and turn on the grid.

Generate a random state-space model with 5 states and create the step response plot with plot handle `h`.

```
rng("default")
sys = rss(5);
h = stepplot(sys);
```

Change the time units to minutes and turn on the grid. To do so, edit properties of the plot handle, `h` using `setoptions`.

```
setoptions(h,'TimeUnits','minutes','Grid','on');
```



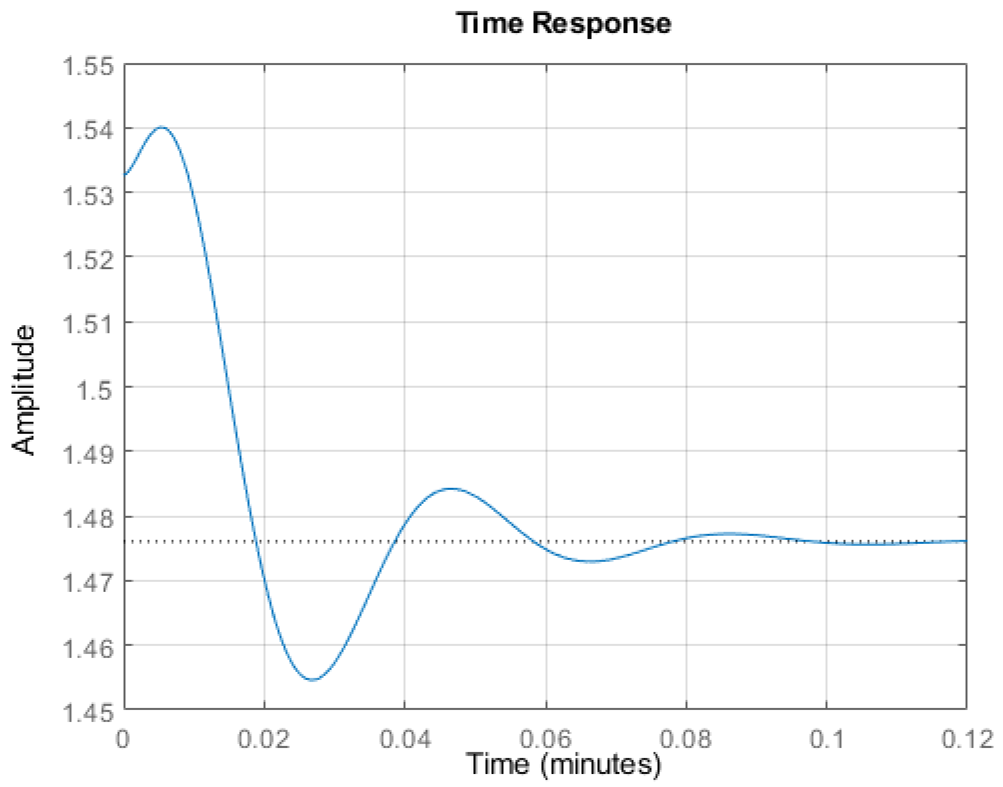
The step plot automatically updates when you call `setoptions`.

Alternatively, you can also use the `timeoptions` command to specify the required plot options. First, create an options set based on the toolbox preferences.

```
plotoptions = timeoptions('cstprefs');
```

Change properties of the options set by setting the time units to minutes and enabling the grid.

```
plotoptions.TimeUnits = 'minutes';
plotoptions.Grid = 'on';
stepplot(sys,plotoptions);
```



You can use the same option set to create multiple step plots with the same customization. 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 `TimeUnits` and `Grid`, override the toolbox preferences.

Customized Step Response Plot at Specified Time

For this example, examine the step response of the following zero-pole-gain model and limit the step plot to `tFinal` = 15 s. Use 15-point blue text for the title. This plot should look the same, regardless of the preferences of the MATLAB session in which it is generated.

```
sys = zpk(-1,[-0.2+3j,-0.2-3j],1)*tf([1 1],[1 0.05]);
tFinal = 15;
```

First, create a default options set using `timeoptions`.

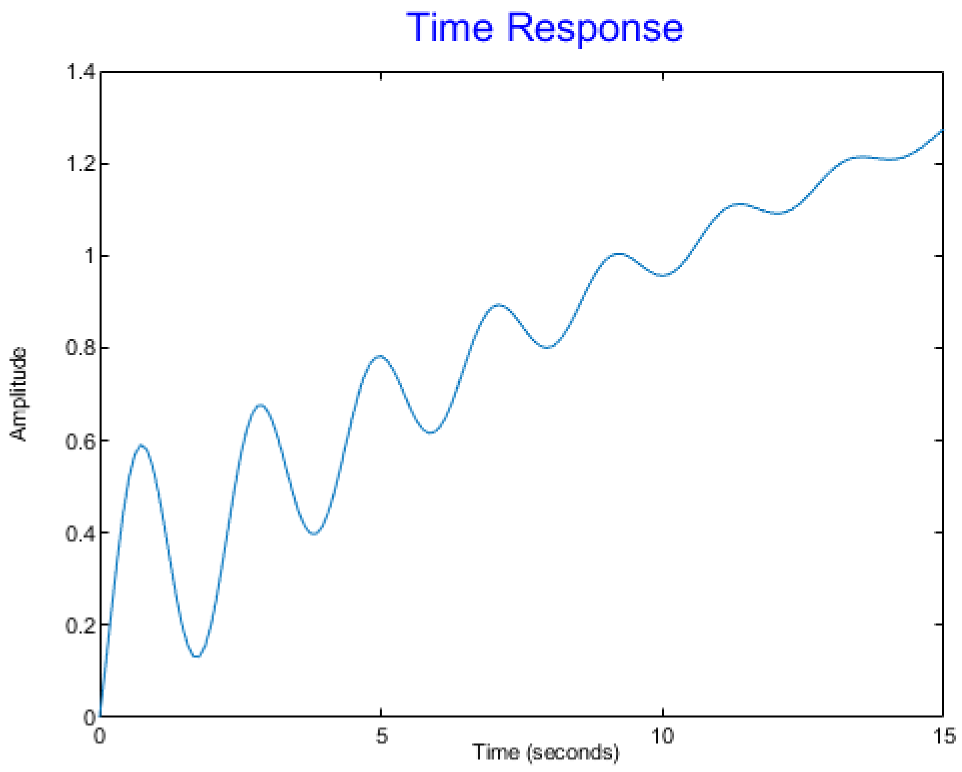
```
plotoptions = timeoptions;
```

Next change the required properties of the options set `plotoptions`.

```
plotoptions.Title.FontSize = 15;
plotoptions.Title.Color = [0 0 1];
```

Now, create the step response plot using the options set `plotoptions`.

```
h = stepplot(sys,tFinal,plotoptions);
```



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

Custom Plot of System Evolution from Initial Condition

By default, `lsimplot` simulates the model assuming all states are zero at the start of the simulation. When simulating the response of a state-space model, use the optional `x0` input argument to specify nonzero initial state values. Consider the following two-state SISO state-space model.

```
A = [-1.5 -3;
      3   -1];
B = [1.3; 0];
C = [1.15 2.3];
D = 0;
sys = ss(A,B,C,D);
```

Suppose that you want to allow the system to evolve from a known set of initial states with no input for 2 s, and then apply a unit step change. Specify the vector `x0` of initial state values, and create the input vector.

```
x0 = [-0.2 0.3];
t = 0:0.05:8;
```

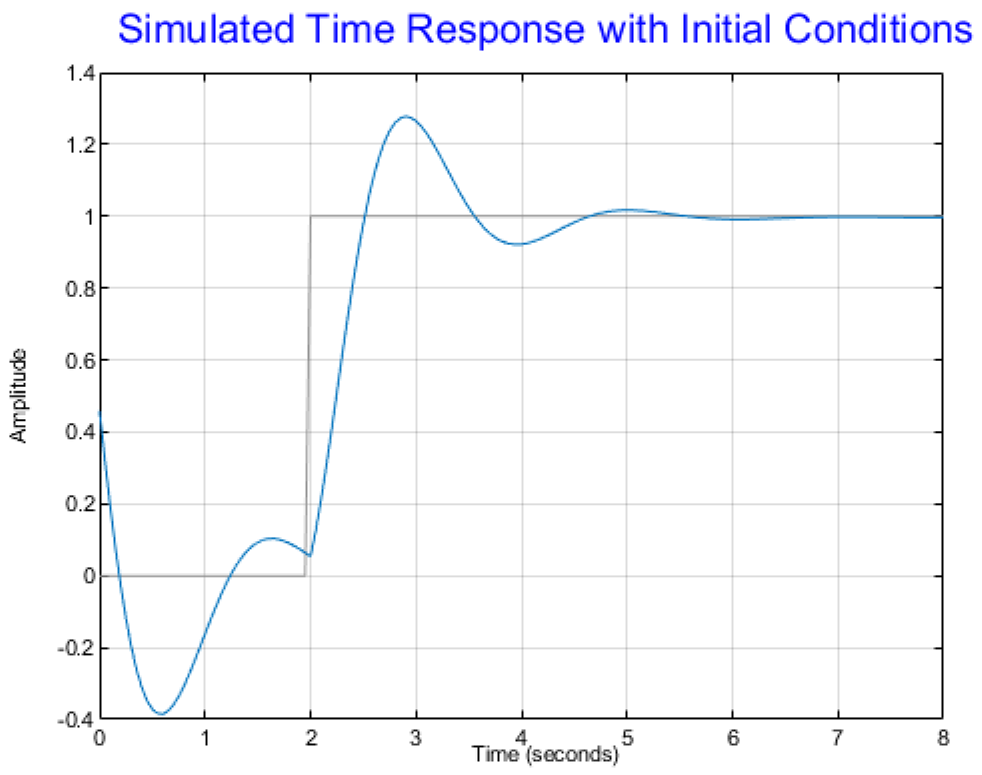
```
u = zeros(length(t),1);
u(t>=2) = 1;
```

First, create a default options set using `timeoptions`.

```
plotoptions = timeoptions;
```

Next change the required properties of the options set `plotoptions` and plot the simulated response with the zero order hold option.

```
plotoptions.Title.FontSize = 15;
plotoptions.Title.Color = [0 0 1];
plotoptions.Grid = 'on';
h = lsimplot(sys,u,t,x0,plotoptions,'zoh');
hold on
title('Simulated Time Response with Initial Conditions')
```



The first half of the plot shows the free evolution of the system from the initial state values `[-0.2 0.3]`. At `t = 2` there is a step change to the input, and the plot shows the system response to this new signal beginning from the state values at that time. Because `plotoptions` begins with a fixed set of options, the plot result is independent of the toolbox preferences of the MATLAB session.

Customized Plot of Simulated Response to Arbitrary Input Signal

For this example, change time units to minutes and turn the grid on for the simulated response plot. Consider the following transfer function.

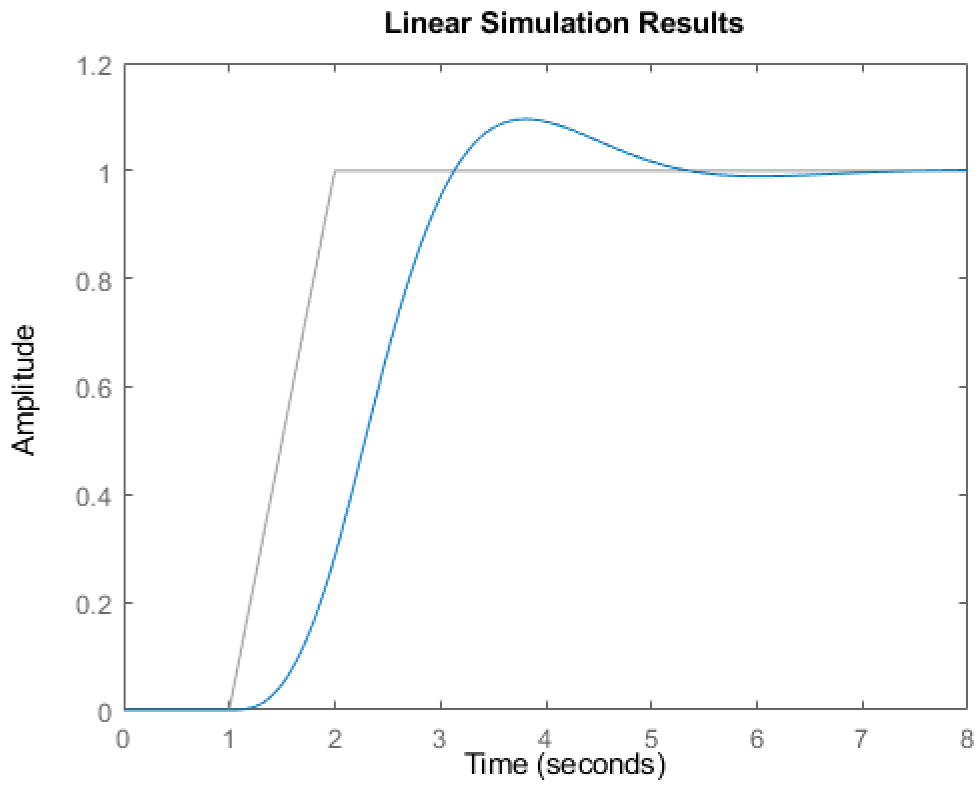
```
sys = tf(3,[1 2 3]);
```

To compute the response of this system to an arbitrary input signal, provide `lsimplot` with a vector of the times `t` at which you want to compute the response and a vector `u` containing the corresponding signal values. For instance, plot the system response to a ramping step signal that starts at 0 at time `t = 0`, ramps from 0 at `t = 1` to 1 at `t = 2`, and then holds steady at 1. Define `t` and compute the values of `u`.

```
t = 0:0.04:8;
u = max(0,min(t-1,1));
```

Use `lsimplot` plot the system response to the signal with a plot handle `h`.

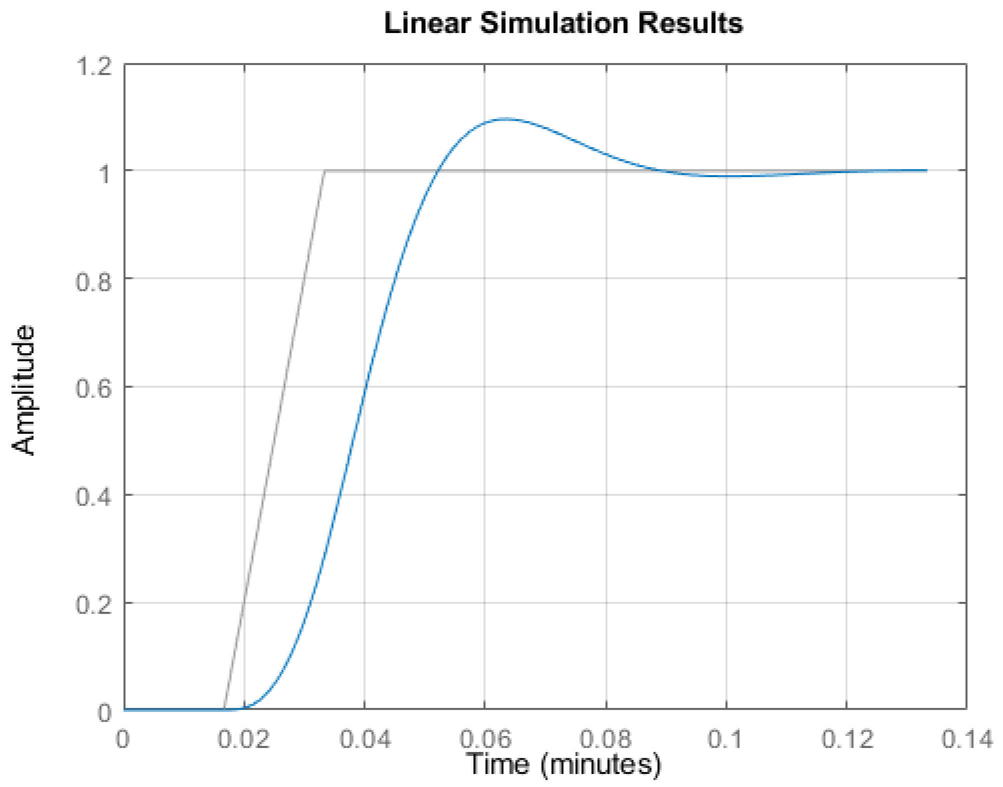
```
h = lsimplot(sys,u,t);
```



The plot shows the applied input `(u,t)` in gray and the system response in blue.

Use the plot handle to change the time units to minutes and to turn the grid on. To do so, edit properties of the plot handle, `h` using `setoptions`.

```
setoptions(h,'TimeUnits','minutes','Grid','on')
```



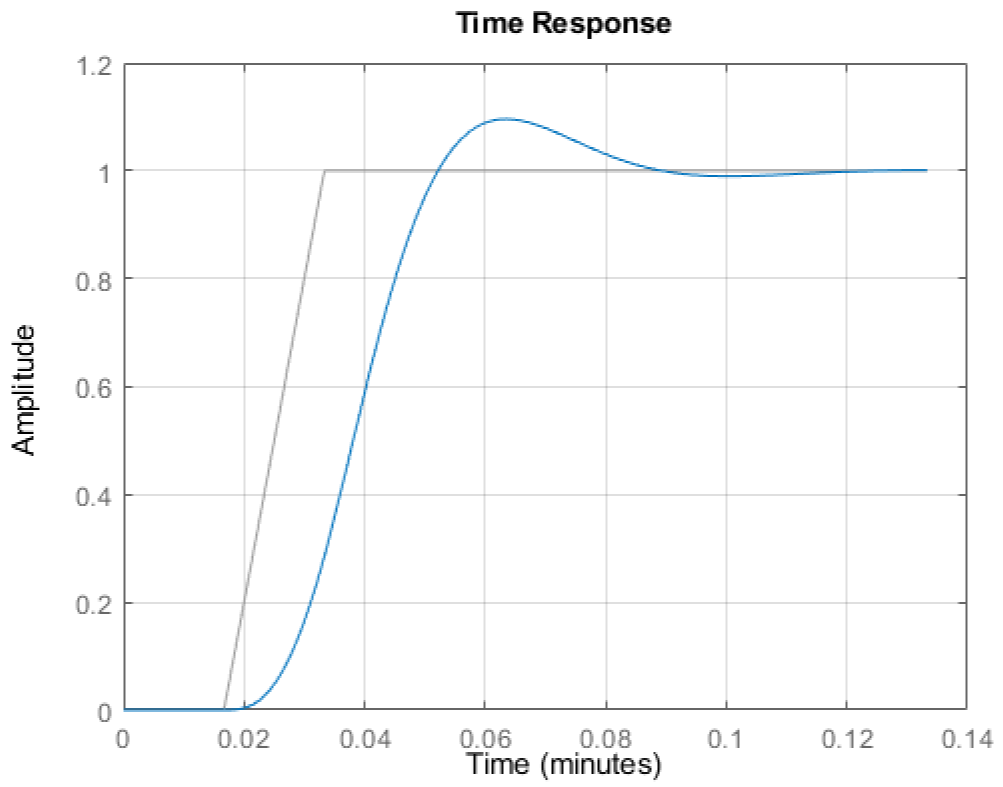
The plot automatically updates when you call `setoptions`.

Alternatively, you can also use the `timeoptions` command to specify the required plot options. First, create an options set based on the toolbox preferences.

```
plotoptions = timeoptions('cstprefs');
```

Change properties of the options set by setting the time units to minutes and enabling the grid.

```
plotoptions.TimeUnits = 'minutes';
plotoptions.Grid = 'on';
lsimplot(sys,u,t,plotoptions);
```



See Also

`getoptions` | `impulseplot` | `initialplot` | `lsimplot` | `setoptions` | `stepplot`

Topics

"Toolbox Preferences Editor"

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

Compute Combined Input-Output Delay for Transfer Function

Create the transfer function model, 1/*s*.

```
sys = tf(1,[1 0]);
```

Specify a 2 second input delay, and a 1.5 second output delay.

```
sys.InputDelay = 2;
sys.Outputdelay = 1.5;
```

Compute the combined input-output delay for `sys`.

```
td = totaldelay(sys)
```

```
td = 3.5000
```

The resulting transfer function has the following form:

$$e^{-2s}x\frac{1}{s}e^{-1.5s} = e^{-3.5s}\frac{1}{s}$$

This result is equivalent to specifying an input-output delay of 3.5 seconds for the original transfer function, 1/*s*.

See Also

`absorbDelay` | `hasdelay`

Introduced in R2012a

translatecov

Translate parameter covariance across model transformation operations

Syntax

```
sys_new = translatecov(fcn,sys)
sys_new = translatecov(fcn,Input1,...,InputN)
```

Description

`sys_new = translatecov(fcn,sys)` transforms `sys` into `sys_new = fcn(sys)`, and translates the parameter covariance of `sys` to the parameter covariance of the transformed model. `fcn` is a transformation function that you specify. The command computes the parameter covariance of `sys_new` by applying the Gauss Approximation formula. To view the translated parameter covariance, use `getcov`.

Applying model transformations directly does not always translate the parameter covariance of the original model to that of the transformed model. For example, `d2c(sys)` does not translate the parameter covariance of `sys`. In contrast, `translatecov(@(x)d2c(x),sys)` produces a transformed model that has the same coefficients as `d2c(sys)` and has the translated parameter covariance of `sys`.

`sys_new = translatecov(fcn,Input1,...,InputN)` returns the model `sys_new = fcn(Input1,...,InputN)` and its parameter covariance. At least one of the `N` inputs must be a linear model with parameter covariance information.

Input Arguments

`fcn`

Model transformation function, specified as a function handle.

For single-input functions, `sys_new = fcn(sys)`. The input to `fcn` must be an identified model with parameter covariance information. Typical single-input transformation operations include:

- Model-type conversion, such as `idpoly` and `idss`. For example, `fcn = @(x)idpoly(x)` converts the model `x` to a polynomial model.
- Sample time conversion, such as `c2d` and `d2c`. For example, `fcn = @(x)c2d(x,Ts)` converts the continuous-time model `x` to discrete-time using the specified sample time `Ts`.

For multi-input functions, `sys_new = fcn(Input1,..InputN)`. At least one of the input arguments `Input1,...,InputN` must be an identified model with parameter covariance information. Typical multi-input operations include merging and concatenation of multiple models. For example, `fcn = @(x,y)[x,y]` performs horizontal concatenation of the models `x` and `y`.

`sys`

Linear model with parameter covariance information, specified as one of the following model types:

- `idtf`
- `idproc`
- `idss`
- `idpoly`
- `idgrey`

The model must contain parameter covariance information, that is `getcov(sys)` must be nonempty.

`Input1,...,InputN`

Multiple input arguments to the translation function `fcn`. At least one of the `N` inputs must be a linear identified model with parameter covariance information. The other inputs can be any MATLAB data type. For an example, see “Translate Parameter Covariance to Closed-Loop Model” on page 1-1769.

Output Arguments

`sys_new`

Model resulting from transformation operation. The model includes parameter covariance information.

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)
```

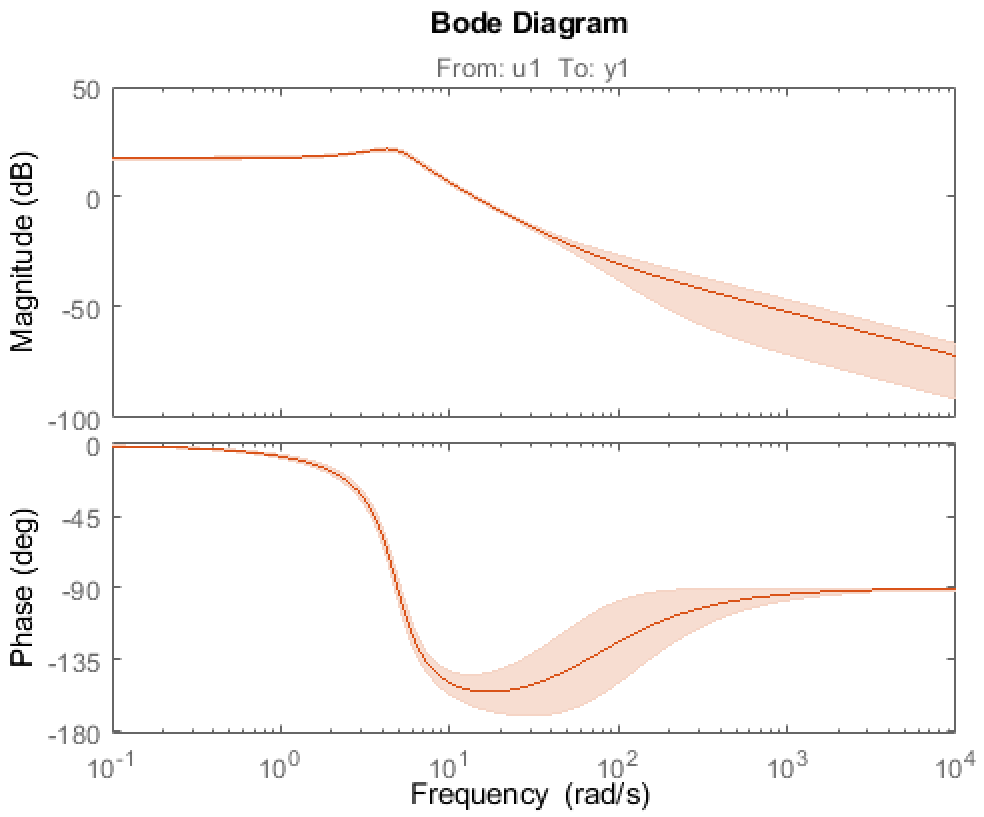
```
ans =

     []
```

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: [36x36 double]
       T: [24x36 double]
    Free: [90x1 logical]
```

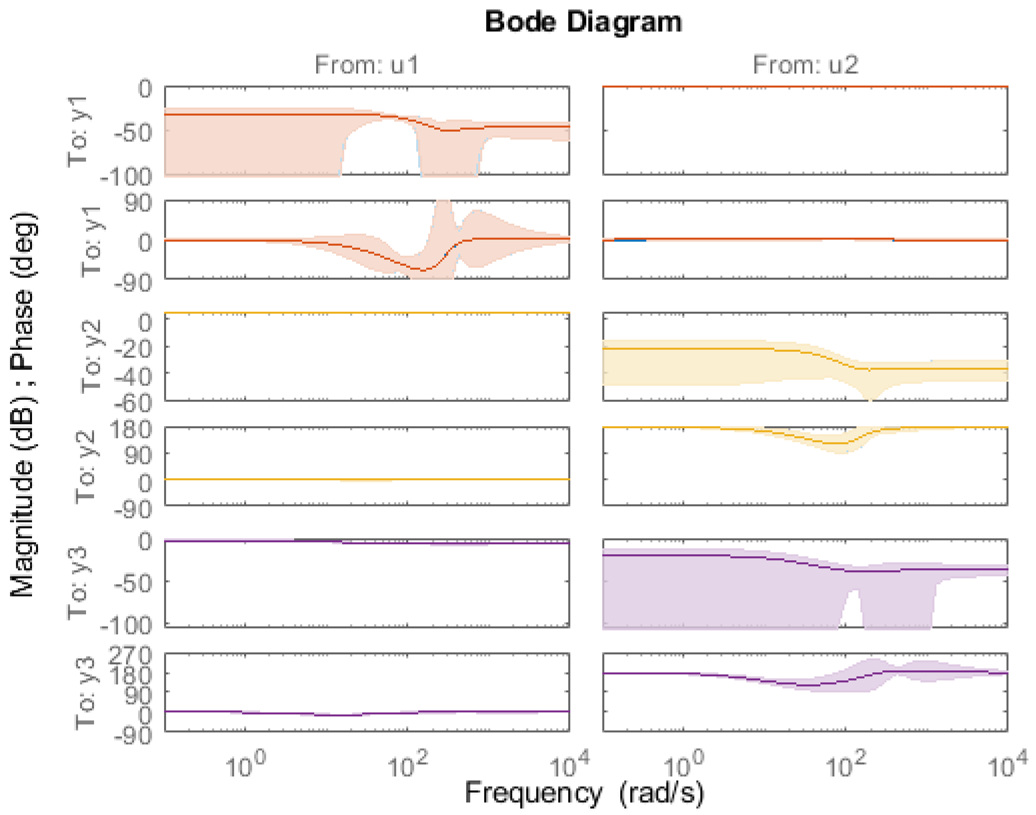
If you combine the estimated models into one 3-output model directly, the covariance information is lost (the output of `getcov` is empty).

```
M1 = [m1;m2;m3];
getcov(M1)
```

```
ans =

     []
```

Compare the confidence bounds.

```
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.

Translate Parameter Covariance to Closed-Loop Model

Consider a closed-loop feedback model consisting of a plant and controller. Translate the parameter covariance of the plant to the closed-loop feedback model.

Estimate a plant as a fourth-order state-space model using estimation data `z1`.

```
load iddata1 z1
Plant = ssest(z1,4);
```

`Plant` contains parameter covariance information.

Create a controller as a continuous-time zero-pole-gain model with zeros, poles, and gain equal to -2, -10, 5, respectively.

```
Controller = zpk(-2,-10,5);
```

Define a transformation function to generate the closed-loop feedback state-space model.

```
fcn = @(x,y)idss(feedback(x,y));
```

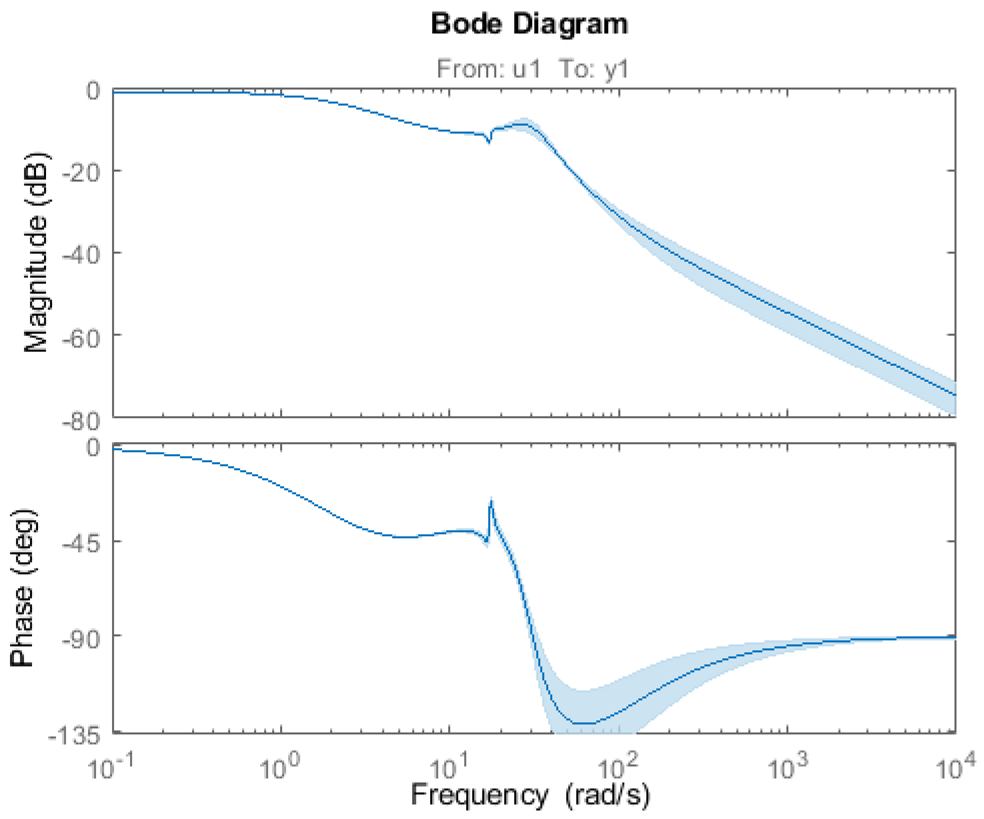
Translate the parameter covariance of the plant to the closed-loop feedback model.

```
sys_new = translatecov(fcn,Plant,Controller);
```

`sys_new` contains the translated parameter covariance information.

Plot the frequency-response of the transformed model `sys_new`, and view the confidence region of the response.

```
h = bodeplot(sys_new);
showConfidence(h);
```



The plot shows the effect of the uncertainty in `Plant` on the closed-loop response.

Tips

- 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.SearchOptions.MaxIterations = 0;
  m_new = ssest(z1,m2,opt)
  ```

  You cannot run a zero-iteration update in the following cases:

- If `MaxIterations` 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.

See Also

`getcov` | `setcov` | `getpvec` | `rsample`

Topics
"What Is Model Covariance?"
"Types of Model Uncertainty Information"

Introduced in R2012b

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.

Property Name	Default	Description
`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

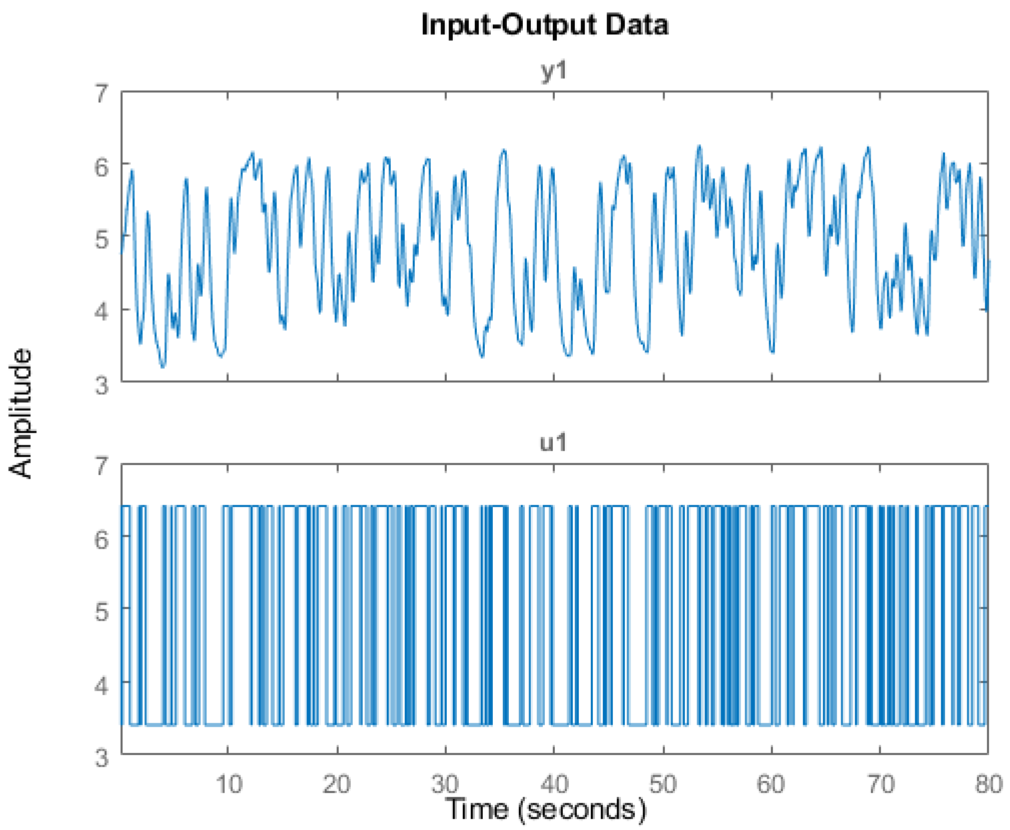
Remove a specified offset from input and output signals.

Load SISO data containing vectors `u2` and `y2`.

```
load dryer2
```

Create a data object with a 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.

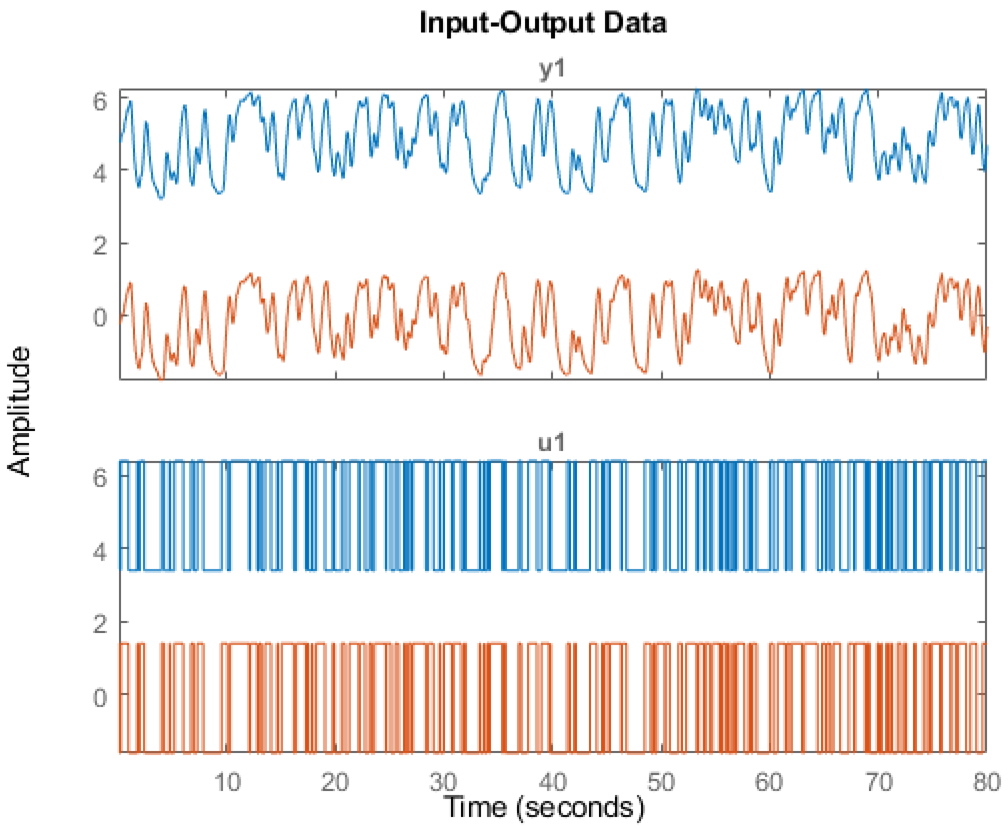
```
T.InputOffset = 5;
T.OutputOffset = 5;
```

Subtract the offsets 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
```

See Also

`detrend` | `getTrend` | `retrend`

Topics

“Handling Offsets and Trends in Data”

Introduced in R2009a

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(StateTransitionFcn,MeasurementFcn)
obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,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(StateTransitionFcn,MeasurementFcn)` creates an unscented Kalman filter object using the specified state transition and measurement functions. Before using the `predict` and `correct` commands, specify the initial state values using dot notation. For example, for a two-state system with initial state values `[1;0]`, specify `obj.State = [1;0]`.

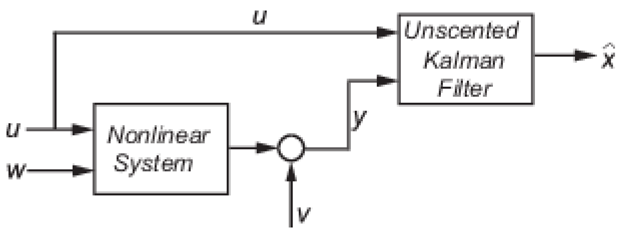
`obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,Name,Value)` specifies additional attributes of the unscented Kalman filter object using one or more `Name,Value` pair arguments. Before using the `predict` and `correct` commands, specify the initial state values using `Name,Value` pair arguments or dot notation.

`obj = unscentedKalmanFilter(Name,Value)` creates an unscented Kalman filter object with properties specified using one or more `Name,Value` pair arguments. Before using the `predict` and `correct` commands, specify the state transition function, measurement function, and initial state values using `Name,Value` pair arguments or dot notation.

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 $\mathtt{u}_s$ and $\mathtt{u}_m$ 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.
`residual`	Return the difference between the actual and predicted measurements.
`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 1-1783.

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.MeasurementFcn = @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 following 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)
```

Input Arguments

`StateTransitionFcn` — State transition function

function handle

State transition function *f*, specified as a function handle. The function calculates the *Ns*-element state vector of the system at time step *k*, given the state vector at time step *k*-1. *Ns* 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 *Ns*-element vector, where *Ns* 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 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 on page 1-1783 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,'ProcessNoise',0.01);
```

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
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 *Ns*-by-*Ns* matrix, where *Ns* 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 *Ns*-by-*Ns* 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 *Ns*, where *Ns* 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 *Ns*-by-*Ns* matrix, where *Ns* is the number of states of the system. If you specify a scalar, the software uses the scalar value to create an *Ns*-by-*Ns* 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
function handle

State transition function *f*, specified as a function handle. The function calculates the *Ns*-element state vector of the system at time step *k*, given the state vector at time step *k*-1. *Ns* 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 on page 1-1783. 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.

Compatibility Considerations

Numerical Changes
Behavior changed in R2020b

Starting in R2020b, numerical improvements in the `unscentedKalmanFilter` algorithm might produce results that are different from the results you obtained in previous versions.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

For more information, see “Generate Code for Online State Estimation in MATLAB”.

Generated code uses an algorithm that is different from the algorithm that the `unscentedKalmanFilter` function uses. You might see some numerical differences in the results obtained using the two methods.

Supports MATLAB Function block: No

See Also

Functions
`predict` | `correct` | `clone` | `extendedKalmanFilter` | `residual`

Blocks
Kalman Filter | Extended Kalman Filter | Unscented Kalman Filter

Topics
"Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter"
"Generate Code for Online State Estimation in MATLAB"
"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"

External Websites
Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

Introduced in R2016b

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

Alphabetically Order States of State-Space Model

Load a previously saved state space model `ssF8` with four states.

```
load ltiexamples
ssF8
```

```
ssF8 =

  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 state-space model.
```

Order the states in alphabetical order.

```
[y,P] = sort(ssF8.StateName);
sys = xperm(ssF8,P)

sys =

  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 state-space 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(sys,J1,...,JN)
```

Description

`Z = zero(sys)` returns the zeros of the single-input, single-output (SISO) dynamic system model, `sys`. The output is expressed as the reciprocal of the time units specified in `sys.TimeUnit`.

`[Z,gain] = zero(sys)` also returns the zero-pole-gain of `sys`.

`[Z,gain] = zero(sys,J1,...,JN)` returns the zeros and gain of the entries in the model array `sys` with subscripts `J1,...,JN`.

Examples

Zeros of Transfer Function

Compute the zeros of the following transfer function:

$$sys(s) = \frac{4.2s^2 + 0.25s - 0.004}{s^2 + 9.6s + 17}$$

```
sys = tf([4.2,0.25,-0.004],[1,9.6,17]);
Z = zero(sys)
```

```
Z = 2×1

   -0.0726
    0.0131
```

Zeros and Gain of Transfer Function

Calculate the zero locations and zero-pole gain of the following transfer function:

$$sys(s) = \frac{4.2s^2 + 0.25s - 0.004}{s^2 + 9.6s + 17}$$

```
sys = tf([4.2,0.25,-0.004],[1,9.6,17]);
[z,gain] = zero(sys)
```

```
z = 2×1

   -0.0726
    0.0131
```

```
gain = 4.2000
```

The zero locations are expressed in second^{-1}, because the time unit of the transfer function (`H.TimeUnit`) is seconds.

Zeros and Gain of Models in an Array

For this example, load a 3-by-1 array of transfer function models.

```
load('tfArray.mat','sys');
size(sys)
```

```
3x1 array of transfer functions.
Each model has 1 outputs and 1 inputs.
```

Find the zeros and gain values of the models in the array.

```
[Z, gain] = zero(sys);
Z(:,:,1,1)
```

```
ans =

  0x1 empty double column vector
```

```
gain(:,:,1,1)
```

```
ans = 1
```

`zero` returns an array each for the zeros and the gain values respectively. Here, `Z(:,:,1,1)` and `gain(:,:,1,1)` corresponds to the zero and the gain value of the first model in the array, that is, `sys(:,:,1,1)`.

Input Arguments

sys — Dynamic system
dynamic system model | model array

Dynamic system, specified as a SISO dynamic system model, or an array of SISO dynamic system models. Dynamic systems that you can use include continuous-time or discrete-time numeric LTI models such as `tf`, `zpk`, or `ss` models.

If `sys` is a generalized state-space model `genss` or an uncertain state-space model `uss`, `zero` returns the zeros of the current or nominal value of `sys`. If `sys` is an array of models, `zero` returns the zeros of the model corresponding to its subscript `J1,...,JN` in `sys`. For more information on model arrays, see “Model Arrays” (Control System Toolbox).

`J1,...,JN` — Indices of models in array whose zeros you want to extract
positive integer

Indices of models in array whose zeros you want to extract, specified as a positive integer. You can provide as many indices as there are array dimensions in `sys`. For example, if `sys` is a 4-by-5 array of dynamic system models, the following command extracts the zeros for entry (2,3) in the array.

```
Z = zero(sys,2,3);
```

Output Arguments

`Z` — Zeros of the dynamic system
column vector | array

Zeros of the dynamic system, returned as a column vector or an array. If `sys` is:

- A single model, then `Z` is a column vector of zeros of the dynamic system model `sys`
- A model array, then `Z` is an array containing the zeros of each model in `sys`

`Z` is expressed as the reciprocal of the time units specified in `sys.TimeUnit`. For example, zero is expressed in 1/minute if `sys.TimeUnit = 'minutes'`.

`gain` — Zero-pole-gain of the dynamic system
scalar

Zero-pole-gain of the dynamic system, returned as a scalar. In other words, `gain` is the value of `K` when the model is written in `zpk` form.

Tips

- 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. To assess the stability of models with internal delays, use `step` or `impulse`.
- To calculate the transmission zeros of a multi-input, multi-output (MIMO) system, use `tzero`.

See Also
`pole` | `pzmap` | `tzero` | `step` | `impulse` | `pzplot`

Topics
"Pole and Zero Locations" (Control System Toolbox)

Introduced in R2012a

zgrid

Generate z-plane grid of constant damping factors and natural frequencies

Syntax

```
zgrid
zgrid(T)
zgrid(zeta,wn)
zgrid(zeta,wn,T)

zgrid( ___ ,'new')
zgrid(AX, ___ )
```

Description

`zgrid` generates a grid of constant damping factors from 0 to 1 in steps of 0.1 and natural frequencies from 0 to `π/T` in steps of `0.1*π/T` for root locus and pole-zero maps. The default steps of `0.1*π/T` represent fractions of the Nyquist frequencies. `zgrid` then plots the grid over the current axis. `zgrid` creates the grid over the plot without altering the current axis limits if the current axis contains a discrete z-plane root locus diagram or pole-zero map. Use this syntax to plot multiple systems with different sample times.

Alternatively, you can select **Grid** from the context menu in the plot window to generate the same z-plane grid.

`zgrid(T)` generates the z-plane grid by using default values for damping factor and natural frequency relative to the sample time `T`.

`zgrid(zeta,wn)` plots a grid of constant damping factor and natural frequency lines for the damping factors and normalized natural frequencies in the vectors `zeta` and `wn`, respectively. When the sample time is not specified, the frequency values in `wn` are interpreted as normalized values, that is, `wn/T`.

`zgrid(zeta,wn,T)` plots a grid of constant damping factor and natural frequency lines for the damping factors and natural frequencies in the vectors `zeta` and `wn`, relative to sample time `T`. `zeta` lines are independent for `T` but the `wn` lines depend on the sample time value. Use this syntax to create the z-plane grid with specific values of `wn`.

`zgrid( ___ ,'new')` clears the current axes first and sets `hold on`.

`zgrid(AX, ___ )` plots the z-plane grid on the `Axes` or `UIAxes` object in the current figure with the handle `AX`. Use this syntax when creating apps with `zgrid` in the App Designer.

Examples

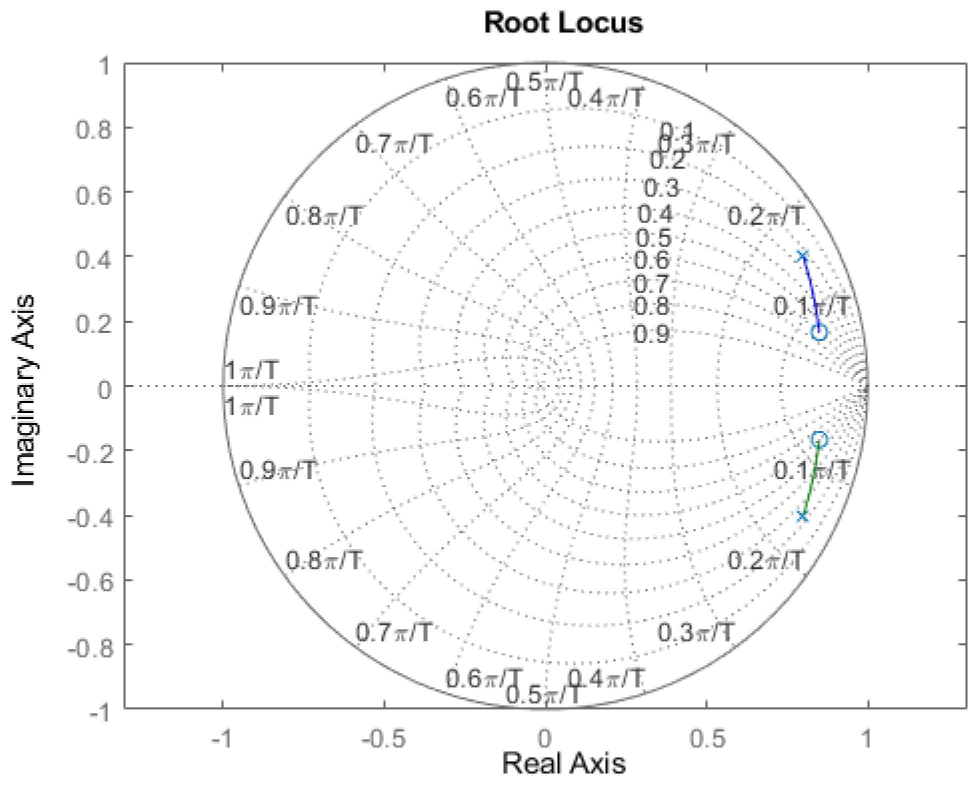
Plot z-plane Grid Lines on the Root Locus

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)
```

```
H =
 
  2 z^2 - 3.4 z + 1.5
  -------------------
   z^2 - 1.6 z + 0.8
 
Sample time: unspecified
Discrete-time transfer function.
```

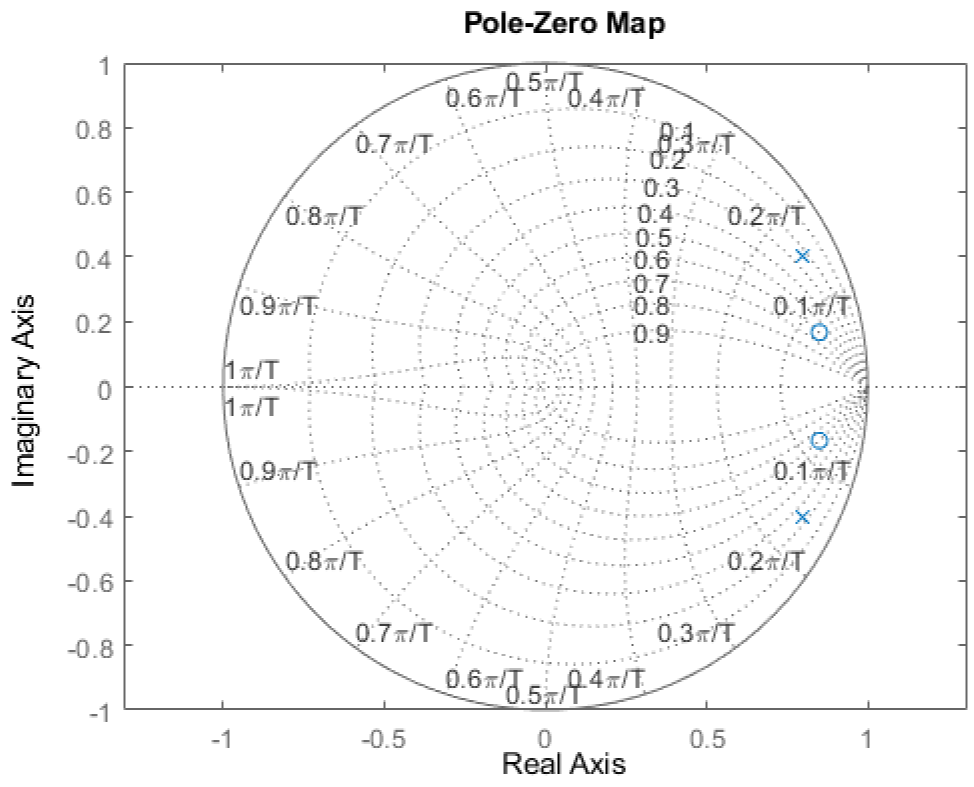
```
rlocus(H)
zgrid
axis equal
```



Normalized and True z-plane Grid Lines on the Pole-Zero Map

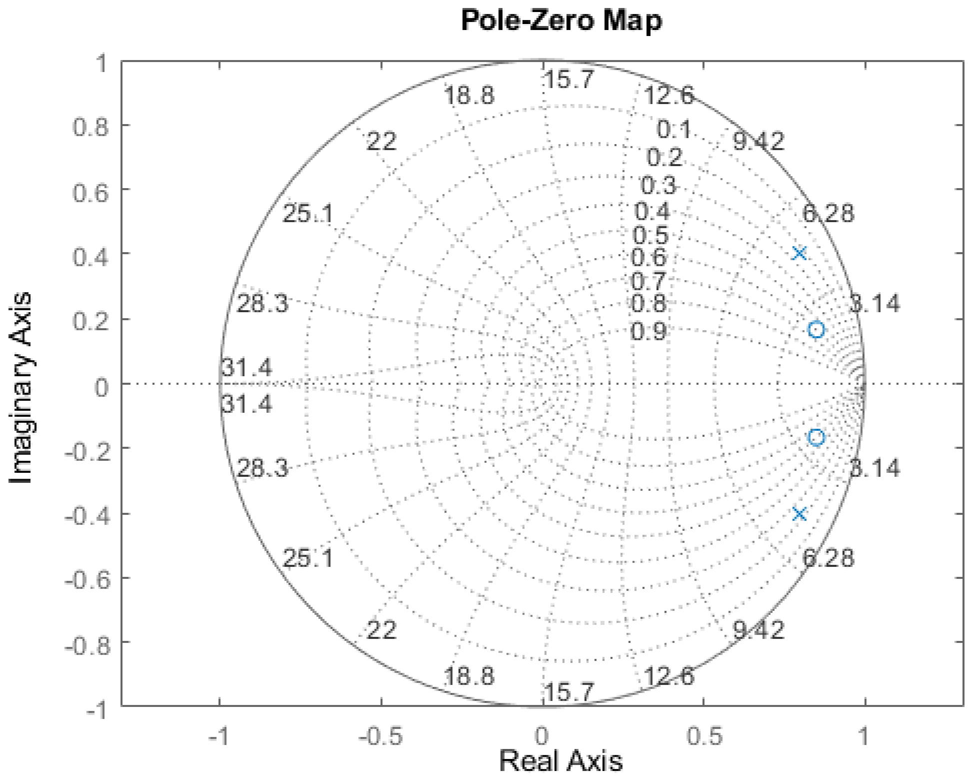
For this example, consider a discrete-time transfer function `sys` with a sample time of 0.1s. Now plot the pole-zero map of `sys` and visualize the default z-plane grid without specifying the sample time.

```
sys = tf([2 -3.4 1.5],[1 -1.6 0.8],0.1);
Ts = 0.1;
figure()
pzmap(sys)
zgrid()
axis equal
```



Observe that the frequencies on the z-plane grid are normalized in terms of $f\frac{\Pi}{T}$. To obtain the true frequency values on the grid, specify the sample time with the `zgrid` command.

```
figure()
pzmap(sys)
zgrid(Ts)
axis equal
```



Now, observe that the frequency values on the plot are true values, that is, they are non-normalized.

Input Arguments

`zeta` — Damping ratio
vector

Damping ratio, specified as a vector in the same order as `wn`.

`wn` — Natural frequency values
vector

Natural frequency values, specified as a vector. Natural frequencies are plotted as true values when `T` is specified. When the sample time is not specified, `zgrid` normalizes the values as `wn/T`.

`T` — Sample time
positive scalar | `-1`

Sample time, specified as:

- A positive scalar representing the sampling period of a discrete-time system. The actual frequency values are displayed on the frequency grid.
- `-1` for a discrete-time system with an unspecified sample time. The frequency values are displayed as normalized values `f*π/T` for the default grid.

`zeta` lines are independent of `T` while `wn` lines are dependent on the sample time. You must specify `T` to plot specific values of `wn`. When the sample time `T` is not specified, the required `wn` values are interpreted as normalized values, that is, `wn/T`.

`AX` — Object handle
`Axes` object | `UIAxes` object

Object handle, specified as an `Axes` or `UIAxes` object. Use `AX` to create apps with `zgrid` in the App Designer.

See Also
`pzmap` | `rlocus` | `sgrid`

Introduced before R2006a

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

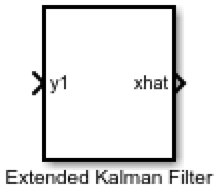
2

Blocks

Extended Kalman Filter

Estimate states of discrete-time nonlinear system using extended Kalman filter

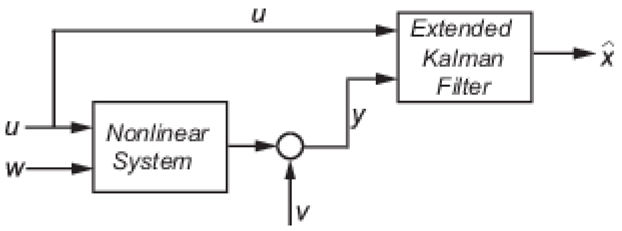
Library: Control System Toolbox / State Estimation
System Identification Toolbox / Estimators



Description

The Extended Kalman Filter block estimates the states 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.



Using the state transition and measurement functions of the system and the extended Kalman filter algorithm, the block produces state estimates $\hat{x}$ for the current time step. For information about the algorithm, see "Extended and Unscented Kalman Filter Algorithms for Online State Estimation".

You create the nonlinear state transition function and measurement functions for the system and specify these functions in the block. The block supports state estimation of a system with multiple sensors that are operating at different sampling rates. You can specify up to five measurement functions, each corresponding to a sensor in the system. 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. For more information, see "State Transition and Measurement Functions" on page 2-14.

Ports

Input

`y1,y2,y3,y4,y5` — Measured system outputs

vector

Measured system outputs corresponding to each measurement function that you specify in the block. The number of ports equals the number of measurement functions in your system. You can specify up to five measurement functions. For example, if your system has two sensors, you specify two measurement functions in the block. The first port **y1** is available by default. When you click **Apply**, the software generates port **y2** corresponding to the second measurement function.

Specify the ports as *N*-dimensional vectors, where *N* is the number of quantities measured by the corresponding sensor. For example, if your system has one sensor that measures the position and velocity of an object, then there is only one port **y1**. The port is specified as a 2-dimensional vector with values corresponding to position and velocity.

Dependencies

The first port **y1** is available by default. Ports **y2** to **y5** are generated when you click **Add Measurement**, and click **Apply**.

Data Types: `single` | `double`

`StateTransitionFcnInputs` — Additional optional input argument to state transition function

scalar | vector | matrix

Additional optional input argument to the state transition function `f` other than the state `x` and process noise `w`. For information about state transition functions see, “State Transition and Measurement Functions” on page 2-14.

Suppose that your system has nonadditive process noise, and the state transition function `f` has the following form:

```
x(k+1) = f(x(k),w(k),StateTransitionFcnInputs)
```

Here `k` is the time step, and `StateTransitionFcnInputs` is an additional input argument other than `x` and `w`.

If you create `f` using a MATLAB function (`.m` file), the software generates the port **StateTransitionFcnInputs** when you click **Apply**. You can specify the inputs to this port as a scalar, vector, or matrix.

If your state transition function has more than one additional input, use a Simulink Function block to specify the function. When you use a Simulink Function block, you provide the additional inputs directly to the Simulink Function block using Inport blocks. No input ports are generated for the additional inputs in the Extended Kalman Filter block.

Dependencies

This port is generated only if both of the following conditions are satisfied:

- You specify `f` in **Function** using a MATLAB function, and `f` is on the MATLAB path.
- `f` requires only one additional input argument apart from `x` and `w`.

Data Types: `single` | `double`

`MeasurementFcn1Inputs,MeasurementFcn2Inputs,MeasurementFcn3Inputs,MeasurementFcn4Inputs,MeasurementFcn5Inputs` — Additional optional input argument to each measurement function

scalar | vector | matrix

Additional optional inputs to the measurement functions other than the state `x` and measurement noise `v`. For information about measurement functions see, “State Transition and Measurement Functions” on page 2-14.

MeasurementFcn1Inputs corresponds to the first measurement function that you specify, and so on. For example, suppose that your system has three sensors and nonadditive measurement noise, and the three measurement functions `h1`, `h2`, and `h3` have the following form:

```
y1[k] = h1(x[k],v1[k],MeasurementFcn1Inputs)

y2[k] = h2(x[k],v2[k],MeasurementFcn2Inputs)

y3[k] = h3(x[k],v3[k])
```

Here `k` is the time step, and `MeasurementFcn1Inputs` and `MeasurementFcn2Inputs` are the additional input arguments to `h1` and `h2`.

If you specify `h1`, `h2`, and `h3` using MATLAB functions (`.m` files) in **Function**, the software generates ports **MeasurementFcn1Inputs** and **MeasurementFcn2Inputs** when you click **Apply**. You can specify the inputs to these ports as scalars, vectors, or matrices.

If your measurement functions have more than one additional input, use Simulink Function blocks to specify the functions. When you use a Simulink Function block, you provide the additional inputs directly to the Simulink Function block using Inport blocks. No input ports are generated for the additional inputs in the Extended Kalman Filter block.

Dependencies

A port corresponding to a measurement function `h` is generated only if both of the following conditions are satisfied:

- You specify `h` in **Function** using a MATLAB function, and `h` is on the MATLAB path.
- `h` requires only one additional input argument apart from `x` and `v`.

Data Types: `single` | `double`

`Q` — Time-varying process noise covariance
scalar | vector | matrix

Time-varying process noise covariance, specified as a scalar, vector, or matrix depending on the value of the **Process noise** parameter:

- **Process noise** is `Additive` — Specify the covariance as a scalar, an *Ns*-element vector, or an *Ns*-by-*Ns* matrix, where *Ns* 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. Specify a vector of length *Ns*, if there is no cross-correlation between process noise terms, but all the terms have different variances.
- **Process noise** is `Nonadditive` — Specify the covariance as a *W*-by-*W* matrix, where *W* is the number of process noise terms in the state transition function.

Dependencies

This port is generated if you specify the process noise covariance as **Time-Varying**. The port appears when you click **Apply**.

Data Types: `single` | `double`

`R1,R2,R3,R4,R5` — Time-varying measurement noise covariance
matrix

Time-varying measurement noise covariances for up to five measurement functions of the system, specified as matrices. The sizes of the matrices depend on the value of the **Measurement noise** parameter for the corresponding measurement function:

- **Measurement noise** is `Additive` — Specify the covariance as an *N*-by-*N* matrix, where *N* is the number of measurements of the system.
- **Measurement noise** is `Nonadditive` — Specify the covariance as a *V*-by-*V* matrix, where *V* is the number of measurement noise terms in the corresponding measurement function.

Dependencies

A port is generated if you specify the measurement noise covariance as **Time-Varying** for the corresponding measurement function. The port appears when you click **Apply**.

Data Types: `single` | `double`

`Enable1,Enable2,Enable3,Enable4,Enable5` — Enable correction of estimated states when measured data is available
scalar

Suppose that measured output data is not available at all time points at the port **y1** that corresponds to the first measurement function. Use a signal value other than `0` at the **Enable1** port to enable the correction of estimated states when measured data is available. Specify the port value as `0` when measured data is not available. Similarly, if measured output data is not available at all time points at the port **y*i*** for the i^{th} measurement function, specify the corresponding port **Enable*i*** as a value other than `0`.

Dependencies

A port corresponding to a measurement function is generated if you select **Add Enable port** for that measurement function. The port appears when you click **Apply**.

Data Types: `single` | `double` | `Boolean`

Output

`xhat` — Estimated states
vector

Estimated states, returned as a vector of size *Ns*, where *Ns* is the number of states of the system. To access the individual states, use the Selector block.

When the **Use the current measurements to improve state estimates** parameter is selected, the block outputs the corrected state estimate $\widehat{x}[k|k]$ at time step `k`, estimated using measured outputs until time `k`. If you clear this parameter, the block returns the predicted state estimate $\widehat{x}[k|k-1]$ for time `k`, estimated using measured output until a previous time `k-1`. Clear this parameter if your filter is in a feedback loop and there is an algebraic loop in your Simulink model.

Data Types: `single` | `double`

`P` — State estimation error covariance
matrix

State estimation error covariance, returned as an *Ns*-by-*Ns* matrix, where *Ns* is the number of states of the system. To access the individual covariances, use the Selector block.

Dependencies

This port is generated if you select **Output state estimation error covariance** in the **System Model** tab, and click **Apply**.

Data Types: `single` | `double`

Parameters

System Model Tab

State Transition

`Function` — State transition function name
`myStateTransitionFcn` (default) | function name

The state transition function calculates the *Ns*-element state vector of the system at time step *k*+1, given the state vector at time step *k*. *Ns* is the number of states of the nonlinear system. You create the state transition function and specify the function name in **Function**. For example, if `vdpStateFcn.m` is the state transition function that you created and saved, specify **Function** as `vdpStateFcn`.

The inputs to the function you create depend on whether you specify the process noise as additive or nonadditive in **Process noise**.

- **Process noise** is `Additive` — The state transition function *f* specifies how the states evolve as a function of state values at previous time step:

  `x(k+1) = f(x(k),Us1(k),...,Usn(k))`,

  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. To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.
- **Process noise** is `Nonadditive` — The state transition function also specifies how the states evolve as a function of the process noise `w`:

  `x(k+1) = f(x(k),w(k),Us1(k),...,Usn(k))`.

For more information, see "State Transition and Measurement Functions" on page 2-14.

You can create *f* using a Simulink Function block or as a MATLAB function (`.m` file).

- You can use a MATLAB function only if *f* has one additional input argument `Us1` other than `x` and `w`.

  ```
  x(k+1) = f(x(k),w(k),Us1(k))
  ```

  The software generates an additional input port **StateTransitionFcnInputs** to specify this argument.

- If you are using a Simulink Function block, specify `x` and `w` using Argument Inport blocks and the additional inputs `Us1,...,Usn` using Inport blocks in the Simulink Function block. You do not provide `Us1,...,Usn` to the Extended Kalman Filter block.

Programmatic Use
Block Parameter: `StateTransitionFcn`
Type: character vector, string
Default: `'myStateTransitionFcn'`

`Jacobian` — Jacobian of state transition function

`off` (default) | `on`

Jacobian of state transition function *f*, specified as one of the following:

- `off` — The software computes the Jacobian numerically. This computation may increase processing time and numerical inaccuracy of the state estimation.
- `on` — You create a function to compute the Jacobian, and specify the name of the function in **Jacobian**. For example, if `vdpStateJacobianFcn.m` is the Jacobian function, specify **Jacobian** as `vdpStateJacobianFcn`. If you create the state transition function *f* using a Simulink Function block, then create the Jacobian using a Simulink Function block. If you create *f* using a MATLAB function (`.m` file), then create the Jacobian using a MATLAB function.

  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 state transition function and must be specified in the same order in both functions. The number of outputs of the Jacobian function depends on the **Process noise** parameter:

  - **Process noise** is `Additive` — The function calculates the partial derivative of the state transition function *f* with respect to the states ($\partial f/\partial x$). The output is an *Ns*-by-*Ns* Jacobian matrix, where *Ns* is the number of states.

    To see an example of a Jacobian function for additive process noise, type `edit vdpStateJacobianFcn` at the command line.
  - **Process noise** is `Nonadditive` — The function must also return a second output that is the partial derivative of the state transition function *f* with respect to the process noise terms ($\partial f/\partial w$). The second output is returned as an *Ns*-by-*W* matrix, where *W* is the number of process noise terms in the state transition function.

Programmatic Use
Block Parameter: `HasStateTransitionJacobianFcn`
Type: character vector
Values: `'off'`,`'on'`
Default: `'off'`
Block Parameter: `StateTransitionJacobianFcn`
Type: character vector, string
Default: `''`

`Process noise` — Process noise characteristics

`Additive` (default) | `Nonadditive`

Process noise characteristics, specified as one of the following values:

- `Additive` — Process noise `w` is additive, and the state transition function *f* that you specify in **Function** has the following form:

  `x(k+1) = f(x(k),Us1(k),...,Usn(k))`,

  where `x(k)` is the estimated state at time `k`, and `Us1,...,Usn` are any additional input arguments required by your state transition function.

- `Nonadditive` — Process noise is nonadditive, and 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+1) = f(x(k),w(k),Us1(k),...,Usn(k))`.

Programmatic Use
Block Parameter: `HasAdditiveProcessNoise`
Type: character vector
Values: `'Additive'`, `'Nonadditive'`
Default: `'Additive'`

`Covariance` — Time-invariant process noise covariance

1 (default) | scalar | vector | matrix

Time-invariant process noise covariance, specified as a scalar, vector, or matrix depending on the value of the **Process noise** parameter:

- **Process noise** is `Additive` — Specify the covariance as a scalar, an *Ns*-element vector, or an *Ns*-by-*Ns* matrix, where *Ns* 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. Specify a vector of length *Ns*, if there is no cross-correlation between process noise terms but all the terms have different variances.
- **Process noise** is `Nonadditive` — Specify the covariance as a *W*-by-*W* matrix, where *W* is the number of process noise terms.

If the process noise covariance is time-varying, select **Time-varying**. The block generates input port **Q** to specify the time-varying covariance.

Dependencies

This parameter is enabled if you do not specify the process noise as **Time-Varying**.

Programmatic Use
Block Parameter: `ProcessNoise`
Type: character vector, string
Default: `'1'`

`Time-varying` — Time-varying process noise covariance

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

If you select this parameter, the block includes an additional input port **Q** to specify the time-varying process noise covariance.

Programmatic Use
Block Parameter: `HasTimeVaryingProcessNoise`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

Initialization

`Initial state` — Initial state estimate

`0` (default) | vector

Initial state estimate value, specified as an *Ns*-element vector, where *Ns* is the number of states in the system. Specify the initial state values based on your knowledge of the system.

Programmatic Use
Block Parameter: `InitialState`
Type: character vector, string
Default: `'0'`

`Initial covariance` — State estimation error covariance

`1` (default) | scalar | vector | matrix

State estimation error covariance, specified as a scalar, an *Ns*-element vector, or an *Ns*-by-*Ns* matrix, where *Ns* is the number of states of the system. If you specify a scalar or vector, the software creates an *Ns*-by-*Ns* diagonal matrix with the scalar or vector elements on the diagonal.

Specify a high value for the covariance when you do not have confidence in the initial state values that you specify in **Initial state**.

Programmatic Use
Block Parameter: `InitialStateCovariance`
Type: character vector, string
Default: `'1'`

Measurement

`Function` — Measurement function name

`myMeasurementFcn` (default) | function name

The measurement function calculates the *N*-element output measurement vector of the nonlinear system at time step *k*, given the state vector at time step *k*. You create the measurement function and specify the function name in **Function**. For example, if `vdpMeasurementFcn.m` is the measurement function that you created and saved, specify **Function** as `vdpMeasurementFcn`.

The inputs to the function you create depend on whether you specify the measurement noise as additive or nonadditive in **Measurement noise**.

- **Measurement noise** is `Additive` — The measurement function *h* specifies how the measurements evolve as a function of state Values:

  `y(k) = h(x(k),Um1(k),...,Umn(k))`,

  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 a sensor for tracking an object, an additional input could be the sensor position.

  To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line.
- **Measurement noise** is `Nonadditive`— The measurement function also specifies how the output measurement evolves as a function of the measurement noise `v`:

  `y(k) = h(x(k),v(k),Um1(k),...,Umn(k))`.

To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

For more information, see “State Transition and Measurement Functions” on page 2-14.

You can create *h* using a Simulink Function block or as a MATLAB function (`.m` file).

- You can use a MATLAB function only if *h* has one additional input argument `Um1` other than `x` and `v`.

  ```
  y[k] = h(x[k],v[k],Um1(k))
  ```

  The software generates an additional input port **MeasurementFcn*i*Inputs** to specify this argument for the *i*th measurement function.
- If you are using a Simulink Function block, specify `x` and `v` using Argument Inport blocks and the additional inputs `Um1,...,Umn` using Inport blocks in the Simulink Function block. You do not provide `Um1,...,Umn` to the Extended Kalman Filter block.

If you have multiple sensors in your system, you can specify multiple measurement functions. You can specify up to five measurement functions using the **Add Measurement** button. To remove measurement functions, use **Remove Measurement**.

Programmatic Use
Block Parameter: `MeasurementFcn1`, `MeasurementFcn2`, `MeasurementFcn3`, `MeasurementFcn4`, `MeasurementFcn5`
Type: character vector, string
Default: `'myMeasurementFcn'`

Jacobian — Jacobian of measurement function
`off` (default) | `on`

Jacobian of measurement function *h*, specified as one of the following:

- `off` — The software computes the Jacobian numerically. This computation may increase processing time and numerical inaccuracy of the state estimation.
- `on` — You create a function to compute the Jacobian of the measurement function *h*, and specify the name of the function in **Jacobian**. For example, if `vdpMeasurementJacobianFcn.m` is the Jacobian function, specify `MeasurementJacobianFcn` as `vdpMeasurementJacobianFcn`. If you create *h* using a Simulink Function block, then create the Jacobian using a Simulink Function block. If you create *h* using a MATLAB function (`.m` file), then create the Jacobian using a MATLAB function.

  The function calculates the partial derivatives of the measurement function *h* 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 **Measurement noise** parameter:

  - **Measurement noise** is `Additive` — 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-*Ns* Jacobian matrix, where *N* is the number of measurements of the system and *Ns* is the number of states.

    To see an example of a Jacobian function for additive measurement noise, type `edit vdpMeasurementJacobianFcn` at the command line.

- **Measurement noise** is `Nonadditive` — 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.

Programmatic Use
Block Parameter: `HasMeasurementJacobianFcn1`, `HasMeasurementJacobianFcn2`, `HasMeasurementJacobianFcn3`, `HasMeasurementJacobianFcn4`,`HasMeasurementJacobianFcn5`
Type: character vector
Values: `'off'`,`'on'`
Default: `'off'`
Block Parameter: `MeasurementJacobianFcn1`, `MeasurementJacobianFcn2`, `MeasurementJacobianFcn3`, `MeasurementJacobianFcn4`, `MeasurementJacobianFcn5`
Type: character vector
Default: `''`

`Measurement noise` — Measurement noise characteristics

`Additive` (default) | `Nonadditive`

Measurement noise characteristics, specified as one of the following values:

- `Additive` — Measurement noise `v` is additive, and the measurement function *h* that you specify in **Function** has the following form:

  `y(k) = h(x(k),Um1(k),...,Umn(k))`,

  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.
- `Nonadditive` — Measurement noise is nonadditive, and 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(k),...,Umn(k))`.

Programmatic Use
Block Parameter: `HasAdditiveMeasurementNoise1`, `HasAdditiveMeasurementNoise2`, `HasAdditiveMeasurementNoise3`, `HasAdditiveMeasurementNoise4`, `HasAdditiveMeasurementNoise5`
Type: character vector
Values: `'Additive'`, `'Nonadditive'`
Default: `'Additive'`

`Covariance` — Time-invariant process noise covariance

`1` (default) | scalar | vector | matrix

Time-invariant process noise covariance, specified as a scalar, vector, or matrix depending on the value of the **Process noise** parameter:

- **Process noise** is `Additive` — Specify the covariance as a scalar, an *Ns*-element vector, or an *Ns*-by-*Ns* matrix, where *Ns* 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. Specify a vector of length *Ns*, if there is no cross-correlation between process noise terms but all the terms have different variances.

- **Process noise** is `Nonadditive` — Specify the covariance as a *W*-by-*W* matrix, where *W* is the number of process noise terms.

If the process noise covariance is time-varying, select **Time-varying**. The block generates input port **Q** to specify the time-varying covariance.

Dependencies

This parameter is enabled if you do not specify the process noise as **Time-Varying**.

Programmatic Use

Block Parameter: `ProcessNoise`
Type: character vector, string
Default: `'1'`

`Time-varying` — Time-varying measurement noise covariance

`off` (default) | `on`

If you select this parameter for the measurement noise covariance of the first measurement function, the block includes an additional input port **R1**. You specify the time-varying measurement noise covariance in **R1**. Similarly, if you select **Time-varying** for the *i*th measurement function, the block includes an additional input port **R*i*** to specify the time-varying measurement noise covariance for that function.

Programmatic Use

Block Parameter: `HasTimeVaryingMeasurementNoise1`, `HasTimeVaryingMeasurementNoise2`, `HasTimeVaryingMeasurementNoise3`, `HasTimeVaryingMeasurementNoise4`, `HasTimeVaryingMeasurementNoise5`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Add Enable Port` — Enable correction of estimated states only when measured data is available

`off` (default) | `on`

Suppose that measured output data is not available at all time points at the port **y1** that corresponds to the first measurement function. Select **Add Enable port** to generate an input port **Enable1**. Use a signal at this port to enable the correction of estimated states only when measured data is available. Similarly, if measured output data is not available at all time points at the port **y*i*** for the *i*th measurement function, select the corresponding **Add Enable port**.

Programmatic Use

Block Parameter: `HasMeasurementEnablePort1`, `HasMeasurementEnablePort2`, `HasMeasurementEnablePort3`, `HasMeasurementEnablePort4`, `HasMeasurementEnablePort5`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

Settings

`Use the current measurements to improve state estimates` — Choose between corrected or predicted state estimate

`on` (default) | `off`

When this parameter is selected, the block outputs the corrected state estimate $\hat{x}[k|k]$ at time step `k`, estimated using measured outputs until time `k`. If you clear this parameter, the block returns the predicted state estimate $\hat{x}[k|k-1]$ for time `k`, estimated using measured output until a previous time `k-1`. Clear this parameter if your filter is in a feedback loop and there is an algebraic loop in your Simulink model.

Programmatic Use
Block Parameter: `UseCurrentEstimator`
Type: character vector
Values: `'off'`, `'on'`
Default: `'on'`

`Output state estimation error covariance` — Output state estimation error covariance
`off` (default) | `on`

If you select this parameter, a state estimation error covariance output port **P** is generated in the block.

Programmatic Use
Block Parameter: `OutputStateCovariance`
Type: character vector
Values: `'off'`,`'on'`
Default: `'off'`

`Data type` — Data type for block parameters
`double` (default) | `single`

Use this parameter to specify the data type for all block parameters.

Programmatic Use
Block Parameter: `DataType`
Type: character vector
Values: `'single'`, `'double'`
Default: `'double'`

`Sample time` — Block sample time
`1` (default) | positive scalar

Block sample time, specified as a positive scalar. If the sample times of your state transition and measurement functions are different, select **Enable multirate operation** in the **Multirate** tab, and specify the sample times in the **Multirate** tab instead.

Dependencies

This parameter is available if in the **Multirate** tab, the **Enable multirate operation** parameter is `off`.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector, string
Default: `'1'`

Multirate Tab

`Enable multirate operation` — Enable specification of different sample times for state transition and measurement functions
`off` (default) | `on`

Select this parameter if the sample times of the state transition and measurement functions are different. You specify the sample times in the **Multirate** tab, in **Sample time**.

Programmatic Use
Block Parameter: `EnableMultirate`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Sample times` — State transition and measurement function sample times
positive scalar

If the sample times for state transition and measurement functions are different, specify **Sample time**. Specify the sample times for the measurement functions as positive integer multiples of the state transition sample time. The sample times you specify correspond to the following input ports:

- Ports corresponding to state transition function — Additional input to state transition function **StateTransitionFcnInputs** and time-varying process noise covariance **Q**. The sample times of these ports must always equal the state transition function sample time, but can differ from the sample time of the measurement functions.
- Ports corresponding to i^{th} measurement function — Measured output **y*i***, additional input to measurement function **MeasurementFcn*i*Inputs**, enable signal at port **Enable*i***, and time-varying measurement noise covariance **R*i***. The sample times of these ports for the same measurement function must always be the same, but can differ from the sample time for the state transition function and other measurement functions.

Dependencies

This parameter is available if in the **Multirate** tab, the **Enable multirate operation** parameter is `on`.

Programmatic Use
Block Parameter: `StateTransitionFcnSampleTime`, `MeasurementFcn1SampleTime1`, `MeasurementFcn1SampleTime2`, `MeasurementFcn1SampleTime3`, `MeasurementFcn1SampleTime4`, `MeasurementFcn1SampleTime5`
Type: character vector, string
Default: `'1'`

More About

State Transition and Measurement Functions

The algorithm computes the state estimates $\hat{x}$ of the nonlinear system using state transition and measurement functions specified by you. You can specify up to five measurement functions, each corresponding to a sensor in the system. 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:

  $$x[k+1] = f(x[k], u_s[k]) + w[k]$$
  $$y[k] = h(x[k], u_m[k]) + v[k]$$

  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 optional input arguments that are denoted by $\texttt{u}_\texttt{s}$ and $\texttt{u}_\texttt{m}$ 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+1)` is linearly related to the process noise `w(k)`, and `y(k)` is linearly related to the measurement noise `v(k)`. For additive noise terms, you do not need to specify the noise terms in the state transition and measurement functions. The software adds the terms to the output of the functions.

- **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:

$$x[k+1] = f(x[k], w[k], u_s[k])$$
$$y[k] = h(x[k], v[k], u_m[k])$$

Compatibility Considerations

Numerical Changes
Behavior changed in R2020b

Starting in R2020b, numerical improvements in the Extended Kalman Filter algorithm might produce results that are different from the results you obtained in previous versions.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The state transition, measurement, and Jacobian functions that you specify must use only the MATLAB commands and Simulink blocks that support code generation. For a list of blocks that support code generation, see “Simulink Built-In Blocks That Support Code Generation” (Simulink Coder). For a list of commands that support code generation, see “Functions and Objects Supported for C/C++ Code Generation” (MATLAB Coder).

Generated code uses an algorithm that is different from the algorithm that the Extended Kalman Filter block itself uses. You might see some numerical differences in the results obtained using the two methods.

See Also

Blocks
Kalman Filter | Unscented Kalman Filter | Particle Filter

Functions
`extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter`

Topics
“What Is Online Estimation?”
“Extended and Unscented Kalman Filter Algorithms for Online State Estimation”

“Validate Online State Estimation in Simulink”
“Troubleshoot Online State Estimation”

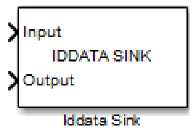
External Websites
Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

Introduced in R2017a

Iddata Sink

Export simulation data as `iddata` object to MATLAB workspace

Library: System Identification Toolbox



Description

The Iddata Sink block exports simulation data as an `iddata` object to the MATLAB workspace. The object stores the input and simulated output signals, sampled at the sample time that you specify. If you simulate your model from the model window, the block exports the object to the MATLAB base workspace. If you simulate the model programmatically, the object is exported to the MATLAB caller workspace. The caller workspace is the workspace of the function that called the currently running function.

Ports

Input

`Input` — Input of `iddata` object

scalar | vector

Input of `iddata` object, specified as a scalar for single-input data. For multichannel data with *Nu* inputs, specify `Input` as a vector of length *Nu*.

Data Types: `double`

`Output` — Output of `iddata` object

scalar | vector

Output of `iddata` object, specified as a scalar for single-output data. For multichannel data with *Ny* outputs, specify `Output` as a vector of length *Ny*.

Data Types: `double`

Parameters

`IDDATA Name` — Name of `iddata` object

`data` (default) | variable name

Name of `iddata` object, specified as a MATLAB variable name. The object is exported with this name to the MATLAB workspace.

`Sample Time` — Sample time in seconds

`0.1` (default) | finite positive number

Sample time in seconds, specified as a finite positive number. The `iddata` object stores the input and output signals, sampled at the sample time that you specify.

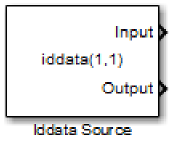
See Also
Iddata Source

Topics
"Simulate Identified Model in Simulink"

Introduced in R2008a

Iddata Source

Import time-domain data stored in `iddata` object in MATLAB workspace
Library: System Identification Toolbox



Description

The Iddata Source block imports the input-output time-domain data stored in an `iddata` object in the MATLAB workspace. You can use this block to import data for simulating a model in Simulink.

Ports

Output

`Input` — Input data stored in `iddata` object
scalar | vector

Input data stored in `iddata` object, returned as a scalar for single-input data and a vector of length *Nu* for multichannel data with *Nu* inputs. If `z` is the `iddata` object, the output at this port at simulation time *t* is `z.InputData` at time *t*. If *t* is greater than `z.SamplingInstants(end)`, the maximum time in `z`, the input data is returned as `0` for each input channel.

Data Types: `double`

`Output` — Output data stored in `iddata` object
scalar | vector

Output data stored in `iddata` object, returned as a scalar for single-output data and as a vector of length *Ny* for multichannel data with *Ny* outputs. If `z` is the `iddata` object, the output at this port at simulation time *t* is `z.OutputData` at time *t*. If *t* is greater than `z.SamplingInstants(end)`, the maximum time in `z`, the output data is returned as `0` for each output channel.

Data Types: `double`

Parameters

`IDDATA object` — Time-domain data to be imported
`iddata(1,1)` (default) | `iddata` object

Time-domain data that is to be imported, specified as an `iddata` object that you have created in the MATLAB workspace. The `iddata` object must contain only one experiment. For a multiple-experiment `iddata` object `z`, to specify an `iddata` object for experiment number `kexp`, specify **IDDATA object** as `getexp(z,kexp)`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

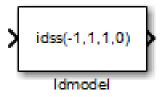
Iddata Sink

Topics
"Simulate Identified Model in Simulink"

Introduced in R2008a

Idmodel

Simulate identified linear model in Simulink software
Library: System Identification Toolbox / Models



Description

The Idmodel block simulates the output of an identified linear model using time-domain input data. The model is a state-space (`idss`), linear grey-box (`idgrey`), polynomial (`idpoly`), transfer function (`idtf`), or process (`idproc`) model that you previously estimated or created. For the simulation of state-space and linear grey-box models, you can specify the initial state values. For other linear models, initial conditions are set to zero. You can also add noise to the simulated output.

Ports

Input

`Port_1(In1)` — Simulation input data
scalar | vector

Simulation input data, specified as a scalar for a single-input model. The data must be time-domain data. 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.

Data Types: `double`

Output

`Port_1(Out1)` — Simulated output
scalar | vector

Simulated output from linear model, returned as a scalar for a single-output model and an *Ny*-element vector for a model with *Ny* outputs.

Data Types: `double`

Parameters

`Identified model` — Linear model to be simulated
`idss(-1,1,1,0,'Ts',1)` (default) | `idss` object | `idgrey` object | `idpoly` object | `idtf` object | `idproc` object

Linear model to be simulated, specified as an `idss`, `idgrey`, `idpoly`, `idtf`, or `idproc` object. The model can be continuous-time or discrete-time, with or without input-output delays. You previously estimate or construct the linear model in the MATLAB workspace.

`Initial states (state space only: idss,idgrey)` — Initial state values

`0` (default) | vector

Initial state values of state-space (`idss`) and linear grey-box (`idgrey`) models, specified as an *Nx*-element vector, where *Nx* is the number of states of the model. 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 simulated output data in the data set `z`, specify `X0`, such that:

```
X0 = findstates(M,z)
```

For linear models other than `idss` or `idgrey`, the block assumes that initial conditions are zero.

If you want to reproduce the simulation results that you get in the Model Output plot window in the System Identification app, or from the `compare` command:

1. If the identified model `m` is not a state-space or grey-box model, convert the model into state-space form (`idss` model), and specify the state-space model `mss` in the block.

   ```
   mss = idss(m);
   ```
2. Compute the initial state values that produce the best fit between the model output and the measured output signal using `findstates`. Specify the prediction horizon as `Inf`, that is, minimize the simulation error.

   ```
   X0 = findstates(mss,z,Inf);
   ```
3. Use the model `mss` and initial states `X0` in the Idmodel block to perform the simulation. Specify the same input signal `z` for simulation that you used as validation data in the app or `compare`.

`Add noise` — Add noise to simulated output

`on` (default) | `off`

When you select this parameter, the block derives the noise amplitude from the linear model property `model.NoiseVariance`. The software filters random Gaussian white noise with the noise transfer function of the model and adds the resulting noise to the simulated model response. If you want to add the same noise every time you run the Simulink model, specify the `Noise seed(s)` property.

For continuous-time models, the ideal variance of the noise term is infinite. In reality, you see a band-limited noise that accounts for 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)` — Add same noise to output for multiple simulations

`[ ]` (default) | nonnegative integer | vector

The **Noise seed(s)** property seeds the random number generator such that the block adds the same noise to the simulated output every time you run the Simulink model. For information about using seeds, see `rng`.

For multi-output models, you can use independent noise realizations that generate the outputs with additive noise. Enter a vector of *Ny* nonnegative integer entries, where *Ny* is the number of output channels.

For random restarts that vary from one simulation to another, specify **Noise seed(s)** as `[]`.

Dependency

To enable this parameter, select `Add noise`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Functions

`sim` | `idgrey` | `idproc` | `idtf` | `idss` | `idpoly` | `findstates`

Blocks

Iddata Sink | Iddata Source

Topics

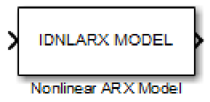
“Simulate Identified Model in Simulink”

Introduced in R2008a

Nonlinear ARX Model

Simulate nonlinear ARX model in Simulink software

Library: System Identification Toolbox / Models



Description

The Nonlinear ARX Model block simulates the output of a nonlinear ARX model using time-domain input data. The model is an `idnlarx` model that you previously estimated or constructed in the MATLAB workspace. You specify initial conditions for the simulation as either steady-state input and output signal levels or as an initial state vector.

Ports

Input

`Port_1(In1)` — Simulation input data

scalar | vector

Simulation input data, specified as a scalar for a single-input model. The data must be time-domain data. 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.

Data Types: `double`

Output

`Port_1(Out1)` — Simulated output

scalar | vector

Simulated output from nonlinear ARX model, returned as a scalar for a single-output model and an *Ny*-element vector for a model with *Ny* outputs.

Data Types: `double`

Parameters

`Model` — Nonlinear ARX model to be simulated

`idnlarx` object

Nonlinear ARX model to be simulated, specified as an `idnlarx` object. You previously estimate or construct the `idnlarx` model in the MATLAB workspace.

Initial conditions — Initial condition specification for simulation

Input and output values (default) | State values

The states of a nonlinear ARX model correspond to the dynamic elements of the nonlinear ARX model structure. The dynamic elements are the model regressors. Regressors can be the delayed input or output variables (standard regressors) or user-defined transformations of delayed input-output variables (custom regressors). For more information about the states of a nonlinear ARX model, see the `idnlarx` reference page.

For simulating nonlinear ARX models, you can specify the initial conditions one of the following:

- `Input and output values` — Specify steady-state input and output signal levels in `Input level` and `Output level`, respectively.
- `State values` — Specify a vector of length equal to the number of states in the model in `Specify initial states as a vector`.

Input level — Steady-state input signal level

0 (default) | scalar

Steady-state input signal level before simulation, specified as a scalar.

Dependency

To enable this parameter, specify `Initial conditions` as `Input and output values`.

Output level — Steady-state output signal level

0 (default) | scalar

Steady-state output signal level before simulation, specified as a scalar.

Dependency

To enable this parameter, specify `Initial conditions` as `Input and output values`.

Specify initial states as a vector — Initial state values

0 (default) | vector

Initial state values of the model, specified as an *Nx*-element vector, where *Nx* is the number of states of the model. This parameter is named **Vector of state values** until you specify `Model`.

If you do not know the initial states, you can estimate these states as follows:

- To simulate the model around a given input level when you do not know the corresponding output level, 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, specify the initial state values as `X0`, where

  ```
  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`, specify `X0`, such that:

  ```
  X0 = findstates(M,z,Inf)
  ```
- To continue a simulation from a previous simulation run, use the simulated input-output values from the previous simulation to compute the initial states `X0` for the current simulation. Use the

`data2state` command to compute `X0`. 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 specify `X0`, such that:

```
X0 = data2state(M,firstSimData)
```

Dependency

To enable this parameter, specify `Initial conditions` as `State values`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Functions

`sim` | `idnlarx` | `idnlarx/findop` | `findstates`

Blocks

Iddata Sink | Iddata Source

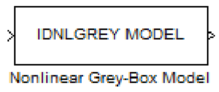
Topics

"Identifying Nonlinear ARX Models"
"Simulate Identified Model in Simulink"

Introduced in R2008a

Nonlinear Grey-Box 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

Functions
`idnlgrey`

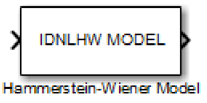
Blocks
Iddata Sink | Iddata Source

Topics
“Estimate Nonlinear Grey-Box Models”

Introduced in R2008a

Hammerstein-Wiener Model

Simulate Hammerstein-Wiener model in Simulink software
Library: System Identification Toolbox / Models



Description

The Hammerstein-Wiener Model block simulates the output of a Hammerstein-Wiener model using time-domain input data. The model is an `idnlhw` model that you previously estimated or constructed in the MATLAB workspace. You specify initial conditions for the simulation as one of the following:

- Zero for all states
- Initial state vector representing the initial states of the linear block

For information about the structure of a Hammerstein-Wiener model, see “What are Hammerstein-Wiener Models?”.

Ports

Input

`Port_1(In1)` — Simulation input data
scalar | vector

Simulation input data, specified as a scalar for a single-input model. The data must be time-domain data. 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.

Data Types: `double`

Output

`Port_1(Out1)` — Simulated output
scalar | vector

Simulated output from Hammerstein-Wiener model, returned as a scalar for a single-output model and as an *Ny*-element vector for a model with *Ny* outputs.

Data Types: `double`

Parameters

`Model` — Hammerstein-Wiener model to be simulated
`idnlhw` object

Hammerstein-Wiener model to be simulated, specified as an `idnlhw` object. You previously estimate or construct the `idnlhw` model in the MATLAB workspace.

`Initial conditions` — Initial condition specification for simulation
`Zero` (default) | `State values`

The states of a Hammerstein-Wiener model correspond to the states of the embedded linear `idpoly` or `idss` model. For more information about the states, see the `idnlhw` reference page. You specify `Initial conditions` as one of the following:

- `Zero` — Specifies zero initial state values, which correspond to a simulation starting from a state of rest.
- `State values` — You specify the state values in **Specify a vector of state values**. Specify the states as a vector of length equal to the number of states in the model.

  If you do not know the initial states, you can estimate these states as follows:

  - To simulate the model 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 specify the initial state values as `X0`, where:

    ```
    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 simulated output data in the data set `z`, specify `X0`, such that:

    ```
    X0 = findstates(M,z)
    ```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Functions
`sim` | `idnlhw` | `idnlhw/findop` | `findstates`

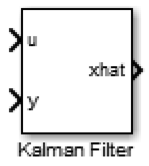
Blocks
Iddata Sink | Iddata Source

Topics
"Identifying Hammerstein-Wiener Models"
"Simulate Identified Model in Simulink"

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

$$\dot{x}(t) = A(t)x(t) + B(t)u(t) + G(t)w(t) \quad \text{(state equation)}$$
$$y(t) = C(t)x(t) + D(t)u(t) + H(t)w(t) + v(t) \quad \text{(measurement equation)}$$

with known inputs u, white process noise w, and white measurement noise v satisfying:

$$E[w(t)] = E[v(t)] = 0$$
$$E[w(t)w^T(t)] = Q(t)$$
$$E[w(t)v^T(t)] = N(t)$$
$$E[v(t)v(^t] = R(t)$$

construct a state estimate $\hat{x}$ that minimizes the state estimation error covariance $P(t) = E[(x - \hat{x})(x - \hat{x})^T]$.

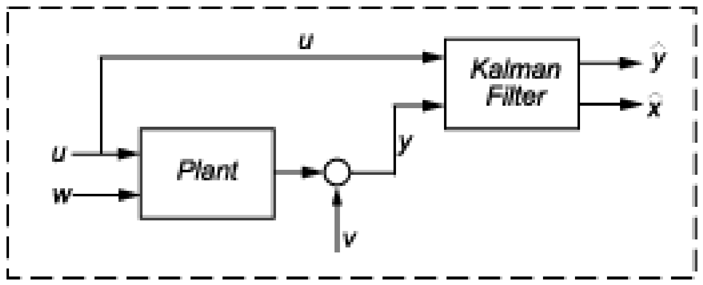
The optimal solution is the Kalman filter with equations

$$L(t) = (P(t)C^T(t) + \bar{N}(t))\bar{R}(^t,$$
$$\dot{P}(t) = A(t)P(t) + P(t)A^T(t) + \bar{Q}(t) - L(t)\bar{R}(t)L^T(t),$$
$$\dot{\hat{x}}(t) = A(t)\hat{x}(t) + B(t)u(t) + L(t)(y(t) - C(t)\hat{x}(t) - D(t)u(t)),$$

where

$$\bar{Q}(t) = G(t)Q(t)G^T(t),$$
$$\bar{R}(t) = R(t) + H(t)N(t) + N^T(t)H^T(t) + H(t)Q(t)H^T(t),$$
$$\bar{N}(t) = G(t)(Q(t)H^T(t) + N(t))\,.$$

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:

$$P = \lim_{t \to \infty} E[(x - \hat{x})(x - \hat{x})^T]\,.$$

Discrete-Time Estimation

Given the discrete plant

$$x[n+1] \;=\; A[n]\,x[n] \;+\; B[n]\,u[n] \;+\; G[n]\,w[n],$$
$$y[n] \;=\; C[n]\,x[n] \;+\; D[n]\,u[n] \;+\; H[n]\,w[n] \;+\; v[n],$$

with known inputs u, white process noise w and white measurement noise v satisfying

$$E[w[n]] = E[v[n]] = 0,$$
$$E[w[n]w^T[n]] = Q[n],$$
$$E[v[n]v^T[n]] = R[n],$$
$$E[w[n]v^T[n]] = N[n]\,.$$

The estimator has the following state equation

$$\hat{x}[n+1|n] = A[n]\hat{x}[n|n-1] + B[n]u[n] + L[n](y[n] - C[n]\hat{x}[n|n-1] - D[n]u[n]),$$

where the gain `L[n]` is calculated through the discrete Riccati equation:

$$L[n] = (A[n]P[n]C^T[n] + \bar{N}[n])(^C,$$
$$M[n] = P[n]C^T[n](^C,$$
$$Z[n] = (I - M[n]C[n])P[n](^I + M[n]\bar{R}[n]M^T[n],$$
$$P[n+1] = (A[n] - \bar{N}[n]\bar{R}^{-1}[n]C[n])Z(^A + \bar{Q}[n] - N[n]\bar{R}^{-1}[n]\bar{N}^T[n],$$

where I is the identity matrix of appropriate size and

$$\bar{Q}[\mathrm{n}] = G[\mathrm{n}]Q[\mathrm{n}]G^{T}[\mathrm{n}],$$
$$\bar{R}[\mathrm{n}] = R[\mathrm{n}] + H[\mathrm{n}]N[\mathrm{n}] + N^{T}[\mathrm{n}]H^{T}[\mathrm{n}] + H[\mathrm{n}]Q[\mathrm{n}]H^{T}[\mathrm{n}],$$
$$\bar{N}[\mathrm{n}] = G[\mathrm{n}](Q[\mathrm{n}]H^{T}[\mathrm{n}] + N[\mathrm{n}]),$$

and

$$P[\mathrm{n}] = E[(x - \widehat{x}[n|n-1])(^{x}],$$
$$Z[\mathrm{n}] = E[(x - \widehat{x}[n|n])(^{x}],$$

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 $\widehat{x}[n|n]$ using all measurement available, including $y[n]$. The filter updates $\widehat{x}[n|n-1]$ with $y[n]$ and outputs:

  $$\widehat{x}[n|n] = \widehat{x}[n|n-1] + M[n](y[n] - C[n]\widehat{x}[n|n-1] - D[n]u[n]),$$
  $$\widehat{y}[n|n] = C[n]\widehat{x}[n|n] + D[n]u[n].$$
- The delayed estimator generates the state estimates $\widehat{x}[n|n-1]$ using measurements up to $y[n-1]$. The filter outputs $\widehat{x}[n|n-1]$ as defined previously, along with the optional output $\widehat{y}[n|n-1]$

  $$\widehat{y}[n|n-1] = C[n]\widehat{x}[n|n-1] + D[n]u[n]$$

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** on page 2-35 • **Use the current measurement y[n] to improve xhat[n]** on page 2-35

Task	Parameters
Specify the system model	**Model source** on page 2-35 in **Model Parameters** tab
Specify initial state estimates	**Source** on page 2-36 in **Model Parameters** tab
Specify noise characteristics	In **Model Parameters** tab: • **Use G and H matrices (default G=I and H=0)** on page 2-37 • **Q** on page 2-37, **Time-invariant Q** on page 2-37 • **R** on page 2-38, **Time-invariant R** on page 2-38 • **N** on page 2-38, **Time-invariant N** on page 2-38
Specify additional inports	In **Options** tab: • **Add input port u** on page 2-38 • **Add input port Enable to control measurement updates** on page 2-38 • **External reset** on page 2-38
Specify additional outports	In **Options** tab: • **Output estimated model output y** on page 2-39 • **Output state estimation error covariance Z** on page 2-39

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 cleared. 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 cleared by default.

By default the block does measurement updates at each time step to improve the state and output estimates $\widehat{x}$ and $\widehat{y}$ 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 $\dot{\widehat{x}}(t) = A(t)\widehat{x}(t) + B(t)u(t)$ for continuous-time Kalman filter and $\widehat{x}[n+1|n] = A[n]\widehat{x}[n|n-1] + B[n]u[n]$ 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 $\widehat{x}$ 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 $\widehat{y}$ output port to the block to output the estimated model outputs. The option is cleared by default.

Output state estimation error covariance 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 2-32 section previously, is provided.

The option is cleared 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.

Port Name	Port Type (In/ Out)	Description
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

$$\bar{Q} = GQG^T$$
$$\bar{R} = R + HN + N^TH^T + HQH^T$$
$$\bar{N} = G(QH^T + N)$$

- The continuous-time Kalman filter cannot be used in Function-Call Subsystems or Triggered Subsystems.

Compatibility Considerations

Kalman Filter block: Numerical changes
Behavior changed in R2021a

Starting in 2021a, numerical improvements in the algorithms used by the Kalman FIlter block might produce results that are different from the results you obtained using previous versions.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Functions
`kalman` | `extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter`

Blocks
Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

Topics
“State Estimation Using Time-Varying Kalman Filter”
“What Is Online Estimation?”
“Validate Online State Estimation in Simulink”
“Troubleshoot Online State Estimation”

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

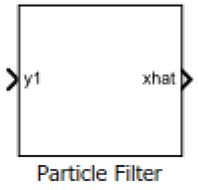
Recursive Polynomial Model Estimator

Introduced in R2014a

Particle Filter

Estimate states of discrete-time nonlinear system using particle filter

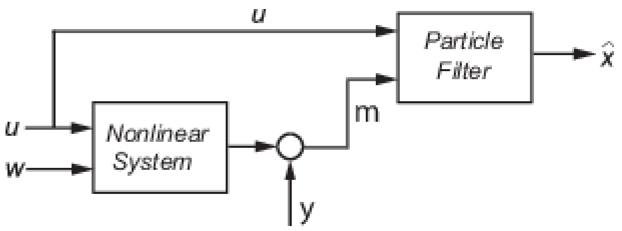
Library: Control System Toolbox / State Estimation
System Identification Toolbox / Estimators



Description

The Particle Filter block estimates the states of a discrete-time nonlinear system using the discrete-time particle filter algorithm.

Consider a plant with states *x*, input *u*, output *m*, process noise *w*, and measurement *y*. Assume that you can represent the plant as a nonlinear system.



The algorithm computes the state estimates $\hat{x}$ of the nonlinear system using the state transition and measurement likelihood functions you specify.

You create the nonlinear state transition function and measurement likelihood functions for the system and specify these functions in the block. The block supports state estimation of a system with multiple sensors that are operating at different sampling rates. You can specify up to five measurement likelihood functions, each corresponding to a sensor in the system.

Ports

Input

y1,y2,y3,y4,y5 — Measured system outputs

vector

Measured system outputs corresponding to each measurement likelihood function that you specify in the block. The number of ports equals the number of measurement likelihood functions in your system. You can specify up to five measurement likelihood functions. For example, if your system has two sensors, you specify two measurement likelihood functions in the block. The first port **y1** is available by default. Click **Add Measurement**, to generate port **y2** corresponding to the second measurement likelihood function.

Specify the ports as *N*-dimensional vectors, where *N* is the number of quantities measured by the corresponding sensor. For example, if your system has one sensor that measures the position and velocity of an object, then there is only one port **y1**. The port is specified as a two-dimensional vector with values corresponding to position and velocity.

Dependencies

The first port **y1** is available by default. Ports **y2** to **y5** are generated when you click **Add Measurement**.

`StateTransitionFcnInputs` — Optional input argument to state transition function

scalar | vector | matrix

Optional input argument to the state transition function `f` other than the state `x`.

If you create `f` using a MATLAB function (`.m` file), the software generates the port **StateTransitionFcnInputs** when you enter the name of your function, and click **Apply**.

If your state transition function has more than one additional input, use a Simulink Function block to specify the function. When you use a Simulink Function block, you provide the additional inputs directly to the Simulink Function block using Inport blocks. No input ports are generated for the additional inputs in the Particle Filter block.

Dependencies

This port is generated only if both of the following conditions are satisfied:

- You specify `f` in **Function** using a MATLAB function, and `f` is on the MATLAB path.
- `f` requires only one additional input argument apart from particles.

`MeasurementLikelihoodFcn1Inputs,...,MeasurementLikelihoodFcn5Inputs` — Optional input argument to each measurement likelihood function

scalar | vector | matrix

Optional inputs to the measurement likelihood functions other than the state `x` and measurement `y`.

MeasurementLikelihoodFcn1Inputs corresponds to the first measurement likelihood function that you specify, and so on.

If you specify two measurement inputs using MATLAB functions (`.m` files) in **Function**, the software generates ports **MeasurementLikelihoodFcn1Inputs** and **MeasurementLikelihoodFcn2Inputs** when you click **Apply**. You can specify the inputs to these ports as scalars, vectors, or matrices.

If your measurement likelihood functions have more than one additional input, use Simulink Function blocks to specify the functions. When you use a Simulink Function block, you provide the additional inputs directly to the Simulink Function block using Inport blocks. No input ports are generated for the additional inputs in the Particle Filter block.

Dependencies

A port corresponding to a measurement likelihood function `h` is generated only if both of the following conditions are satisfied:

- You specify measurement input `h` in **Function** using a MATLAB function, and `h` is on the MATLAB path.
- `h` requires only one additional input argument apart from particles and measurement.

`Enable1,Enable2,Enable3,Enable4,Enable5` — Enable correction of estimated states when measured data is available

scalar

Enable correction of estimated states when measured data is available.

For example, consider that measured output data is not available at all time points at the port **y1** that corresponds to the first measurement likelihood function. Then, use a signal value other than `0` at the **Enable1** port to enable the correction of estimated states when measured data is available. Specify the port value as `0` when measured data is not available. Similarly, if measured output data is not available at all time points at the port **y*i*** for the *i*th measurement likelihood function, specify the corresponding port **Enable*i*** as a value other than `0`.

Dependencies

If you select **Add Enable port** for a measurement likelihood function, a port corresponding to that measurement likelihood function is generated. The port appears when you click **Apply**.

Output

`xhat` — Estimated states

vector

Estimated states, returned as a vector of size *Ns*, where *Ns* is the number of states of the system. To access the individual states, use the Selector block.

When the **Use the current measurements to improve state estimates** parameter is selected, the block outputs the corrected state estimate $\hat{x}[k|k]$ at time step `k`, estimated using measured outputs until time `k`. If you clear this parameter, the block returns the predicted state estimate $\hat{x}[k|k-1]$ for time `k`, estimated using measured output until a previous time `k-1`. Clear this parameter if your filter is in a feedback loop and there is an algebraic loop in your Simulink model.

P — State estimation error covariance

matrix

State estimation error covariance, returned as an *Ns*-by-*Ns* matrix, where *Ns* is the number of states of the system. To access the individual covariances, use the Selector block.

You can output the error covariance only if you select **Output state estimation error covariance** in the **Block outputs, Multirate** tab, and click **Apply**.

Dependencies

This parameter is available if in the **Block outputs, Multirate** tab, the **State estimation method** parameter is set to `'Mean'`.

`Particles` — Particle values used for state estimation

array

Particle values used for state estimation, returned as an *Ns*-by-*Np* or *Np*-by-*Ns* array. *Ns* is the number of states of the system, and *Np* is the number of particles.

- If the `StateOrientation` parameter is specified as `'column'`, then **Particles** is returned as an *Ns*-by-*Np* array.
- If the `StateOrientation` parameter is specified as `'row'`, then **Particles** is returned as an *Np*-by-*Ns* array.

Dependencies

This port is generated if you select **Output all particles** in the **Block outputs, Multirate** tab, and click **Apply**.

`Weights` — Particle weights used for state estimation

vector

Particle weights used for state estimation, returned as a 1-by-*Np* or *Np*-by-1 vector, where *Np* is the number of particles used for state estimation.

- If the `StateOrientation` parameter is specified as `'column'`, then **Weights** is returned as a 1-by-*Np* vector, where each weight is associated with the particle in the same column in the `Particles` array.
- If the `StateOrientation` parameter is specified as `'row'`, then **Weights** is returned as a *Np*-by-1 vector, where each weight is associated with the particle in the same row in the `Particles` array.

Dependencies

This port is generated if you select **Output weights** in the **Block outputs, Multirate** tab, and click **Apply**.

Parameters

System Model Tab

State Transition

`Function` — State transition function name

`'vdpParticleFilterStateFcn'` (default) | function name

The particle filter state transition function calculates the particles at time step *k+1*, given particles at time step *k* per the dynamics of your system and process noise. This function has the syntax:

```
particlesNext = f(particles, param1, param2, ...)
```

where, *particles* and *particlesNext* have dimensions *Ns*-by-*Np* if **State Orientation** is specified as `'column'`, or *Np*-by-*Ns* if **State Orientation** is specified as `'row'`. Also, `param_i` represents optional input arguments you may specify. For more information on optional input arguments, see “StateTransitionFcnInputs” on page 2-0 .

You create the state transition function and specify the function name in **Function**. For example, if `vdpParticleFilterStateFcn.m` is the state transition function that you created and saved, specify **Function** as `'vdpParticleFilterStateFcn'`.

You can create **Function** using a Simulink Function block or as a MATLAB function (`.m` file).

Programmatic Use

Block Parameter: `StateTransitionFcn`
Type: character vector, string
Default: `'vdpParticleFilterStateFcn'`

Initialization

`Number of Particles` — Number of particles used in the filter

1000 (default) | positive scalar integer

Number of particles used in the filter, specified as a positive scalar integer. Each particle represents a state hypothesis in the system. A higher number of particles increases the state estimation accuracy, but also increases the computational effort required to run the filter.

Programmatic Use

Block Parameter: `NumberOfParticles`
Type: positive scalar integer
Default: `1000`

`Distribution` — Initial distribution of particles

`'Gaussian'` (default) | `'Uniform'` | `'Custom'`

Initial distribution of particles, specified as `'Gaussian'`, `'Uniform'`, or `'Custom'`.

If you choose `'Gaussian'`, the initial set of particles or state hypotheses are distributed per the multivariate Gaussian distribution, where you specify the **Mean** and **Covariance**. The initial weight of all particles is assumed to be equal.

If you choose `'Uniform'`, the initial set of particles are distributed per the uniform distribution, where you specify the upper and lower **State bounds**. The initial weight of all particles is assumed to be equal.

`'Custom'` allows you to specify your own set of initial particles and their weights. You can use arbitrary probability distributions for **Particles** and **Weights** to initialize the filter.

Programmatic Use

Block Parameter: `InitialDistribution`
Type: character vector
Values: `'Gaussian'`, `'Uniform'`, `'Custom'`
Default: `'Gaussian'`

`Mean` — Initial mean value of particles

[0;0] (default) | vector

Initial mean value of particles, specified as a vector. The number of states to be estimated defines the length of the vector.

Dependencies

This parameter is available if in the **System model** tab, the **Distribution** parameter is set to `Gaussian`.

Programmatic Use

Block Parameter: `InitialMean`
Type: array
Default: `[0,0]`

`Covariance` — Initial covariance of particles

1 (default) | scalar | vector | matrix

Initial covariance of particles, specified as a scalar, vector, or matrix.

If **Covariance** is specified as:

- A scalar, then it must be positive. The covariance is assumed to be a [*Ns Ns*] matrix with this scalar on the diagonals. Here, *Ns* is the number of states.
- A vector, then each element must be positive. The covariance is assumed to be a [*Ns Ns*] matrix with the elements of the vector on the diagonals.
- A matrix, then it must be positive semidefinite.

Dependencies

This parameter is available if in the **System model** tab, the **Distribution** parameter is set to `Gaussian`.

Programmatic Use
Block Parameter: `InitialCovariance`
Type: scalar, vector, or matrix
Default: `1`

`Circular Variables` — Circular variables used for state estimation
0 (default) | scalar | vector

Circular variables used for state estimation, specified as a scalar, or *Ns*-element vector, where *Ns* is the number of states.

If **Circular Variables** is specified as a scalar, the software extends it to a vector where each element is equal to this scalar. Circular (or angular) distributions use a probability density function with a range of [`-π π`]. Use circular variables if some of the states in your system represent angular quantities like the orientation of an object.

Programmatic Use
Block Parameter: `CircularVariables`
Type: scalar, vector
Default: `0`

`State Orientation` — Orientation of input system states
`'column'` (default) | `'row'`

Orientation of system states, specified as `'column'` or `'row'`.

If **State Orientation** is specified as:

- `'column'`, then the first input argument to the state transition and measurement likelihood function is [*Ns Np*]. In this case, i^{th} column of this matrix is the i^{th} particle (state hypothesis). Also, the states estimates **xhat** is output as a [*Ns* 1] vector. Here, *Ns* is the number of states, and *Np* is the number of particles.
- `'row'`, then the first input argument to the state transition and measurement likelihood function is [*Np Ns*], and each row of this matrix contains a particle. Also, the states estimates **xhat** is output as a [1 *Ns*] vector.

Programmatic Use
Block Parameter: `StateOrientation`
Type: character vector
Values: `'column'`, `'row'`
Default: `'column'`

`State bounds` — Initial bounds on system states

[-3 3;-3 3] (default) | array

Initial bounds on system states, specified as an *Ns*-by-2 array, where *Ns* is the number of states.

The i^{th} row lists the lower and upper bound of the uniform distribution for the initial distribution of particles of the i^{th} state.

Dependencies

This parameter is available if in the **System model** tab, the **Distribution** parameter is set to `Uniform`.

Programmatic Use

Block Parameter: `InitialStateBounds`
Type: array
Default: `[-3 3;-3 3]`

`Particles` — Custom particle distribution for state estimation

[] (default) | array

Custom particle distribution for state estimation, specified as an *Ns*-by-*Np* or *Np*-by-*Ns* array. *Ns* is the number of states of the system, and *Np* is the number of particles.

- If the `StateOrientation` parameter is specified as `'column'`, then **Particles** is an *Ns*-by-*Np* array.
- If the `StateOrientation` parameter is specified as `'row'`, then **Particles** is an *Np*-by-*Ns* array.

Dependencies

This parameter is available if in the **System model** tab, the **Distribution** parameter is set to `Custom`.

Programmatic Use

Block Parameter: `InitialParticles`
Type: array
Default: `[]`

`Weights` — Custom particle weight values for state estimation

[] (default) | positive vector

Custom particle weight values for state estimation, specified as a 1-by-*Np* or *Np*-by-1 positive vector, where *Np* is the number of particles used for state estimation.

- If the `StateOrientation` parameter is specified as `'column'`, then **Weights** is a 1-by-*Np* vector. Each weight in the vector is associated with the particle in the same column in the `Particles` array.
- If the `StateOrientation` parameter is specified as `'row'`, then **Weights** is a *Np*-by-1 vector. Each weight in the vector is associated with the particle in the same row in the `Particles` array.

Dependencies

This parameter is available if in the **System model** tab, the **Distribution** parameter is set to `Custom`.

Programmatic Use
Block Parameter: `InitialWeights`
Type: positive vector
Default: `[]`

Measurement

`Function` — Measurement likelihood function name

`'vdpMeasurementLikelihoodFcn'` (default) | function name

The measurement likelihood function calculates the likelihood of particles (state hypotheses) using the sensor measurements. For each state hypothesis (particle), the function first calculates an *Nm*-element measurement hypothesis vector. Then the likelihood of each measurement hypothesis is calculated based on the sensor measurement and the measurement noise probability distribution. this function has the syntax:

```
likelihood = h(particles, measurement, param1, param2, ...)
```

where, *likelihood* is an *Np*-element vector, where *Np* is the number of particles. *particles* have dimensions *Ns*-by-*Np* if **State Orientation** is specified as `'column'`, or *Np*-by-*Ns* if **State Orientation** is specified as `'row'`. *measurement* is an *Nm*-element vector where, *Nm* is the number of measurements your sensor provides. *param_i* represents optional input arguments you may specify. For more information on optional input arguments, see “MeasurementLikelihoodFcn1Inputs,...,MeasurementLikelihoodFcn5Inputs” on page 2-0 .

You create the measurement likelihood function and specify the function name in **Function**. For example, if `vdpMeasurementLikelihoodFcn.m` is the measurement likelihood function that you created and saved, specify **Function** as `'vdpMeasurementLikelihoodFcn'`.

You can create **Function** using a Simulink Function block or as a MATLAB function (`.m` file).

- You can use a MATLAB function only if *h* has zero or one additional input argument `param_i` other than **Particles** and **Measurement**.

  The software generates an additional input port **MeasurementLikelihoodFcn*i*Inputs** to specify this argument for the i^{th} measurement likelihood function, and click **Apply**.
- If you are using a Simulink Function block, specify `x` and `y` using Argument Inport blocks and the additional inputs `param_i` using Inport blocks in the Simulink Function block. You do not provide `param_i` to the Particle Filter block.

If you have multiple sensors in your system, you can specify multiple measurement likelihood functions. You can specify up to five measurement likelihood functions using the **Add Measurement** button. To remove measurement likelihood functions, use **Remove Measurement**.

Programmatic Use
Block Parameter: `MeasurementLikelihoodFcn1`, `MeasurementLikelihoodFcn2`, `MeasurementLikelihoodFcn3`, `MeasurementLikelihoodFcn4`, `MeasurementLikelihoodFcn5`
Type: character vector, string
Default: `'vdpMeasurementLikelihoodFcn'`

`Add Enable Port` — Enable correction of estimated states only when measured data is available

`off` (default) | `on`

Suppose that measured output data is not available at all time points at the port **y1** that corresponds to the first measurement likelihood function. To generate an input port **Enable1**, select **Add Enable**

port. Use a signal at this port to enable the correction of estimated states only when measured data is available. Similarly, if measured output data is not available at all time points at the port **y*i*** for the i^{th} measurement likelihood function, select the corresponding **Add Enable port**.

Programmatic Use
Block Parameter: `HasMeasurementEnablePort1`, `HasMeasurementEnablePort2`, `HasMeasurementEnablePort3`, `HasMeasurementEnablePort4`, `HasMeasurementEnablePort5`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

Resampling

`Resampling method` — Method used for particle resampling
`'Multinomial'` (default) | `'Systemic'` | `'Stratified'`

Method used for particle resampling, specified as one of the following:

- `'Multinomial'` — Multinomial resampling, also called simplified random sampling, generates `N` random numbers independently from the uniform distribution in the open interval `(0,1)` and uses them to select particles proportional to their weight.
- `'Stratified'` — Stratified resampling divides the whole population of particles into subsets called strata. It pre-partitions the `(0,1)` interval into `N` disjoint sub-intervals of size `1/N`. The random numbers are drawn independently in each of these sub-intervals and the sample indices chosen in the strata.
- `'Systematic'` — Systematic resampling is similar to stratified resampling as it also makes use of strata. One distinction is that it only draws one random number from the open interval `(0,1/N)` and the remaining sample points are calculated deterministically at a fixed `1/N` step size.

Programmatic Use
Block Parameter: `ResamplingMethod`
Type: character vector
Values: `'Multinomial'`, `'Systemic'`, `'Stratified'`
Default: `'Multinomial'`

`Trigger method` — Method to determine when resampling occurs
`'Ratio'` (default) | `'Interval'`

Method to determine when resampling occurs, specified as either `'Ratio'` or `'Interval'`. The `'Ratio'` value triggers resampling based on the ratio of effective total particles. The `'Interval'` value triggers resampling at regular time steps of the particle filter operation.

Programmatic Use
Block Parameter: `TriggerMethod`
Type: character vector
Values: `'Ratio'`, `'Interval'`
Default: `'Ratio'`

`Minimum effective particle ratio` — Minimum desired ratio of the effective number of particles to the total number of particles
0.5 (default) | positive scalar

Minimum desired ratio of the effective number of particles to the total number of particles, specified as a positive scalar. The effective number of particles is a measure of how well the current set of particles approximates the posterior distribution. A lower effective particle ratio implies that a lower number of particles are contributing to the estimation and resampling is required.

If the ratio of the effective number of particles to the total number of particles falls below the minimum effective particle ratio, a resampling step is triggered.

Specify minimum effective particle ratio as any value from 0 through 1.

Dependencies

This parameter is available if in the **System model** tab, the **Trigger method** parameter is set to `Ratio`.

Programmatic Use
Block Parameter: `MinEffectiveParticleRatio`
Type: scalar
Values: Range `[0,1]`
Default: `0.5`

`Sampling Interval` — Fixed interval between resampling

1 (default) | positive scalar integer

Fixed interval between resampling, specified as a positive scalar integer. The sampling interval determines during which correction steps the resampling is executed. For example, a value of two means the resampling is executed every second correction step. A value of `inf` means that resampling is never executed.

Dependencies

This parameter is available if in the **System model** tab, the **Trigger method** parameter is set to `Interval`.

Programmatic Use
Block Parameter: `SamplingInterval`
Type: positive scalar integer
Default: `1`

Random Number Generator Options

`Randomness` — Whether the random numbers are repeatable

`'Repeatable'` (default) | `'Not repeatable'`

Whether the random numbers are repeatable, specified as either `'Repeatable'` or `'Not repeatable'`. If you want to be able to produce the same result more than once, set **Randomness** to `'Repeatable'`, and specify the same random number generator seed value in **Seed**.

Programmatic Use
Block Parameter: `Randomness`
Type: character vector
Values: `'Repeatable'`, `'Not repeatable'`
Default: `'Repeatable'`

Seed — Seed value for repeatable random numbers

0 (default) | scalar

Seed value for repeatable random numbers, specified as a scalar.

Dependencies

This parameter is available if in the **System model** tab, the **Randomness** parameter is set to `'Repeatable'`.

Programmatic Use

Block Parameter: `Seed`
Type: scalar
Default: `0`

Settings

`Data type` — Data type for block parameters

`double` (default) | `single`

Use this parameter to specify the data type for all block parameters.

Programmatic Use

Block Parameter: `DataType`
Type: character vector
Values: `'single'`, `'double'`
Default: `'double'`

`Sample time` — Block sample time

`1` (default) | positive scalar

Block sample time, specified as a positive scalar.

Use the **Sample time** parameter if your state transition and all measurement likelihood functions have the same sample time. Otherwise, select the **Enable multirate operation** option in the **Multirate** tab, and specify sample times in the same tab.

Dependencies

This parameter is available if in the **Block output, Multirate** tab, the **Enable multirate operation** parameter is `off`.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector, string
Default: `'1'`

Block Outputs, Multirate Tab

Outputs

`State Estimation Method` — Method used for extracting a state estimate from particles

`'Mean'` (default) | `'MaxWeight'` | `'None'`

Method used for extracting a state estimate from particles, specified as one of the following:

- `'Mean'` — The Particle Filter block outputs the weighted mean of the particles, depending on the parameters **Weights** and **Particles**, as the state estimate.
- `'Maxweight'` — The Particle Filter block outputs the particle with the highest weight as the state estimate.
- `'None'` — Use this option to implement a custom state estimation method by accessing all particles using the **Output all particles** parameter from the **Block outputs, Multirate** tab.

Programmatic Use
Block Parameter: `StateEstimationMethod`
Type: character vector, string
Values: `'Mean'`, `'MaxWeight'`, `'None'`
Default: `'Mean'`

`Output all particles` — Output all particles

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

If you select this parameter, an output port for particles used in the estimation, **Particles** is generated in the block.

- If the `StateOrientation` parameter is specified as `'column'`, then the particles are output as an *Ns*-by-*Np* array. *Ns* is the number of states of the system, and *Np* is the number of particles.
- If the `StateOrientation` parameter is specified as `'row'`, then the particles are output as an *Np*-by-*Ns* array.

Programmatic Use
Block Parameter: `OutputParticles`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Output weights` — Output particle weights

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

If you select this parameter, an output port for particle weights used in the estimation, **Weights** is generated in the block.

- If the `StateOrientation` parameter is specified as `'column'`, then the particle weights are output as a 1-by-*Np* vector. Here, where each weight is associated with the particle in the same column in the `Particles` array. *Np* is the number of particles used for state estimation.
- If the `StateOrientation` parameter is specified as `'row'`, then the particle weights are output as a *Np*-by-1 vector.

Programmatic Use
Block Parameter: `OutputWeights`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Output state estimation error covariance` — Output state estimation error covariance

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

If you select this parameter, a state estimation error covariance output port, **P** is generated in the block.

Dependencies

This parameter is available if in the **Block outputs, Multirate** tab, the **State estimation method** parameter is set to `'Mean'`.

Programmatic Use
Block Parameter: `OutputStateCovariance`
Type: character vector

Values: `'off'`, `'on'`
Default: `'off'`

`Use the current measurements to improve state estimates` — Option to use current measurements for state estimation

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

When this parameter is selected, the block outputs the corrected state estimate $\hat{x}[k|k]$ at time step `k`, estimated using measured outputs until time `k`. If you clear this parameter, the block returns the predicted state estimate $\hat{x}[k|k-1]$ for time `k`, estimated using measured output until a previous time `k-1`. Clear this parameter if your filter is in a feedback loop and there is an algebraic loop in your Simulink model.

Programmatic Use
Block Parameter: `UseCurrentEstimator`
Type: character vector
Values: `'on'`, `'off'`
Default: `'on'`

Multirate

`Enable multirate operation` — Enable specification of different sample times for state transition and measurement likelihood functions

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

Select this parameter if the sample times of the state transition or any of the measurement likelihood functions differ from the rest. You specify the sample times in the **Multirate** tab, in **Sample time**.

Programmatic Use
Block Parameter: `EnableMultirate`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Sample times` — State transition and measurement likelihood function sample times

positive scalar

If the sample times for state transition and measurement likelihood functions are different, specify **Sample time**. Specify the sample times for the measurement functions as positive integer multiples of the state transition sample time. The sample times you specify correspond to the following input ports:

- Ports corresponding to state transition function — Additional input to state transition function **StateTransitionFcnInputs**. The sample times of these ports must always equal the state transition function sample time, but can differ from the sample time of the measurement likelihood functions.
- Ports corresponding to i^{th} measurement likelihood function — Measured output **y*i***, additional input to measurement likelihood function **MeasurementLikelihoodFcn*i*Inputs**, enable signal at port **Enable*i***. The sample times of these ports for the same measurement likelihood function must always be the same, but can differ from the sample time for the state transition function and other measurement likelihood functions.

Dependencies

This parameter is available if in the **Block outputs, Multirate** tab, the **Enable multirate operation** parameter is `on`.

Programmatic Use
Block Parameter: `StateTransitionFcnSampleTime`, `MeasurementLikelihoodFcn1SampleTime1`, `MeasurementLikelihoodFcn1SampleTime2`, `MeasurementLikelihoodFcn1SampleTime3`, `MeasurementLikelihoodFcn1SampleTime4`, `MeasurementLikelihoodFcn1SampleTime5`
Type: character vector, string
Default: `'1'`

References

[1] T. Li, M. Bolic, P.M. Djuric, "Resampling Methods for Particle Filtering: Classification, implementation, and strategies," *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 70-86, May 2015.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The state transition and measurement likelihood functions that you specify must use only the MATLAB commands and Simulink blocks that support code generation. For a list of blocks that support code generation, see “Simulink Built-In Blocks That Support Code Generation” (Simulink Coder). For a list of commands that support code generation, see “Functions and Objects Supported for C/C++ Code Generation” (MATLAB Coder).

See Also

Blocks
Kalman Filter | Unscented Kalman Filter | Extended Kalman Filter

Functions
`particleFilter` | `extendedKalmanFilter` | `unscentedKalmanFilter`

Topics
“Parameter and State Estimation in Simulink Using Particle Filter Block”
“Validate Online State Estimation in Simulink”
“Troubleshoot Online State Estimation”
“Estimate States of Nonlinear System with Multiple, Multirate Sensors”

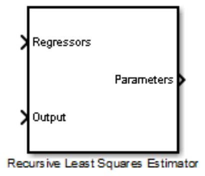
External Websites
Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

Introduced in R2018a

Recursive Least Squares Estimator

Estimate model coefficients using recursive least squares (RLS) algorithm

Library: System Identification Toolbox / Estimators



Description

The Recursive Least Squares Estimator estimates the parameters of a system using a model that is linear in those 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 can provide both infinite-history [1] and finite-history [2] (also known as sliding-window), estimates for θ. For more information on these methods, see “Recursive Algorithms for Online Parameter Estimation”.

The block supports several estimation methods and data input formats. Configurable options in the block include:

- Sample-based or frame-based data format — See the **Input Processing** parameter.
- Infinite-history or finite- history estimation — See the **History** parameter.
- Multiple infinite-history estimation methods — See the **Estimation Method** parameter.
- Initial conditions, enable flag, and reset trigger — See the **Initial Estimate**, **Add enable port**, and **External Reset** parameters.

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.

For example, suppose that 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.

Ports

Input

`Regressors` — Regressors signal

vector | matrix

Regressors input signal $H(t)$. The **Input Processing** and **Number of Parameters** parameters define the dimensions of the signal:

- Sample-based input processing and N estimated parameters — 1-by-N vector

- Frame-based input processing with *M* samples per frame and *N* estimated parameters — *M*-by-*N* matrix

Data Types: `single` | `double`

`Output` — Measured output
scalar | vector

Measured output signal *y*(*t*). The **Input Processing** parameter defines the dimensions of the signal:

- Sample-based input processing — Scalar
- Frame-based input processing with *M* samples per frame — *M*-by-1 vector

Data Types: `single` | `double`

`Enable` — Enable estimation updates
`true` (default) | `false`

External signal that allows you to enable and disable estimation updates. If the signal value is:

- `true` — Estimate and output the parameter values for the time step.
- `false` — Do not estimate the parameter values, and output the most recent previously estimated value.

Dependencies

To enable this port, select the **Add enable port** parameter.

Data Types: `single` | `double` | `Boolean` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

`Reset` — Reset trigger
scalar

Reset parameter estimation to its initial conditions. The value of the **External reset** parameter determines the trigger type. The trigger type dictates whether the reset occurs on a signal that is rising, falling, either rising or falling, level, or on level hold.

Dependencies

To enable this port, select any option other than `None` in the **External reset** dropdown.

Data Types: `single` | `double` | `Boolean` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`InitialParameters` — Initial parameter estimates
vector

Initial parameter estimates, supplied from a source external to the block. The block uses this inport at the beginning of the simulation or when you trigger an algorithm reset using the **Reset** signal.

The **Number of Parameters** parameter defines the dimensions of the signal. If there are *N* parameters, the signal is *N*-by-1.

Dependencies

To enable this port, set **History** to `Infinite` and **Initial Estimate** to `External`.

Data Types: `single` | `double`

`InitialCovariance` — Initial covariance of parameters

positive scalar (default) | vector of positive scalars | symmetric positive-definite matrix

Initial parameter covariances, supplied from a source external to the block. For details, see the **Parameter Covariance Matrix** parameter.The block uses this inport at the beginning of the simulation or when you trigger an algorithm reset using the **Reset** signal.

Dependencies

To enable this port, set the following parameters:

- **History** to `Infinite`
- **Estimation Method** to `Forgetting Factor` or `Kalman Filter`
- **Initial Estimate** to `External`

Data Types: `single` | `double`

`InitialRegressors` — Initial values of the regressors

matrix

Initial values of the regressors in the initial data window when using finite-history (sliding-window) estimation, supplied from an external source. The **Window length** parameter *W* and the **Number of Parameters** parameter *N* define the dimensions of this signal, which is *W*-by-*N*.

The **InitialRegressors** signal controls the initial behavior of the algorithm. The block uses this inport at the beginning of the simulation or whenever the **Reset** signal triggers.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Dependencies

To enable this port, set **History** to `Finite` and **Initial Estimate** to `External`.

Data Types: `single` | `double`

`InitialOutputs` — Initial value of the measured output buffer

vector

Initial set of output measurements when using finite-history (sliding-window) estimation, supplied from an external source. The signal to this port must be a *W*-by-1 vector, where *W* is the window length.

The **InitialOutputs** signal controls the initial behavior of the algorithm. The block uses this inport at the beginning of the simulation or whenever the **Reset** signal triggers.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Dependencies

To enable this port, set **History** to `Finite`, and **Initial Estimate** to `External`.

Data Types: `single` | `double`

Output

`Parameters` — Estimated parameters
vector

Estimated parameters $\theta(t)$, returned as an N-by-1 vector where N is the number of parameters.

Data Types: `single` | `double`

`Error` — Estimation error
scalar | vector

Estimation error, returned as:

- Scalar — Sample-based input processing
- M-by-1 vector — Frame-based input processing with M samples per frame

Dependencies

To enable this port, select the **Output estimation error** parameter.

Data Types: `single` | `double`

`Covariance` — Parameter estimation error covariance P
matrix

Parameter estimation error covariance P, returned as an N-by-N matrix, where N is the number of parameters. For details, see the **Output Parameter Covariance Matrix** parameter.

Dependencies

To enable this port:

- If **History** is `Infinite`, set **Estimation Method** to `Forgetting Factor` or `Kalman Filter`.
- Whether **History** is `Infinite` or `Finite`, select the **Output parameter covariance matrix** parameter.

Data Types: `single` | `double`

Parameters

Model Parameters

`Initial Estimate` — Source of initial parameter estimates
`None` (default) | `Internal` | `External`

Specify how to provide initial parameter estimates to the block:

- `None` — Do not specify initial estimates.
  - If **History** is `Infinite`, the block uses `1` as the initial parameter estimate.
  - If **History** is `Finite`, the block calculates the initial parameter estimates from the initial **Regressors** and **Outputs** signals.

Specify **Number of Parameters**, and also, if **History** is `Infinite`, **Parameter Covariance Matrix**.

- `Internal` — Specify initial parameter estimates internally to the block
  - If **History** is `Infinite`, specify the **Initial Parameter Values** and **Parameter Covariance Matrix** parameters.
  - If **History** is `Finite`, specify the **Number of Parameters**, the **Initial Regressors**, and the **Initial Outputs** parameters.
- `External` — Specify initial parameter estimates as an input signal to the block.

  Specify the **Number of Parameters** parameter. Your setting for the **History** parameter determines which additional signals to connect to the relevant ports:

  - If **History** is `Infinite` — **InitialParameters** and **InitialCovariance**
  - If **History** is `Finite` — **InitialRegressors** and **InitialOutputs**

Programmatic Use

Block Parameter: `InitialEstimateSource`
Type: character vector, string
Values: `'None'`, `'Internal'`, `'External'`
Default: `'None'`

`Number of Parameters` — Number of parameters to estimate

2 (default) | positive integer

Specify the number of parameters to estimate in the model, equal to the number of elements in the parameter $\theta(t)$ vector.

Dependencies

To enable this parameter, set either:

- **History** to `Infinite` and **Initial Estimate** to either `None` or `External`
- **History** to `Finite`

An alternative way to specify the number of parameters N to estimate is by using the **Initial Parameter Values** parameter, for which you define an initial estimate vector with N elements. This approach covers the one remaining combination, where **History** is `Infinite` and **Initial Estimate** is `Internal`. For more information, see **Initial Parameter Values**.

Programmatic Use

Block Parameter: `InitialParameterData`
Type: positive integer
Default: 2

`Parameter Covariance Matrix` — Initial parameter covariance

`1e4` (default) | scalar | vector | matrix

Specify **Parameter Covariance Matrix** as a:

- Real positive scalar, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real positive scalars, $[\alpha_1,\ldots,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,\ldots,\alpha_N]$ as the diagonal elements.

- *N*-by-*N* symmetric positive-definite matrix.

Here, *N* is the number of parameters to be estimated.

Dependencies

To enable this parameter, set the following parameters:

- **History** to `Infinite`
- **Initial Estimate** to `None` or `Internal`
- **Estimation Method** to `Forgetting Factor` or `Kalman Filter`

Programmatic Use
Block Parameter: `P0`
Type: scalar, vector, or matrix
Default: `1e4`

`Initial Parameter Values` — Initial values of the parameters to estimate

`[1 1]` (default) | vector

Specify initial parameter values as a vector of length *N*, where *N* is the number of parameters to estimate.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Initial Estimate** to `Internal`.

Programmatic Use
Block Parameter: `InitialParameterData`
Type: real vector
Default: `[1 1]`

`Initial Regressors` — Initial values of the regressors buffer

`0` (default) | matrix

Specify the initial values of the regressors buffer when using finite-history (sliding window) estimation. The **Window length** parameter *W* and the **Number of Parameters** parameter *N* define the dimensions of the regressors buffer, which is *W*-by-*N*.

The **Initial Regressors** parameter controls the initial behavior of the algorithm. The block uses this parameter at the beginning of the simulation or whenever the **Reset** signal triggers.

When the initial value is set to `0`, the block populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Dependencies

To enable this parameter, set **History** to `Finite` and **Initial Estimate** to `Internal`.

Programmatic Use
Block Parameter: `InitialRegressors`
Type: real matrix

Default: `0`

`Initial Outputs` — Initial values of the measured outputs buffer

`0` (default) | vector

Specify initial values of the measured outputs buffer when using finite-history (sliding-window) estimation. This parameter is a *W*-by-1 vector, where *W* is the window length.

When the initial value is set to `0`, the block populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

The **Initial Outputs** parameter controls the initial behavior of the algorithm. The block uses this parameter at the beginning of the simulation or whenever the **Reset** signal triggers.

Dependencies

To enable this parameter, set **History** to `Finite` and **Initial Estimate** to `Internal` .

Programmatic Use

Block Parameter: `InitialOutputs`
Type: real vector
Default: `0`

Input Processing and Sample Time

`Input Processing` — Choose sample-based or frame-based processing

`Sample-based` (default) | `Frame-based`

- `Sample-based` processing operates on signals streamed one sample at a time.
- `Frame-based` processing operates on signals containing samples from multiple time steps. Many machine sensor interfaces package multiple samples and transmit these samples together in frames. `Frame-based` processing allows you to input this data directly without having to first unpack it.

Specifying frame-based data adds an extra dimension of *M* to some of your data inports and outports, where *M* is the number of time steps in a frame. These ports are:

- **Regressors**
- **Output**
- **Error**

For more information, see the port descriptions in “Ports” on page 2-58.

Programmatic Use

Block Parameter: `InputProcessing`
Type: character vector, string
Values: `'Sample-based'`, `'Frame-based'`
Default: `'Sample-based'`

`Sample Time` — Block sample time

`-1` (default) | positive scalar

Specify the data sample time, whether by individual samples for sample-based processing (t_s), or by frames for frame-based processing ($t_f = Mt_s$), where M is the frame length. When you set **Sample Time** to its default value of -1, the block inherits its t_s or t_f based on the signal.

Specify **Sample Time** as a positive scalar to override the inheritance.

Programmatic Use
Block Parameter: `Ts`
Type: real scalar
Default: `-1`

Algorithm and Block Options

Algorithm Options

`History` — Choose infinite or finite data history

`Infinite` (default) | `Finite`

The **History** parameter determines what type of recursive algorithm you use:

- `Infinite` — Algorithms in this category aim to produce parameter estimates that explain all data since the start of the simulation. These algorithms retain the history in a data summary. The block maintains this summary within a fixed amount of memory that does not grow over time.

  The block provides multiple algorithms of the `Infinite` type. Selecting this option enables the **Estimation Method** parameter with which you specify the algorithm.
- `Finite` — Algorithms in this category aim to produce parameter estimates that explain only a finite number of past data samples. The block uses all of the data within a finite window, and discards data once that data is no longer within the window bounds. This method is also called sliding-window estimation.

  Selecting this option enables the **Window Length** parameter that sizes the sliding window.

For more information on recursive estimation methods, see “Recursive Algorithms for Online Parameter Estimation”

Programmatic Use
Block Parameter: `History`
Type: character vector, string
Values: `'Infinite'`, `'Finite'`
Default: `'Infinite'`

`Window Length` — Window size for finite sliding-window estimation

`200` (default) | positive integer

The **Window Length** parameter determines the number of time samples to use for the sliding-window estimation method. Choose a window size that balances estimation performance with computational and memory burden. Sizing factors include the number and time variance of the parameters in your model. Always specify **Window Length** in samples, even if you are using frame-based input processing.

Window Length must be greater than or equal to the number of estimated parameters.

Suitable window length is independent of whether you are using sample-based or frame-based input processing. However, when using frame-based processing, **Window Length** must be greater than or equal to the number of samples (time steps) contained in the frame.

Dependencies

To enable this parameter, set **History** to `Finite`.

Programmatic Use
Block Parameter: `WindowLength`
Type: positive integer
Default: `200`

`Estimation Method` — Recursive estimation algorithm

`Forgetting Factor` (default) | `Kalman Filter` | `Normalized Gradient` | `Gradient`

Specify the estimation algorithm when performing infinite-history estimation. When you select any of these methods, the block enables additional related parameters.

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and normalized gradient methods. However, these more intensive methods have better convergence properties than the gradient methods. For more information about these algorithms, see "Recursive Algorithms for Online Parameter Estimation".

Programmatic Use
Block Parameter: `EstimationMethod`
Type: character vector, string
Values: `'Forgetting Factor'`,`'Kalman Filter'`,`'Normalized Gradient'`,`'Gradient'`
Default: `'Forgetting Factor'`

`Forgetting Factor` — Discount old data using forgetting factor

1 (default) | positive scalar in (0 1] range

The forgetting factor λ specifies if and how much old data is discounted in the estimation. 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 [0.98 0.995] range.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Estimation Method** to `Forgetting Factor`.

Programmatic Use
Block Parameter: `AdaptationParameter`
Type: scalar
Values: (0 1] range
Default: `1`

`Process Noise Covariance` — Process noise covariance for Kalman filter estimation method

1 (default) | nonnegative scalar | vector of nonnegative scalars | symmetric positive semidefinite matrix

Process Noise Covariance prescribes the elements and structure of the noise covariance matrix for the Kalman filter estimation. Using N as the number of parameters to estimate, specify the **Process Noise Covariance** as one of the following:

- Real nonnegative scalar, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ as the diagonal elements.
- N-by-N symmetric positive semidefinite matrix.

The Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. **Process Noise Covariance** is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to constant coefficients, or parameters. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, expect the larger values to result in noisier parameter estimates. The default value is 1.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Estimation Method** to `Kalman Filter`.

Programmatic Use

Block Parameter: `AdaptationParameter`
Type: scalar, vector, matrix
Default: `1`

`Adaptation Gain` — Adaptation gain specification for gradient estimation methods

`1` (default) | positive scalar

The adaptation gain γ scales the influence of new measurement data on the estimation results for the gradient and normalized gradient methods. When your measurements are trustworthy, or in other words have a high signal-to-noise ratio, specify a larger value for γ. However, setting γ too high can cause the parameter estimates to diverge. This divergence is possible even if the measurements are noise free.

When **Estimation Method** is `NormalizedGradient`, **Adaptation Gain** should be less than 2. With either gradient method, if errors are growing in time (in other words, estimation is diverging), or parameter estimates are jumping around frequently, consider reducing **Adaptation Gain**.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Estimation Method** to `Normalized Gradient` or to `Gradient`.

Programmatic Use

Block Parameter: `AdaptationParameter`
Type: scalar
Default: `1`

`Normalization Bias` — Bias for adaptation gain scaling for normalized gradient estimation method

`eps` (default) | 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, the near-zero denominator can cause jumps in the estimated parameters. **Normalization Bias** is the term introduced to the denominator to prevent these jumps. Increase **Normalization Bias** if you observe jumps in estimated parameters.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Estimation Method** to `Normalized Gradient`.

Programmatic Use
Block Parameter: `NormalizationBias`
Type: scalar
Default: `eps`

Block Options

`Output estimation error` — Add Error outport to block

`off` (default) | on

Use the **Error** outport signal to validate the estimation. For a given time step *t*, the estimation error *e*(*t*) is calculated as:

$$e(t) = y(t) - y_{est}(t),$$

where *y*(*t*) is the measured output that you provide, and $y_{est}(t)$ is the estimated output using the regressors *H*(*t*) and parameter estimates *θ*(*t*-1).

Programmatic Use
Block Parameter: `OutputError`
Type: character vector, string
Values: `'off'`,`'on'`,
Default: `'off'`

`Output parameter covariance matrix` — Add covariance outport to block

`off` (default) | on

Use the **Covariance** outport signal to examine parameter estimation uncertainty. The software computes parameter covariance `P` assuming that the residuals, *e*(*t*), are white noise, and the variance of these residuals is 1.

The interpretation of `P` depends on the estimation approach you specify in **History** and **Estimation Method** as follows:

- If **History** is `Infinite`, then your **Estimation Method** selection results in:
  - `Forgetting Factor` — (R_2/2)`P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals. The block outputs the residuals in the **Error** port.
  - `Kalman Filter` — R_2`P` is the covariance matrix of the estimated parameters, and R_1 / R_2 is the covariance matrix of the parameter changes. Here, R_1 is the covariance matrix that you specify in **Parameter Covariance Matrix**.
  - `Normalized Gradient` or `Gradient` — Covariance *P* is not available.
- If **History** is `Finite` (sliding-window estimation) — $R_2\,P$ is the covariance of the estimated parameters. The sliding-window algorithm does not use this covariance in the parameter-estimation process. However, the algorithm does compute the covariance for output so that you can use it for statistical evaluation.

Programmatic Use
Block Parameter: `OutputP`
Type: character vector, string
Values: `'off'`,`'on'`
Default: `'off'`

`Add enable port` — Add Enable inport to block

`off` (default) | on

Use the **Enable** signal to provide 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.

- You can use this option, for example, when or if:
  - Your regressors or output signal become too noisy, or do not contain information at some time steps
  - Your system enters a mode where the parameter values do not change in time

Programmatic Use
Block Parameter: `AddEnablePort`
Type: character vector, string
Values: `'off'`,`'on'`
Default: `'off'`

`External reset` — Specify trigger for external reset

`None` (default) | `Rising` | `Falling` | `Either` | `Level` | `Level hold`

Set the **External reset** parameter to both add a **Reset** inport and specify the inport signal condition that triggers a reset of algorithm states to their specified initial values. Reset the estimation, for example, if parameter covariance is becoming too large because of lack of either sufficient excitation or information in the measured signals.

Suppose that 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.

If the block is disabled at *t* and you reset the block, the block outputs the values specified in **Initial Estimate**.

Specify this option as one of the following:

- `None` — Algorithm states and estimated parameters are not reset.
- `Rising` — Trigger 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` — Trigger 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` — Trigger reset when the control signal is either rising or falling.
- `Level` — Trigger 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` — Trigger 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.

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector, string
Values: `'None'`,`'Rising'`,`'Falling'`, `'Either'`, `'Level'`, `'Level hold'`
Default: `'None'`

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999, pp. 363–369.

[2] Zhang, Q. "Some Implementation Aspects of Sliding Window Least Squares Algorithms." *IFAC Proceedings*. Vol. 33, Issue 15, 2000, pp. 763-768.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

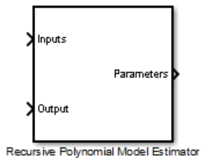
Recursive Polynomial Model Estimator | Kalman Filter

Topics
“Estimate Parameters of System Using Simulink Recursive Estimator Block”
“Online Recursive Least Squares Estimation”
“Preprocess Online Parameter Estimation Data in Simulink”
“Validate Online Parameter Estimation Results in Simulink”
“Generate Online Parameter Estimation Code in Simulink”
“Recursive Algorithms for Online Parameter Estimation”

Introduced in R2014a

Recursive Polynomial Model Estimator

Estimate input-output and time-series polynomial model coefficients
Library: System Identification Toolbox / Estimators



Description

Model Structures

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 — $A(q)y(t) = B(q)u(t - n_k) + e(t)$
- ARMAX — $A(q)y(t) = B(q)u(t - n_k) + C(q)e(t)$
- OE — $y(t) = \frac{B(q)}{F(q)}u(t - n_k) + e(t)$
- BJ — $y(t) = \frac{B(q)}{F(q)}u(t - n_k) + \frac{C(q)}{D(q)}e(t)$

q is the time-shift operator and nk is the input 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 correspond to the maximum number of time shifts, as represented by the exponent of q. For instance, the order na is represented in the $A(q)$ polynomial by:

$1 + a_1q^{-1} + a_2q^{-2} + \dots + a_{na}q^{-na}$.

An equivalent representation applies to the $C(q)$, $D(q)$, and $F(q)$ polynomials and their corresponding orders nc, nd, and nf.

The $B(q)$ polynomial is unique with respect to the others, because this polynomial operates on the input and contains the system zeros. For $B(q)$, the order nb is the order of the polynomial $B(q) + 1$:

$b_1 + b_2q^{-1} + b_3q^{-2} + \dots + b_{nb}q^{-(nb-1)}$.

The orders na, nb, nc, nd, nf, and input delay nk are known ahead of time. Specify these values as block parameters. Provide $u(t)$ and $y(t)$ through the **Inputs** and **Outputs** inports, respectively. The block estimates the set of $A(q)$, $B(q)$, $C(q)$, $D(q)$, and $F(q)$ coefficients that the model structure uses and outputs them in the **Parameters** outport. During the estimation, the block constrains the estimated C, D, and F polynomials to a stable region with roots in the unit disk, while allowing the estimated *A* and *B* polynomials to be unstable. The **Parameters** outport provides a bus signal with the following elements:

- A — Vector containing [1 $a_1(t)$ ... $a_{na}(t)$].
- B — Vector containing [0_1 ... 0_{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 that you want to estimate the coefficients for the following SISO ARMAX model:

$y(t) + a_1y(t - 1) +...+a_{na}y(t - na) = b_1u(t - nk) + ... + b_{nb}u(t - nb - nk + 1) + e(t) + c_1e(t - 1) + ... + c_{nc}e(t - nc)$

y, *u*, *na*, *nb*, *nc*, and *nk* are known quantities that you provide to the block. For each time step, *t*, the block estimates the *A*, *B*, and *C* parameter values, constraining only the *C* polynomial to a stable region. The block then outputs these estimated values using the **Parameters** outport.

Block Capabilities

The block supports several estimation methods and data input formats. The block can provide both infinite-history [1] and finite-history [2] (also known as sliding-window) estimates for θ. Configurable options in the block include:

- Multiple inputs (ARX model structure only) — See the **Inputs** port.
- Sample-based or frame-based data format — See the **Input Processing** parameter.
- Multiple infinite-history estimation methods [1] — See the **Estimation Method** parameter.
- Infinite-history (all model structures) or finite-history (AR, ARX, or OE model structures only) — See the **History** parameter.
- Initial conditions, enable flag, and reset trigger — See the **Initial Estimate**, **Add enable port**, and **External Reset** parameters.

For more information on the estimation methods, see “Recursive Algorithms for Online Parameter Estimation”.

Ports

Input

`Inputs` — Input signal

vector | matrix

Input signal $u(t)$. The **Input Processing** parameter and the number of inputs *nu* define the dimensions of the signal. Only the ARX model structure can have multiple inputs, with *nu* greater than 1.

- Sample-based input processing and *nu* inputs — *nu*-by-1 vector
- Frame-based input processing with *M* samples per frame and *nu* inputs — *M*-by-*nu* matrix

Dependencies

To enable this port, set the **Model Structure** parameter to `ARX`, `ARMAX`, `BJ`, or `OE`.

Data Types: `single` | `double`

`Output` — Measured output signal
scalar | vector

Measured output signal *y*(*t*). The **Input Processing** parameter defines the dimensions of the signal:

- Sample-based input processing — Scalar
- Frame-based input processing with *M* samples per frame — *M*-by-1 vector

Data Types: `single` | `double`

`Enable` — Enable estimation updates
`true` (default) | `false`

External signal that allows you to enable and disable estimation updates. If the signal value is:

- `true` — Estimate and output the parameter values for the time step.
- `false` — Do not estimate the parameter values, and output the most recent previously estimated value.

Dependencies

To enable this port, select the **Add enable port** parameter.

Data Types: `single` | `double` | `Boolean` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

`Reset` — Reset trigger
scalar

Reset parameter estimation to its initial conditions. The value of the **External reset** parameter determines the trigger type. The trigger type dictates whether the reset occurs on a signal that is rising, falling, either rising or falling, level, or on level hold.

Dependencies

To enable this port, select any option other than `None` in the **External reset** dropdown.

Data Types: `single` | `double` | `Boolean` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`InitialParameters` — Initial parameter estimates
`bus` object

Initial parameter estimates, supplied from a source external to the block. The block uses this inport at the beginning of the simulation or when you trigger an algorithm reset using the **Reset** signal.

For information on the contents of the **InitialParameters** bus object, see the **Parameters** port description.

Dependencies

To enable this port, set **History** to `Infinite` and **Initial Estimate** to `External`.

Data Types: `single` | `double`

`InitialCovariance` — Initial covariance of parameters
positive scalar (default) | vector of positive scalars | symmetric positive-definite matrix

Initial parameter covariances, supplied from a source external to the block. For details, see the **Parameter Covariance Matrix** parameter. The block uses this inport at the beginning of the simulation or when you trigger an algorithm reset using the **Reset** signal.

Dependencies

To enable this port, set the following parameters:

- **History** to `Infinite`
- **Estimation Method** to `Forgetting Factor` or `Kalman Filter`
- **Initial Estimate** to `External`

Data Types: `single` | `double`

`InitialInputs` — Initial values of the inputs
matrix

Initial set of inputs when using finite-history (sliding-window) estimation, supplied from an external source.

- If **Model Structure** is `ARX`, then the signal to this port must be a (*W*–1+max(*nb*)+max(*nk*))-by-*nu* matrix, where *W* is the window length and *nu* is the number of inputs. *nb* is the vector of *B*(*q*) polynomial orders and *nk* is the vector of input delays.
- If **Model Structure** is `OE`, then the signal to this port must be a (*W*–1+*nb*+*nk*)-by-1 vector, where *W* is the window length. *nb* is the vector of *B*(*q*) polynomial orders and *nk* is the vector of input delays.

The block uses this inport at the beginning of the simulation or whenever the **Reset** signal triggers.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Dependencies

To enable this port, set:

- **History** to `Finite`
- **Model Structure** to `ARX` or `OE`
- **Initial Estimate** to `External`

Data Types: `single` | `double`

`InitialOutputs` — Initial values of the measured outputs buffer
vector

Initial set of output measurements when using finite-history (sliding-window) estimation, supplied from an external source.

- If **Model Structure** is `AR` or `ARX`, then the signal to this port must be a (*W*+*na*)-by-1 vector, where *W* is the window length and *na* is the polynomial order of *A*(*q*).

- If **Model Structure** is `OE`, then the signal to this port must be a (*W*+*nf*)-by-1 vector, where *W* is the window length and *nf* is the polynomial order of *F*(*q*).

The block uses this inport at the beginning of the simulation or whenever the **Reset** signal triggers.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

Dependencies

To enable this port, set:

- **History** to `Finite`
- **Model Structure** to `AR`, `ARX`, or `OE`
- **Initial Estimate** to `External`

Data Types: `single` | `double`

Output

`Parameters` — Estimated parameters
`bus` object

Estimated polynomial coefficients, returned as a bus. The bus contains an element for each of the *A*, *B*, *C*, *D*, and *F* polynomials that correspond to the structure that you specify in **Model Structure** (see “Model Structures” on page 2-71).

Each bus element is a vector signal containing the associated polynomial coefficients. For example, the *A* element contains [1 $a_1(t)$ ... $a_{na}(t)$].

Estimated *C*, *D*, and *F* values are constrained to be stable discrete-time polynomials. That is, these polynomials all have roots within the unit circle. Estimated *A* and *B* polynomials are allowed to be unstable.

Data Types: `single` | `double`

`Error` — Estimation error
scalar | vector

Estimation error, returned as:

- Scalar — Sample-based input processing
- *M*-by-1 vector — Frame-based input processing with *M* samples per frame

Dependencies

To enable this port, select the **Output estimation error** parameter.

Data Types: `single` | `double`

`Covariance` — Parameter estimation error covariance P
matrix

Parameter estimation error covariance *P*, returned as an *N*-by-*N* matrix, where *N* is the number of parameters. For details, see the **Output Parameter Covariance Matrix** parameter.

Dependencies

To enable this port:

- Select the **Output parameter covariance matrix** parameter.
- If **History** is `Infinite`, set **Estimation Method** to `Forgetting Factor` or `Kalman Filter`.

Data Types: `single` | `double`

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` — Source of initial parameter estimates
`None` (default) | `Internal` | `External`

Specify how to provide initial parameter estimates to the block:

- `None` — Do not specify initial estimates.

  The block uses `0` as the initial parameter estimate.

  Specify the parameters that the block enables based on your choice of model structure and estimation method.

  - Specify the set of **Number of Parameters ()** parameters that the block enables based on your **Model Structure**. For instance, if your setting for **Model Structure** is `AR`, specify the **Number of Parameters in A(q) (na)** parameter.
  - Specify the **Input Delay (nk)** parameter that the block enables when your model structure uses a $B(q)$ element.
  - Specify the **Parameter Covariance Matrix** if **Estimation Method** is `Forgetting Factor` or `Kalman Filter`.
- `Internal` — Specify initial parameter estimates internally to the block.
  - Specify the initial parameter values **Initial ()** parameters that the block enables based on your **Model Structure** and **History**. For instance, if your setting for **Model Structure** is `AR` and **History** is `Infinite`, specify the **Initial A(q)** parameter.

  - Specify the **Input Delay (nk)** parameter that the block enables when your model structure uses a $B(q)$ element.
  - Specify the **Parameter Covariance Matrix** parameter if **Estimation Method** is `Forgetting Factor` or `Kalman Filter`.
  - Specify the **Initial Inputs** parameter (`ARX` and `OE` only) and the **Initial Outputs** parameter (`ARX`, `AR`, and `OE`) if **History** is `Finite`.
- `External` — Specify initial parameter estimates as an input signal to the block.

  Specify the **Number of Parameters ()** parameters that the block enables based on your **Model Structure**. Your setting for **Model Structure** and for the **History** parameter determines which signals to connect to the relevant ports:

  - If **History** is `Infinite` — **InitialParameters** and **InitialCovariance**
  - If **History** is `Finite` — **InitialOutputs** for the `AR`, `ARX`, and `OE` model structures, and **InitialInputs** for the `ARX` and `OE` model structures

Programmatic Use
Block Parameter: `InitialEstimateSource`
Type: character vector, string
Values: `'None'`, `'Internal'`, `'External'`
Default: `'None'`

`Number of Parameters in A(q) (na)` — Number of estimated parameters in the *A*(*q*) polynomial

1 (default) | non-negative integer

Specify the number of estimated parameters *na* in the $A(q)$ polynomial.

Dependencies

To enable this parameter, either:

- Set **History** to `Infinite`, **Model Structure** to `AR`, `ARX`, `ARMA`, or `ARMAX`, and **Initial Estimate** to `None` or `External`.
- Set **History** to `Finite`, **Model Structure** to `AR` or `ARX`, and **Initial Estimate** to `None` or `External`.

Programmatic Use
Block Parameter: `A0`
Type: non-negative integer
Default: `1`

`Number of Parameters in B(q) (nb)` — Number of estimated parameters in the *B*(*q*) polynomial

1 (default) | vector of positive integers

Specify the number of estimated parameters *nb* in the $B(q)$ polynomial.

For MISO systems using an ARX model structure, specify *nb* as a vector with as many elements as there are inputs. Each element of this vector represents the order of the $B(q)$ polynomial associated with the corresponding input. For example, suppose that you have a two-input MISO system whose $B(q)$ elements are: $\begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} 0 & b_{11} & 0 \\ 0 & b_{21} & b_{22} \end{bmatrix}$. The zero at the beginning of each polynomial represents a single input delay for each input (see the **Initial B(q)** parameter description). The trailing zero in B_1 is for equalizing the length of the polynomials and has no impact on estimation. *nb* for each polynomial is equal to the number of estimated parameters following the initial zero, or 1 for input 1 and 2 for input 2. Specify **Number of Parameters in B(q) (nb)** as `[1 2]`, and **Input Delay (nk)** as `[1 1]`.

Dependencies

To enable this parameter, either:

- Set **History** to `Infinite`, **Model Structure** to `ARX`, `ARMAX`, `BJ`, or `OE`, and **Initial Estimate** to `None` or `External`.
- Set **History** to `Finite` with **Model Structure** of `ARX` or `OE` and **Initial Estimate** to `None` or `External`.

Programmatic Use
Block Parameter: `B0`
Type: positive integer
Default: `1`

`Number of Parameters in C(q) (nc)` — Number of estimated parameters in the *C*(*q*) polynomial

`1` (default) | positive integer

Specify the number of estimated parameters *nc* in the *C*(*q*) polynomial.

Dependencies

To enable this parameter, set **History** to `Infinite`, **Model Structure** to `ARMA`, `ARMAX`, or `BJ`, and **Initial Estimate** to `None` or `External`

Programmatic Use
Block Parameter: `C0`
Type: positive integer
Default: `1`

`Number of Parameters in D(q) (nd)` — Number of estimated parameters in the *D*(*q*) polynomial

`1` (default) | positive integer

Specify the number of estimated parameters *nd* in the *D*(*q*) polynomial.

Dependencies

To enable this parameter, set **History** to `Infinite`, **Model Structure** to `BJ`, and **Initial Estimate** to `None` or `External`.

Programmatic Use
Block Parameter: `D0`
Type: positive integer
Default: `1`

`Number of Parameters in F(q) (nf)` — Number of estimated parameters in the *F*(*q*) polynomial

`1` (default) | positive integer

Specify the number of estimated parameters *nf* in the *F*(*q*) polynomial.

Dependencies

To enable this parameter, set **Initial Estimate** to `None` or `External` and either:

- **History** to `Infinite`, **Model Structure** to `OE` or `BJ`, and **Initial Estimate** to `None` or `External`
- **History** to `Finite`, **Model Structure** to `OE`, and **Initial Estimate** to `None` or `External`

.

Programmatic Use
Block Parameter: `F0`
Type: positive integer
Default: `1`

`Input Delay (nk)` — Input delay
`1` (default) | vector of non-negative integers

Specify the input delay as an integer representing the number of time steps that occur before the input affects the output. This delay is also called the *dead time* in the system. The block encodes the input delay as fixed leading zeros of the $B(q)$ polynomial. For more information, see the $B(q)$ parameter description.

For MISO systems with ARX model structure, specify *nk* as a vector with elements specifying the delay for each input. This vector is of length *nu*, where *nu* is the number of inputs.

For example, suppose that you have a two-input MISO system whose $B(q)$ elements are: $\begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} 0 & b_{11} & b_{12} \\ 0 & 0 & b_{21} \end{bmatrix}$. The zeros at the beginning of the polynomials represent a single-sample delay for the first input, and a two-sample delay for the second input (see the **Initial B(q)** parameter description). *nb* for each polynomial is equal to the number of estimated parameters following the initial zeros, or 2 for input 1 and 1 for input 2. Specify **Input Delay (nk)** as `[1 2]`, and **Number of Parameters in B(q) (nb)** as `[2 1]`.

Dependencies

To enable this parameter, either:

- Set **History** to `Infinite`, **Model Structure** to `ARX`, `ARMAX`, `OE`, or `BJ`, and **Initial Estimate** to `None` or `External`.
- Set **History** to `Finite`, **Model Structure** to `ARX` or `OE`, and **Initial Estimate** to `None` or `External`.

Programmatic Use
Block Parameter: `nk`
Type: non-negative integer vector
Default: `1`

`Parameter Covariance Matrix` — Initial parameter covariance
`1e4` (default) | scalar | vector | matrix

- Real positive scalar, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real positive scalars, $[\alpha(a),\alpha(b),\alpha(c), \alpha(d), \alpha(f)]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha(a),\alpha(b),\alpha(c),\alpha(d), \alpha(f)]$ as the diagonal elements. $\alpha(a)$ is a vector of the

covariance for each coefficient of the A polynomial. Similarly, $\alpha(b)$, $\alpha(c)$, $\alpha(d)$ and $\alpha(f)$ 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 — $N = nb + nc + nd + nf$

Dependencies

To enable this parameter, set

- **History** to `Infinite`
- **Initial Estimate** to `None` or `Internal`
- **Estimation Method** to `Forgetting Factor` or `Kalman Filter`

The block uses this parameter at the beginning of the simulation or whenever the **Reset** signal triggers.

Programmatic Use
Block Parameter: `P0`
Type: scalar, vector, or matrix
Default: `1e4`

`Initial A(q)` — Initial values of the *A*(*q*) polynomial coefficients

`[1 eps]` (default) | vector

Specify the initial estimate of the $A(q)$ polynomial coefficients as a row vector of length na+1.

The leading coefficient of A must be `1`.

Dependencies

To enable this parameter, set:

- **Model Structure** to `AR`, `ARX`, `ARMA`, or `ARMAX`
- **Initial Estimate** to `Internal`

Programmatic Use
Block Parameter: `A0`
Type: real vector
Default: `[1 eps]`

`Initial B(q)` — Initial values of the *B*(*q*) polynomial coefficients

`[0 eps]` (default) | vector | matrix

Specify the initial estimate of the *B*(*q*) polynomial coefficients as a row vector of length *nb*+*nk*. For multiple-input models, specify a matrix where the *i*th row corresponds to the *i*th input.

The block counts the leading zeros in *B*(*q*) and interprets them as input delay *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 *nb* 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]`.

Dependencies

To enable this parameter, set:

- **Model Structure** to `ARX`, `ARMAX`, `OE`, or `BJ`
- **Initial Estimate** to `Internal`

Programmatic Use
Block Parameter: `B0`
Type: real vector or matrix
Default: `[0 eps]`

`Initial C(q)` — Initial values of the *C*(*q*) polynomial coefficients

`[1 eps]` (default) | vector

Specify the initial estimate of the *C*(*q*) polynomial coefficients 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.

Dependencies

To enable this parameter, set:

- **History** to `Infinite`
- **Model Structure** to `ARMA`, `ARMAX`, or `BJ`
- **Initial Estimate** to `Internal`

Programmatic Use
Block Parameter: `C0`
Type: real vector
Default: `[1 eps]`

`Initial D(q)` — Initial values of the *D*(*q*) polynomial coefficients

`[1 eps]` (default) | vector

Specify the initial estimate of the *D*(*q*) polynomial coefficients 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.

Dependencies

To enable this parameter, set:

- **History** to `Infinite`
- **Model Structure** to `BJ`
- **Initial Estimate** to `Internal`

Programmatic Use
Block Parameter: `D0`
Type: real vector
Default: `[1 eps]`

`Initial F(q)` — Initial values of the *F*(*q*) polynomial coefficients

`[1 eps]` (default) | vector

Specify the initial estimate of the $F(q)$ polynomial coefficients 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.

Dependencies

To enable this parameter, set:

- **Model Structure** to `OE` or to `BJ`
- **Initial Estimate** to `Internal`

Programmatic Use
Block Parameter: `F0`
Type: real vector
Default: `[1 eps]`

`Initial Inputs` — Initial values of the inputs buffer

`0` (default) | matrix

Specify initial values of the inputs buffer when using finite-history (sliding window) estimation. The buffer dimensions accommodate the specified window length, the regressors associated with polynomials within that window, the input delays, and the number of inputs. These elements drive a matrix size of:

- ARX model structure — $(W–1+\max(nb)+\max(nk))$-by-nu
- OE model structure — $(W–1+nb+nk)$-by-1

where W is the window length and nu is the number of inputs. nb is the vector of $B(q)$ polynomial orders and nk is the vector of input delays.

When the initial value is set to `0`, the block populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

The block uses this parameter at the beginning of the simulation or whenever the **Reset** signal triggers.

Dependencies

To enable this parameter, set

- **History** to `Finite`
- **Model Structure** to `ARX` or `OE`
- **Initial Estimate** to `Internal`

.

Programmatic Use
Block Parameter: `InitialInputs`
Type: real matrix
Default: `0`

`Initial Outputs` — Initial values of the measured outputs buffer

`0` (default) | vector

Specify initial values of the measured outputs buffer when using finite-history (sliding-window) estimation. The buffer dimensions accommodate the specified window length and the regressors associated with the polynomials within that window.

- AR or ARX model structure — (W+na)-by-1 vector, where W is the window length and na is the polynomial order of $A(q)$.
- OE model structure — (W+nf)-by-1 vector, where W is the window length and nf is the polynomial order of $F(q)$.

When the initial value is set to `0`, the block populates the buffer with zeros.

If the initial buffer is set to `0` or does not contain enough information, you see a warning message during the initial phase of your estimation. The warning should clear after a few cycles. The number of cycles it takes for sufficient information to be buffered depends upon the order of your polynomials and your input delays. If the warning persists, you should evaluate the content of your signals.

The block uses this parameter at the beginning of the simulation or whenever the **Reset** signal triggers.

Dependencies

To enable this parameter, set:

- **History** to `Finite`
- **Model Structure** to `AR`, `ARX`, or `OE`
- **Initial Estimate** to `Internal`

Programmatic Use
Block Parameter: `InitialOutputs`
Type: real vector
Default: `0`

Input Processing and Sample Time

`Input Processing` — Choose sample-based or frame-based processing

`Sample-based` (default) | `Frame-based`

- `Sample-based` processing operates on signals streamed one sample at a time.
- `Frame-based` processing operates on signals containing samples from multiple time steps. Many machine sensor interfaces package multiple samples and transmit these samples together in frames. `Frame-based` processing allows you to input this data directly without having to first unpack it.

Specifying frame-based data adds an extra dimension of *M* to some of your data inports and outports, where *M* is the number of time steps in a frame. These ports are:

- **Inputs**
- **Output**
- **Error**

For more information, see the port descriptions in “Ports” on page 2-72.

Programmatic Use
Block Parameter: `InputProcessing`
Type: character vector, string
Values: `'Sample-based'`, `'Frame-based'`
Default: `'Sample-based'`

`Sample Time` — Block sample time

`–1` (default) | positive scalar

Specify the data sample time, whether by individual samples for sample-based processing (t_s), or by frames for frame-based processing ($t_f = Mt_s$), where *M* is the frame length. When you set **Sample Time** to its default value of –1, the block inherits its t_s or t_f based on the signal.

Specify **Sample Time** as a positive scalar to override the inheritance.

Programmatic Use
Block Parameter: `Ts`
Type: real scalar
Values: `–1`, positive scalar
Default: `–1`

Algorithm and Block Options

Algorithm Options

`History` — Choose infinite or finite data history

`Infinite` (default) | `Finite`

The **History** parameter determines what type of recursive algorithm you use:

- `Infinite` — Algorithms in this category aim to produce parameter estimates that explain all data since the start of the simulation. These algorithms retain the history in a data summary. The block maintains this summary within a fixed amount of memory that does not grow over time.

  The block provides multiple algorithms of the `Infinite` type. Selecting this option enables the **Estimation Method** parameter with which you specify the algorithm.
- `Finite` — Algorithms in this category aim to produce parameter estimates that explain only a finite number of past data samples. The block uses all of the data within a finite window, and discards data once that data is no longer within the window bounds. This method is also called sliding-window estimation.

  The block provides one algorithm of the `Finite` type. You can use this option only with the `AR`, `ARX`, and `OE` model structures.

  Selecting this option enables the **Window Length** parameter.

For more information on recursive estimation methods, see “Recursive Algorithms for Online Parameter Estimation”

Programmatic Use
Block Parameter: `History`
Type: character vector, string
Values: `'Infinite'`, `'Finite'`
Default: `'Infinite'`

`Window Length` — Window size for finite sliding-window estimation

`200` (default) | positive integer

The **Window Length** parameter determines the number of time samples to use for the finite-history (sliding-window) estimation method. Choose a window size that balances estimation performance with computational and memory burden. Sizing factors include the number and time variance of the parameters in your model. Always specify **Window Length** in samples, even if you are using frame-based input processing.

Window Length must be greater than or equal to the number of estimated parameters.

Suitable window length is independent of whether you are using sample-based or frame-based input processing. However, when using frame-based processing, **Window Length** must be greater than or equal to the number of samples (time steps) contained in the frame.

Dependencies

To enable this parameter, set **History** to `Finite`.

Programmatic Use
Block Parameter: `WindowLength`
Type: positive integer
Default: `200`

`Estimation Method` — Recursive estimation algorithm

`Forgetting Factor` (default) | `Kalman Filter` | `Normalized Gradient` | `Gradient`

Specify the estimation algorithm when performing infinite-history estimation. When you select any of these methods, the block enables additional related parameters.

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and normalized gradient methods. However, these more intensive methods have better convergence properties than the gradient methods. For more information about these algorithms, see “Recursive Algorithms for Online Parameter Estimation”.

Programmatic Use
Block Parameter: `EstimationMethod`
Type: character vector, string
Values: `'Forgetting Factor'`,`'Kalman Filter'`,`'Normalized Gradient'`,`'Gradient'`
Default: `'Forgetting Factor'`

`Forgetting Factor` — Discount old data using forgetting factor

1 (default) | positive scalar in (0 1] range

The forgetting factor λ specifies if and how much old data is discounted in the estimation. 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 [0.98 0.995] range.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Estimation Method** to `Forgetting Factor`.

Programmatic Use
Block Parameter: `AdaptationParameter`
Type: scalar
Values: (0 1] range
Default: `1`

`Process Noise Covariance` — Process noise covariance for Kalman filter estimation method

1 (default) | nonnegative scalar | vector of nonnegative scalars | symmetric positive semidefinite matrix

Process Noise Covariance prescribes the elements and structure of the noise covariance matrix for the Kalman filter estimation. Using N as the number of parameters to estimate, specify the **Process Noise Covariance** as one of the following:

- Real nonnegative scalar, α — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real nonnegative scalars, $[\alpha_1,...,\alpha_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[\alpha_1,...,\alpha_N]$ as the diagonal elements.
- N-by-N symmetric positive semidefinite matrix.

The Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. **Process Noise Covariance** is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to constant coefficients, or parameters. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, expect the larger values to result in noisier parameter estimates. The default value is 1.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Estimation Method** to `Kalman Filter`.

Programmatic Use
Block Parameter: `AdaptationParameter`
Type: scalar, vector, matrix
Default: `1`

`Adaptation Gain` — Adaptation gain specification for gradient estimation methods
`1` (default) | positive scalar

The adaptation gain *γ* scales the influence of new measurement data on the estimation results for the gradient and normalized gradient methods. When your measurements are trustworthy, or in other words have a high signal-to-noise ratio, specify a larger value for *γ*. However, setting *γ* too high can cause the parameter estimates to diverge. This divergence is possible even if the measurements are noise free.

When **Estimation Method** is `NormalizedGradient`, **Adaptation Gain** should be less than 2. With either gradient method, if errors are growing in time (in other words, estimation is diverging), or parameter estimates are jumping around frequently, consider reducing **Adaptation Gain**.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Estimation Method** to `Normalized Gradient` or to `Gradient`.

Programmatic Use
Block Parameter: `AdaptationParameter`
Type: scalar
Default: `1`

`Normalization Bias` — Bias for adaptation gain scaling for normalized gradient estimation method
`eps` (default) | 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, the near-zero denominator can cause jumps in the estimated parameters. **Normalization Bias** is the term introduced to the denominator to prevent these jumps. Increase **Normalization Bias** if you observe jumps in estimated parameters.

Dependencies

To enable this parameter, set **History** to `Infinite` and **Estimation Method** to `Normalized Gradient`.

Programmatic Use
Block Parameter: `NormalizationBias`
Type: scalar
Default: `eps`

Block Options

`Output estimation error` — Add Error outport to block

`off` (default) | on

Use the **Error** outport signal to validate the estimation. For a given time step t, the estimation error $e(t)$ is calculated as:

$$e(t) = y(t) - y_{est}(t),$$

where $y(t)$ is the measured output that you provide, and $y_{est}(t)$ is the estimated output using the regressors $H(t)$ and parameter estimates $\theta(t\text{-}1)$.

Programmatic Use
Block Parameter: `OutputError`
Type: character vector, string
Values: `'off'`,`'on'`,
Default: `'off'`

`Output parameter covariance matrix` — Add covariance outport to block

`off` (default) | on

Use the **Covariance** outport signal to examine parameter estimation uncertainty. The software computes parameter covariance `P` assuming that the residuals, $e(t)$, are white noise, and the variance of these residuals is 1.

The interpretation of `P` depends on the estimation approach you specify in **History** and **Estimation Method** as follows:

- If **History** is `Infinite`, then your **Estimation Method** selection results in:
  - `Forgetting Factor` — $(R_2/2)$`P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals. The block returns these residuals through the **Error** port.
  - `Kalman Filter` — R_2`P` is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Here, R_1 is the covariance matrix that you specify in **Parameter Covariance Matrix**.
  - `Normalized Gradient` or `Gradient` — Covariance `P` is not available.
- If **History** is `Finite` (sliding-window estimation) — R_2`P` is the covariance of the estimated parameters. The sliding-window algorithm does not use this covariance in the parameter-estimation process. However, the algorithm does compute the covariance for output so that you can use it for statistical evaluation.

Programmatic Use
Block Parameter: `OutputP`
Type: character vector, string
Values: `'off'`,`'on'`
Default: `'off'`

`Add enable port` — Add Enable inport to block

`off` (default) | on

Use the **Enable** signal to provide 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 output contains the last estimated parameter values.

You can use this option, for example, when or if:

- Your regressors or output signal become too noisy, or do not contain information at some time steps
- Your system enters a mode where the parameter values do not change in time

Programmatic Use
Block Parameter: `AddEnablePort`
Type: character vector, string
Values: `'off'`,`'on'`
Default: `'off'`

`External reset` — Specify trigger for external reset

`None` (default) | `Rising` | `Falling` | `Either` | `Level` | `Level hold`

Set the **External reset** parameter to both add a **Reset** inport and specify the inport signal condition that triggers a reset of algorithm states to their specified initial values. Reset the estimation, for example, if parameter covariance is becoming too large because of lack of either sufficient excitation or information in the measured signals. The **External reset** parameter determines the timing for the reset.

Suppose that 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.

If the block is disabled at *t* and you reset the block, the block output contains the values specified in **Initial Estimate**.

Specify this option as one of the following:

- `None` — Algorithm states and estimated parameters are not reset.
- `Rising` — Trigger 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` — Trigger 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` — Trigger reset when the control signal is either rising or falling.
- `Level` — Trigger 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` — Trigger 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.

Programmatic Use
Block Parameter: `ExternalReset`
Type: character vector, string

Values: `'None'`,`'Rising'`,`'Falling'`, `'Either'`, `'Level'`, `'Level hold'`
Default: `'None'`

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999, pp. 363–369.

[2] Zhang, Q. "Some Implementation Aspects of Sliding Window Least Squares Algorithms." *IFAC Proceedings*. Vol. 33, Issue 15, 2000, pp. 763–768.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also
Recursive Least Squares Estimator | Kalman Filter

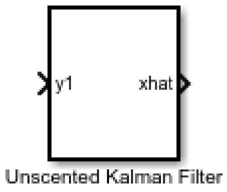
Topics
“Estimate Parameters of System Using Simulink Recursive Estimator Block”
“Online Recursive Least Squares Estimation”
“Preprocess Online Parameter Estimation Data in Simulink”
“Validate Online Parameter Estimation Results in Simulink”
“Generate Online Parameter Estimation Code in Simulink”
“Recursive Algorithms for Online Parameter Estimation”

Introduced in R2014a

Unscented Kalman Filter

Estimate states of discrete-time nonlinear system using unscented Kalman filter

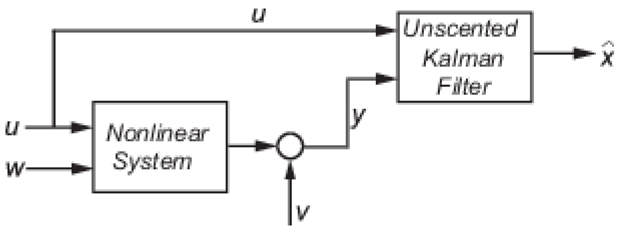
Library:	Control System Toolbox / State Estimation System Identification Toolbox / Estimators



Description

The Unscented Kalman Filter block estimates the states 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.



Using the state transition and measurement functions of the system and the unscented Kalman filter algorithm, the block produces state estimates $\hat{x}$ for the current time step. For information about the algorithm, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”.

You create the nonlinear state transition function and measurement functions for the system and specify these functions in the block. The block supports state estimation of a system with multiple sensors that are operating at different sampling rates. You can specify up to five measurement functions, each corresponding to a sensor in the system. For more information, see “State Transition and Measurement Functions” on page 2-103.

Ports

Input

`y1,y2,y3,y4,y5` — Measured system outputs

vector

Measured system outputs corresponding to each measurement function that you specify in the block. The number of ports equals the number of measurement functions in your system. You can specify up to five measurement functions. For example, if your system has two sensors, you specify two measurement functions in the block. The first port **y1** is available by default. When you click **Apply**, the software generates port **y2** corresponding to the second measurement function.

Specify the ports as *N*-dimensional vectors, where *N* is the number of quantities measured by the corresponding sensor. For example, if your system has one sensor that measures the position and velocity of an object, then there is only one port **y1**. The port is specified as a 2-dimensional vector with values corresponding to position and velocity.

Dependencies

The first port **y1** is available by default. Ports **y2** to **y5** are generated when you click **Add Measurement**, and click **Apply**.

Data Types: `single` | `double`

`StateTransitionFcnInputs` — Additional optional input argument to state transition function

scalar | vector | matrix

Additional optional input argument to the state transition function `f` other than the state `x` and process noise `w`. For information about state transition functions see, “State Transition and Measurement Functions” on page 2-103.

Suppose that your system has nonadditive process noise, and the state transition function `f` has the following form:

`x(k+1) = f(x(k),w(k),StateTransitionFcnInputs)`.

Here `k` is the time step, and `StateTransitionFcnInputs` is an additional input argument other than `x` and `w`.

If you create `f` using a MATLAB function (`.m` file), the software generates the port **StateTransitionFcnInputs** when you click **Apply**. You can specify the inputs to this port as a scalar, vector, or matrix.

If your state transition function has more than one additional input, use a Simulink Function block to specify the function. When you use a Simulink Function block, you provide the additional inputs directly to the Simulink Function block using Inport blocks. No input ports are generated for the additional inputs in the Unscented Kalman Filter block.

Dependencies

This port is generated only if both of the following conditions are satisfied:

- You specify `f` in **Function** using a MATLAB function, and `f` is on the MATLAB path.
- `f` requires only one additional input argument apart from `x` and `w`.

Data Types: `single` | `double`

`MeasurementFcn1Inputs,MeasurementFcn2Inputs,MeasurementFcn3Inputs,MeasurementFcn4Inputs,MeasurementFcn5Inputs` — Additional optional input argument to each measurement function

scalar | vector | matrix

Additional optional inputs to the measurement functions other than the state `x` and measurement noise `v`. For information about measurement functions see, “State Transition and Measurement Functions” on page 2-103.

MeasurementFcn1Inputs corresponds to the first measurement function that you specify, and so on. For example, suppose that your system has three sensors and nonadditive measurement noise, and the three measurement functions `h1`, `h2`, and `h3` have the following form:

```
y1[k] = h1(x[k],v[k],MeasurementFcn1Inputs)

y2[k] = h2(x[k],v[k],MeasurementFcn2Inputs)

y3[k] = h3(x[k],v[k])
```

Here `k` is the time step, and `MeasurementFcn1Inputs` and `MeasurementFcn2Inputs` are the additional input arguments to `h1` and `h2`.

If you specify `h1`, `h2`, and `h3` using MATLAB functions (`.m` files) in **Function**, the software generates ports **MeasurementFcn1Inputs** and **MeasurementFcn2Inputs** when you click **Apply**. You can specify the inputs to these ports as scalars, vectors, or matrices.

If your measurement functions have more than one additional input, use Simulink Function blocks to specify the functions. When you use a Simulink Function block, you provide the additional inputs directly to the Simulink Function block using Inport blocks. No input ports are generated for the additional inputs in the Unscented Kalman Filter block.

Dependencies

A port corresponding to a measurement function `h` is generated only if both of the following conditions are satisfied:

- You specify `h` in **Function** using a MATLAB function, and `h` is on the MATLAB path.
- `h` requires only one additional input argument apart from `x` and `v`.

Data Types: `single` | `double`

Q — Time-varying process noise covariance

scalar | vector | matrix

Time-varying process noise covariance, specified as a scalar, vector, or matrix depending on the value of the **Process noise** parameter:

- **Process noise** is `Additive` — Specify the covariance as a scalar, an *Ns*-element vector, or an *Ns*-by-*Ns* matrix, where *Ns* 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. Specify a vector of length *Ns*, if there is no cross-correlation between process noise terms, but all the terms have different variances.
- **Process noise** is `Nonadditive` — Specify the covariance as a *W*-by-*W* matrix, where *W* is the number of process noise terms in the state transition function.

Dependencies

This port is generated if you specify the process noise covariance as **Time-Varying**. The port appears when you click **Apply**.

Data Types: `single` | `double`

`R1,R2,R3,R4,R5` — Time-varying measurement noise covariance

matrix

Time-varying measurement noise covariances for up to five measurement functions of the system, specified as matrices. The sizes of the matrices depend on the value of the **Measurement noise** parameter for the corresponding measurement function:

- **Measurement noise** is `Additive` — Specify the covariance as an *N*-by-*N* matrix, where *N* is the number of measurements of the system.
- **Measurement noise** is `Nonadditive` — Specify the covariance as a *V*-by-*V* matrix, where *V* is the number of measurement noise terms in the corresponding measurement function.

Dependencies

A port is generated if you specify the measurement noise covariance as **Time-Varying** for the corresponding measurement function. The port appears when you click **Apply**.

Data Types: `single` | `double`

`Enable1,Enable2,Enable3,Enable4,Enable5` — Enable correction of estimated states when measured data is available

scalar

Suppose that measured output data is not available at all time points at the port **y1** that corresponds to the first measurement function. Use a signal value other than `0` at the **Enable1** port to enable the correction of estimated states when measured data is available. Specify the port value as `0` when measured data is not available. Similarly, if measured output data is not available at all time points at the port **y*i*** for the *i*th measurement function, specify the corresponding port **Enable*i*** as a value other than `0`.

Dependencies

A port corresponding to a measurement function is generated if you select **Add Enable port** for that measurement function. The port appears when you click **Apply**.

Data Types: `single` | `double` | `Boolean`

Output

`xhat` — Estimated states

vector

Estimated states, returned as a vector of size *Ns*, where *Ns* is the number of states of the system. To access the individual states, use the Selector block.

When the **Use the current measurements to improve state estimates** parameter is selected, the block outputs the corrected state estimate $\hat{x}[k|k]$ at time step `k`, estimated using measured outputs until time `k`. If you clear this parameter, the block returns the predicted state estimate $\hat{x}[k|k-1]$ for time `k`, estimated using measured output until a previous time `k-1`. Clear this parameter if your filter is in a feedback loop and there is an algebraic loop in your Simulink model.

Data Types: `single` | `double`

P — State estimation error covariance

matrix

State estimation error covariance, returned as an *Ns*-by-*Ns* matrix, where *Ns* is the number of states of the system. To access the individual covariances, use the Selector block.

Dependencies

This port is generated if you select **Output state estimation error covariance** in the **System Model** tab, and click **Apply**.

Data Types: `single` | `double`

Parameters

System Model Tab

State Transition

`Function` — State transition function name

`myStateTransitionFcn` (default) | function name

The state transition function calculates the *Ns*-element state vector of the system at time step *k*+1, given the state vector at time step *k*. *Ns* is the number of states of the nonlinear system. You create the state transition function and specify the function name in **Function**. For example, if `vdpStateFcn.m` is the state transition function that you created and saved, specify **Function** as `vdpStateFcn`.

The inputs to the function you create depend on whether you specify the process noise as additive or nonadditive in **Process noise**.

- **Process noise** is `Additive` — The state transition function *f* specifies how the states evolve as a function of state values at previous time step:

  `x(k+1) = f(x(k),Us1(k),...,Usn(k))`,

  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. To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.
- **Process noise** is `Nonadditive` — The state transition function also specifies how the states evolve as a function of the process noise `w`:

  `x(k+1) = f(x(k),w(k),Us1(k),...,Usn(k))`.

For more information, see “State Transition and Measurement Functions” on page 2-103.

You can create *f* using a Simulink Function block or as a MATLAB function (`.m` file).

- You can use a MATLAB function only if *f* has one additional input argument `Us1` other than `x` and `w`.

  `x(k+1) = f(x(k),w(k),Us1(k))`

  The software generates an additional input port **StateTransitionFcnInputs** to specify this argument.
- If you are using a Simulink Function block, specify `x` and `w` using Argument Inport blocks and the additional inputs `Us1,...,Usn` using Inport blocks in the Simulink Function block. You do not provide `Us1,...,Usn` to the Unscented Kalman Filter block.

Programmatic Use
Block Parameter: `StateTransitionFcn`
Type: character vector, string
Default: `'myStateTransitionFcn'`

`Process noise` — Process noise characteristics

`Additive` (default) | `Nonadditive`

Process noise characteristics, specified as one of the following values:

- `Additive` — Process noise `w` is additive, and the state transition function *f* that you specify in **Function** has the following form:

  `x(k+1) = f(x(k),Us1(k),...,Usn(k))`,

  where `x(k)` is the estimated state at time `k`, and `Us1,...,Usn` are any additional input arguments required by your state transition function.
- `Nonadditive` — Process noise is nonadditive, and 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+1) = f(x(k),w(k),Us1(k),...,Usn(k))`.

Programmatic Use
Block Parameter: `HasAdditiveProcessNoise`
Type: character vector
Values: `'Additive'`, `'Nonadditive'`
Default: `'Additive'`

`Covariance` — Time-invariant process noise covariance

`1` (default) | scalar | vector | matrix

Time-invariant process noise covariance, specified as a scalar, vector, or matrix depending on the value of the **Process noise** parameter:

- **Process noise** is `Additive` — Specify the covariance as a scalar, an *Ns*-element vector, or an *Ns*-by-*Ns* matrix, where *Ns* 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. Specify a vector of length *Ns*, if there is no cross-correlation between process noise terms but all the terms have different variances.
- **Process noise** is `Nonadditive` — Specify the covariance as a *W*-by-*W* matrix, where *W* is the number of process noise terms.

If the process noise covariance is time-varying, select **Time-varying**. The block generates input port **Q** to specify the time-varying covariance.

Dependencies

This parameter is enabled if you do not specify the process noise as **Time-Varying**.

Programmatic Use
Block Parameter: `ProcessNoise`
Type: character vector, string
Default: `'1'`

`Time-varying` — Time-varying process noise covariance

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

If you select this parameter, the block includes an additional input port **Q** to specify the time-varying process noise covariance.

Programmatic Use
Block Parameter: `HasTimeVaryingProcessNoise`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

Initialization

`Initial state` — Initial state estimate

`0` (default) | vector

Initial state estimate value, specified as an *Ns*-element vector, where *Ns* is the number of states in the system. Specify the initial state values based on your knowledge of the system.

Programmatic Use
Block Parameter: `InitialState`
Type: character vector, string
Default: `'0'`

`Initial covariance` — State estimation error covariance

`1` (default) | scalar | vector | matrix

State estimation error covariance, specified as a scalar, an *Ns*-element vector, or an *Ns*-by-*Ns* matrix, where *Ns* is the number of states of the system. If you specify a scalar or vector, the software creates an *Ns*-by-*Ns* diagonal matrix with the scalar or vector elements on the diagonal.

Specify a high value for the covariance when you do not have confidence in the initial state values that you specify in **Initial state**.

Programmatic Use
Block Parameter: `InitialStateCovariance`
Type: character vector, string
Default: `'1'`

Unscented Transformation Parameters

`Alpha` — Spread of sigma points

`1e-3` (default) | scalar value between 0 and 1

The unscented Kalman filter algorithm treats the state of the system as a random variable with a mean state value and variance. 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 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. Specify as a scalar value between 0 and 1 (`0` < **Alpha** `<= 1`). 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 typically 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”.

Programmatic Use
Block Parameter: `Alpha`
Type: character vector, string
Default: `'1e-3'`

`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 the optimal choice.

For more information, see the description for **Alpha**.

Programmatic Use
Block Parameter: `Beta`
Type: character vector, string
Default: `'2'`

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 description for **Alpha**.

Programmatic Use
Block Parameter: `Kappa`
Type: character vector, string
Default: `'0'`

Measurement

`Function` — Measurement function name

`myMeasurementFcn` (default) | function name

The measurement function calculates the *N*-element output measurement vector of the nonlinear system at time step *k*, given the state vector at time step *k*. You create the measurement function and specify the function name in **Function**. For example, if `vdpMeasurementFcn.m` is the measurement function that you created and saved, specify **Function** as `vdpMeasurementFcn`.

The inputs to the function you create depend on whether you specify the measurement noise as additive or nonadditive in **Measurement noise**.

- **Measurement noise** is `Additive` — The measurement function *h* specifies how the measurements evolve as a function of state Values:

  `y(k) = h(x(k),Um1(k),...,Umn(k))`,

  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 a sensor for tracking an object, an additional input could be the sensor position.

  To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line.
- **Measurement noise** is `Nonadditive`— The measurement function also specifies how the output measurement evolves as a function of the measurement noise `v`:

  `y(k) = h(x(k),v(k),Um1(k),...,Umn(k))`.

  To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

For more information, see “State Transition and Measurement Functions” on page 2-103.

You can create *h* using a Simulink Function block or as a MATLAB function (`.m` file).

- You can use a MATLAB function only if *h* has one additional input argument `Um1` other than `x` and `v`.

  `y[k] = h(x[k],v[k],Um1(k))`

  The software generates an additional input port **MeasurementFcnInput** to specify this argument.
- If you are using a Simulink Function block, specify `x` and `v` using Argument Inport blocks and the additional inputs `Um1,...,Umn` using Inport blocks in the Simulink Function block. You do not provide `Um1,...,Umn` to the Unscented Kalman Filter block.

If you have multiple sensors in your system, you can specify multiple measurement functions. You can specify up to five measurement functions using the **Add Measurement** button. To remove measurement functions, use **Remove Measurement**.

Programmatic Use
Block Parameter: `MeasurementFcn1`, `MeasurementFcn2`, `MeasurementFcn3`, `MeasurementFcn4`, `MeasurementFcn5`
Type: character vector, string
Default: `'myMeasurementFcn'`

`Measurement noise` — Measurement noise characteristics

`Additive` (default) | `Nonadditive`

Measurement noise characteristics, specified as one of the following values:

- `Additive` — Measurement noise `v` is additive, and the measurement function *h* that you specify in **Function** has the following form:

  `y(k) = h(x(k),Um1(k),...,Umn(k))`,

  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.
- `Nonadditive` — Measurement noise is nonadditive, and 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(k),...,Umn(k))`.

Programmatic Use
Block Parameter: `HasAdditiveMeasurementNoise1`, `HasAdditiveMeasurementNoise2`, `HasAdditiveMeasurementNoise3`, `HasAdditiveMeasurementNoise4`, `HasAdditiveMeasurementNoise5`
Type: character vector
Values: `'Additive'`, `'Nonadditive'`
Default: `'Additive'`

`Covariance` — Time-invariant measurement noise covariance

`1` (default) | matrix

Time-invariant measurement noise covariance, specified as a matrix. The size of the matrix depends on the value of the **Measurement noise** parameter:

- **Measurement noise** is `Additive` — Specify the covariance as an *N*-by-*N* matrix, where *N* is the number of measurements of the system.
- **Measurement noise** is `Nonadditive` — Specify the covariance as a *V*-by-*V* matrix, where *V* is the number of measurement noise terms.

If the measurement noise covariance is time-varying, select **Time-varying**. The block generates input port **R*i*** to specify the time-varying covariance for the *i*th measurement function.

Dependencies

This parameter is enabled if you do not specify the process noise as **Time-Varying**.

Programmatic Use
Block Parameter: `MeasurementNoise1`, `MeasurementNoise2`, `MeasurementNoise3`, `MeasurementNoise4`, `MeasurementNoise5`
Type: character vector, string
Default: `'1'`

`Time-varying` — Time-varying measurement noise covariance

`off` (default) | `on`

If you select this parameter for the measurement noise covariance of the first measurement function, the block includes an additional input port **R1**. You specify the time-varying measurement noise covariance in **R1**. Similarly, if you select **Time-varying** for the *i*th measurement function, the block includes an additional input port **R*i*** to specify the time-varying measurement noise covariance for that function.

Programmatic Use
Block Parameter: `HasTimeVaryingMeasurementNoise1`, `HasTimeVaryingMeasurementNoise2`, `HasTimeVaryingMeasurementNoise3`, `HasTimeVaryingMeasurementNoise4`, `HasTimeVaryingMeasurementNoise5`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Add Enable Port` — Enable correction of estimated states only when measured data is available

`off` (default) | `on`

Suppose that measured output data is not available at all time points at the port **y1** that corresponds to the first measurement function. Select **Add Enable port** to generate an input port **Enable1**. Use a signal at this port to enable the correction of estimated states only when measured data is available. Similarly, if measured output data is not available at all time points at the port **y*i*** for the *i*th measurement function, select the corresponding **Add Enable port**.

Programmatic Use
Block Parameter: `HasMeasurementEnablePort1`, `HasMeasurementEnablePort2`, `HasMeasurementEnablePort3`, `HasMeasurementEnablePort4`, `HasMeasurementEnablePort5`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

Settings

`Use the current measurements to improve state estimates` — Choose between corrected or predicted state estimate

`on` (default) | `off`

When this parameter is selected, the block outputs the corrected state estimate $\hat{x}[k|k]$ at time step `k`, estimated using measured outputs until time `k`. If you clear this parameter, the block returns the predicted state estimate $\hat{x}[k|k-1]$ for time `k`, estimated using measured output until a previous time `k-1`. Clear this parameter if your filter is in a feedback loop and there is an algebraic loop in your Simulink model.

Programmatic Use
Block Parameter: `UseCurrentEstimator`
Type: character vector
Values: `'off'`, `'on'`
Default: `'on'`

`Output state estimation error covariance` — Output state estimation error covariance

`off` (default) | `on`

If you select this parameter, a state estimation error covariance output port **P** is generated in the block.

Programmatic Use
Block Parameter: `OutputStateCovariance`
Type: character vector
Values: `'off'`,`'on'`
Default: `'off'`

`Data type` — Data type for block parameters
`double` (default) | `single`

Use this parameter to specify the data type for all block parameters.

Programmatic Use
Block Parameter: `DataType`
Type: character vector
Values: `'single'`, `'double'`
Default: `'double'`

`Sample time` — Block sample time
`1` (default) | positive scalar

Block sample time, specified as a positive scalar. If the sample times of your state transition and measurement functions are different, select **Enable multirate operation** in the **Multirate** tab, and specify the sample times in the **Multirate** tab instead.

Dependencies

This parameter is available if in the **Multirate** tab, the **Enable multirate operation** parameter is `off`.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector, string
Default: `'1'`

Multirate Tab

`Enable multirate operation` — Enable specification of different sample times for state transition and measurement functions
`off` (default) | `on`

Select this parameter if the sample times of the state transition and measurement functions are different. You specify the sample times in the **Multirate** tab, in **Sample time**.

Programmatic Use
Block Parameter: `EnableMultirate`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Sample times` — State transition and measurement function sample times
positive scalar

If the sample times for state transition and measurement functions are different, specify **Sample time**. Specify the sample times for the measurement functions as positive integer multiples of the state transition sample time. The sample times you specify correspond to the following input ports:

- Ports corresponding to state transition function — Additional input to state transition function **StateTransitionFcnInputs** and time-varying process noise covariance **Q**. The sample times of these ports must always equal the state transition function sample time, but can differ from the sample time of the measurement functions.
- Ports corresponding to i^{th} measurement function — Measured output **y*i***, additional input to measurement function **MeasurementFcn*i*Inputs**, enable signal at port **Enable*i***, and time-

varying measurement noise covariance **R*i***. The sample times of these ports for the same measurement function must always be the same, but can differ from the sample time for the state transition function and other measurement functions.

Dependencies

This parameter is available if in the **Multirate** tab, the **Enable multirate operation** parameter is `on`.

Programmatic Use
Block Parameter: `StateTransitionFcnSampleTime`, `MeasurementFcn1SampleTime1`, `MeasurementFcn1SampleTime2`, `MeasurementFcn1SampleTime3`, `MeasurementFcn1SampleTime4`, `MeasurementFcn1SampleTime5`
Type: character vector, string
Default: `'1'`

More About

State Transition and Measurement Functions

The algorithm computes the state estimates $\hat{x}$ of the nonlinear system using state transition and measurement functions specified by you. You can specify up to five measurement functions, each corresponding to a sensor in the system. 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:

  $$x[k+1] = f(x[k], u_s[k]) + w[k]$$
  $$y[k] = h(x[k], u_m[k]) + v[k]$$

  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 optional 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+1)` is linearly related to the process noise `w(k)`, and `y(k)` is linearly related to the measurement noise `v(k)`. For additive noise terms, you do not need to specify the noise terms in the state transition and measurement functions. The software adds the terms to the output of the functions.
- **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:

  $$x[k+1] = f(x[k], w[k], u_s[k])$$
  $$y[k] = h(x[k], v[k], u_m[k])$$

Compatibility Considerations

Numerical Changes

Behavior changed in R2020b

Starting in R2020b, numerical improvements in the Unscented Kalman Filter algorithm might produce results that are different from the results you obtained in previous versions.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The state transition and measurement functions that you specify must use only the MATLAB commands and Simulink blocks that support code generation. For a list of blocks that support code generation, see “Simulink Built-In Blocks That Support Code Generation” (Simulink Coder). For a list of commands that support code generation, see “Functions and Objects Supported for C/C++ Code Generation” (MATLAB Coder).

Generated code uses an algorithm that is different from the algorithm that the Unscented Kalman Filter block itself uses. You might see some numerical differences in the results obtained using the two methods.

See Also

Blocks

Kalman Filter | Extended Kalman Filter | Particle Filter

Functions

`extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter`

Topics

“What Is Online Estimation?”
“Extended and Unscented Kalman Filter Algorithms for Online State Estimation”
“Validate Online State Estimation in Simulink”
“Troubleshoot Online State Estimation”

External Websites

Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

Introduced in R2017a