COMSOL SCRIPT

Fredrik Gustafsson

SIGNALS & SYSTEMS LAB

REFERENCE GUIDE

VERSION 1.1

COMSOL

How to contact COMSOL:

Benelux
COMSOL BV
Röntgenlaan 19
2719 DX Zoetermeer
The Netherlands
Phone: +31 (0) 79 363 4230
Fax: +31 (0) 79 361 4212
info@femlab.nl
www.femlab.nl

Denmark
COMSOL A/S
Diplomvej 376
2800 Kgs. Lyngby
Phone: +45 88 70 82 00
Fax: +45 88 70 80 90
info@comsol.dk
www.comsol.dk

Finland
COMSOL OY
Arabianranta 6
FIN-00560 Helsinki
Phone: +358 9 2510 400
Fax: +358 9 2510 4010
info@comsol.fi
www.comsol.fi

France
COMSOL France
WTC, 5 pl. Robert Schuman
F-38000 Grenoble
Phone: +33 (0)4 76 46 49 01
Fax: +33 (0)4 76 46 07 42
info@comsol.fr
www.comsol.fr

Germany
FEMLAB GmbH
Berliner Str. 4
D-37073 Göttingen
Phone: +49-551-99721-0
Fax: +49-551-99721-29
info@femlab.de
www.femlab.de

Italy
COMSOL S.r.l.
Via Vittorio Emanuele II, 22
25122 Brescia
Phone: +39-030-3793800
Fax: +39-030-3793899
info.it@comsol.com
www.it.comsol.com

Norway
COMSOL AS
Søndre gate 7
NO-7485 Trondheim
Phone: +47 73 84 24 00
Fax: +47 73 84 24 01
info@comsol.no
www.comsol.no

Sweden
COMSOL AB
Tegnérgatan 23
SE-111 40 Stockholm
Phone: +46 8 412 95 00
Fax: +46 8 412 95 10
info@comsol.se
www.comsol.se

Switzerland
FEMLAB GmbH
Technoparkstrasse 1
CH-8005 Zürich
Phone: +41 (0)44 445 2140
Fax: +41 (0)44 445 2141
info@femlab.ch
www.femlab.ch

United Kingdom
COMSOL Ltd.
UH Innovation Centre
College Lane
Hatfield
Hertfordshire AL10 9AB
Phone:+44-(0)-1707 284747
Fax: +44-(0)-1707 284746
info.uk@comsol.com
www.uk.comsol.com

United States
COMSOL, Inc.
1 New England Executive Park
Suite 350
Burlington, MA 01803
Phone: +1-781-273-3322
Fax: +1-781-273-6603

COMSOL, Inc.
10850 Wilshire Boulevard
Suite 800
Los Angeles, CA 90024
Phone: +1-310-441-4800
Fax: +1-310-441-0868

COMSOL, Inc.
744 Cowper Street
Palo Alto, CA 94301
Phone: +1-650-324-9935
Fax: +1-650-324-9936

info@comsol.com
www.comsol.com

For a complete list of international representatives, visit
www.comsol.com/contact

Company home page
www.comsol.com

COMSOL user forums
www.comsol.com/support/forums

Signals & Systems Lab Reference Guide

Version: October 2007 COMSOL 3.4

CONTENTS

Chapter 1: Introduction

Chapter 2: Command Reference

1

Introduction

The documentation set for the Signals & Systems Lab consists of a printed book, the *Signals & Systems Lab User's Guide*, and this *Signals & Systems Lab Reference Guide*. Both books are available in PDF and HTML versions from the COMSOL Help Desk. This book contains detailed information about all commands provided by the Signals & Systems Lab. The commands are listed in alphabetical order, and the information is arranged under the headings Purpose, Syntax, Description, and Example(s).

2

Command Reference

Summary of Commands

Purpose The autoregressive (AR) model object.

Syntax

`m=ar(na)`	AR model structure
`m=ar(a)`	AR model with parameter vector

Description The AR model is defined by

$$y(t) = -a_1 y(t-1) - \ldots - a_{n_a} y(t-n_a) + e(t)$$

$$y(t) = \frac{1}{1 + a_1 q^{-1} + a_2 q^{-2} + \ldots + a_{n_a} q^{-n_a}} e(t)$$

The AR model is a special case of the ARX structure, where there is no deterministic input. The AR object inherits all methods of the ARX object, which in turn inherits some methods from the ARMAX object. The AR object has a somewhat simplified constructor compared to the ARX object.

A multiple output AR model shares the same a polynomial for all output channels by convention.

For the methods and fields of the AR model object, see `arx` on page 9.

Example Generate an AR structure and an AR model:

```
ar(4)
Unspecified Volatility
ar([1 1 1])
Discrete time AR(2) model (fs=1):
(q^2+q+1) y(t) =  + e(t)
ar(4)
Unspecified Volatility
ar([1 1 1])
Discrete time AR(2) model (fs=1):
(q^2+q+1) y(t) =  + e(t)
```

See Also `arx`, `arx.estimate`, `fir`

Purpose The AutoRegressive model with eXogenous input (ARX) object.

Syntax The `arx` object constructor supports the following sets of input arguments:

`arx(nn)`	Constructor of ARX structure, where nn=[na nb nk]
`arx(b,a)`	Constructor of ARX model parameters, implicitly also defining the structure
`arx(nn,th,P)`	Constructor of uncertain ARX model defined by a parameter vector and associated covariance matrix P
`arx(nn,th,thMC)`	Constructor of uncertain ARX model defined by a parameter vector Monte Carlo samples

Description The ARX model is defined as

$$y(t) = -a_1 y(t-1) - \ldots - a_{n_a} y(t-n_a)$$
$$+ b_0 u(t-n_k) + \ldots + b_{n_b} u(t-n_b-n_k) + e(t)$$

$$y(t) = q^{n_k} \frac{b_0 + b_1 q^{-1} + b_2 q^{-2} + \ldots + b_{n_b} q^{-n_b}}{1 + a_1 q^{-1} + a_2 q^{-2} + \ldots + a_{n_a} q^{-n_a}} u(t)$$
$$+ \frac{1}{1 + a_1 q^{-1} + a_2 q^{-2} + \ldots + a_{n_a} q^{-n_a}} e(t)$$

Note that the ARX model is specified either by the (a, b) polynomials, or by the parameter vector θ with structural parameters (n_a, n_b, n_k).

A multiple output ARX model shares the same a polynomial for all output channels by convention, but has one b polynomial for each input-output combination.

The methods of the ARX object include:

`arrayread`	Picks out subsystems from MIMO systems. Example: G2=G(:,2);
`display`	The overloaded display function gives an ASCII-formatted printout
`estimate`	Estimates an ARX model of specified structure from a signal SIG object
`info`	Displays user-specified information about the signal names
`rand`	Returns a random ARX model of specified structure or a number of samples from an uncertain (typically estimated) model
`size`	Returns the sizes of the model structure nn
`symbolic`	Returns a symbolic string expression for the ARX structure
`tex`	LaTeX code for displaying the model

The ARX object has the following fields that you can specify:

b	Matrix of dimension (*ny, nb, nu*) with numerator polynomials of order *nb*, where each entry specifies the numerator polynomial from input *i* to output *j*.
a	Common denominator polynomial for all input-output channels
th	Parameter vector of free parameters in b and a
P	Covariance matrix of th
nn	Structure parameters [na nb nk nu ny]
MC	Number of Monte Carlo samples
pe	Noise variance (Gaussian assumption) or PDF object (default 1)
fs	User specified sampling frequency (default 1)
name	User specified name
marker	Time instants of interest
tlabel	Time label
ylabel	User-specified names on output signals
ulabel	User-specified names on input signals
markerlabel	Label for the marker
method	For estimated models, the method and design parameters are saved here
desc	User-specified description of the signal

Example

Construct an empty structure, a certain ARX model from polynomials, and an uncertain model using the parameter vector:

```
arx([2 2 1])
Unspecified ARX(2,2,1)

arx([4 5 6],[1 2 3])
Discrete time ARX(2,3,0) model (fs=1):
(q^2+2*q+3) y(t) = (4*q^2+5*q+6) u(t) + e(t)

arx([2 3 0],[2 3 4 5 6]',0.1*eye(5))
Discrete time ARX(2,3,0) model (fs=1):
(q^2+2*q+3) y(t) = (4*q^2+5*q+6) u(t) + e(t)

Parameter vector and uncertainties [std=sqrt(P(i,i)]
    2      3      4      5      6
    0.316  0.316  0.316  0.316  0.316
```

See also

`ar`, `arx.estimate`, `fir`, `tf`

Purpose

Estimate an ARX model from data in a signal object.

Syntax

```
mhat=estimate(mstruc,z,Property1,Value1,...)
```

Description

The ARX model is estimated using the least-squares (LS) algorithm as described in "Simulation and Estimation of ARX Models" on page 199. The input parameters are given in the following table:

ARGUMENT	DESCRIPTION
z	signal object with output input data
ms	gives the model structure, typically ms=arx([na nb nk nu ny])

Use `ms=ar(na)` to estimate an AR model, and `ms=fir(nb)` to estimate a FIR model.

`ms` can also include the prior. For example, the following two lines would give the same result except for some missing data at the border:

```
m=estimate(ms,z(1:end))
m=estimate(estimate(ms,z(1:100)), z(101:end))
```

The uncertainty is stored in the covariance matrix `P` provided by LS algorithm. However, in case that Monte Carlo simulations of the signal are available, the corresponding point estimate of the parameter vector `th` are stored in the columns of `P` and used in subsequent functions for representing uncertainty. That is, Monte Carlo uncertainty has precedence to the one represented by the covariance matrix.

Example

Create a structure, and generate a random model for this. Then, you perform two experiments. In the first one, simulate one realization of a PRBS response and then estimate the model. Then simulate 30 realizations and estimate the model. In the first case, the uncertainty is provided by the LS method's covariance matrix. In the second case, the uncertainty is computed from the 30 models estimated from the 30 different signal realizations.

```
ms=arx([2 2 1]);
m=rand(ms)
Discrete time ARX(2,2,1) model (fs=1):
(q^2-0.468*q+0.14) y(t) = (q-0.999) u(t) + e(t)

z=simulate(m,getsignal('prbs',100));
mhat=estimate(ms,z)
Discrete time ARX(2,2,1) model (fs=1):
(q^2-0.475*q+0.0994) y(t) = (0.827*q-0.765) u(t) + e(t)

Parameter vector and uncertainties [std=sqrt(P(i,i)]
```

```
    -0.475  0.0994  0.827  -0.765
    0.0906  0.0881  0.15   0.156
m.MC=30;
z=simulate(m,getsignal('prbs',100));
mhat=estimate(ms,z)
Discrete time ARX(2,2,1) model (fs=1):
(q^2-0.416*q-0.0236) y(t) = (1.1*q-1.03) u(t) + e(t)

Parameter vector and uncertainties [std of MC samples]
    -0.416  -0.0236  1.1    -1.03
    0.0712  0.0973   0.127  0.126
```

See Also arx

Purpose The beta distribution.

Syntax `X=betadist(a,b)`

Description The probability density function of the beta distribution, and its first two moments, are given by

$$p(x;a,b) = \frac{x^{a-1}(1-x)^{b-1}}{\mathrm{int}_0^1 u^{a-1}(1-u)^{b-1}du}$$

$$= \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}x^{a-1}(1-x)^{b-1} = \frac{1}{\beta(a,b)}x^{a-1}(1-x)^{b-1},$$

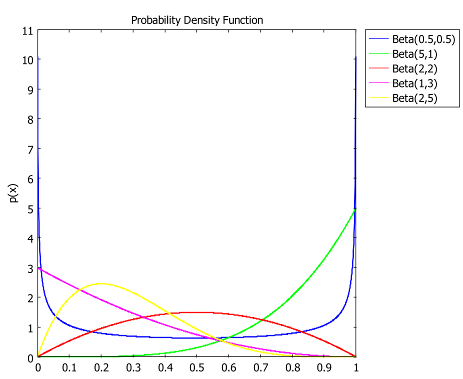
$$E(X) = \frac{a}{a+b},$$

$$\mathrm{Var}(X) = \frac{ab}{(a+b)^2(a+b+1)}.$$

Both `a` and `b` must be positive. This is a child of `pdfclass`, and all of its methods apply to this distribution, in particular `pdfclass.estimate` and the plot functions.

Example Some sample distributions:

```
a=[0.5 5 2 1 2];
b=[0.5 1 2 3 5];
for i=1:5; X{i}=betadist(a(i),b(i)); end
plot(X{:})
```



See Also `pdfclass`

Purpose

The chi2 distribution.

Syntax

`X=chi2dist(n)`

Description

The probability density function of the chi2 distribution, and its first two moments, are given by

$$p(x;n) = \ = \frac{1}{2^{k/2}\Gamma(k/2)} x^{k/2-1} e^{-x/2}, \ x > 0,$$
$$E(X) = k,$$
$$\mathrm{Var}(X) = 2k.$$

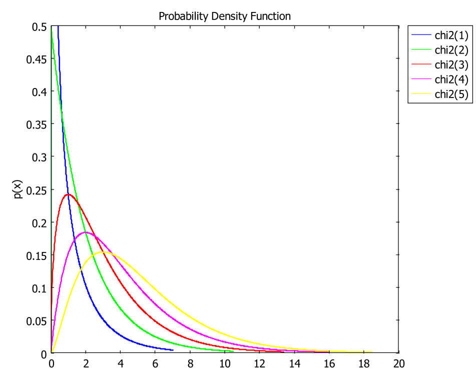
`n` must be a positive integer. This is a child of `pdfclass`, and all of its methods apply to this distribution, in particular `pdfclass.estimate` and the plot functions.

The additivity property of the chi2 distribution is implemented symbolically.

Example

Some sample distributions:

```
for k=1:5; X{k}=chi2dist(k); end
plot(X{:})
set(gca,'Ylim',[0 0.5])
```



See Also

`pdfclass`

Purpose The constructor for the covariance function object.

Syntax The following table shows the available syntax for the `covf` object constructor:

`R=covf`	Empty object (for estimation)
`R=covf(R,tau)`	Definition with a 3D matrix with elements R(tau,i,j)
`R=covf(R,tau,RMC)`	As above, with MC simulations in RMC(k,tau,i,j)
`R=covf(s)`	Conversions from SIG and LTI (ARMAX, SS) objects

Description The covariance function is defined as

$$R_{ij}(\tau) = E[y_i(t)y_j(t-\tau)]$$

Its content is stored in a 3D matrix $R(\tau,i,j)$. Monte Carlo simulations are stored in a similar matrix $R(k,\tau,i,j)$, where the first dimension is the Monte Carlo sample number k. The lags are contained in the vector τ. For a scalar signal, `R` is a column vector, and `RMC` is a matrix.

Optional public fields:

FIELD NAME	DESCRIPTION
`fs`	Sampling frequency
`MC`	Number of Monte Carlo simulations taken when converting from other uncertain objects
`name`	Name of the signal that will appear in plot titles
`tlabel`	Optional label for time
`ylabel`	Optional label for signal amplitude
`desc`	Optional description

Change these with `R.fs=fs`, and so on.

METHOD	DESCRIPTION
`ESTIMATE`	Estimates the covariance function from a signal in a SIG object
`SIZE`	Returns the sizes of R, [ny,ntau]=size(R)
`PLOT`	Plots the covariance function

The covariance function describes how a stochastic signal correlates with itself , and the definition for stationary stochastic processes is:

$$R(\tau) = E[s(t)s(t-\tau)]$$

The covariance function is closely related to spectral analysis, because the spectrum is defined as the Fourier transform of the covariance function.

$$\Phi(f) = \mathrm{FT}[R(\tau)]$$

The Signals and Systems Lab represents the covariance function in the COVF object for discrete time signals. The direct construction uses the syntax

```
R=covf(R,tau,RMC)
```

where each column of `R` contains one covariance function, `tau` is the vector of time lags and `RMC` contains Monte Carlo realizations of the covariance function. Here, `R` is `(ntau,ny*ny)`, `tau` is `(ntau,1)`, and `RMC` is `(MC,ntau,ny*ny)`.

Normally, the covariance function is estimated from a signal using the unbiased estimate

$$\hat{R}_{ij}(\tau) = \frac{1}{N-|\tau|}\sum_{t=1}^{N-|\tau|} y_i(t)y_j(t-\tau)$$

For coherence in syntax, there are three different ways to do the same thing, as the following table summarizes:

`c=estimate(covf,y,Property1,Value1,...)`	Explicit call
`c=covf(y,Property1,Value1,...)`	Implicit call
`c=sig2covf(y,Property1,Value1,...)`	Direct low-level call

Alternatively, it is possible to compute the covariance function from a stochastic signal model, which can be certain or uncertain. In the latter case, MC data are generated in `RMC`. For ARX models and stochastic state-space models, the algorithm neglects the deterministic input-output part. That is, only the AR part that is used for ARX models is kept here. The theoretical covariance function is in both cases computed using a state-space realization using the following algorithm:

1. `R(0) = C * Pibar * C' + R`
2. `R(tau) = C * A^tau * Pibar * C' + C * A^(tau-1) * S, tau=1,2,...,taumax`, where `Pibar` is the controllability Gramian (see `gram`)

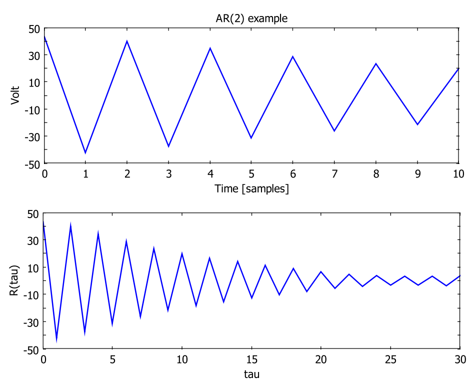
When the covariance function is computed from a stochastic process represented by a SIG object with Monte Carlo data, these random realizations are propagated to random realizations of the COV object contained in the field `RMC`. A similar situation occurs when an uncertain model is converted to a covariance function.

Each sample of the uncertain model is converted to covariance function. In both cases, the covariance function can be regarded as a stochastic variable for each time lag. Using the `plot` function, you can add confidence bounds or scatter plot of the realizations. Further, the following operations are possible:

mean/E	Returns the mean of the Monte Carlo data	c=E(C)
std	Returns the standard deviation of the Monte Carlo data	sigma=std(C)
var	Returns the variance of the Monte Carlo data	sigma2=var(C)
rand	Return one random COV object or a cell array of random COV objects	c=rand(C,10)
fix	Remove the Monte Carlo simulations from the object	c=fix(C)

Example Compare the following direct and indirect ways to create a COVF object:

```
N=1000;
y=filter(1,[1 1.6 0.64],randn(N,1));
for tau=0:10;
    R(1+tau)=sum(y(1:N-tau)'*y(1+tau:N))/(N-tau);
end
c1=covf(R,0:10);
c1.name='AR(2) example';
c1.tlabel='Time [samples]';
c1.ylabel='Volt';
ysig=sig(y);
c2=covf(ysig);
subplot(2,1,1), plot(c1)
subplot(2,1,2), plot(c2)
```



See Also `covf.estimate`, `covf.plot`, `sig.sig2covf`, `ss.ss2covf`

Purpose Estimate the covariance function from a SIG object.

Syntax The are three different functions call for computing the covariance function `c`:

`c=estimate(covf,y,Property1,Value1,...)`	Explicit call
`c=covf(y,Property1,Value1,...)`	Implicit call
`c=sig2covf(y,Property1,Value1,...)`	Direct low-level call

Description The function implements the unbiased estimate

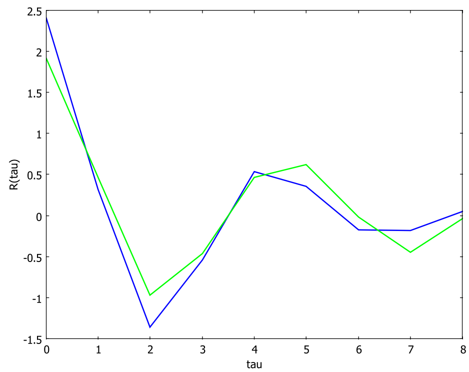
$$\hat{R}_{ij}(\tau) = \frac{1}{N-|\tau|} \sum_{t=1}^{N-|\tau|} y_i(t) y_j(t-\tau)$$

The following table shows the available property/value options:

`taumax`	`{30}`	Maximum lag for which the covariance function is computed
`fs`	`{y.fs}`	Sampling frequency (overrides fs specified in y)
`MC`	`{100}`	Number of Monte Carlo simulations to compute confidence bound
`method`	`{'direct'}`	Direct summation in the time domain
	`'conv'`	Summation in the time domain using conv
	`'freq'`	Convolution computed in the frequency domain

Example Simulate a signal from a random AR model, and estimate the corresponding covariance function. Plot the result together with the true function.

```
m0=rand(arx(4));
y=simulate(m0,100);
csighat=estimate(covf,y);  % Equivalent to  csighat=covf(y);
c0=covf(m0);
plot(c0,csighat)
set(gca,'Xlim',[0 8])
```



See Also covf, covf.plot, sig.sig2covf, ss.ss2covf

Purpose Plot the covariance function from COV objects.

Syntax

```
plot(c1,c2,...,Property1,Value1,...)
```

Description Use `covf.plot` to illustrate one or more covariance functions at the same time. For covariance functions computed from estimated models, you can add a confidence bound based on Monte Carlo simulations of the covariance function. The Monte Carlo data can also be shown in a scatter plot, where all simulated random covariance functions appear with half the line width along with the nominal one. The input objects `c` can be LTI, SIG, or COV objects, except for the first one, which must be a COV object in order to get the correct `plot` method.

For SIG objects, `covf.sig2covf` is invoked, and similarly `covf.ss2covf` for LTI objects.

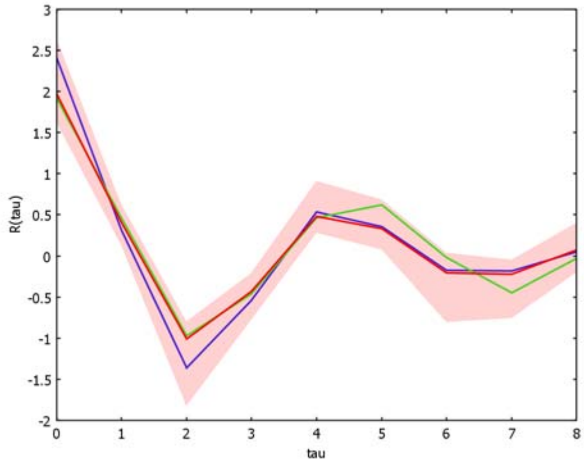
TABLE 2-1: COVPLOT PROPERTIES

PROPERTY	VALUE	DESCRIPTION
conf	[{0},100]	Confidence level (0 mean no levels plotted) from MC data
scatter	'on' | {'off'}	Scatter plot of MC data
taumax	{30}	Maximum lag for which the covariance function is computed
interval	{'pos'} |	Positive `tau=0:taumax` or
	'sym'	symmetric `tau=-taumax:taumax` lag interval
axis	{gca}	Axis handle where plot is added
col	{'bgrmyk'}	Colors in order of appearance
fontsize	14	Font size
linewidth	2	Line width
Xlim	{}	Limits on x axis
Ylim	{}	Limits on y axis
legend	{}	Legend text

Example Generate an AR(4) system, simulate data, and compare true covariance function to the estimated one:

```
m0=rand(arx(4));
y=simulate(m0,100);
mhat=estimate(arx(4),y);
csighat=covf(y);
cmhat=covf(mhat);
c0=covf(m0);
```

```
plot(c0,csighat,cmhat)
set(gca,'Xlim',[0 8])
```



See Also covf, covf.estimate, covf.plot, sig.sig2covf, ss.ss2covf

Purpose Signals & Systems Lab signal database.

Syntax `y=dbsignal(name,help)`

Description If you provide any second input argument `help`, the function displays a help text and shows some introductory plots. The following sets of data are available:

TABLE 2-2: SIGNALS DATABASE

NAME	DESCRIPTION
`bach`	A piece of music performed by a cellular phone
`carpath`	Car position obtained by dead-reckoning of wheel velocities
`current`	Current in an overloaded transformer
`eeg_human`	The EEG signal y shows the brain activity of a human test subject
`eeg_rat`	The EEG signal y shows the brain activity of a rat.
`ekg`	An EKG signal showing human heartbeats.
`equake`	Earthquake data where each of the 14 columns shows one time series.
`ess`	Human speech signal of 's' sound
`fricest`	Data z for a linear regression model used for friction estimation.
`fuel`	Data y=z from measurements of instantaneous fuel consumption.
`genera`	The number of genera on earth during 560 million years
`highway`	Measurements of car positions from a helicopter hovering over a highway
`pcg`	An PCG signal showing human heartbeats.
`photons`	Number of detected photons in X-ray and gamma-ray observatories.
`planepath`	Measurements y=p of aircraft position.

See Also `getsignal`, `sig`

Purpose Representations of nonparametric distributions.

Syntax `X=empdist(x)`

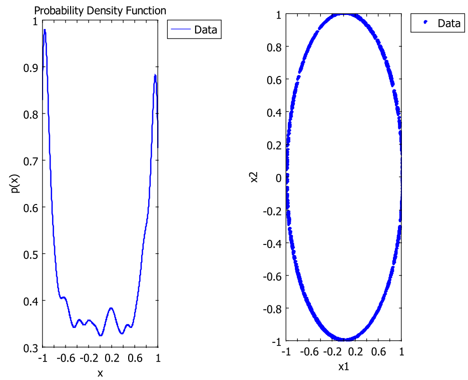
Description The data vector `x` contains samples from a univariate or multivariate distribution. The methods are inherited from PDFCLASS, but there are several methods that require numerical implementations.

METHOD	DESCRIPTION	
E	The expectation estimator	
MEAN	The expectation estimator	
VAR	The variance estimator	
STD	The standard deviation estimator	
VAR	The variance estimator	
SKEW	The skewness estimator	
KURT	The kurtosis estimator	
ESTIMATE	Estimate a parametric density function from a list of PDF objects	dist=estimate(X,pdflist)
RAND	Generate random numbers in a vector if X is scalar or (nx,N) matrix if X is a stochastic vector	x=rand(X,N)
ERF	Evaluate the error function I(x)=P(X<x) numerically, where X is scalar	I=erf(X,x)
ERFINV	Evaluate the inverse error function I(x)=P(X<x) numerically, where X is scalar	x=erfinv(X,I)
CDF	The cumulative density function (X scalar)	P=cdf(X,x)
PDF	The probability density function, obtained by histogram smoothing	p=pdf(X,x)

Example Example of a univariate and a multivariate statistical variable represented as an empirical distribution.

```
u=2*pi*rand(1000,1);
U=empdist(u);
X=sin(U)
Empirical data vector of size 1 with 1000 samples
subplot(1,2,1), plot(X)
Y=cos(U)
Empirical data vector of size 1 with 1000 samples
Z=[X;Y]
Empirical data vector of size 2 with 1000 samples
```

```
subplot(1,2,2), plot2(Z)
```



See Also `pdfclass`

Purpose Generate standard LTI objects.

Syntax `m=exlti(ex);`

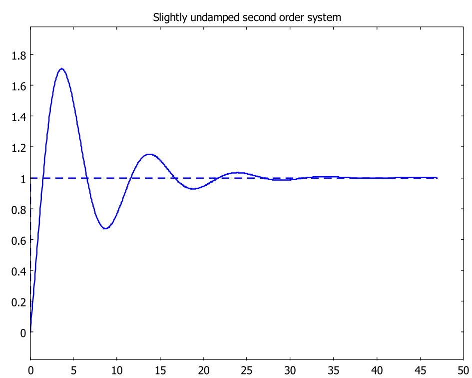
Description Standard examples of LTI objects of different structures are returned depending on the string argument `ex`.The following table lists some of the available options:

TABLE 2-3: SOME OPTIONS FOR CREATING LTI OBJECTS WITH EXLTI

EX	TYPE	TIME	DESCRIPTION
`tf1c`	`tf`	`cont`	DC motor
`tf1d`	`tf`	`disc`	DC motor
`tf2c`	`tf`	`cont`	Slightly undamped second-order system
`tf2d`	`tf`	`disc`	Slightly undamped second-order system

Example Load a continuous-time second-order transfer function and display its step response:

```
m=exlti('tf2c')

               0.62*s+0.41
     Y(s) = -------------- U(s)
              s^2+0.3*s+0.41

step(m);
```



See Also `ss`, `tf`, `getfilter`, `exnl`

Purpose Generate standard NL objects.

Syntax `m=exnl(ex,opt1)`

Description All demos in the manual are included as predefined examples here. There are also many standard motion models as used in target tracking and navigation applications. See `help exnl` for a list of options.

See Also `exlti`

Purpose The exponential distribution.

Syntax `X=expdist(mu)`

Description The probability density function of the exponential distribution, and its first two moments, are given by

$$p(x;\mu) = \ = \frac{1}{\mu}e^{-x/\mu}, \ \ x > 0$$

$$E(X) = \mu,$$

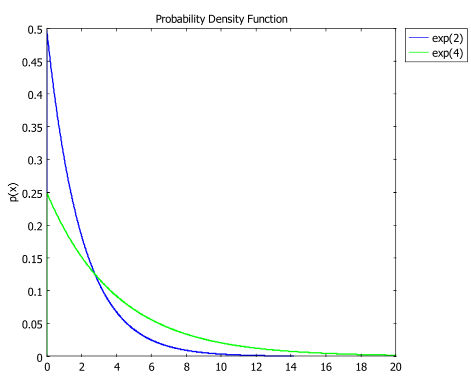
$$\mathrm{Var}(X) = \mu^2.$$

`mu` must be positive. This is a child of `pdfclass`, and all of its methods apply to this distribution, in particular `pdfclass.estimate` and the plot functions.

The multiplicative scale property of the exponential distribution is implemented symbolically.

Example Illustration of the scaling property:

```
X=expdist(2);
Y=2*X
exp(4)
plot(X,Y)
```



See Also `pdfclass`

Purpose Generate RARX object examples of length N.

Syntax `m=exrarx(ex,N);`

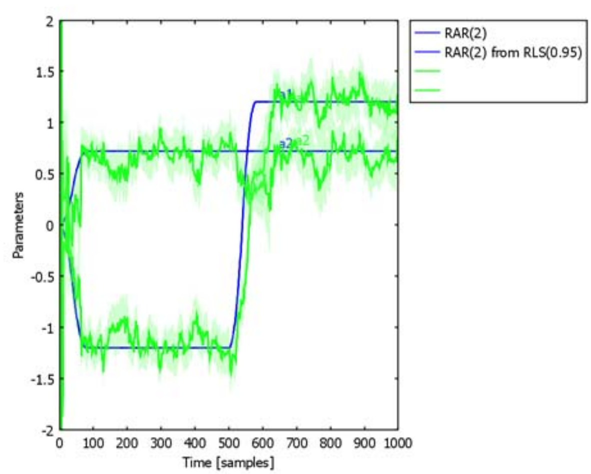
Description The function returns various examples of LTV objects of RARX form of different model structures and orders. The size of the LTV object is rescaled with `N`, which denotes the number of samples of the time-varying model (default 500). Some examples of string options for `ex`:

TABLE 2-4: RARX OBJECT EXAMPLES

EX	DESCRIPTION
`mean1`	Three changes in the mean (1,2,4) of length 200
`mean2`	As mean1, with softer transitions
`mean3`	As mean1, with random means N(0,10)
`ar1`	An abrupt switch between two AR(2) models
`ar2`	A soft switch between two AR(2) models

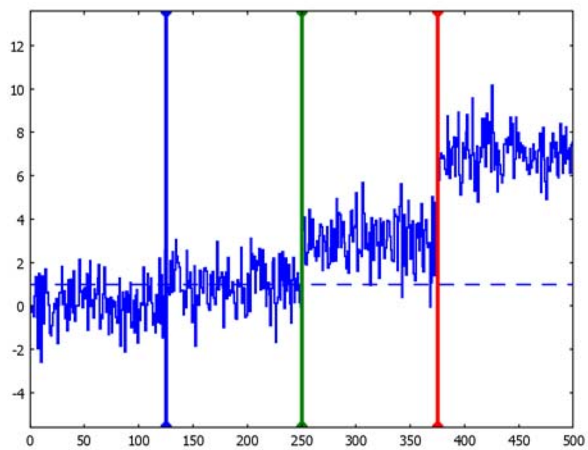
Example Show the time-varying AR(2) model, and compare with one estimated from simulated data.

```
m=exrarx('rar2',1000);
y=simulate(m);
mhat=estimate(rarx(2),y,'adg',0.95);
plot(m,mhat,'Ylim',[-2 2])
```



Generate a change in the mean model as a time-varying FIR(1) system fed by a unit input signal:

```
mt=exrarx('mean1',500);
u=getsignal('ones',length(mt));
y=simulate(mt,u);
plot(y)
```



See Also `rarx`

Purpose The F distribution.

Syntax `X=fdist(d1,d2)`

Description The probability density function of the F distribution, and its first two moments, are given by

$$p(x;d_1,d_2) = \ = \frac{1}{\beta(d_1/2,d_2/2)}\left(\frac{d_1x}{d_1x+d_2}\right)^{d_1/2}\left(1-\frac{d_1x}{d_1x+d_2}\right)^{d_2/2}\frac{1}{x}, \quad x>0$$

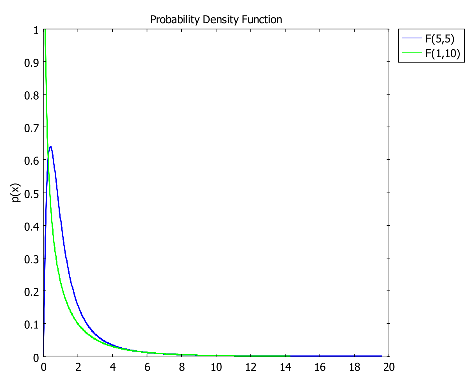
$$E(X) = \frac{d_2}{d_2-2}, \ d_2>2$$

$$\mathrm{Var}(X) = \frac{2d_2^2(d_1+d_2-2)}{d_1(d_2-2)^2(d_2-4)}, \ d_2>4.$$

Both `d1` and `d2` must be positive integers. This is a child of `pdfclass`, and all of its methods apply to this distribution, in particular `pdfclass.estimate` and the plot functions.

Example Some sample distributions:

```
d1=[5 1]; d2=[5 10];
for i=1:2; X{i}=fdist(d1(i),d2(i)); end
plot(X{:})
set(gca,'Ylim',[0 1])
```



See Also `pdfclass`

Purpose

Noncausal implementation of a filter using forward and backward filtering.

Syntax

`y=filtfilt(b,a,u,M);`	Zero-phase noncausal filtering
`y=filtfilt(bf,af,bb,ab,u,M);`	General noncausal filtering

Description

For transfer functions with poles both inside and outside the unit circle, a stable implementation of a filter must be noncausal. The typical use is for zero-phase filters defined as $|H(z)|^2$ for some stable transfer function $H(z)$. The implementation is based on forward-backward filtering. The following lines show the core of the function.

```
x=filter(b,a,u);
xr=x(end:-1:1);
yr=filter(b,a,xr);
y=yr(end:-1:1);
```

If you provide an input argument `M`, `filtfilt` attempts to minimize the transients and remove the influence of the order of application of the forward and backward filter. An approximation of the optimal initial conditions is used, where `M` denotes the number of samples in both ends that are used. With `M` = 0, `filtfilt` computes a default value for `M` from the impulse response.

If you want different causal and noncausal filters, use

```
y=filtfilt(bf,af,bb,ab,u,M);
```

Here, `bf/af` is the forward filter, and `bb/ab` is the backward filter. Default is `bb=bf` and `ab=af`.

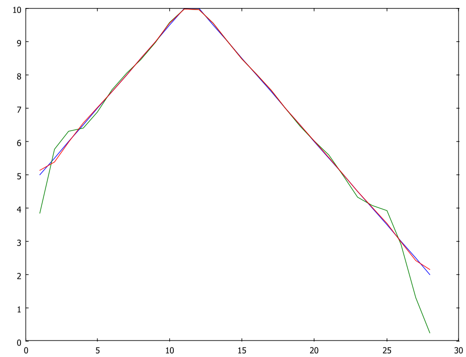
You can judge the effect of transients by comparing the forward-backward and backward-forward filtered sequences

```
[yfb,ybf]=filtfilt(b,a,u,M);
```

Example

Filter a sequence of data with nonzero initial conditions.

```
u=[5:0.5:10, 10:-0.5:2]';
G=getfilter(4,0.5,'fs',2);
y=filtfilt(G.b,G.a,u);
yM=filtfilt(G.b,G.a,u,10);
plot([u y yM])
```



Note that the straightforward implementation gives noticeable transients because of unknown initial conditions that are taken as zero by default. The effect is in particular clear when there are initial nonzero signal values. The initial filter state is estimated in the red curve, where the transient is almost invisible.

See Also

`tf.filtfilt`

Purpose The Finite Impulse Response (FIR) model object.

Syntax

`m=fir([nb nk])`	FIR model structure
`m=fir(b)`	FIR model with parameter vector

Description The FIR model is defined by

$$y(t) = b_0 u(t-n_k) + \ldots + b_{n_b} u(t-n_b-n_k) + e(t)$$

$$y(t) = q^{-n_k}(b_0 + b_1 q^{-1} + b_2 q^{-2} + \ldots + b_{n_b} q^{-n_b}) u(t) + e(t)$$

The FIR model is a special case of the ARX structure that is linear in the parameters. The FIR object inherits all methods of the ARX object, which in turn inherits some methods from the ARMAX object. The FIR object has a somewhat simplified constructor compared to the ARX object.

For the second usage of the constructor, add initial zeros in the b polynomial as `b=[0,0,...,0,b0,...,bnb]`.

Note the ambiguity for FIR models of the type `b=[b0 b1]`, where b has the same size as `[nb nk]`. Use the ARX constructor in such cases.

FIR shares all methods and its representation with the ARX object.

Example Create an empty structure, a certain model, and an uncertain model:

```
fir(4)
Unspecified FIR(4,0)

fir([1 2 3 4])
Discrete time FIR(4,0) model (fs=1):
1 y(t) = (q^3+2*q^2+3*q+4) u(t) + e(t)

fir([1 2 3 4],eye(4))
Discrete time FIR(4,0) model (fs=1):
1 y(t) = (q^3+2*q^2+3*q+4) u(t) + e(t)

Parameter vector and uncertainties [std=sqrt(P(i,i)]
    1  2  3  4
    1  1  1  1
```

See Also `arx`, `ar`

Purpose Construct a frequency object from LTI models.

Syntax

```
Hf=freq(H,f,fs,HMC)
Hf=freq(m)
```

`Hf=freq(H,f,fs,HMC)`	Direct definition
`Hf=freq(m)`	Conversion from LTI objects

Description The input arguments to the constructor are defined as follows:

TABLE 2-5: INPUT ARGUMENTS TO FREQ

ARG	DESCRIPTION
H	frequency response of a transfer function H(f,yind,uind)
f	frequency values
fs	sampling frequency
HMC	Monte Carlo samples of H organized as HMC(mc,f,uind,yind)

The FREQ object contains the following fields that you can set:

TABLE 2-6: FIELDS IN THE FREQ OBJECT

FIELD	DESCRIPTION
MC	Number of Monte Carlo samples for uncertain models
fs	Sampling frequency
name	Name of system
ulabel	Array of label for the inputs
ylabel	Array of label for the outputs
tlabel	Label for time
desc	Description of the system

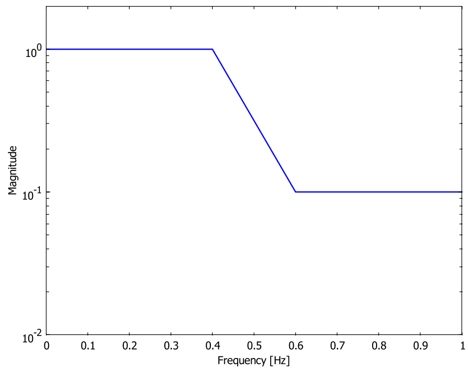
These labels are inherited from systems on TF or SS form when transformed to FREQ objects. The FREQ object inherits the plot functions from the LTI object:

PLOT	DESCRIPTION
bode	Bode diagram
bodeamp	Bode amplitude plot
bodephase	Bode phase plot
nyquist	Nyquist plot

Example Create an approximation of an ideal low-pass filter:

```
Hf=freq([1 1 0.1 0.1]',[0 0.4 0.6 1],2)
FREQ object: Untitled
  SISO transfer function frequency response
```

```
   Number of frequency points: 4
   Number of Monte Carlo samples: 0
bodeamp(Hf,'Ylim',[0.01 2]);
```



See Also `ss.ss2freq`, `tf.tf2freq`, `lti.bode`

Purpose The Fourier Transform (FT) object

Syntax

`Y=ft(y)`	Conversion from SIG object
`Y=ft(Y,f,YMC)`	Direct definition

Description The Fourier Transform is represented by a transform vector and a frequency vector with field names `Y` and `f`, respectively. Monte Carlo samples are stored in a matrix with field name `YMC`. The sampling frequency is used by the plot method for correct axis scalings. Here, Y is of size `(nf,ny)`, `f` is `nf`, and `YMC` is of size `(MC,nf,ny)`.

The usual plot variants are overloaded (see `ft.plot` on page 39 for further information and options).

PLOT FUNCTION	DESCRIPTION
`plot`	Plot with linear axes
`loglog`	Plot with logarithmic axes
`semilogy`	Plot with linear frequency axis and logarithmic amplitude axis
`semilogx`	Plot with logarithmic frequency axis and linear amplitude axis

You can use the overloaded operators in the following table to further control what is plotted using, for instance, `plot(angle(Y))`.

OPERATORS	DESCRIPTION
`abs`	Absolute value
`angle`	Angle, or phase
`real`	Real part
`imag`	Imaginary part

Furthermore, indexing a FT object by `Y(freqind,yind)` selects the frequency indices in `freqind` and the subsignals in `yind` for multivariate signals.

When the FT is computed from a stochastic process represented by a SIG object with Monte Carlo data, these random realizations are propagated to random realizations of the FT object contained in the field `YMC`. That is, the Fourier Transform can be regarded as a stochastic variable at each frequency. The `plot` function allows confidence bounds or scatter plot of the realizations to be added. Further, the following operations are possible:

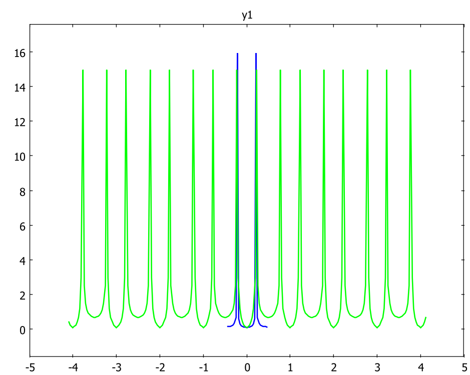
mean/E	Returns the mean of the Monte Carlo data	E(Y)
std	Returns the standard deviation of the Monte Carlo data	std(Y)

var	Returns the variance of the Monte Carlo data	var(Y)
rand	Return one random FT object or a cell array of random FT objects	rand(Y,10)
fix	Remove the Monte Carlo simulations from the object	fix(Y)

Example

Generate a sampled sinusoid whose frequency is not a DFT bin, and construct a FT object in the two available ways. The conversion uses zero padding that reveals the leakage effects of the finite rectangular window that is implicitly applied to an infinitely long sinusoid.

```
t=(0:31)';
f1=0.22;
y=sin(2*pi*f1*t);
Y=fft(y);
Y1=ft(Y(1:16),(0:15)/32);  % Direct definition
Y2=ft(sig(y,t));           % Conversion
plot(Y1,Y2)
```



See Also

`ft.plot`, `freq`

Purpose

Plot Fourier transforms

Syntax

```
plot(Y1,Y2,...,Property1,Value1,...)
```

Description

`Yi` are Fourier Transform (FT) objects.

The following property and value pairs are available:

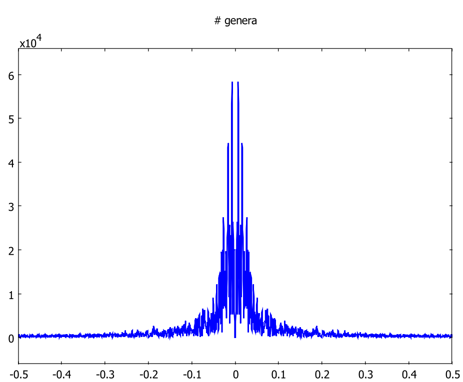
TABLE 2-7: FT.PLOT PROPERTIES

PROPERTY	VALUE	DESCRIPTION
`type`	`1,2,{3}`	Interval for f, 1:[0,fs], 2:[0,fs/2], 3:[-fs/2,fs/2]
`plottype`	`'plot'|{'semilogy'}|'semilogx'|'loglog'`	This is the plot function used in feval
`Xlim`	`{}`	Limits on x axis
`Ylim`	`{}`	Limits on y axis
`axis`	`{gca}`	Axis handle where plot is added
`col`	`{'bgrmyk'}`	Colors in order of appearance
`fontsize`	`{14}`	Font size
`linewidth`	`{2}`	Line width

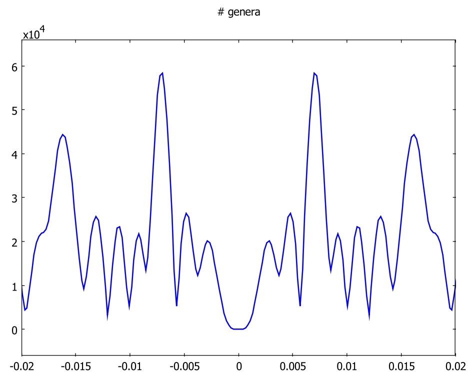
Example

Load a real data example, detrend the data, and plot the DTFT. By zooming in on the low-frequency part, two resonance peaks are visible.

```
load genera
yd=detrend(y1,3);
Y=ft(yd);
plot(Y)    % Default view
```




```
plot(Y,'Xlim',[-0.02 0.02])
```



See Also `ft`

Purpose The gamma distribution.

Syntax `X=gammadist(a,b)`

Description The probability density function of the gamma distribution, and its first two moments, are given by

$$p(x;a,b) = \ = x^{a-1}\frac{e^{-x/b}}{b^a\Gamma(a)},\quad x > 0, a, b > 0$$

$$E(X) = ab,$$

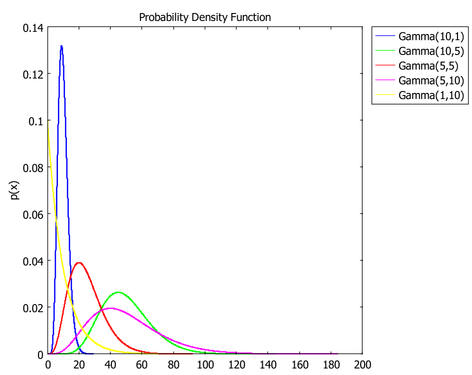
$$\mathrm{Cov}(X) = ab^2.$$

`a` and `b` must be positive. This is a child of `pdfclass`, and all of its methods apply to this distribution, in particular `pdfclass.estimate` and the plot functions.

The scale property `t*gammdist(a,b)=gammadist(a,t*b)` is implemented symbolically.

Example Some sample distributions

```
a=[10 10 5 5 1]; b=[1 5 5 10 10];
for i=1:length(a); X{i}=gammadist(a(i),b(i)); end
plot(X{:})
```



See Also `pdfclass`

Purpose

Compute an approximation of ideal transfer functions of type LP, HP, BS, BP

Syntax

`m=getfilter(n,fc,Property1,Value1,...)` TF object
`[b,a]=getfilter(n,fc,Property1,Value1,...)` Polynomial form

Description

The function computes both continuous-time and discrete-time filters. If the sampling frequency `fs = NaN`, then the continuous-time filter is returned. If you provide an `fs`, the cutoff frequencies must satisfy `fc<fs/2`. The default is `fs=2`, so fc is a normalized cutoff frequency in the interval [0, 1].

Input parameters:

TABLE 2-8: INPUT PARAMETERS

`fc`	Cutoff frequency or vector of frequencies, to be normalized by sampling frequency
`n`	Filter order

Optional parameters:

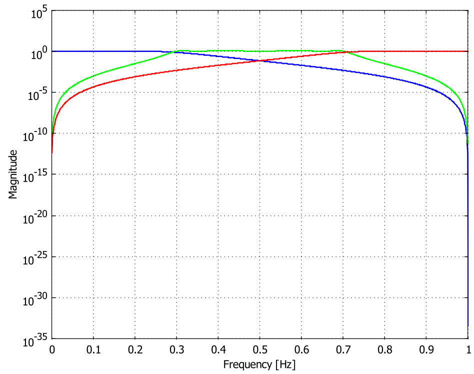
TABLE 2-9: OPTIONAL PARAMETERS

PROPERTY	VALUE	DESCRIPTION
`type`	`{'LP'} | 'HP' | 'BP'`	Type of filter
`alg`	`{'butter'} | 'cheby1'`	Algorithm
`fs`	`{2}`	Sampling frequency, fs=0 corresponds to a continuous filter.
`ripple`	`{0.5}`	Ripple in decibels for Chebyshev

Example

Compute a filter bank consisting of one LP, one BP, and one HP filter:

```
H1=getfilter(4,0.3,'alg','butter');
H2=getfilter(4,[0.3 0.7],'alg','cheby1','type','bp');
H3=getfilter(4,0.7,'type','hp');
bodeamp(H1,H2,H3)
grid
```



See Also `lti.bode`

getsignal

Purpose Generate standard signals as SIG objects.

Syntax `y=getsignal(ex,N,opt1,opt2);`

Description Standard examples of SIG objects of different properties are returned depending on the value of the string `ex`. The optional `N` defines the number of samples for discrete-time signals and time interval for continuous-time signals.

Discrete-time signal examples:

EXAMPLE	DESCRIPTION
`ones`	A unit signal [1 … 1] with nu=opt1 dimensions
`zeros`	A zero signal [0 … 0] with nu=opt1 dimensions
`pulse`	A single unit pulse [0 1 0...0]
`step`	A unit step [0 1 … 1]
`ramp`	A unit ramp with an initial zero and N/10 trailing ones
`square`	Square wave of length opt1
`sawtooth`	Sawtooth wave of length opt1
`pulsetrain`	Pulse train of length opt1
`sinc`	sin(pi*t)/(pi*t) with t=k*T where T=opt1
`diric`	The periodic sinc function sin(N*pi*t)/(N*sin(pi*t)) with t=k*T where T=opt1 and N=opt2
`prbs`	Pseudo-random binary sequence with basic period length opt1 (default N/100) and transition probability opt2 (default 0.5)
`gausspulse`	sin(pi*t)*p(t;sigma) with t=k*T where p is the Gaussian pdf, T=opt1 and sigma=opt2
`chirp1`	sin(pi*(t+a*t^2) with t=k*T where T=opt1 and a=opt2
`sin1`	One sinusoid in noise
`sin2`	Two sinusoids in noise
`sin2n`	Two sinusoids in LP noise

Continuous-time signal examples:

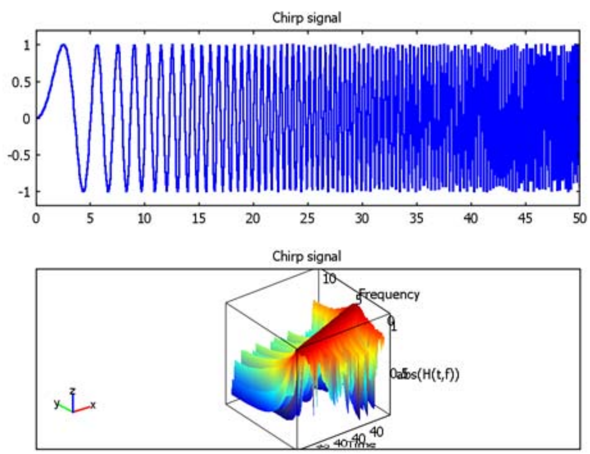
EXAMPLE	DESCRIPTION
`cones`	A unit signal [1 1] with t=[0 N] and ny=opt1
`czeros`	A zero signal [0 0] with t=[0 N] and ny=opt1
`impulse`	A single unit impulse [0 1 0 0] with t=[0 0 0 N]
`cstep`	A unit step [0 1 1] with t=[0 0 N]

EXAMPLE	DESCRIPTION
`csquare`	Square wave of length N and period length opt1
`impulsetrain`	Pulse train of length N and period length opt1
`cprbs`	Pseudo-random binary sequence with basic period length opt1 (default N/100) and transition probability opt2 (default 0.5)

Example

Generate a chirp signal, plot it, and look at its time-frequency description. The momentary frequency of a chirp is linearly increasing in time, but due to aliasing it will after a certain time be folded back to below the Nyquist frequency.

```
s=getsignal('chirp1');
subplot(2,1,1), plot(s)
subplot(2,1,2), surf(tfd(s))
```



See Also

`dbsignal`

Purpose Compute data window.

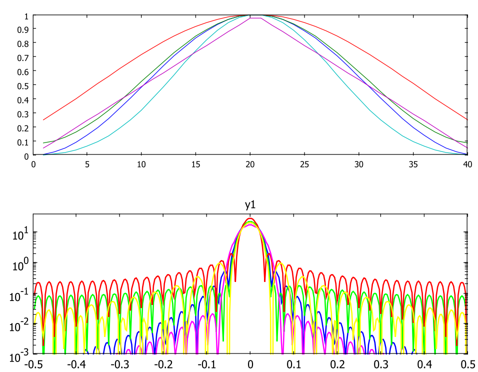
Syntax `w=getwindow(N,type,n)`

Description `N` is the length of the desired window, and type is one of `'box'`, `'hanning'`, `'hamming'`, `'kaiser'`, `'blackman'`, `'bartlett'`, or `'spline'`, generating a box, Hanning, Hamming, Kaiser, Blackman, Bartlett, or spline window, respectively. Here, the spline window type uses a uniform window convolved with itself `n` times.

This function is used internally in the SIG method `window` (which applies the window to a signal and further supports MIMO and MC simulations).

Example Compare five different windows in the time and frequency domains, respectively:

```
w1=getwindow(40,'hanning');
w2=getwindow(40,'hamming');
w3=getwindow(40,'kaiser');
w4=getwindow(40,'blackman');
w5=getwindow(40,'bartlett');
subplot(2,1,1), plot([w1 w2 w3 w4 w5],'linewidth',1)
W1=ft(sig(w1));
W2=ft(sig(w2));
W3=ft(sig(w3));
W4=ft(sig(w4));
W5=ft(sig(w5));
subplot(2,1,2)
semilogy(W1,W2,W3,W4,W5,'type',3,'Ylim',[1e-3 40])
```



See Also `sig`

Purpose Rescale the values in the matrix H uniformly to the interval $[0, 1]$.

Syntax

```
H=histeq(H);
```

Description In histogram equalization, a monotonous mapping is applied such that the original values in H are mapped to values in $[0, 1]$ so that all values are evenly spread as in a uniform distribution. The TFD plot functions all use `histeq` for improved visibility.

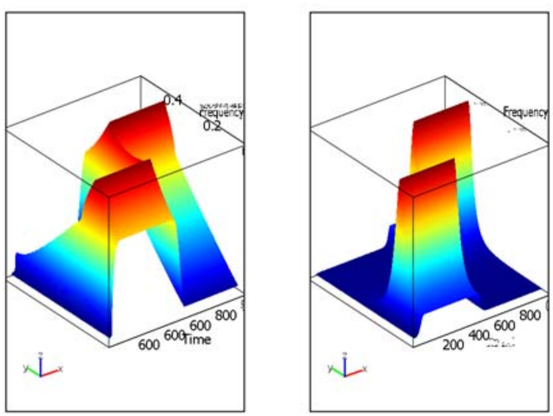
Example Map a random vector to uniformly distributed values in $[0, 1]$:

```
u=rand(1,6)
u =
    0.8729    0.0887    0.1474    0.8319    0.7369    0.8680
histeq(u)
ans =
         1    0.1667    0.3333    0.6667    0.5000    0.8333
```

The vector is normalized to values in the set $\{k/6\}$, $k = 1–6$.

The next example shows how the normalization of the z value in a 3D plot provides better visibility:

```
mt=exrarx('rar2',1000);
Mt=tfd(mt);
subplot(1,2,1),  surf(Mt,'histeq','on')
subplot(1,2,2),  surf(Mt,'histeq','off')
```



Because the height values in the left surf plot are rescaled, details between the minimum value and the maximum value are more visible.

See Also `rarx.surf`, `tfdplot`

Purpose Parent Linear Time-Invariant (LTI) model object.

Syntax The constructor is not available.

Description This object has no constructor. It contains plot methods that are in common for its children (SS, TF, ARX, FREQ):

FUNCTION	DESCRIPTION
`BODE`	Plots the Bode diagram of amplitude and phase
`BODEAMP`	Plots the Bode diagram of amplitude only
`BODEPHASE`	Plots the Bode diagram of phase only
`NYQUIST`	Plots the Nyquist curve
`ZPPLOT`	Plots the zeros and poles
`RLPLOT`	Plots the root locus

See Also `ss`, `tf`, `freq`, `lti.bode`, `lti.nyquist`, `lti.zpplot`, `lti.rlplot`

Purpose

Plot the Bode diagram of a system.

Syntax

`bode(G1,G2,...,Property1,Value1,...)`

Description

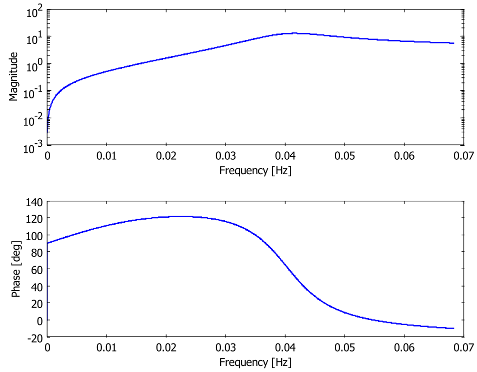
The system is first converted to a FREQ object by `F=freq(G)`. Then `bode` provides separate plots of the amplitude and phase in two subplots.

PROPERTY	VALUE	DESCRIPTION
`fmax`	`{'auto'}`	Maximum frequency
`plottype`	`{'plot'} | 'semilogx' | 'semilogy' | 'loglog'`	Decides if the x and y axes are plotted in linear or logarithmic scale.
`MC`	`{30}`	Number of Monte Carlo simulations
`conf`	`[0,100] {0}`	Confidence level (default 0, no levels plotted) from MC data
`conftype`	`{1} | 2`	1=shaded confidence region, 2=dashed bounds and median
`scatter`	`'on' | {'off'}`	Scatter plot of MC data
`col`	`{'bgrmyk'}`	Colors in order of appearance
`Xlim`		Limits on x axis
`Ylim`		Limits on y axis
`linewidth`	`{2}`	Line width on plots
`fontsize`	`{14}`	Font size
`title`	`{'on'}|'off'`	Display the title of the (first) model

Example

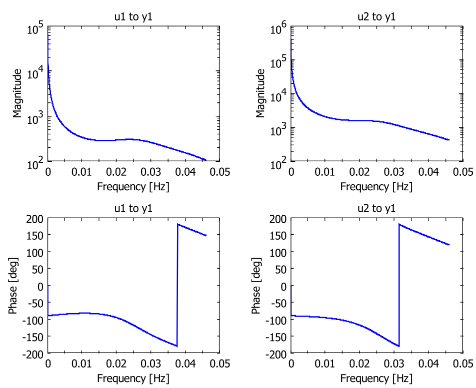
Bode diagram of random model:

```
G=rand(tf(4))

          s(s^3+4.7*s^2+5.9*s+0.67)
   Y(s) = -------------------------------- U(s)
       s^4+1.9*s^3+1.7*s^2+0.24*s+0.098

bode(G)
```



The same for a MIMO system:

```
G=rand(tf([4 4 2 2]));
bode(G)
```



See Also `lti`, `lti.nyquist`, `lti.zpplot`, `lti.rlplot`, `ss`, `tf`

Purpose

Plot the Nyquist curve of a system.

Syntax

`nyquist(G1,G2,...,Property1,Value1,...)`

Description

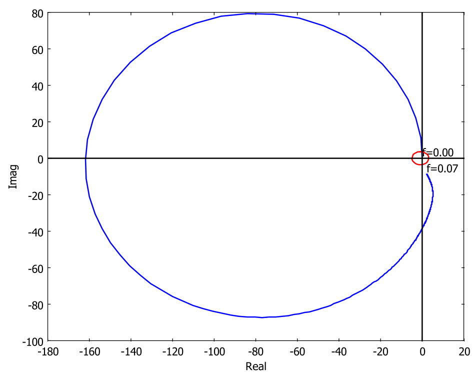
The system is first converted to a FREQ object by `freq(G)`. Then the function plots the complex numbers in $G(f)$ in the complex plane.

PROPERTY	VALUE	DESCRIPTION
`fmax`	`{'auto'}`	Maximum frequency
`plottype`	`{'plot'} | 'semilogx' | 'semilogy' | 'loglog'`	Decides if the x and y axes are plotted in linear or logarithmic scale.
`MC`	`{30}`	Number of Monte Carlo simulations
`conf`	`[0,100] {0}`	Confidence level (default 0, no levels plotted) from MC data
`conftype`	`{1} | 2`	1=shaded confidence region, 2=dashed bounds and median
`scatter`	`'on' | {'off'}`	Scatter plot of MC data
`axis`	`{gca}`	Axis handle where plot is added (does not apply for MIMO that creates subplots)
`col`	`{'bgrmyk'}`	Colors in order of appearance
`Xlim`		Limits on x axis
`Ylim`		Limits on y axis
`linewidth`	`{2}`	Line width on plots
`fontsize`	`{14}`	Font size
`title`	`{'on'}|'off'`	Display the title of the (first) model

Example

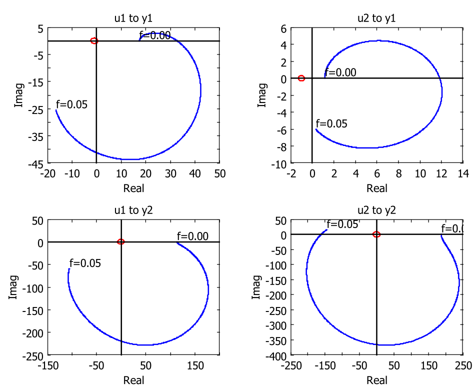
Nyquist curve of random model:

```
G=rand(tf(4))

                  s(s^3+4.7*s^2+5.9*s+0.67)
   Y(s) = ------------------------------------- U(s)
            s^4+1.5*s^3+0.29*s^2-0.004*s+2.5e-005

nyquist(G)
```



MIMO system:

```
G=rand(tf([4 4 1 2 2]));
nyquist(G)
```



See Also `lti`, `lti.bode`, `lti.zpplot`, `lti.rlplot`, `ss`, `tf`

Purpose

Plot the root locus of a system.

Syntax

`rlplot(G1,G2,...,Property1,Value1,...)`

Description

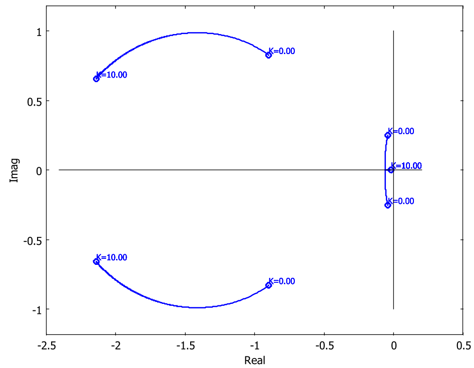
This function plots the poles of the system with K times unit feedback, corresponding to something like `Gc = feedback(G,K*eye(ny))`, in the complex plane as a function of the feedback gain K.

PROPERTY	VALUE	DESCRIPTION
`Kmax`	`{'auto'}`	Maximum K
`Kgrid`	`{'auto'}`	Grid for K
`MC`	`{30}`	Number of Monte Carlo simulations
`scatter`	`'on' | {'off'}`	Scatter plot of MC data
`axis`	`{gca}`	Axis handle where plot is added (does not apply for subplots in Bode diagrams of type 'both' or for MIMO)
`col`	`{'bgrmyk'}`	Colors in order of appearance
`Xlim`		Limits on x axis
`Ylim`		Limits on y axis
`linewidth`	`{2}`	Line width on plots
`fontsize`	`{14}`	Font size
`title`	`{'on'}|'off'`	Display the title of the (first) model

Example

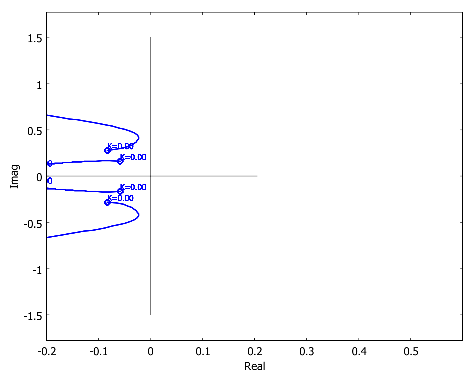
Root locus of random model:

```
G=rand(tf(4))

            s(s^3+4.7*s^2+5.9*s+0.67)
   Y(s) = -------------------------------- U(s)
          s^4+1.9*s^3+1.7*s^2+0.24*s+0.098

rlplot(G)
```



Root locus of square MIMO model:

```
G22=rand(tf([4 4 1 2 2]));
rlplot(G22,'Xlim',[-0.2 0.6])
```



See Also `lti`, `lti.bode`, `lti.nyquist`, `lti.zpplot`, `lti.rlplot`, `ss`, `tf`

Purpose

Plot the zeros and poles of a system.

Syntax

`zpplot(G1,G2,...,Property1,Value1,...)`

Description

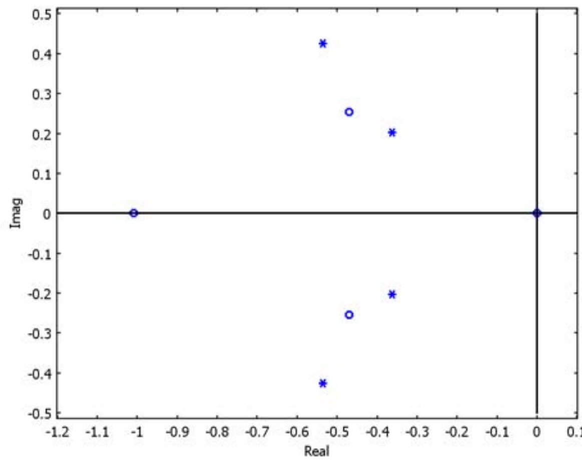
The system is first converted to zeros and poles using `zpk(G)`. These are then illustrated as circles and stars, respectively, in the complex plane.

PROPERTY	VALUE	DESCRIPTION
`MC`	`{30}`	Number of Monte Carlo simulations
`scatter`	`'on'|{'off'}`	Scatter plot of MC data
`axis`	`{gca}`	Axis handle where plot is added (does not apply for MIMO)
`col`	`{'bgrmyk'}`	Colors in order of appearance
`Xlim`		Limits on x axis
`Ylim`		Limits on y axis
`linewidth`	`{2}`	Line width on plots
`fontsize`	`{14}`	Font size
`title`	`{'on'}|'off'`	Display the title of the (first) model

Example

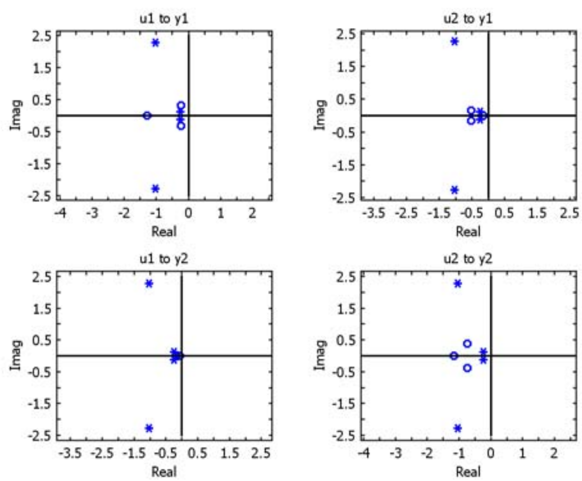
Zero-pole plot of random model:

```
G=rand(tf(4))

            s(s^3+1.9*s^2+1.2*s+0.29)
   Y(s) = -------------------------------- U(s)
         s^4+1.8*s^3+1.4*s^2+0.52*s+0.081

zpplot(G)
```



Zero-pole plot of MIMO model:

```
G=rand(tf([4 4 1 2 2]));
zpplot(G)
```



See Also `lti`, `lti.bode`, `lti.nyquist`, `lti.rlplot`, `ss`, `tf`

Purpose Stable implementation of arbitrary transfer function on polynomial form.

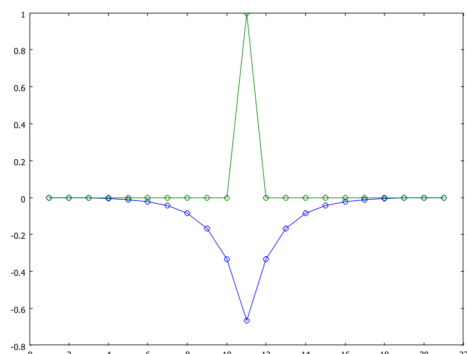
Syntax `y=ncfilter(b,a,u)`

Description The zeros of the a polynomial can be arbitrary (both inside and outside the unit circle). A stable but noncausal implementation is applied to the input vector u, using the following algorithm:

1. Split the roots of `a` and `b` into two groups; the ones inside the unit circle and the ones outside the unit circle. This gives the polynomials `af`, `ab`, `bf`, and `bb`, and the gain `k`.
2. Call the function `filtfilt` with `y=k*filtfilt(bf,af,bb,ab,u,M)`.

Example

```
a=poly([0.5 2]);
u=[zeros(10,1);1;zeros(10,1)];
y=ncfilter(1,a,u);
plot([y u],'-o')
```



See Also `tf.ncfilter`, `filtfilt`

Purpose The Gaussian (normal) distribution

Syntax `X=ndist(m,P)`

Description The probability density function of the Gaussian (normal) distribution, and its first two moments, are given by

$$p(x;\mu, P) = \, = \frac{1}{(2\pi \det((P))^{n/2})} e^{-0.5(x-\mu)^{-1}(x-m)},$$

$$E(X) = \mu,$$

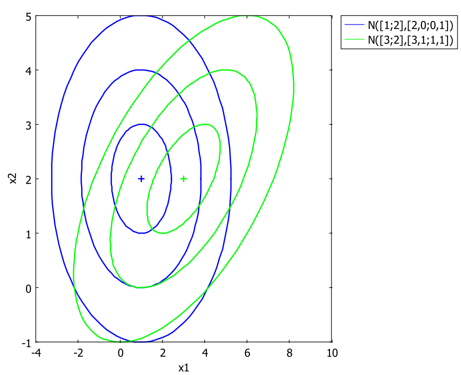
$$\mathrm{Cov}(X) = P.$$

`P` must be a positive definite matrix. This is a child of `pdfclass`, and all of its methods apply to this distribution, in particular `pdfclass.estimate` and the plot functions.

The linearity property of the normal distribution is implemented symbolically.

Example Illustration of the linearity property:

```
X=ndist([1;2],[2 0;0 1]);
Y=[1 1;0 1]*X
N([3;2],[3,1;1,1])
plot2(X,Y)
```



See Also `pdfclass`

Purpose Monte Carlo approximation of a nonlinear mapping.

Syntax `Y=mceval(X,f,NMC,varargin)`

Description The mean and covariance of the nonlinear mapping defined by `Y=f(X,varargin{:})` are approximated using Monte Carlo sampling.

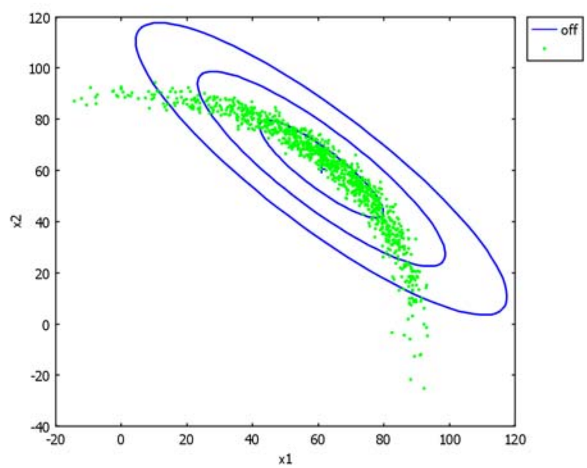
Example Compute an approximation to a quadratic form of a Gaussian variable.

```
X=ndist(0,1)
N(0,1)
h=inline('(x+1).^2');
Ymc = mceval(X,h,1000)
N(1.93,5.52)
```

The correct mean and variance are 2 and 6, respectively. The accuracy increases with increased number of samples.

Convert Gaussian distributed range and bearing to Cartesian coordinates

```
R=90+ndist(0,5);
Phi=pi/4+ndist(0,0.1);
Nut=mceval([R;Phi],inline('[x(1,:).*cos(x(2,:));
x(1,:).*sin(x(2,:))]'))
N([60.9;60.6],[355,-314;-314,362])
plot2(Nut,[R*cos(Phi);R*sin(Phi)],'legend','off')
```



See Also `ndist.uteval`, `ndist.tt1eval`, `nl.nltf`

Purpose First-order Taylor approximation of a nonlinear mapping.

Syntax `X=tt1eval(X1,X2)`

Description The mean and covariance of the nonlinear mapping defined by `Y=f(X,varargin{:})` are approximated using a first-order Taylor expansion.

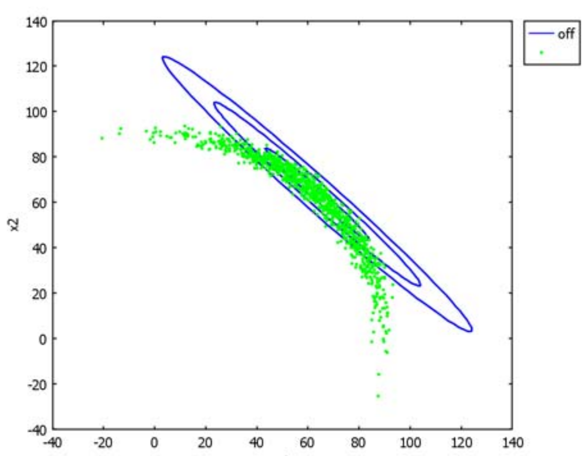
Example Compute an approximation to a quadratic form of a Gaussian variable.

```
X=ndist(0,1)
N(0,1)
h=inline('(x+1).^2');
Ymc = tt1eval(X,h)
N(1,4)
```

The correct mean and variance are 2 and 6, respectively. The accuracy increases with increased number of samples.

Convert Gaussian distributed range and bearing to Cartesian coordinates

```
R=90+ndist(0,5);
Phi=pi/4+ndist(0,0.1);
Nut=tt1eval([R;Phi],inline('[x(1,:).*cos(x(2,:));
x(1,:).*sin(x(2,:))]'))
N([63.6;63.6],[408,-403;-403,408])
plot2(Nut,[R*cos(Phi);R*sin(Phi)],'legend','off')
```



See Also `ndist.mceval`, `ndist.uteval`, `nl.nltf`

Purpose Unscented transform approximation of a nonlinear mapping.

Syntax `X=uteval(X1,X2)`

Description The mean and covariance of the nonlinear mapping defined by `Y=f(X,varargin{:})` are approximated using the unscented transform.

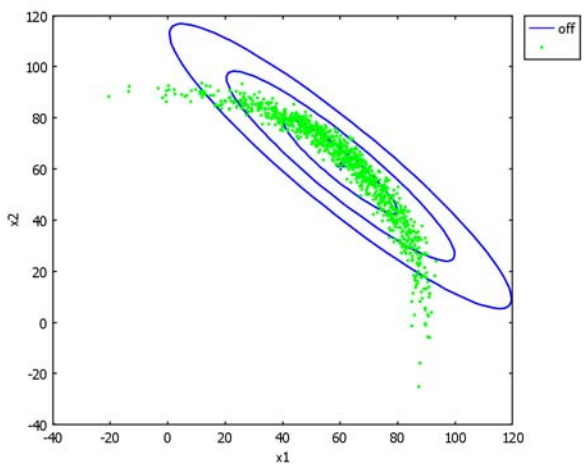
Example Compute an approximation to a quadratic form of a Gaussian variable.

```
X=ndist(0,1)
N(0,1)
h=inline('(x+1).^2');
Ymc = uteval(X,h)
N(2,6)
```

The correct mean and variance are 2 and 6, respectively.

Convert Gaussian distributed range and bearing to Cartesian coordinates:

```
R=90+ndist(0,5);
Phi=pi/4+ndist(0,0.1);
Nut=mceval([R;Phi],inline('[x(1,:).*cos(x(2,:));
x(1,:).*sin(x(2,:))]'))
N([60.1;61.1],[393,-342;-342,344])
plot2(Nut,[R*cos(Phi);R*sin(Phi)],'legend','off')
```



See Also `ndist.mceval`, `ndist.tt1eval`, `nl.nltf`

Purpose NL is the model object for nonlinear time-invariant systems.

Syntax `m=nl(f,h,nn,fs)`

Description The general definition of the NL model is in continuous time

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), u(t); \theta) + v(t), \\ y(t_k) &= h(t_k, x(t_k), u(t_k); \theta) + e(t_k), \\ x(0) &= x_0. \end{aligned}$$

and in discrete time

$$\begin{aligned} x_{k+1} &= f(k, x_k, u_k; \theta) + v_k, \\ y_k &= h(k, x_k, u_k, e_k; \theta). \end{aligned}$$

The involved signals and functions are:

- x denotes the state vector.
- t is time, and t_k denotes the sampling times that are monotonously increasing. For discrete time models, k refers to time kT, where T is the sampling interval.
- u is a known (control) input signal.
- v is an unknown stochastic input signal specified with its probability density function $p_v(v)$.
- e is a stochastic measurement noise specified with its probability density function $p_e(e)$.
- x_0 is the known or unknown initial state. In the latter case, it may be considered as a stochastic variable specified with its probability density function $p_0(x_0)$.
- θ contains the unknown parameters in the model. There might be prior information available, characterized with its mean and covariance.

For deterministic systems, when v and e are not present above, these model definitions are quite general. The only restriction from a general stochastic nonlinear model is that both process noise v and measurement noise e have to be *additive*.

The constructor `m=nl(f,h,nn)` has three mandatory arguments:

- The argument f defines the dynamics and is entered in one of the following ways:
  - A string. Example: `f='-th*x(1,:).^2';`.
  - An inline function. Example:

```
f=inline('-x(1,:).^2','t','x','u','th');
```

- An M-file. Example:

```
function f=fun(t,x,u,th)
f=-th*x(1,:).^2;
```

It is important to use the standard model parameter names `t`, `x`, `u`, and `th`. For inline functions and M-files, the number of arguments must be all these four even if some of them are not used, and the order of the arguments must follow this convention. The complete indexing above (also avoiding `end`) is recommended, though a simplified notation as `f='-th*x^2';` also works in most cases.

- `h` is defined analogously to `f` above.
- `nn=[nx,nu,ny,nth]` denotes the orders of the input parameters. These must be consistent with the entered `f` and `h`. This apparently trivial information must be provided by the user, since it is hard to unambiguously interpret all combinations of input dimensions that are possible otherwise. All other tests are done by the constructor, which calls both functions `f` and `h` with zero inputs of appropriate dimensions according to `nn`, and validates the dimensions of the returned outputs.

All other parameters are set separately:

- `pv`, `pe`, `px0` are distributions for the process noise, measurement noise and initial state, respectively. All of these are entered as objects in the `pdfclass`, or as covariance matrices when a Gaussian distribution is assumed.
- `th`, `P` are the fields for the parameter vector and optional covariance matrix. The latter option is used to represent uncertain systems. Only the second order property of model uncertainty is currently supported for NL objects, in contrast to the LTI objects SS and TF.
- `fs` is similarly to the LTI objects the sampling frequency, where the convention is that `fs=NaN` means continuous time systems (which is used by default). All NL objects are set to continuous time models in the constructor, and the user has to specify a numeric value of `fs` after construction if a discrete model is wanted.
- `xlabel`, `thlabel`, `ulabel`, `ylabel`, and `name` are used to name the variables and the model, respectively. These names are inherited after simulation in the SIG object, for instance.

The methods of the NL object are listed below.

TABLE 2-10: METHODS FOR THE NL OBJECT

METHOD	DESCRIPTION
ARRAYREAD	Used to pick out sub-systems by indexing. Ex: m(2,:) picks out the dynamics from all inputs to output number 2. Only the output dimension can be decreased for NL objects, as opposed to LTI objects.
DISPLAY	Returns an ascii formatted version of the NL model
ESTIMATE	Estimates/calibrates the parameters in an NL system using data
SIMULATE	Simulates the NL system using daspk
NL2SS	Returns a linearized state space model
EKF	Implements the extended Kalman filter for state estimation
NLTF	Implements a class of Riccati-free filters, where the unscented Kalman filter and extended Kalman filter are special cases
PF	Implements the particle filter for state estimation
CRLB	Computes the Cramer-Rao Lower Bound for state estimation

Example

A simple nonlinear system with one parameter:

```
m=nl('-th(1)*x(1,:).^2-th(2)*x(1,:)','x',[1 0 1 2])
NL object
 dx/dt = -th(1)*x(1,:).^2-th(2)*x(1,:)
     y = x
   x0' = [0]
   th' = [0         0]
```

The van der Pol system with measurement noise:

```
f='[x(2,:);(1-x(1,:).^2).*x(2,:)-x(1,:)]';
h='x';
m=nl(f,h,[2 0 2 0]);
m.name='Van der Pol system';
m.x0=[2;0];
m.pe=ndist([0;0],0.1*eye(2));
m
NL object: Van der Pol system
 dx/dt = [x(2,:);(1-x(1,:).^2).*x(2,:)-x(1,:)]
     y = x + N([0;0],[0.1,0;0,0.1])
   x0' = [2         0]
```

See Also

ss

Purpose

Compute the parametric Cramer Rao lower bound for state estimation.

Syntax

```
x=crlb(m,z,,Property1,Value1,...)
```

Description

The Cramer Rao lower bound (CRLB) is defined as the minimum covariance any unbiased estimator can achieve. The parametric CRLB for the NL model can be computed as

$$
\begin{aligned}
P_{k+1|k} &= A_k P_{k|k} A_k^T + \bar{Q}_k, \\
P_{k+1|k+1} &= P_{k+1|k} - P_{k+1|k} C_k^T (C_{k+1|k} C_k^T + \bar{R}_k)^{-1} C_{k+1|k}, \\
A_k^T &= \frac{\partial}{\partial x_k} f^T(x_k), \\
C_k^T &= \frac{\partial}{\partial x_k} h^T(x_k).
\end{aligned}
$$

The state and measurement noise covariances Q and R are here replaced by the overlined matrices. For Gaussian noise, these coincide. Otherwise, Q and R are scaled with the *intrinsic accuracy* of the noise distribution, which is strictly smaller than one for non-Gaussian noise.

Here, the gradients are defined at the true states. That is, the parametric CRLB can only be computed for certain known trajectories. The code is essentially the same as for the EKF, with the difference that the true state taken from the input SIG object.

The arguments are as follows:

- `m` is a NL object defining the model.
- `z` is a SIG object defining the true state x. The outputs y and inputs u are not used for CRLB computation, but passed to the output SIG object.
- `x` is a SIG object with covariance lower bound `Pxcrlb=x.Px` for the states, and `Pxcrlb=x.Px` for the outputs, respectively.

The optional parameters are summarized in the table below.

TABLE 2-11: OPTIONAL PARAMETERS IN THE CRLB FUNCTION

PROPERTY	VALUE	DESCRIPTION
k	k>0 {0}	Prediction horizon: 0 for filter (default), 1 for one-step ahead predictor.
P0	{[]}	Initial covariance matrix. Scalar value scales identity matrix. Empty matrix gives a large identity matrix.
x0	{[]}	Initial state matrix. Empty matrix gives a zero vector.

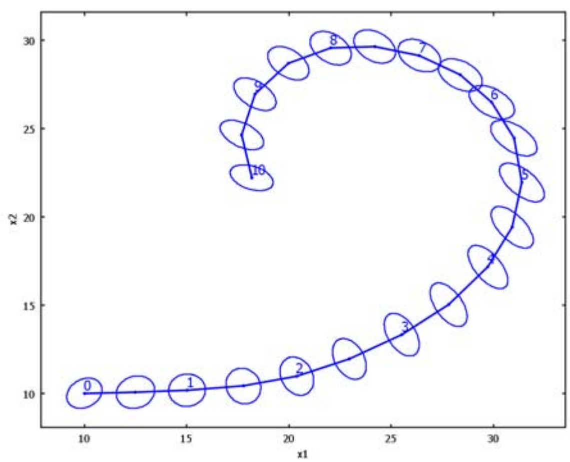
TABLE 2-11: OPTIONAL PARAMETERS IN THE CRLB FUNCTION

PROPERTY	VALUE	DESCRIPTION
Q	{[]}	Process noise covariance (overrides the value in m.Q). Scalar value scales m.Q.
R	{[]}	Measurement noise covariance (overrides the value in m). Scalar value scales m.R.

Example

The CRLB for a nonlinear tracking model is computed for one realization of a simulated trajectory.

```
m=exnl('ctpv2d'); % coordinated turn model
z=simulate(m,10); % ten seconds trajectory
zcrlb=crlb(m,z);
xplot2(zcrlb,'conf',90)
```



According to the theory, no nonlinear filter can compute better estimates than these confidence ellipsoids indicate.

See Also

`nl.ekf`, `nl.nltf`, `nl.pf`

Purpose Implementation of the extended Kalman filter (EKF) for state estimation.

Syntax

```
[x,V]=ekf(m,z,Property1,Value1,...)
```

Description The EKF implements the following recursion (where some of the arguments to f and h are dropped for simplicity):

$$\hat{x}_{k+1|k} = f(\hat{x}_{k|k})$$

$$P_{k+1|k} = f'_x(\hat{x}_{k|k})P_{k|k}(f'_x(\hat{x}_{k|k}))^T + f'_v(\hat{x}_{k|k})Q_k(f'_x(\hat{x}_{k|k}))^T$$

$$S_k = h'_x(\hat{x}_{k|k-1})P_{k|k-1}(h'_x(\hat{x}_{k|k-1}))^T + h'_e(\hat{x}_{k|k-1})R_k(h'_e(\hat{x}_{k|k-1}))^T$$

$$K_k = P_{k|k-1}(h'_x(\hat{x}_{k|k-1}))^T S_k^{-1}$$

$$\varepsilon_k = y_k - h(\hat{x}_{k|k-1})$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k\varepsilon_k$$

$$P_{k|k} = P_{k|k-1} - P_{k|k-1}(h'_x(\hat{x}_{k|k-1}))^T S_k^{-1} h'_x(\hat{x}_{k|k-1})P_{k|k-1}.$$

The EKF can be expected to perform well when the linearization error is small. Here small relates both to the state estimation error and the degree of nonlinearity in the model. As a rule of thumb, EKF works well in the following cases:

- The model is almost linear.
- The SNR is high and the filter does converge. In such cases, the estimation error will be small, and the neglected rest term in a linearization becomes small.
- If either process or measurement noise are multimodel (many peaks), then EKF may work fine, but nevertheless perform worse than nonlinear filter approximations as the particle filter.

Design guidelines include the following useful tricks to mitigate linearization errors:

- Increase the state noise covariance `Q` to compensate for higher-order nonlinearities in the state dynamic equation.
- Increase the measurement noise covariance `R` to compensate for higher-order nonlinearities in the measurement equation.

The arguments are as follows:

- `m` is a NL object defining the model.

- `z` is a SIG object with measurements y, and inputs u if applicable. The state field is not used by the EKF.
- `x` is a SIG object with state estimates. `xhat=x.x` and signal estimate `yhat=x.y`.

The optional parameters are summarized in the table below.

TABLE 2-12: OPTIONAL PARAMETERS IN THE EKF FUNCTION

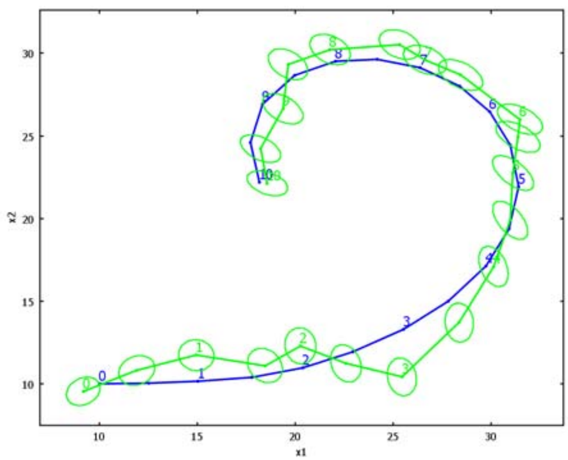
PROPERTY	VALUE	DESCRIPTION
k	k>0 {0}	Prediction horizon: 0 for filter (default), 1 for one-step ahead predictor.
P0	{[]}	Initial covariance matrix. Scalar value scales identity matrix. Empty matrix gives a large identity matrix.
x0	{[]}	Initial state matrix. Empty matrix gives a zero vector.
Q	{[]}	Process noise covariance (overrides the value in m.Q). Scalar value scales m.Q.
R	{[]}	Measurement noise covariance (overrides the value in m). Scalar value scales m.R.

The main difference to the KF is that EKF does not predict further in the future than one sample, and that smoothing is not implemented. Further, there is no square root filter implemented, and there is no such thing as a stationary EKF.

Example

Simulate a trajectory for a nonlinear target tracking model, apply the EKF to estimate the state, and plot the position estimates together with the true trajectory.

```
m=exnl('ctpv2d'); % coordinated turn model
z=simulate(m,10); % ten seconds trajectory
zhat=ekf(m,z);    % EKF state estimation
xplot2(z,zhat,'conf',90);
```



See Also `nl`, `nl.nltf`, `nl.pf`, `nl.crlb`

Purpose Estimate, or calibrate, the parameters in an NL system using measured data.

Syntax `[mhat,res]=estimate(m,z,property1,value1,...)`

Description This function estimates the parameters and initial state in an NL object in either discrete

$$x_{k+1} = f(k, x_k, u_k; \theta) + v_k,$$
$$y_k = h(k, x_k, u_k, e_k; \theta).$$

or continuous time

$$\dot{x}(t) = f(t, x(t), u(t); \theta) + v(t),$$
$$y(t_k) = h(t_k, x(t_k), u(t_k); \theta) + e(t_k),$$
$$x(0) = x_0.$$

In contrast to other model estimation methods, quite good an initial estimate is required here, which motives the term model calibration rather than model estimation.

The optimization methods using the NLS algorithm as implemented in `nls` leads to the following calibration algorithm:

1 Initialize the parameter vector $\eta = (\theta, x_0)$ using the values of `x0` and `th` in the NL object.

2 Iterate in i:

$$\eta^{i+1} = \eta^i - \mu^j (J(\eta^i) J^T(\eta^i))^{-1} J(\eta^i) \varepsilon(\eta^i)$$

3 In each iteration, check if the cost function has decreased. Otherwise, half the step size μ until either the cost function decreases or `maxhalf` iterations in the line search is reached.

4 Continue until j = `maxiter` or the relative change in cost function is smaller than `ctol`, or the maximum element in the gradient J is smaller than `gtol`.

A numeric gradient J is computed according to

$$\frac{\partial}{\partial \eta_i} \hat{y}(t_k, \eta) \approx \frac{\hat{y}(t_k, \eta + \mathrm{he}_i) - \hat{y}(t_k, \eta - \mathrm{he}_i)}{2h},$$

which is evaluated for each unit vector.

The internal parameters in the algorithm are controlled by property-value pairs for both usages `mhat=estimate(m,z,Property1,Value1,...)` and `mhat=nls(m,z,Property1,Value1,...)`. The property-value pairs are listed in the table below.

TABLE 2-13: PROPERTY-VALUE PAIRS FOR THE NLS FUNCTION

PROPERTY	VALUE {DEFAULT}	DESCRIPTION
thmask	{ones(1,nth)}	Binary search mask for parameter vector
x0mask	{ones(1,nx)}	Binary search mask for initial state vector
x0	Cell array	In case z is a cell with multiple data sets, different known initial conditions can be set. x0{i} is the initial state for z{i}
alg		Optimization algorithm
	{'gn'}	Gauss-Newton
	'rgn'	Robust Gauss-Newton, where the Hessian *H=JJ'* is robustified by adding a small identity matrix.
	'lm'	Levenberg-Marquardt, where the line search is replaced by a region search.
	'sd'	steepest-descent, where the Hessian is replaced with the identity matrix.
disp	{0}|1	Display status of the iterations
maxiter	{50}	Maximum number of iterations in search direction
maxhalf	{50}	Maximum number of iterations in the line search.
gtol	{1e-4}	Tolerance for the gradient.
ctol	{1e-4}	Minimum relative decrease in the cost function before the search is terminated.
svtol	{1e-4}	Lower bound for the singular values of the Jacobian in robust Gauss-Newton
numgrad	(0)|1	Force a numerical computation of the gradient even if a gradient m.J is specified

The direct call to NLS enables a second output structure `[mhat,res]=nls(m,z)`, which gives access to internal variables, as summarized in the following table.

TABLE 2-14: INTERNAL FIELDS IN THE NLS STRUCTURE

FIELD NAME	DESCRIPTION
res.TH	Parameter values at each iteration
res.V	Value of the cost function at each iteration

TABLE 2-14: INTERNAL FIELDS IN THE NLS STRUCTURE

FIELD NAME	DESCRIPTION
res.dV	The gradient at each iteration
res.m	The obtained model at each iteration as a cell array
res.sl	The step sizes at each iteration
res.sol	The obtained solution (thhat)
res.term	Text string with cause of termination
res.P	Covariance (estimated) for the parameters
res.Rhat	Covariance (estimated) for the measurements

Example

Define a first order parametric NL object, simulate data and use these to calibrate a model with the same structure but with uncertain initial state and parameters.

```
m0=nl('-th(1)*x^2-th(2)*x','x',[1 0 1 2]);
NL constructor warning: try to vectorize f for increased speed
m0.th=[1;1];
m0.x0=1;
m0.fs=1;
z=simulate(m0,0:10);
m=m0;           % No model error
m.x0=0.8;       % Prior on initial state
m.th=[0.9;1.1]; % Prior on parameters
mhat=estimate(m,z)
NL object:  (calibrated from data)
x(t+1) = -th(1)*x^2-th(2)*x
     y = x + N(0,1.47)
   x0' = [0.67] + N(0,1.1e-008)
   th' = [1.5        1.7]
   std = [0.0014       0.57]
```

See Also

nl, nls.m

Purpose

Compute a linearized model using first-order Taylor expansion around a nominal state value.

Syntax

```
[mout,zout]=nl2ss(m,z)
```

Description

The linear state-space model is defined by the following Taylor expansion

```
x+ = f(z.t,z.x,z.u) + df(z.t,z.x,z.u)/dx *(x(t)-z.x)
+ df(z.t,z.x,z.u)/du *(u(t)-z.u) +v(t)
y(t) = h(z.t,z.x,z.u) + dh(z.t,z.x,z.u)/dx *(x(t)-z.x)
+ dh(z.t,z.x,z.u)/du *(u(t)-z.u) + e(t)
```

Here `x+` denotes either dx/dt or $x(t+1)$ for continuous and discrete time models, respectively. The numeric gradients $A = df/dx$, $B = df/du$, $C = dh/dx$, and $D = dh/dx$ are computed around the linearization point specified in the SIG object `z.x` (one sample). The model is then

```
 x+  = ux(t) + A*x(t) + B*u(t) + v(t)
y(t) = uy(t) + C*x(t) + D*u(t) + e(t)
```

where the extra inputs are defined as

```
ux(t) = f(z.t,z.x,z.u) - A * z.x - B * z.u
uy(t) = h(z.t,z.x,z.u) - C * z.x - D * z.u
```

Using an augmented input vector `ua(t)=[u' ux' uy']'`, the returned model is

```
mout <-> [A, [B I 0], C, [D 0 I]]
zout.y = z.y
zout.x = z.x
zout.u = [z.u; ux; uy];
```

Example

A simple first-order nonlinear system is linearized around the state $x = 2$:

```
mnl=nl('-x(1,:).^2-x(1,:)','x',[1 0 1 0])
NL object
 dx/dt = -x(1,:).^2-x(1,:)
     y = x
   x0' = [0]

zin=sig(0,mnl.fs,[],2);
[mss,zout]=nl2ss(mnl,zin);
mss
d/dt x(t) = -5 x(t) + (1  0) u(t)

y(t) = 1 x(t) + (0  1) u(t)


zout.u
ans =
```

```
    4   0
```

In the state dynamics, the augmented input signal in `zout` consists of the constant term that occurs in the Taylor expansion; the output dynamics is unaffected because it is linear already.

See Also `nl`, `ss.ss2nl`

Purpose

Implementation of the unscented (UKF) and extended (EKF) Kalman filters using nonlinear transformations completely avoiding Riccati equations.

Syntax

`[x,V]=nltf(m,z,Property1,Value1,...)`

Description

The main arguments are:

- `m` is the NL object specifying the model.
- `z` is an input SIG object with measurements.
- `x` is an output SIG object with state estimates `xhat=x.x` and signal estimate `yhat=x.y`.

The algorithm with script notation basically works as follows:

1 Time update:
   a Let `xbar = [x;v] = N([xhat;0];[P,0;0,Q])`
   b Transform approximation of `x(k+1) = f(x,u)+v` gives `xhat, P`
2 Measurement update:
   a Let `xbar = [x;e] = N([xhat;0];[P,0;0,R])`
   b Transform approximation of `z(k) = [x; y] = [x; h(x,u)+e]` provides `zhat=[xhat; yhat]` and `Pz=[Pxx Pxy; Pyx Pyy]`
   c The Kalman gain is `K=Pxy*inv(Pyy)`
   d `xhat = xhat+K*(y-yhat)` and `P = P-K*Pyy*K'`

The transform in 1b and 2b can be chosen arbitrarily using the `uteval`, `tt1eval`, `tt2eval`, and `mceval` in the `ndist` object.

Note: the NL object must be a function of indexed states, so always write for instance `x(1,:)` or `x(1:nx,:)` (avoid using `end`), even for scalar systems. The reason is that the state vector is augmented, so any nonindexed `x` will cause errors.

TABLE 2-15: PROPERTY-VALUE PAIRS FOR THE NLTF FUNCTION

PROPERTY	VALUE	DESCRIPTION
k	k>0 {0}	Prediction horizon:
		0 for filter (default)
		1 for one-step ahead predictor,
P0	{[]}	Initial covariance matrix. Scalar value scales identity matrix. Empty matrix gives a large identity matrix
x0	{[]}	Initial state matrix (overrides the value in m.x0). Empty matrix gives a zero vector

TABLE 2-15: PROPERTY-VALUE PAIRS FOR THE NLTF FUNCTION

PROPERTY	VALUE	DESCRIPTION
Q	{[]}	Process noise covariance (overrides m.Q). Scalar value scales m.Q
R	{[]}	Measurement noise covariance (overrides m.R). Scalar value scales m.R
tup		Non-linear transformation in the time update
	{'ut'}	The unscented Kalman filter (UKF) based on the uteval method of the ndist object
	'tt1'	Variant of the EKF, based on a first order Taylor expansion in the tt1eval method of the ndist object
	'tt2'	Second order corrected EKF, based on a second order Taylor expansion in the tt1eval method of the ndist object
	'mc'	Monte Carlo version of the EKF, based on Monte Carlo sampling in the mceval method of the ndist object
mup		Non-linear transformation in the measurement update
	{'ut'}	The unscented Kalman filter (UKF) based on the uteval method of the ndist object
	'tt1'	Variant of the EKF, based on a first order Taylor expansion in the tt1eval method of the ndist object
	'tt2'	Second order corrected EKF, based on a second order Taylor expansion in the tt1eval method of the ndist object
	'mc'	Monte Carlo version of the EKF, based on Monte Carlo sampling in the mceval method of the ndist object
ukftype	{'std'} | 'wan'	Standard or WAN unscented transform
ukfpar	{[]}	Parameters in UKF
		For std, par=w0 {w0=1-n/3}
		For wan, par=[beta,alpha,kappa] {[2 1e-3 0]}
NMC	{100}	Number of Monte Carlo samples for mceval

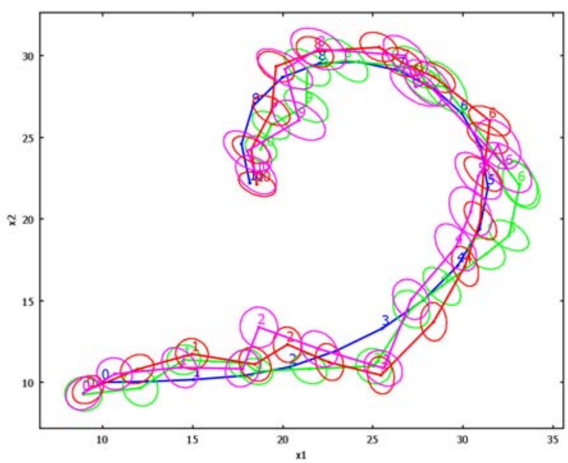
One important difference to running the standard EKF in common for all these filters is that the initial covariance must be chosen carefully. It cannot be taken as a huge identity matrix, which works well when a Riccati equation is used. The problem is most easily explained for the Monte Carlo method. If P0 is large, random number all over the state space are generated and propagated by the measurement relation. Most certainly, none of these come close the observed measurement, and the problem is obvious. Otherwise, the same caution as for the EKF should be taken, where Q and R can be increased to mitigate the effect of (first and second order, respectively) linearization errors.

As another guideline, always use tt1 when the dynamic model or measurement relation is linear.

Example

The most natural combination of nonlinear transformation combinations are evaluated on a target tracking example:

```
m=exnl('ctpv2d');  % coordinated turn model
z=simulate(m,10);  % ten seconds trajectory
zukf=genfilt(m,z); % UKF state estimation
zekf=genfilt(m,z,'tup','tt1','mup','tt1'); % EKF variant
zmc=genfilt(m,z,'tup','mc','mup','mc'); % EKF variant
xplot2(z,zukf,zekf,zmc,'conf',90);
```



See Also

nl, nl.ekf, nl.nltf, nl.crlb

Purpose Implementation of a particle filter for state estimation in nonlinear systems.

Syntax

```
zhat=pf(m,z,Property1,Value1,...)
```

Description The `pf` method of the NL object implements the standard SIR filter. The principal code is given below:

```
y=z.y.';
u=z.u.';
xp=ones(Np,1)*m.x0.' + rand(m.px0,Np);    % Initialization
for k=1:N;
   % Time update
   v=rand(m.pv,Np);                        % Random process noise
   xp=m.f(k,xp,u(:,k),m.th).'+v;           % State prediction
   % Measurement update
   yp=m.h(k,xp,u(k,:).',m.th).';          % Measurement prediction
   w=pdf(m.pe,repmat(y(:,k).',Np,1)-yp);  % Likelihood
   xhat(k,:)=mean(repmat(w(:),1,Np).*xp); % Estimation
   [xp,w]=resample(xp,w);                  % Resampling
   xMC(:,k,:)=xp;                         % MC uncertainty repr.
end
zhat=sig(yp.',z.t,u.',xhat.',[],xMC);
```

The arguments are as follows:

- `m` is a NL object defining the model.
- `z` is a SIG object with measurements y, and inputs u if applicable. The state field is not used by the EKF.
- `x` is a SIG object with state estimates. `xhat=x.x` and signal estimate `yhat=x.y`.

The optional parameters are summarized in the table below.

TABLE 2-16: OPTIONAL PARAMETERS FOR THE PF FUNCTION

PROPERTY	VALUE	DESCRIPTION
Np	Np>0 {0}	Number of particles
k	k=0,1	Prediction horizon:
		0 for filter (default)
		1 for one-step ahead predictor,
	sampling	
	{'simple'}	Standard algorithm
	'systematic'	
	'residual'	

TABLE 2-16: OPTIONAL PARAMETERS FOR THE PF FUNCTION

PROPERTY	VALUE	DESCRIPTION
	'stratified'	
animate	{[]},ind	Animate states x(ind)

The particle filter suffers from some divergence problems caused by sample impoverishment. In short, this implies that one or a few particles are contributing to the estimate, while all the others have almost zero weight. Some mitigation tricks are useful to know:

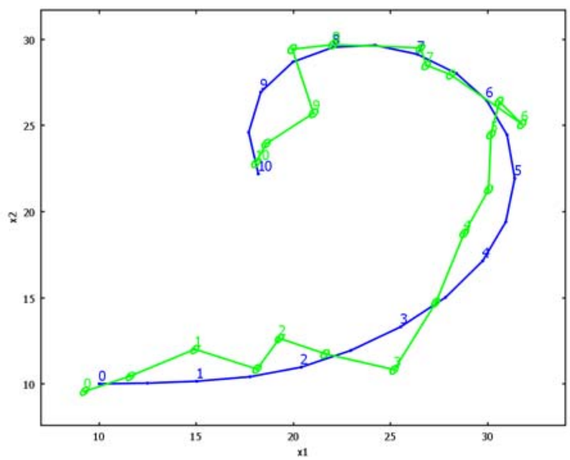
- *Medium SNR*. The SIR PF usually works alright for medium signal to noise ratios (SNR). That is, the state noise and measurement noise are comparable in some diffuse measure.
- *Low SNR*. When the state noise is very small, the total state space is not explored satisfactorily by the particles, and some extra excitation needs to be injected to spread out the particles. Dithering (or jittering or roughening) is one way to robustify the PF in this case, and the trick is to increase the state noise in the PF model.
- *High SNR*. Using the dynamic state model as proposal density is a good idea generally, but it should be remembered that it is theoretically unsound when the signal to noise ratio is very high. What happens when the measurement noise is very small is that most or even all particles obtained after the time prediction step get zero weight from the likelihood function. In such cases, try to increase the measurement noise in the PF model.

As another user guideline, try out the PF on a subset of the complete measurement record. Start with a small number of particles (100 is default). Increase an order of magnitude and compare the results. One of the examples below illustrate how the result eventually will be consistent. Then, run the PF on the whole data set, after having extrapolated the computation time from the smaller subset. Generally, the PF is linear in both the number of particles and number of samples, which facilitates estimation of computation time.

Example

Apply the PF to a simulated target tracking scenario, and plot the estimated target position:

```
m=exnl('ctpv2d'); % Coordinated turn model
z=simulate(m,10); % Ten seconds trajectory
mpf=m;            % Estimation model
mpf.pv=5*m.pv;   % Dithering
zpf=pf(mpf,z,'Np',1000);    % PF state estimation
xplot2(z,zpf,'conf',90);    % Position estimate
```



See Also nl.ekf, nl.nltf

Purpose Simulate a NL system using DASPK.

Syntax

```
z=simulate(m,z,Property1,Value1,...)
z=simulate(m,T,Property1,Value1,...)
```

Description For a discrete-time systems, the system recursions are evaluated in a for-loop for `t=t0:tfinal`.

For continuous-time systems, the input SIG object `u` defines the simulation time. If the NL system contains no input, use `T=[t0 tfinal]` instead of `u`. `z` is the simulated SIG object, where `z.x` is the solution to

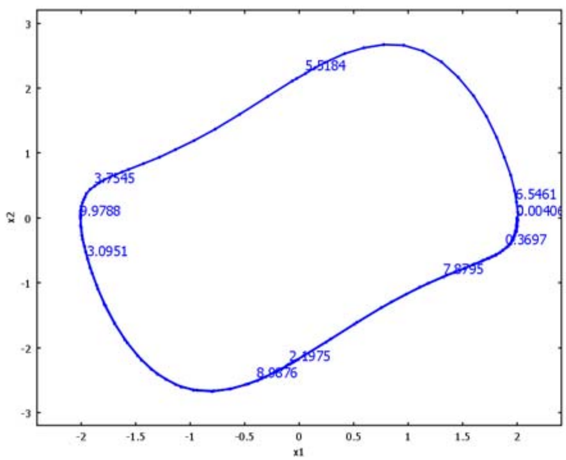
```
x' = m.f(t, x, u.y; m.th)
y = m.h(t, x, u.y; m.th)
x(0) = m.x0
```

The initial values `x0` and nominal parameters `th` are the ones defined in the NL object. The time instants in `z.t` are the ones generated by the ODE solver when `T=[t0 tfinal]`, otherwise the function computes the output `z.y` at the time instants specified in `T` or `u.t`.

PROPERTY	VALUE	DESCRIPTION
MC	{30}	Number of MC simulations when th uncertain

Example Simulate the van der Pol system, and plot the state trajectory:

```
m=exnl('vdp')
NL object: Van der Pol system
 dx/dt = [x(2,:);(1-x(1,:).^2).*x(2,:)-x(1,:)]
     y = x
   x0' = [2         0]

z=simulate(m,10)
SIG object with continuous time stochastic state space data (no
input)
  Sizes:       N = 164,  ny = 2, nx = 2
  MC is set to: 30
  #MC samples:  0
xplot2(z)
```



See Also `nl`, `ss.simulate`

Purpose

Numerical solver for the nonlinear least-squares (NLS) problem.

Syntax

```
[mhat,res] = nls(m,z,Property1,Value1,...)
```

Description

The nonlinear least-squares (NLS) algorithm implements various versions of the Gauss-Newton method for parameter estimation. A general problem formulation is to estimate the initial state and parameter vector in the model

$$y = H(x_0, \theta, u, e, v)$$

based on observations of y and u. Special cases include pure optimization

$$0 = H(\theta, e)$$

and data fitting

$$y = H(\theta, e)$$

The least-squares framework is appropriate whenever H is a vector. These problems can all be recast to the general nonlinear model object NL in either discrete time

$$\begin{aligned} x_{k+1} &= f(k, x_k, u_k; \theta) + v_k, \\ y_k &= h(k, x_k, u_k, e_k; \theta). \end{aligned}$$

or continuous time

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), u(t); \theta) + v(t), \\ y(t_k) &= h(t_k, x(t_k), u(t_k); \theta) + e(t_k), \\ x(0) &= x_0. \end{aligned}$$

There are basically three uses of `nls`:

1. `[m,res]=nls(h)` for pure optimization, where `h` is an inline function with `th` as a parameter.
2. `[m,res]=nls(h,y)` for data fitting, where `h` is either an inline function or a structure where one field is called `h` and contains an inline function with `th` as a parameter. The advantage with the latter usage is that more information about the problem can be provided in other fields. In all cases, `y` is a SIG object.
3. `[m,res]=nls(m,y)` for model calibration. Here `m` is an NL object, and all of its specified parameters and initial states are estimated by default using the data in the SIG object `y`. This is also known as gray-box identification, as opposed to black-box identification where standard model structures as ARX have to be used.

In gray-box identification, the model might be partially known, and the physical parameters are identified directly. The term model calibration is used here to stress that quite a good initial value of the parameters is generally needed for the algorithms to converge.

In general, the NLS parameter estimation problem can always be specified as an NL object `m=nl(f,h,nn)`, and the parameters are estimated with the NL method `mhat=estimate(m,z)`, or equivalently, `mhat=nls(m,z)`.

The following NLS algorithm as implemented in `nls`:

1. Initialize the parameter vector $\eta = (\theta, x_0)$ using the values of `x0` and `th` in the NL object.
2. Iterate in i:

$$\eta^{i+1} = \eta^i - \mu^j (J(\eta^i) J^T(\eta^i))^{-1} J(\eta^i)\varepsilon(\eta^i)$$

3. In each iteration, check if the cost function has decreased. Otherwise, half the step size μ until either the cost function decreases or `maxhalf` iterations in the line search is reached.
4. Continue until j = `maxiter` or the relative change in cost function is smaller than `ctol`, or the maximum element in the gradient J is smaller than `gtol`.

There is some support for entering a symbolic gradient J to the NL object for pure estimation problems, where $J = dh/d\theta$. Otherwise, a numeric gradient is computed according to

$$\frac{\partial}{\partial \eta_i}\hat{y}(t_k, \eta) \approx \frac{\hat{y}(t_k, \eta + he_i) - \hat{y}(t_k, \eta - he_i)}{2h},$$

which is evaluated for each unit vector.

The internal parameters in the algorithm are controlled by property-value pairs for both usages `mhat=estimate(m,z,Property1,Value1,...)` and `mhat=nls(m,z,Property1,Value1,...)`. The property-value pairs are listed in the table below.

TABLE 2-17: PROPERTY/VALUE PAIRS FOR THE NLS STRUCTURE

PROPERTY	VALUE {DEFAULT}	DESCRIPTION
thmask	`{ones(1,nth)}`	Binary search mask for parameter vector
x0mask	`{ones(1,nx)}`	Binary search mask for initial state vector

TABLE 2-17: PROPERTY/VALUE PAIRS FOR THE NLS STRUCTURE

PROPERTY	VALUE {DEFAULT}	DESCRIPTION
x0	Cell array	In case z is a cell with multiple data sets, different known initial conditions can be set. x0{i} is the initial state for z{i}
alg		Optimization algorithm
	`{'gn'}`	Gauss-Newton
	`'rgn'`	Robust Gauss-Newton, where the Hessian *H=JJ'* is robustified by adding a small identity matrix.
	`'lm'`	Levenberg-Marquardt, where the line search is replaced by a region search.
	`'sd'`	steepest-descent, where the Hessian is replaced with the identity matrix.
disp	`{0}|1`	Display status of the iterations
maxiter	`{50}`	Maximum number of iterations in search direction
maxhalf	`{50}`	Maximum number of iterations in the line search.
gtol	`{1e-4}`	Tolerance for the gradient.
ctol	`{1e-4}`	Minimum relative decrease in the cost function before the search is terminated.
svtol	`{1e-4}`	Lower bound for the singular values of the Jacobian in robust Gauss-Newton
numgrad	`{0}|1`	Force a numerical computation of the gradient even if a gradient m.J is specified

The direct call to NLS enables a second output structure `[mhat,res]=nls(m,z)`, which gives access to internal variables, as summarized in the following table.

TABLE 2-18: INTERNAL FIELDS IN THE NLS STRUCTURE

FIELD NAME	DESCRIPTION
res.TH	Parameter values at each iteration
res.V	Value of the cost function at each iteration
res.dV	The gradient at each iteration
res.m	The obtained model at each iteration as a cell array
res.sl	The step sizes at each iteration
res.sol	The obtained solution (thhat)
res.term	Text string with cause of termination
res.P	Covariance (estimated) for the parameters
res.Rhat	Covariance (estimated) for the measurements

Example

Pure optimization of vector-valued function:

```
m.h=inline('[th(1)-1; 2*th(2)+2*th(2)-4]','th');
m.J=inline('[1 0;0 2]','th');
tic, res=nls(m); toc    % Gradient J specified, symbolic
derivative

Elapsed time:  0.171 s

thstar=res.th
thstar =
  1
  1
tic, res=nls(m.h); toc  % No gradient J specified, numeric
derivative

Elapsed time:  0.000 s

thstar=res.th
thstar =
  1
  1
```

Providing the symbolic gradient speeds up computation time.

Curve fitting:

```
m.h=inline('th(1)*(1-exp(-th(2)*t))','t','th');  % Curve model
m.th=[2;0.5];                % True parameters
z.t=(0:0.2:3)';              % Time vector
y=m.h(z.t,m.th);             % True curve
z.y=y+0.1*randn(size(z.t)); % Measurements of curve
m.th=m.th+0.3*randn(2,1);   % Perturbed initial values
yinit=m.h(z.t,m.th);         % Inital curve
[mhat,res]=nls(m,z);         % Calibrated curve
mhat.th
ans =
     2.1455
     0.4607
yhat=mhat.h(z.t,mhat.th);   % Estimated curve
plot(z.t,y,'b',z.t,z.y,'g.-',z.t,yinit,'r',z.t,yhat,'k')
legend({'True curve','Measurements','Initial curve','Estimated
curve'})
```

See Also

`nl`, `nl.estimate`

Purpose Generic probability density function (PDF) class.

Syntax `p=pdfclass`

Description PDFCLASS is the parent of all other PDFs. The constructor for this object is only used for listing its children using `list(pdfclass)`. The generic methods are listed in the following table:

METHOD	DESCRIPTION	
ARRAYREAD	Pick out parts of a stochastic vector	`Xi=X(i)`
LIST	Lists of classes that inherits PDFCLASS	`l=list(pdfclass)`
VERTCAT	Create a multivariate stochastic vector from stochastic variables/vectors	`X=[X1;X2]`
RAND	Generate random numbers in a vector of length N, or a (nX,N) matrix when X is a stochastic vector	`x=rand(X,N)`
ERF	Evaluates the error function I(x)=P(X<x) numerically	`I=erf(X,x)`
ERFINV	Evaluate the inverse error function I(x)=P(X<x) numerically	`x=erfinv(X,I)`
CDF	The cumulative density function	`P=cdf(X,x)`
PDF	The probability density function	`p=pdf(X,x)`
FUN	All conceivable functions and operators can be applied to a stochastic variable	ex: `Z=sin(X)`
EVALFUN	Implements functions of one variable, which can be used for user-defined functions and M-files	
EVALFUN2	Implements functions of two variables, which can be used for user-defined functions and M-files	
PLOT	Illustrate the PDF of X	
CDFPLOT	Illustrate the cumulative density function (CDF) of X	
ERFPLOT	Illustrate the error function ERF of X	
PLOT2	Illustrate the PDF of X in a two dimensional plot	`plot2(X1,[i,j])`
SURF	Illustrate the PDF of X in a two dimensional plot	`surf(X1,[i,j])`

Children of PDFCLASS include the following specific distributions:

FUNCTION	PURPOSE
`empdist(x)`	The empirical distribution defined by a set of samples
`ndist(mu,P)`	The normal, or Gaussian, distribution
`udist(a,b)`	The uniform distribution
`expdist(mu)`	The exponential distribution
`tdist(n)`	Student's t distribution
`gammadist(a,b)`	The Gamma distribution
`betadist(a,b)`	The Beta distribution
`fdist(d1,d2)`	The F-distribution

Type `list(pdfclass)` to list all children of PDFCLASS in the search path, which also includes user-defined distributions (see "Defining Your Own Distributions" on page 249).

Each distribution is characterized by the following methods:

`LENGTH`	The length of the stochastic vector X
`DESC`	Return a description of the distribution defined by the class
`E`	The expectation operator
`MEAN`	The expectation operator
`STD`	The standard deviation operator
`VAR`	The variance operator
`SKEW`	The skewness operator
`KURT`	The kurtosis operator

Furthermore, each PDF can have methods for specific symbolic operations, such as addition of Gaussian variables and multiplication of a matrix and a Gaussian vector.

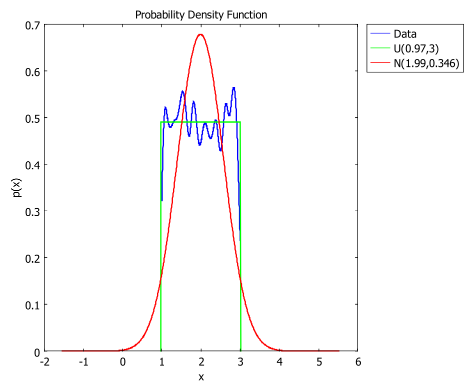
`ERF`	Compute the error function	`x=erf(X,alpha)`
`ERFINV`	Compute the inverse error function	`alpha=erf(X,x)`
`ESTIMATE`	Estimate a parametric density function from an empirical distribution	`Xhat=estimate(X, Xemp)`
`ESTIMATE`	Estimate a parametric density function from a list of PDF objects	`dist=estimate(Xemp,Xlist)`

The error function is defined as $I(x) = P(X<x)$. The `erf` and `erfinv` methods generalize the built-in functions with the same names to non-Gaussian distributions.

Example

Define distributions, estimate distributions and visualize them:

```
XO=udist(1,3);
x=rand(XO,1000);
Xemp=empdist(x);
Xhat1=estimate(udist,Xemp);
Xhat2=estimate(ndist,Xemp);
plot(Xemp,Xhat1,Xhat2)
```



Compute moments and error functions:

```
[E(XO), E(Xemp), E(Xhat1),  E(Xhat2)]
ans =
          2     1.9887     1.9887     1.9887
[std(XO), std(Xemp), std(Xhat1),  std(Xhat2)]
ans =
     0.5774     0.5885     0.5885     0.5885
[erf(XO,2.9), erf(Xemp,2.9), erf(Xhat1,2.9), erf(Xhat1,2.9)]
ans =
     0.9510     0.9544     0.9479     0.9479
[erfinv(XO,0.9), erfinv(Xemp,0.9), erfinv(Xhat1,0.9),
erfinv(Xhat1,0.9)]
ans =
     2.7982     2.8061     2.8023     2.8023
```

See Also

`empdist`, `pdfclass.estimate`

Purpose Estimate free parameters in a probability density function from a set of samples.

Syntax

`X=estimate(Xdist,Xemp)`	Adapt the distribution Xdist to Xemp
`X=estimate(Xemp,pdflist)`	Find the best distribution fit in pdflist

Description The `estimate` method of a specific distribution computes the free parameters that give an exact fit of the first moments. This is referred to as a moment-based estimator.

For example, the uniform distribution contains two free parameters (the interval bounds). These are computed to give the same mean and variance as the sample average and variance. This uniform distribution is completely different than what the maximum likelihood estimate provides, which is a larger interval that contains all samples.

In the second case of usage, a structure of PDF objects is provided as the second argument, and the `estimate` method of `empdist` is used. The free parameters in each distribution are estimated, and the one that gives the best least-squares fit of the cumulative distribution function is chosen as output.

$$\hat{F}(x;\theta) = \arg\min \sum_{i=1}^{N} |F(x_i;\hat{\theta} - \hat{F}_{\mathrm{emp}}(x_i)|^2$$

$$= \arg\min \sum_{i=1}^{N} |F\left(x_i;\hat{\theta} - \frac{i}{N}\right)|^2$$

Example See the example in `pdfclass`.

See Also `pdfclass`, `empdist`

Purpose

Optimal fusion of two independent unbiased estimates of the same variable.

Syntax

`X=fusion(X1,X2)`

Description

Assume there are two estimates (or measurements) of a stochastic variable x:

$$E(\hat{x}_1) = E(\hat{x}_2) = x,$$
$$\mathrm{cov}(\hat{x}_1) = P_1,$$
$$\mathrm{cov}(\hat{x}_2) = P_2.$$

If these are independent, the fused estimate is given by

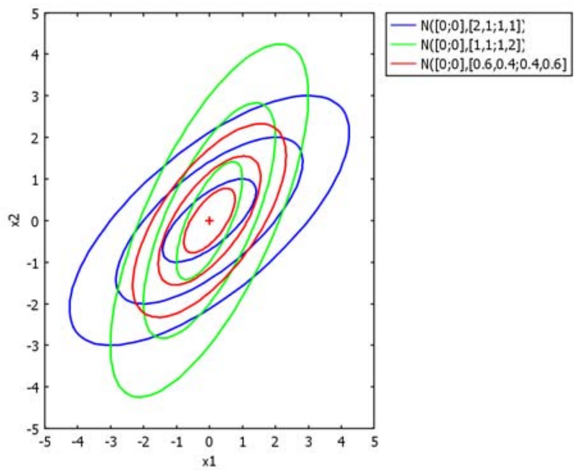
$$P = (P_1^{-1} + P_2^{-1})^{-1},$$
$$\hat{x} = P(P_1^{-1}\hat{x}_1 + P_2^{-1}\hat{x}_2).$$

The output `X` is packed as a Gaussian `ndist` object.

Example

Generate two different Gaussian distributions with the same mean, and compute the optimal combination of this information:

```
X1=ndist([0;0],[2 1;1 1]);
X2=ndist([0;0],[1 1;1 2]);
X=fusion(X1,X2)
N([0;0],[0.6,0.4;0.4,0.6])
plot2(X1,X2,X)
```



See Also `pdfclass.safefusion`, `nl.estimate`

Purpose Conservative fusion of two possibly dependent unbiased estimates of the same variable using covariance intersection.

Syntax `X=safefusion(X1,X2)`

Description Assume there are two estimates (or measurements) of a stochastic variable x:

$$E(\hat{x}_1) = E(\hat{x}_2) = x,$$
$$\mathrm{cov}(\hat{x}_1) = P_1,$$
$$\mathrm{cov}(\hat{x}_2) = P_2.$$

If these are dependent, the fused estimate is given by

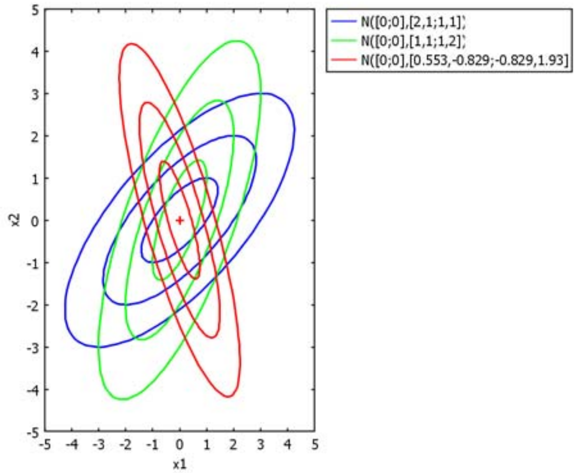
$$K = P_{12}P_2^{-1},$$
$$P = P_1 - KP_2K^T,$$
$$\hat{x} = \hat{x}_1 + K(\hat{x}_2 - \hat{x}_1).$$

`safefusion` computes the fused estimate under a worst case assumption using covariance intersection techniques.

The output `X` is packed as a Gaussian `ndist` object.

Example Generate two different Gaussian distributions with the same mean, and compute the optimal combination of this information

```
X1=ndist([0;0],[2 1;1 1]);
X2=ndist([0;0],[1 1;1 2]);
X=safefusion(X1,X2)
N([0;0],[0.553,-0.829;-0.829,1.93])
plot2(X1,X2,X)
```



See Also `pdfclass.fusion`, `nl.estimate`

Purpose

Analysis and training tool for amplitude distribution.

Syntax

`pdftool(y)`	Analysis mode
`pdftool`	Training mode

Description

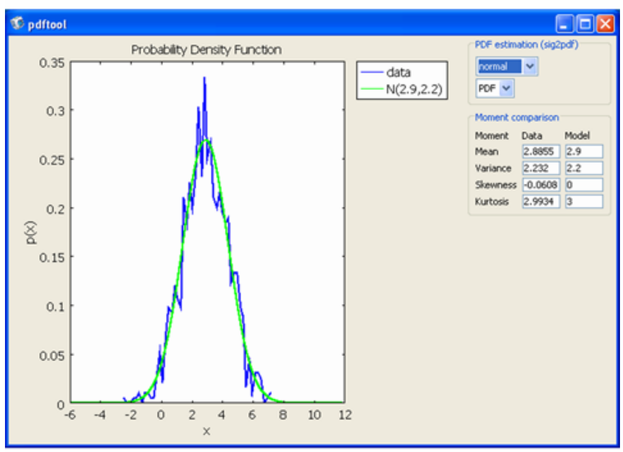
In analysis mode, the `pdftool` calls the functions `sig2pdf` and `pdfplot` interactively. The data histogram (as a plot rather than bar graph) is always shown. You choose the distribution, and the tool estimates its parameters to get the best fit of the first moments (this is the definition of the moment-based estimator).

In training mode, the function `pdf2sig` is used to simulate a number of samples from a distribution of your choice. Otherwise, it works as in the analysis mode. One difference is that there is a truth here, so the true distribution and moments are displayed.

Example

Given a data vector `y`, fit a Gaussian distribution to its amplitude histogram:

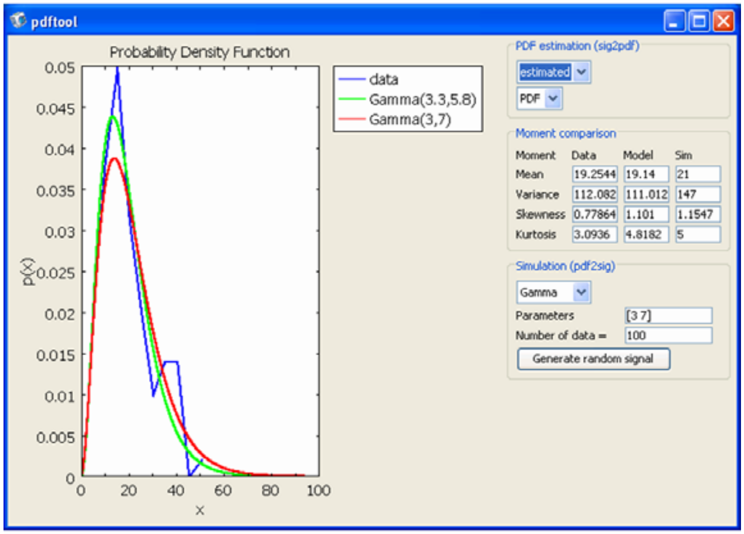
```
pdftool(y)
```



The fit of the histogram and parametric distribution, and the different moments, can be used to validate the chosen distribution.

To start in training mode, omit any arguments.

```
pdftool
```


pdftool
Probability Density Function
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
p(x)
0
20
40
60
80
100
x
data
Gamma(3.3,5.8)
Gamma(3,7)
PDF estimation (sig2pdf)
estimated
PDF
Moment comparison
Moment Data Model Sim
Mean 19.2544 19.14 21
Variance 112.082 111.012 147
Skewness 0.77864 1.101 1.1547
Kurtosis 3.0936 4.8182 5
Simulation (pdf2sig)
Gamma
Parameters [3 7]
Number of data = 100
Generate random signal

Purpose Construct a RARX model object.

Syntax The RARX model object constructor supports the following calling syntaxes:

m=rarx(nn)	Empty structure, for estimate and rand
m=rarx(nn,th)	Parameter vector (certain model)
m=rarx(nn,th,P)	Parameter vector and covariance
m=rarx(nn,th,P,lambda)	Parameter vector, covariance and measurement noise
m=rarx(nn,th,thMC)	MC samples instead of covariance

Description RARX is a time-varying ARX model, where the first dimension of `th` and `P` is time. One main difference to the ARX object is that `b` and `a` polynomials are not explicitely saved, and that the noise variance `lambda` is a time-varying parameter.

Protected fields: `th`, `P`, `nn`, and `lambda`

Public fields: `MC`, `pe`, `fs`, `name`, `desc`, `marker`, `method`, `xlabel`, `ulabel`, `ylabel`, `tlabel`, and `markerlabel`

The arguments are defined below:

ARGUMENT	DIMENSION	DESCRIPTION
nn	Row vector	See the following table
th	(N,na+nb)	Time-varying parameter vector
P	(N,na+nb,na+nb)	Time-varying covariance matrix for parameter uncertainty
thMC	(MC,N,na+nb)	Monte Carlo representation of parameter uncertainty as alternative to P
lambda	(N,1)	Time-varying scaling of noise variance

The model order is defined in the following table:

nn=[na nb nk]	The structure of the RARX model
nn=na	The order of a RAR model
nn=[]	Implies a volatility model y(t)=e(t), var(e(t))=lambda(t)
nn=[na nb nk nu ny]	Implies a MIMO RARX model
nn=[na nb nk nu ny np]	Also models a polynomial trend or order np. np=0 corresponds to constant, np=1 to a linear trend and so on.

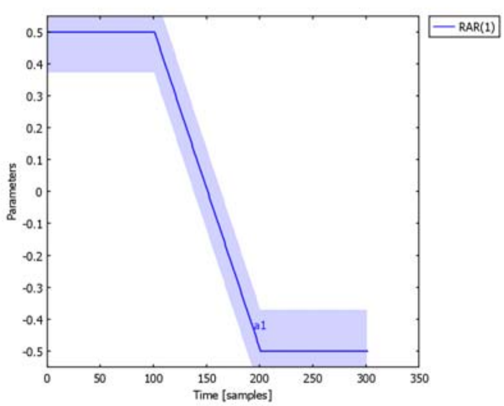
Overloaded methods:

METHOD	DESCRIPTION
ARRAYREAD	Pick out time intervals or subsystems from MIMO systems
	sys=arrayread(t,i,j)
	t is the time index/indices
	i is the row index/indices, corresponding to the outputs
	j is the column index/indices, corresponding to the inputs
DISPLAY	Script window display
SYMBOLIC	Return a symbolic string expression for the ARX structure
LENGTH	Return the number of time points N
SIZE	Return the sizes nn=[na,nb,nk,nu,ny]
	[na,nb,nk,nu,ny]=size(s) Complete structure nn
	[ny,nu]=size(s) MIMO size

Example

Define a RAR(1) model with constant uncertainty:

```
MC=30;
nn=1;  % RAR(1) model
th=[0.5*ones(100,1); (0.5:-0.01:-0.5)' ; -0.5*ones(100,1)];
P=0.01*ones(length(th),1);
mt=rarx(nn,th,P);
plot(mt)
```



See Also

rarx, rarx.surf, exrarx, rarx.estimate, rarx.expand, rarx.rarx2tfd, rarx.simulate, rarx.surf, rarx.contour

Purpose Illustrate a RARX object graphically in the frequency domain using `contour`.

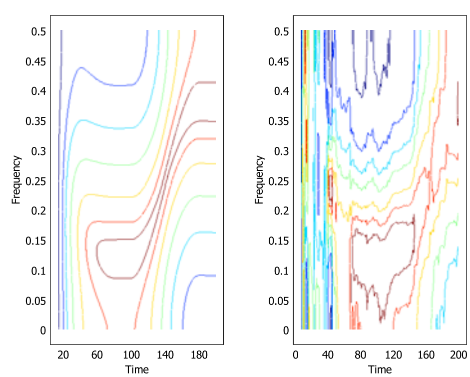
Syntax `contour(mt1,mt2,...,Property1,Value1,...)`

Description Illustrates the transfer function part in one RARX. Note that you cannot view multiple models and confidence intervals for this function.

PROPERTY	VALUE/ {DEFAULT}	DESCRIPTION
`histeq`	`{'on'} | 'off'`	Histogram equalization of tfd value (see histeq)
`decfactor`	`{N/200}`	Decimation factor (to reduce computations)
`t0`	`{1}`	First time value to remove transients
`axis`	`{gca}`	Axis handle where plot is added
`col`	`{'bgrmyk'}`	Colors in order of appearance
`f`	`{128}`	Frequency vector for tfd plot. f=Nf gives Nf linearly spaced values.

Example Load an example RARX model with a time-varying AR(2) structure, simulate it, and estimate an uncertain RARX object of the same AR(2) structure with an adaptive filter (RLS with forgetting factor 0.95). Then compare the true and the estimated model in various ways.

```
mt=exrarx('rar2',200);
z=simulate(mt);
mthat=estimate(rarx(2),z,'adg',0.95);
subplot(1,2,1), contour(mt)
subplot(1,2,2), contour(mthat)
```



See Also `rarx`, `rarx.surf`, `exrarx`, `rarx.estimate`, `rarx.expand`, `rarx.rarx2tfd`, `rarx.simulate`, `rarx.surf`

Purpose Estimate a time-varying ARX model by adaptive filtering.

Syntax

```
m=estimate(ms,z,Property1,Value1,...)
```

Description The RARX method `estimate` implements a number of standard recursive algorithms for adaptive filtering such as LMS, RLS, and Kalman filters.

Required input parameters:

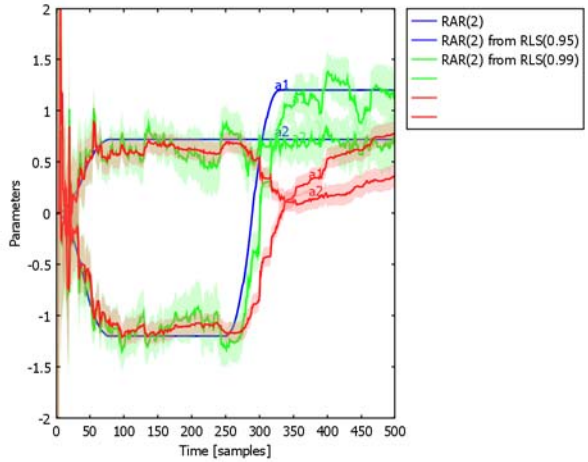
ARGUMENT	DESCRIPTION
`z`	Output (and input) data
`ms`	Model structure, for instance, rarx([2 2 1]) for recursive ARX(2,2,1)

Optional parameters:

PROPERTY	VALUE/ {DEFAULT}	DESCRIPTION
`adg`	`{0.95}`	Adaption gain, forgetting factor, step size or Q. For state space models adg is Q.
`adm`	`{}`	Adaptation method: LS, RLS, WLS, LMS, NLMS or KF. An intelligent default value between RLS, WLS and LMS is made based on the value of adg.
`fflam`	`{1}`	Forgetting factor for estimating noise variance lambda
`decfactor`	`{1}`	Decimation factor to save time
`P0`	`{1e6}`	Initial covariance (or scaling of I) for RLS and KF
`th0`	`{0}`	Initial value for theta

Example Using an example RARX model:

```
m=exrarx('rar2',500);
z=simulate(m);
mhat1=estimate(rarx(2),z,'adg',0.95,'adm','rls');
mhat2=estimate(rarx(2),z,'adg',0.99,'adm','rls');
plot(m,mhat1,mhat2,'Ylim',[-2 2])
```



See Also `rarx`, `rarx.surf`, `exrarx`, `rarx.expand`, `rarx.rarx2tfd`, `rarx.simulate`, `rarx.surf`, `rarx.contour`

Purpose Create RARX objects from a set of ARX models.

Syntax

```
m=expand(ms,jumps,th,P,lambda,Property1,Value1,...)
```

Description A time-varying model defined by one parameter vector for each segment is expanded to one parameter vector for each time instant.

Required input arguments:

ARGUMENT	DESCRIPTION
`ms`	Model structure, typically rarx([na nb nk])
`jumps`	Vector of length njumps with end points for each segment. Convention: jumps(end)=N, that is, implicit definition of total time span
`th`	Each row in the (njumps,nth) matrix contains the parameter vector for each segment j.
`P`	P(j,:,:) contains the covariance matrix for th(j,:) in segment j
`lambda`	lambda(j) contains the measurement noise in segment j

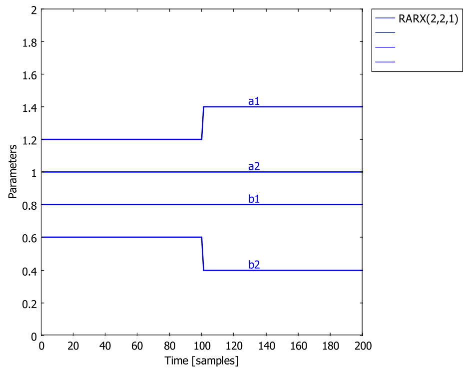
Both `P` and `lambda` can be empty matrices.

Optional parameters for smoothing the transitions:

PROPERTY	VALUE
`ip`	Interpolation method for the changes in TH and Lambda
	'off' No interpolation (default)
	'ic' Integrated change from jumptime to jumptime+L
`ipL`	Interpolation window size (default 10% of average segment length)
`ipn`	Interpolation order. n=1 gives linear interpolation. (Default 1).

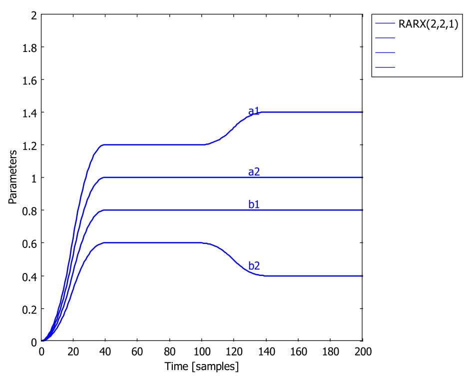
Example A convenient way to create a RARX model for simulation is to concatenate the parameters from ARX models over different segments:

```
TH=[1.2 1 0.8 0.6; 1.4 1 0.8 0.4];
m=expand(rarx([2 2 1]),[100 200],TH)
Certain RARX(2,2,1)
plot(m,'Ylim',[0 2])
```



You obtain softer transitions by interpolation:

```
TH=[1.2 1 0.8 0.6; 1.4 1 0.8 0.4];
m=expand(rarx([2 2 1]),[100
200],TH,[],1,'ip','on','ipn',2,'ipL',20)
Certain RARX(2,2,1)
plot(m,'Ylim',[0 2])
```



See Also `rarx`, `rarx.surf`, `exrarx`, `rarx.estimate`, `rarx.rarx2tfd`, `rarx.simulate`, `rarx.surf`, `rarx.contour`

Purpose Illustrate the parameters in a RARX object in a plot.

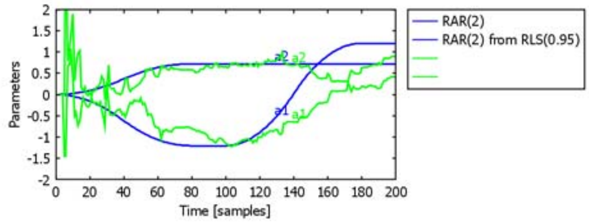
Syntax `plot(mt1,mt2,...,Property1,Value1,...)`

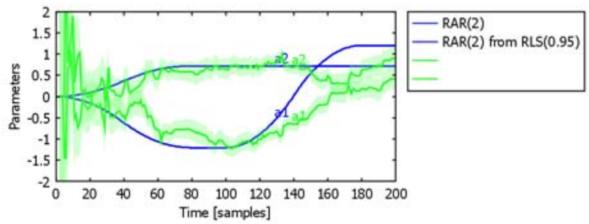
Description Illustrates parameters in one or more RARX objects at the same time. You can add confidence intervals of the estimated parameters.

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
decfactor	{N/200}	Decimation factor (to reduce computations)
t0	{1}	First time value to remove transients
conf	[0,100] {0}	Confidence level for par view (0 means no levels)
conftype	1|{2}	1 for upper and lower bound lines, 2 for confidence area
axis	{gca}	Axis handle where plot is added
col	{'bgrmyk'}	Colors in order of appearance
f	{128}	Frequency vector for tfd plot. f=Nf gives Nf linearly spaced values.

Example Load an example RARX model with a time-varying AR(2) structure, simulate it, and estimate an uncertain RARX object of the same AR(2) structure with an adaptive filter (RLS with forgetting factor 0.95). Then compare the true and the estimated model in a parameter plot without and with confidence intervals, respectively.

```
mt=exrarx('rar2',200);
z=simulate(mt);
mthat=estimate(rarx(2),z,'adg',0.95);
subplot(2,1,1), plot(mt,mthat,'view','par','conf',0,'Ylim',[-2
2])
subplot(2,1,2), plot(mt,mthat,'view','par','conf',90,'Ylim',[-2
2])
```





See Also `rarx`

Purpose

Convert a time-varying ARX model to a Time-Frequency Description (TFD).

Syntax

`Yt=rarx2tfd(mt)`	Explicit call
`Yt=tfd(mt)`	Implicit call

Description

Converts an LTV object to its corresponding time-varying frequency description (TFD). This is an extension of `arx2freq` for time-invariant ARX models. RARX contains one model for each time instant, which is converted into the frequency domain.

Without output argument, this function calls `ltv.plot`.

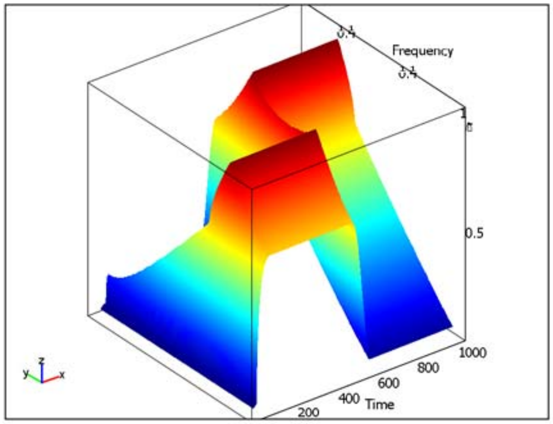
Optional input parameters:

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`decfactor`	`{N/200}`	Decimation factor (to reduce computations)
`t0`	`{1}`	First time value to remove transients
`f`	`{128}`	Frequency vector for tfdplot. f=Nf gives Nf linearly spaced values.

Example

Load a time-varying ARX object of AR(2) structure and compute its theoretical time-frequency description (TFD):

```
m=exrarx('rar2',1000);
Yt=tfd(m);
surf(Yt)
```



See Also `rarx`, `rarx.surf`, `exrarx`, `rarx.estimate`, `rarx.expand`, `rarx.rarx2tfd`, `rarx.simulate`, `rarx.surf`, `rarx.contour`

Purpose

Simulate a RARX model.

Syntax

`z=simulate(mt,Property1,Value1,...)` RAR model without input
`z=simulate(mt,u,Property1,Value1,...)` RARX model with input

Description

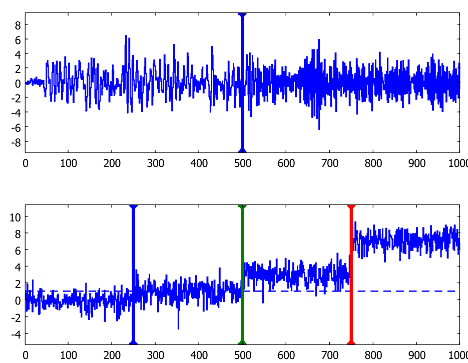
`mt` is the RARX object to simulate. It contains a time-varying parameter vector, model structure, and model order information. You can obtain the RARX object from, for instance, `exrarx` or `estimate(rarx(nn),z)`.

The output is a SIG object.

Example

Load two examples, one without and one with input signal, simulate them, and plot the result:

```
mt1=exrarx('rar2',1000);
z1=simulate(mt1);
mt2=exrarx('mean1',1000);
u=getsignal('ones',1000);
z2=simulate(mt2,u);
subplot(2,1,1)
plot(z1)
subplot(2,1,2)
plot(z2)
```



See Also

`rarx`, `rarx.surf`, `exrarx`, `rarx.estimate`, `rarx.rarx2tfd`, `rarx.surf`, `rarx.contour`

Purpose Illustrate a RARX object graphically in the frequency domain using `surf`.

Syntax

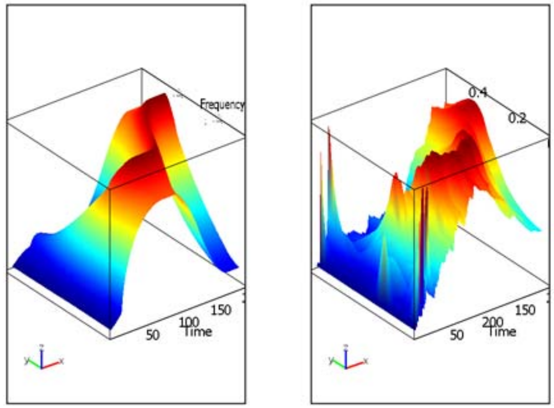
```
surf(mt1,mt2,...,Property1,Value1,...
```

Description Illustrates the transfer function part in one RARX. Note that you cannot view multiple models and confidence intervals for this function.

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`histeq`	`{'on'} | 'off'`	Histogram equalization of tfd value (see histeq)
`decfactor`	`{N/200}`	Decimation factor (to reduce computations)
`t0`	`{1}`	First time value to remove transients
`axis`	`{gca}`	Axis handle where plot is added
`col`	`{'bgrmyk'}`	Colors in order of appearance
`f`	`{128}`	Frequency vector for tfd plot. f=Nf gives Nf linearly spaced values.

Example Load an example RARX model with a time-varying AR(2) structure, simulate it, and estimate an uncertain RARX object of the same AR(2) structure with an adaptive filter (RLS with forgetting factor 0.95). Then compare the true and the estimated model in various ways:

```
mt=exrarx('rar2',200);
z=simulate(mt);
mthat=estimate(rarx(2),z,'adg',0.95);
subplot(1,2,1), surf(mt)
subplot(1,2,2), surf(mthat)
```



See Also `rarx`, `rarx.surf`, `exrarx`, `rarx.estimate`, `rarx.expand`, `rarx.rarx2tfd`, `rarx.simulate`, `rarx.contour`

Purpose

The signal object SIG.

Syntax

The signal object supports the following input data:

`sig(y,fs)`	Uniformly sampled time series y[k]=y(k/fs)
`sig(y,t)`	Nonuniformly sampled time series y(t)
`sig(y,t,u)`	Uniformly sampled I/O system
`sig(y,t,u,x)`	Uniformly sampled state-space system
`sig(y,fs,u,x,yMC,xMC)`	MC data arranged in an array

Description

The constructor of the SIG object basically converts a vector signal to an object, where certain other information can be provided. The main advantages of using a signal object rather than just vectors are:

- Defining stochastic signals from PDFCLASS objects is highly simplified, using calls as `yn=y+ndist(0,1);`. Monte Carlo simulations are here generated as a background process.
- Standard operations as `+,-,.*,./` can be applied to the SIG object just as you would have done to a vector signal, where these operations are also applied to the Monte Carlo simulations.
- All plot functions accepts multiple signals that do not need to have the same time vector. The plot functions can visualize the Monte Carlo data as confidence bounds or scatter plots.
- The additional information that you put into the SIG object is used subsequently to get correct time axis in plots and frequency axis in Fourier transform plots. Further, you can obtain appropriate plot titles and legends automatically.

The basic use of the constructor is `y=sig(yvec,fs)` for discrete-time signals and `y=sig(yvec,tvec)` for continuous-time signals, respectively. Continuous-time signals are represented by nonuniform time points and the corresponding signal value, with the following two conventions:

- Steps and other discontinuities are represented by two (2) identical time stamps with different signal values. For instance, `t=[0 1 1 2]'; y=[0 0 1 1]', z=sig(y,t);` defines a unit step.
- Impulses are represented by three (3) identical time stamps where the middle signal value represents the area of the impulse. For instance, `t=[0 1 1 1 2]'; y=[0 0 10 0]', z=sig(y,t);` defines a unit impulse.

These conventions influence how the Signals & Systems Lab visualize continuous-time signals in plots, but also how it performs simulations.

The obtained SIG object can be seen as a structure with the following field names:

- `sig.y` is the signal itself.
- `sig.fs` is the sampling frequency in Hertz. Continuous-time signals have `fs=NaN` by convention.
- `sig.t` contains the sampling times (uniformly or nonuniformly sampled). If this is provided, it overrides the sampling frequency.
- `sig.u` is the input signal, if applicable.
- `sig.x` is the state vector for simulated data.
- `sig.name` is a one-line identifier that can be used for plot legends.
- `sig.desc` can contain a more detailed description of the signal.
- `sig.marker` contains optional user-specified markers indicating points of interest in the signal. For instance, the markers can indicate points where the signal dynamics changes or known faults in systems.
- `sig.yMC` and `sig.xMC` contain Monte Carlo simulations arranged as an array.
- `sig.ylabel`, `sig.xlabel`, `sig.ulabel`, `sig.tlabel`, and `sig.markerlabel` contain labels for plots.

The data fields `y`, `t`, `u`, `x`, `yMC`, and `xMC` are protected, and you cannot overwrite or change them. All other fields are open for both reading and writing.

METHOD	SYNTAX	DESCRIPTION
`arrayread`	`z=z(t,i,j)`	Pick out subsignals from SIG systems, where t is time, i output, and j input indices. z(t1:t2) picks out a time interval and is equivalent to z(t1:t2,:,:).
`horzcat`	`z=horzcat(z1,z2)` or `z=[z1 z2]`	Concatenate two SIG objects to larger output dimension. The time vectors must be equal.
`vertcat`	`z=vertcat(z1,z2)` or `z=[z1;z2]`	Concatenate two SIG objects in time. The number of inputs and outputs must be the same.
`append`	`z=append(z1,z2)`	Concatenate two SIG objects to MIMO signals

The following operators are overloaded:

OPERATOR	DESCRIPTION	EXAMPLE
plus	Adds a constant, a vector, another signal, or noise from a PDFCLASS object	y=sin(t)+1+expdist(1)
minus	Subtracts a constant, a vector, another signal, or noise from a PDFCLASS object	y=sin(t)-1-expdist(1)
times	Multiply a constant, a vector, another signal, or noise from a PDFCLASS object	y=udist(0.9,1.1)*sin(t)
rdivide	Divide a signal with a constant, a vector, another signal, or noise from a PDFCLASS object. divide and mrdivide are also mapped to rdivide for convenience.	y=sin(t)/2
mean/E	Returns the mean of the Monte Carlo data	y=E(Y)
std	Returns the standard deviation of the Monte Carlo data	sigma=std(Y)
var	Returns the variance of the Monte Carlo data	sigma2=var(Y)
rand	Return one random SIG object or a cell array of random SIG objects	y=rand(Y,10)
fix	Remove the Monte Carlo simulations from the object	y=fix(Y)

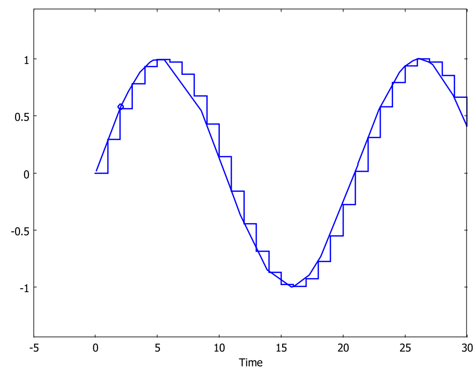
Example

The following example shows signal construction of a sinusoid with three types of data:

- Uniformly sampled data
- Nonuniformly sampled data
- Monte Carlo data:

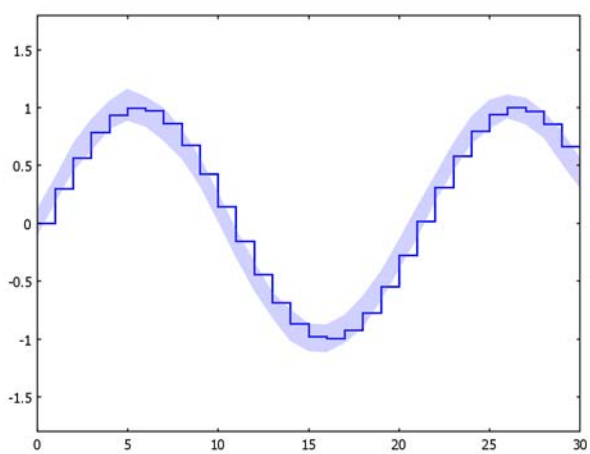
```
fs=1;
t1=(0:1:30)'*fs;
y1=sin(0.3*t1);
z1=sig(y1,fs)
SIG object with discrete time (fs = 1) time series
  Sizes:       N = 31,  ny = 1
  MC is set to: 30
  #MC samples:  0
t2=sort(30*rand(31,1));
y2=sin(0.3*t2);
z2=sig(y2,t2)
SIG object with continuous time time series
  Sizes:       N = 31,  ny = 1
```

```
  MC is set to: 30
  #MC samples:  0
plot(z1), hold on
stem(z2), hold off
```



To add Monte Carlo simulations, simply either create the yMC matrix yourself,

```
MC=100;
yMC=repmat(y1',MC,1)+0.1*randn(MC,length(y1));
z3=sig(y1,fs,[],[],yMC);
plot(z3,'conf',90)
```



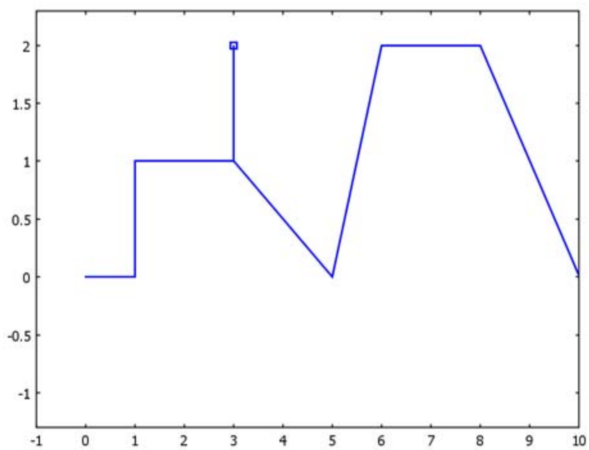
or add any distribution belonging to the PDFCLASS family,

```
z4=z2+0.1*ndist(0,1)
SIG object with continuous time time series
  Sizes:       N = 31,  ny = 1
  MC is set to: 30
  #MC samples:  30
```

The results should be identical.

To create a continuous-time signal with steps and impulses, use the convention that you repeat the time twice for steps and three times for impulses as the following example illustrates:

```
t=   [0 1 1 3 3 3 5 6 8 10]';
yvec=[0 0 1 1 2 1 0 2 2  0]';
y=sig(yvec,t);
plot(y)
```



See Also `sig.detrend`, `sig.interp`, `sig.resample`, `sig.sig2covf`

Purpose Remove trends in nonstationary data series.

Syntax `[yd,trend,lsfit]=detrend(y,order)`

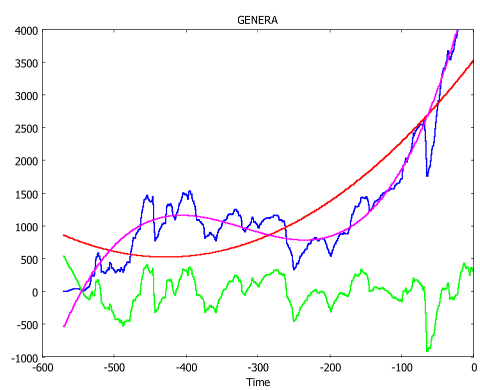
Description A polynomial model of a certain order is estimated by the least-squares method and subtracted from the data.

TABLE 2-19: INPUT ARGUMENTS FOR THE DETREND FUNCTION

ARGUMENT	DESCRIPTION
`y`	Input data as SIG object
`order`	Order of polynomial, 0 (default) for subtracting mean; order 1 for subtracting linear trend, and so on
`yd`	Detrended data as SIG object
`trend`	The estimated trend as SIG object
`lsfit`	Least-squares loss function

Example Load a real-data example and remove a quadratic and cubic trend, respectively. The least-squares fit reveals what the eye can see; the cubic trend fits data better.

```
load genera
[yd2,trend2,lsfit2]=detrend(y1,2);
[yd3,trend3,lsfit3]=detrend(y1,3);
[lsfit2,lsfit3]
ans =
   3.4e+005   6.0e+004
plot(y1,yd3,trend2,trend3,'Ylim',[-1000 4000])
```



See Also `sig` ,`sig.interp`, `sig.resample`, `sig.sig2covf`

Purpose

Interpolate $y_1(t_1)$ to $y_2(t_2)$.

Syntax

```
y2=interp(y1,t2,Property1,Value1,...)
```

Description

Interpolation is based on either a band-limited assumption, where perfect reconstruction and resampling can be done, a spline interpolation, or using an assumption of intersample behavior. This can be a zero-order hold for piecewise constant signals or a first-order hold for piecewise linear signals.

TABLE 2-20: INTERP PROPERTIES

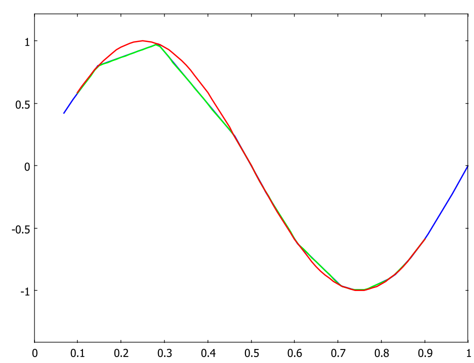
PROPERTY	VALUE	DESCRIPTION
`method`	`'BL' | {'hold'} | 'spline'`	Type of interpolation
`degree`	`{0} | 1`	Degree in hold function

Note that the output `y2` is by default a continuous-time signal. If `t2` is uniformly sampled, set the field `fs` after interpolation, or used the method `y2=sample(y1,fs)`.

Example

Compute random sampling times in $[0, 1]$, and get the sinusoidal values at these points. Resample the signal and compare the methods:

```
t1=sort(rand(20,1));  % Non-uniformly sampled data
y1=sig(sin(2*pi*t1),t1);
t2=0.1:0.01:0.9;
y2FOH=interp(y1,t2,'method','hold','degree',1);
y2spline=interp(y1,t2,'method','spline');
plot(y1,y2FOH,y2spline)
```



See Also

`sig`, `sig.detrend`, `sig.resample`, `sig.sig2covf`

Purpose Plot a signal.

Syntax

```
plot(z1,z2,...,Property1,Value1,...)
```

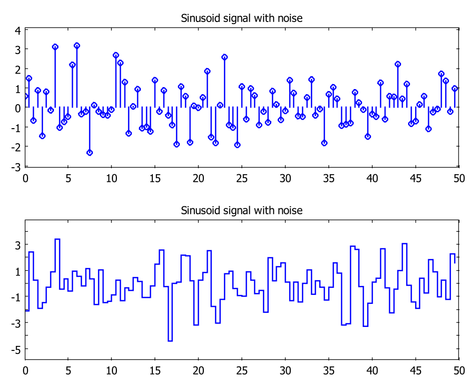
Description This function extends the usual `plot` functions with `staircase` and `stem` plot options. Further, if the input signal is a signal structure, supporting information such as sampling frequency, title, and variable names are displayed in the plot.

TABLE 2-21: SIGNAL PLOT FUNCTION PROPERTIES

PROPERTY	VALUE	DESCRIPTION
`interval`	`{1:N}`	Time interval for focus
`axis`	`{gca}`	Axis handle where plot is added
`conf`	`[{0},100]`	Confidence level from MC, 0 means no levels plotted
`scatter`	`'on'|{'off'}`	Scatter plot of MC data
`col`	`{'bgrmyk'}`	Colors in order of appearance in sig1,sig2,...
`type`	`'staircase' | {'interp'} | 'stem'`	Type of plot for sampled signals
`legend`	`{}`	Legend text

Example Load a sinusoid signal example, and make a stem and staircase plot, respectively:

```
s1=getsignal('sin1',100);
subplot(2,1,1),  stem(s1)
s2=getsignal('sin2',100);
subplot(2,1,2),  staircase(s2)
```



See Also `sig`

Purpose Resample uniformly sampled signal using a band-limitation assumption.

Syntax `y=resample(y,n,m)`

Description Resampling definition: $y[k] = y(kT)$ to $y[l] = y(l \cdot n/m \cdot T)$

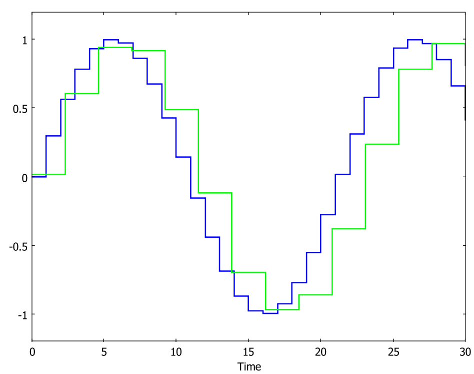
`resample(y,n,1)` decimates a factor n

`resample(y,1,m)` upsamples a factor m

Anti-alias filtering applied if $n/m > 1$. For decimation, when $m = 1$, the method `y=decimate(y,n)` applies.

Example Simulate a sinusoid and resample it at $7/3$ times lower sampling frequency:

```
t1=(0:1:30)';
y1=sig(sin(0.3*t1));
y2=resample(y1,7,3);
plot(y1,y2)
```



See Also `sig`, `sig.detrend`, `sig.interp`, `sig.sig2covf`

Purpose Estimate a covariance function from a SIG object.

Description See `covf.estimate` on page 18.

Purpose Compute the Fourier transform from a SIG object.

Description See `ft` on page 37.

Purpose Estimate the spectrum from a SIG object.

Syntax Use the following calls to estimate the spectrum from a SIG object:

`Phi=estimate(spec,y,Property1,Value1,...)`	Explicit call
`Phi=spec(y,Property1,Value1,...)`	Implicit call
`Phi=sig2spec(y,Property1,Value1,...)`	Direct low-level call

Description See `spec.estimate` on page 135.

Purpose Estimate a Time-Frequency Description (TFD) from a SIG object.

Syntax Use the following calls to estimate a TFD from a SIG object:

`Yt=estimate(tfd,y,Property1,Value1,...)`	Explicit call
`Yt=tfd(y,Property1,Value1,...)`	Implicit call
`Yt=sig2tfd(y,Property1,Value1,...)`	Direct low-level call

Description See `tfd.estimate` on page 226.

See Also `tfd.estimate`

Purpose Pick out the input from a SIG object to be the output in a new SIG object.

Syntax

```
y=u2y(u)
```

Description The SIG object `u` is supposed to consist of `(y,x,u)`, where `x` might be empty. The new SIG object `y` is then `(y,[],[])`. This might be useful in making simulation of systems represented by a signal triplet, where only the input signal should be used in the simulation. Another purpose is for plotting the input only. All descriptive information is inherited automatically.

Example Recover the input from a simulation of a state space model.

```
G=rand(ss([3 1 0 2],1));
u=getsignal('prbs');
z=simulate(G,10)
SIG object with discrete time (fs = 1) input-output state space
data
  Sizes:       N = 10,  ny = 2, nu = 1, nx = 3
u=u2y(z)
SIG object with discrete time (fs = 1) time series
  Sizes:       N = 10,  ny = 1
  MC is set to: 30
  #MC samples:  0
```

See Also `sig.x2y`

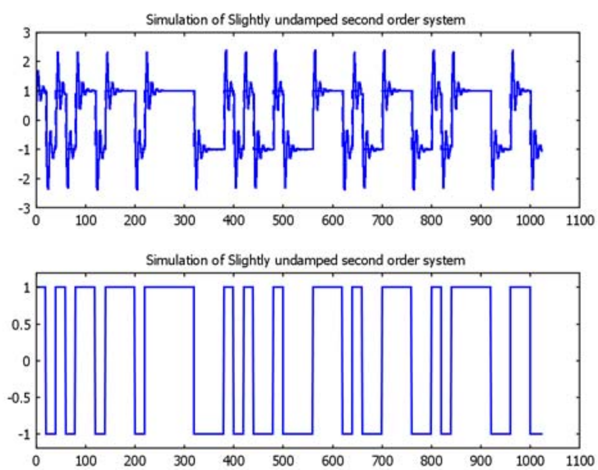
Purpose Plot the input signal in a SIG object.

Syntax `uplot(z)`

Description In a SIG object `z=(y,x,u)`, the normal plot function shows `y(i)` for each input `u(j)` in a subplot, when an input signal is included in the SIG object. The `uplot` function converts the input to a new signal object (similar to `u2y(z)`), and then calls the plot function. That is, the behavior is similar to `plot(u2y(z))`. Labels and other descriptive information are inherited.

Example Simulate the spring response of a PRBS signal, then compare the two plot alternatives.

```
G=exlti('tf2d');
u=getsignal('prbs');
z=simulate(G,u)
SIG object with discrete time (fs = 1) input-output data
  Name:       Simulation of Slightly undamped second order system
  Sizes:       N = 1024,  ny = 1, nu = 1
subplot(2,1,1), plot(z)
subplot(2,1,2), uplot(z)
```



See Also `sig`, `sig.u2y`, `sig.plot`

Purpose Apply a data window to a SIG object.

Syntax

```
yw=window(y,type)
```

Description The function applies a data window to a signal and basically performs the following operations:

1. `w=getwindow(length(y),type);`
2. `yw=y.*w;`

and repeats this for all signal dimensions and Monte Carlo realizations.

See Also `getwindow`

Purpose Pick out the state from a SIG object to be the output in a new SIG object.

Syntax `y=x2y(x)`

Description The SIG object `u` is supposed to consist of `(y,x,u)`, where `u` can be empty. The new SIG object `y` is then `(x,x,u)`. All descriptive information is inherited automatically.

Example Recover the state from a simulation of a state-space model.

```
G=rand(ss([3 1 0 2],1));
u=getsignal('prbs');
z=simulate(G,10)
SIG object with discrete time (fs = 1) input-output state space
data
  Sizes:       N = 10,  ny = 2, nu = 1, nx = 3
x=x2y(z)
SIG object with discrete time (fs = 1) input-output state space
data
  Sizes:       N = 10,  ny = 3, nu = 1, nx = 3
```

See Also `sig.u2y`

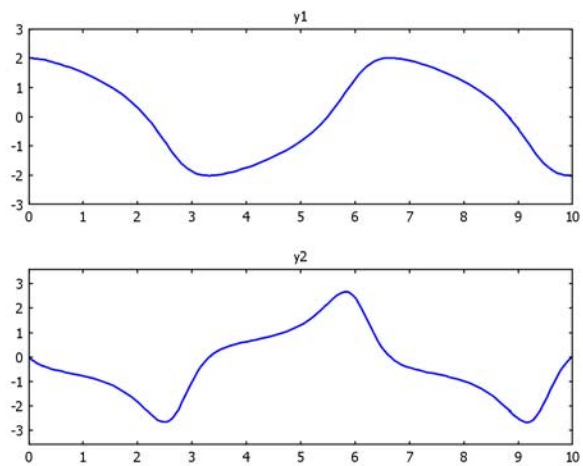
Purpose Plot the states in a SIG object as subplots.

Syntax `xplot(z,Property1,Value1,...)`

Description This is essentially equivalent to `plot(x2y(z))`. That means that the property-value pairs are the same as for `sig.plot`.

Example Plot the states of a simulation of the van der Pol system.

```
m=exnl('vdp');
z=simulate(m,10);
xplot(z)
```



See Also `sig.xplot2`, `sig.plot`, `sig.x2y`

Purpose Plot two states in a SIG object as a state trajectory.

Syntax

```
xplot2(z,Property1,Value1,...,ind)
```

Description This is essentially the same as `plot(z.x(ind(1)),z.x(ind(2)))`, but with some additional features:

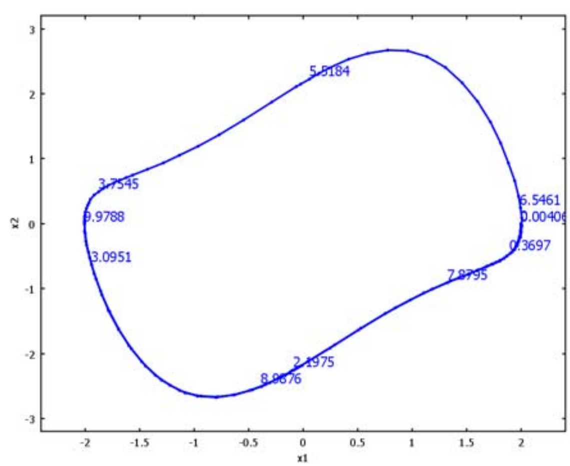
- An uncertain state represented by a covariance in the first place, or Monte Carlo samples in the second place, is illustrated with covariance ellipsoids.
- Time instants along the trajectory are automatically added.

The property/value pairs are the same as for `sig.plot`, with one additional item.

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`tlabel`	`{10}`	Text label for tlabel time instants

Example Plot the states of a simulation of the van der Pol system.

```
m=exnl('vdp');
z=simulate(m,10);
xplot2(z)
```



See Also `sig.xplot`, `sig.plot`, `sig.x2y`

Purpose

Create a spectrum object.

Syntax

The following calls create a spectrum object (SPEC object):

`S=spec`	Empty object
`S=spec(Phi,f)`	Phi(f,1:ny,1:ny)
`S=spec(Phi,f,PhiMC)`	PhiMC(1:MC,f,1:ny,1:ny)
`S=spec(s)`	Conversions from SIG and LTI (ARMAX, ARX, SS) objects

Description

The SPEC object has the following fields:

TYPE	FIELDS
`protected`	`Phi`, `PhiMC`, `f`
`public`	`fs`, `MC`, `name`, `xlabel`, `ulabel`, `ylabel`, `desc`, `method`

The spectrum is defined as the Fourier transform of the covariance function:

$$\Phi(f) = \mathrm{FT}[R(\tau)]$$

The covariance function, in turn, is defined for stationary stochastic processes:

$$R(\tau) = E[s(t)s(t-\tau)]$$

The *periodogram* is the basic input to spectral estimation methods, and it can equivalently be defined in two different ways:

$$\hat{\Phi}(f) = \mathrm{FT}[\hat{R}(\tau)], \qquad \hat{R}(\tau) = \frac{1}{N}\sum_k s[k]s[k-\tau]$$

$$\hat{\Phi}(f) = \frac{T}{N}|\mathrm{TDFT}[s[k]]|^2$$

The three methods to smooth the periodogram implemented in `sig2spec` are:

- Direct smoothing using a low-pass filter approximation and `filtfilt` performs noncausal zero-phase low-pass filtering to avoid frequency shifts in the spectral estimate. The low-pass filter is a running average of M samples, which after `filtfilt` becomes a triangular averaging window.

- Windowing the covariance function estimate using a standard window of size M and type that you choose from the options in `getwindow`. This is called *Blackman-Tukey's method*.
- Segmenting the data into M segments, computing the periodogram on each segment, and then averaging. This is referred to as the *Welch method*.

The basic design parameter that you tune to trade off resolution to noise reduction is basically the same for all three methods: the number of elements in averaging, the size of the window and the number of segments are all related. The design parameter M is therefore in the same order for all methods, but the result is not exactly the same.

Alternatively, if the stationary process is generated by a known model as filtered white noise, then the spectrum is given by

$$s[k] = H(q)e[k] \Rightarrow \Phi(f) = \sigma_e^2 |H(e^{i2\pi f}|^2.$$

This opens up for model-based approaches, where you estimate a model from the signal and then convert it to a spectrum.

When the spectrum is computed from a stochastic process represented by a SIG object with Monte Carlo data, these random realizations are propagated to random realizations of the SPEC object contained in the field `PhiMC`. A similar situation occurs when an uncertain model is converted to a spectrum. Each sample of the uncertain model is converted to spectrum. In both cases, the spectrum can be regarded as a stochastic variable at each frequency. Using the `plot` function, you can add confidence bounds and scatter plots of the realizations.

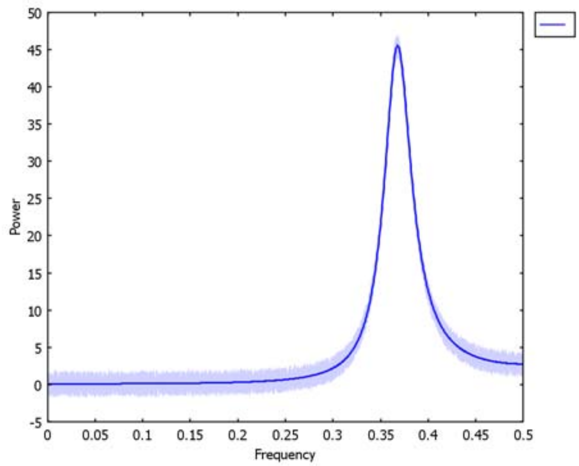
The following operations are available:

OPERATOR	DESCRIPTION	EXAMPLE
`mean/E`	Return the mean of the Monte Carlo data	`Phi=E(PHI)`
`std`	Return the standard deviation of the Monte Carlo data	`Phi=std(PHI)`
`var`	Return the variance of the Monte Carlo data	`Phi=var(PHI)`
`rand`	Return one random SPEC object or a cell array of random SPEC objects	`Phi=rand(PHI,10)`
`fix`	Remove the Monte Carlo simulations from the object	`Phi=fix(PHI)`

Example

Straightforward definition of a SPEC object.

```
MC=30;
H=tf(1,[1 1.2 0.8],1);
Hf=freq(H);
f=Hf.f;
Phi=abs(Hf.H).^2;
PhiMC=4*(-0.5+rand(MC,size(Phi,1)))+repmat(Phi',MC,1);
Phi=spec(Phi,f,PhiMC);
plot(Phi);
```



See Also

`spec.plot`, `spectool`, `ss.ss2spec`

Purpose Spectral estimation of a SIG object

Syntax Use the following calls to perform a spectral estimation of a SIG object:

`Phi=estimate(spec,y,Property1,Value1,...)`	Explicit call
`Phi=spec(y,Property1,Value1,...)`	Implicit call
`Phi=sig2spec(y,Property1,Value1,...)`	Direct low-level call

Description The starting point for spectral analysis is the periodogram. This can be smoothed by the Welch method or Blackman-Tukey's methods, or by a direct low-pass filter using `filtfilt`.

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`MC`	`{100}`	Number of Monte Carlo simulations used in lti2cov
`M`	`{min([N/5 max([N/10 30])])}`	Smoothing parameter, M=1 recovers the periodogram, larger M gives less detail but better averaging. Default is length(y)/30
`method`	`1 | 'periodogram'`	Squared magnitude of Fourier transform
	`2 | 'Blackman-Tukey'`	Smoothed version of periodogram with window win (default: 'hamming') with width M
	`3 | 'Welch'`	Averaged periodogram over different signal segments of length M, windowed by win (default: 'hamming')
	`4 | 'smoothing'`	Applies a smoothing window directly on the periodogram
`overlap`	`{0}`	Overlap used in Welch method (default: 0)
`fs`	`{2}`	Sampling frequency, scales the frequency axle f (default: fs=2). Overrides the fs specified in struct y.
`win`	`'hamming'`	Window used in method 2 and 3. See help window for options (default: 'hamming')

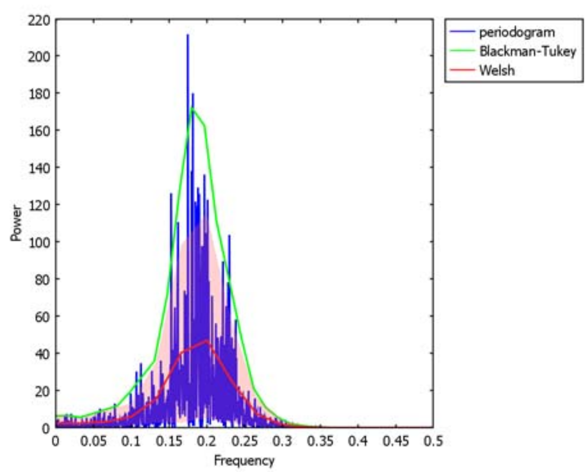
The estimation algorithm computes uncertainty in the Welch method by interpreting the periodogram from the different segments as Monte Carlo data. For smoothed periodogram, the variance over each position of the sliding window is used as uncertainty.

If Monte Carlo realizations of the signal are available, the uncertainty is computed from these instead. This applies to all methods of spectral estimation.

Example

Generate a random ARMA(5,5) model and 1000 samples from it. Then, compare the Welch method and Blackman-Tukey's method with the periodogram. Note the confidence region automatically added for the Welch method:

```
rand('state',2)
Gstruc=tf(5);
Gstruc.fs=1;
G=rand(Gstruc);
y=filter(G,sig(randn(2000,1)));
Phi1=estimate(spec,y,'M',20,'method','periodogram');
Phi2=estimate(spec,y,'M',30,'method','blackman');
Phi3=estimate(spec,y,'M',30,'method','welch');
plot(Phi1,Phi2,Phi3);
```



See Also

`spec`

Purpose Plot spectra estimated from signals or computed from LTI models

Syntax `plot(Phi1,Phi2,...,Property1,Value1,...)`

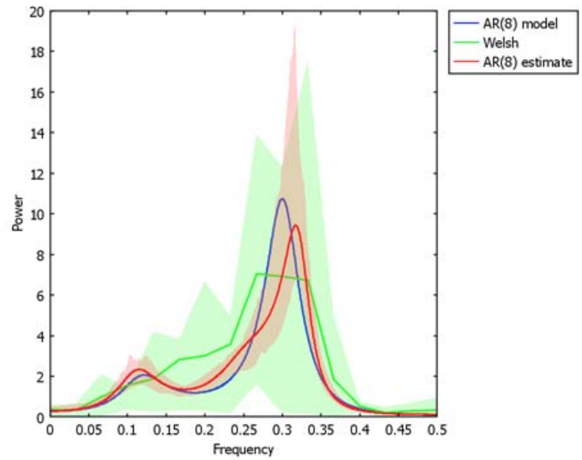
Description Illustrates one or more spectra at the same time.

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
plottype	'plot' | 'semilogx'	Type of plot
	'semilogy' | 'loglog'	
MC	{30}	Number of Monte Carlo simulations
conf	[{0},100]	Confidence level (0 means no levels plotted) from MC data
scatter	'on' | {'off'}	Scatter plot of MC data
xlim	[fmin fmax]	Focus on frequency axis
axis	{gca}	Axis handle where plot is added
linewidth	{2}	Line width
fontsize	{14}	Font size
conftype	1 | {2}	Confidence area (1) or lines (2)
legend	{'on'} | 'off'	Display spectrum data as legend
col	{'bgrmyck'}	Colors in order of appearance

The `plottype` options are also implemented as methods, so the call `semilogy(Phi)` is possible.

Example Generate an AR(8) model and estimate its spectrum from simulated data. Then compare the spectra.

```
m0=rand(ar(8));
y=simulate(m0,256);
M=30;
Phiw=spec(y,’M’,M,’method’,’welch’);
mhat=estimate(ar(8),y);
Phi0=spec(m0);
Phiar8hat=spec(mhat);
plot(Phi0,Phiw,Phiar8hat);
```



See Also `spec`

Purpose Tool for analysis and training of spectral estimation.

Syntax

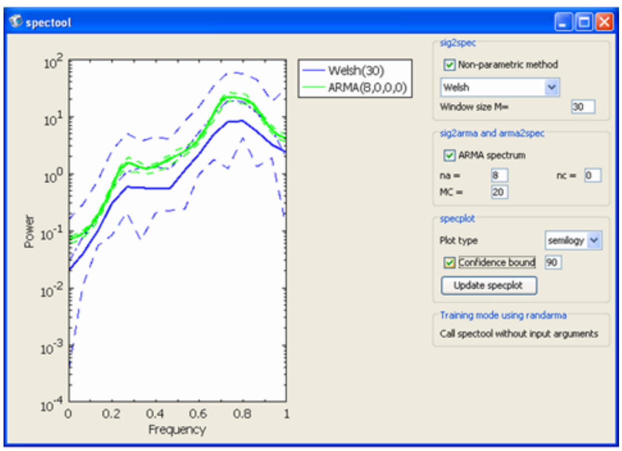
```
spectool(y)
spectool
```

Description Use `spectool` without input arguments to start the user interface in training mode. Use `spectool(y)` for spectral estimation of the signal `y`.

Example Suppose that you have a signal `y` that you want to perform a spectral estimation of. THen start the tool with

```
spectool(y)
```

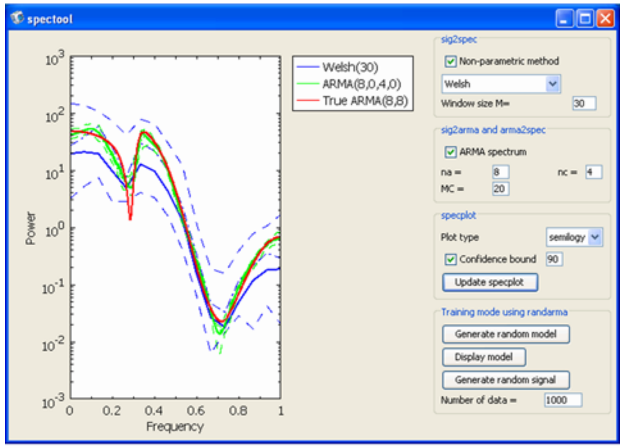
and after selecting the **Confidence bounds** and **ARMA estimation** check boxes, you get the following figure.



In training mode, you simulate a chosen ARMA model yourself. Call `spectool` without input arguments,

```
spectool
```

and the following GUI appears:



Here you can randomize models and generate new realizations of data from the model with varying number of samples. You can then try to fit a model-based spectral estimate as well as possible to simulated data. If you want to cheat, you can at any time display the model currently used.

Purpose

The state-space (SS) model object.

Syntax

There are different ways to construct an SS object:

`SS([nx,nu,nv,ny])`	Empty structure (fs=0 by convention)
`SS([nx,nu,nv,ny],fs)`	Empty structure with sampling frequency
`SS(A,B,C,D,fs)`	Deterministic input-output model
`SS(A,B,C,D,Q,R,S,fs)`	Stochastic input-output model
`SS(A,B,C,D,Q,R,fs)`	As above, without matrix S
`SS(A,[],C,[],Q,R,S,fs)`	Stochastic time series model (no input)
`SS([],[],[],D,fs)`	Static noise-free model
`SS(marx,MC)`	Conversion from ARX model where MC is the number of MC samples from uncertain ARX model

There are also special constructors:

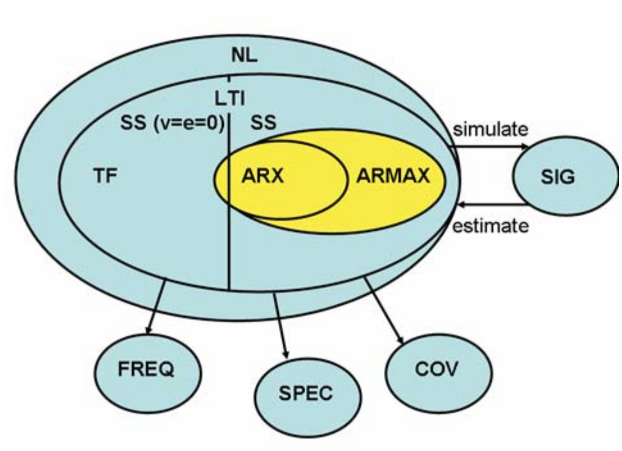
`SS('unit')`	Defines the unit system y=u
`SS('delay')`	Defines the unit delay system y(t)=u(t-1)
`SS('sum')`	Defines the summator (integrator approx.) y(t)=y(t-1)+u(t)
`SS('int')`	Defines the integrator G(s)=1/s

Description

Overloaded functions (methods) include:

- A display function (`display`) used whenever you request a workspace printout.
- A plot function (`ltiplot`) invoked when you type `plot(G)`, `bode(G)`, and other plot commands.
- A simulation function (`simulate`) to produce a `SIG` object.
- An estimation function (`estimate`) to produce a model from a `SIG` object.
- A filter function (`kalman`) invoked when you type `filter(G,y)` for estimating and predicting system state.
- Operators for basic model operations such as `+`, `-`, `*`, `/`, and `feedback`.

The following picture illustrates the structure of the different linear time-invariant (LTI) models. The SS model is the most general one, which implies that it is possible to uniquely transform all other models inside the LTI ellipse to an SS object.



The following section provides a more detailed and comprehensive description of the different methods.

Overloaded operators include:

`plus`	+G	Unitary plus
`uminus`	-G	Unitary minus
`plus`	G1+G2	Parallel connection with summation at output
`minus`	G1-G2	Parallel connection with difference at output
`power`	G.^n	Repeated multiplication
`mpower`	G^n	Repeated multiplication
`inv`	inv(G)	Inverse of square systems
`eq`	G1==G2	Test for equality
`mrdivide`	1/G	Right inverse of a system
`mldivide`	G\1	Left inverse of a system
`mtimes`	G1*G2	Series connection of two models
`times`	G1.*G2	Elementwise multiplication of two models
`feedback`	feedback(G1,G2)	Feedback connection of two systems
`diag`	diag(G1,G2,...)	Append independent models

append	append(G1,G2)	Append independent models
ctranspose	G.'	Reverse the inputs with outputs
transpose	G'	Reverse the inputs with outputs
horzcat	[G1 G2]	Horizontal concatenation
vertcat	[G1;G2]	Vertical concatenation
arrayread	G(i,j)	Pick out subsystems by indexing

These operators do not have a separate reference page.

Certain methods require an input signal to be feasible (such as `plus`, `mtimes`, and `feedback`). For stochastic models, inversion as done in `inv`, for instance, is impossible to define. The rule of thumb is that a method is applicable to stochastic systems in the Signals and Systems Lab whenever it is conceivable.

System analysis tools:

ss.ctrb	Computes the controllability matrix
ss.obsv	Computes the observability matrix
ss.gram	Computes the controllability (observability) Gramian
ss.lqe	Solves the continuous time stationary Riccati equation
ss.dlqe	Solves the discrete time stationary Riccati equation

Overloaded model conversions (SS to SS):

ss.c2d	Convert a continuous-time SS to a discrete-time SS
ss.d2c	Convert a discrete-time SS to a continuous-time SS
ss.minreal	Compute a minimal realization
ss.modred	Compute a reduced-order model using a balanced realization
ss.balreal	Compute the balanced realization

Methods to transform to other object types:

ss.ss2freq	Compute frequency domain response G(f)
ss.zpk	Compute the zeros, poles, and gain
ss.ss2covf	Compute the covariance function of stochastic SS models
ss.ss2spec	Compute the spectrum function of stochastic SS models
ss.ss2tf	Compute the transfer function
ss.impulse	Compute the analytic impulse response

`ss.step`	Compute the analytic step response
`ss.simulate`	Simulate a signal y=G(u) from an LTI state-space model

You can view an LTI object in different ways using the following plots:

- Bode diagram of amplitude and phase as a function of frequency f of $G(f)$. There are options that you can use to plot only the amplitude, only the phase, or both.
- Nyquist curve, where the plot shows $G(f)$ as a complex function.
- Pole-zero plots, which show the poles and zeros of $G(s)$ in the complex plane.
- Root locus for $G(s)$, which is a plot of the poles of the closed-loop system, using a constant feedback K, as a function of K. For SISO TF objects, the poles are the roots of the equation $A(s) + KB(s) = 0$. For MIMO state-space models, the closed loop poles are determined by the eigenvalues to the matrix $A - B(1/kI + D)^{-1}C$.

`lti.bode(G1,G2,...)`	Bode plot of amplitude and phase curves
`lti.bodeamp(G1,G2,...)`	Bode diagram of phase curve
`lti.bodephase(G1,G2,...)`	Bode diagram of phase curve
`lti.nyquist(G1,G2,...)`	Nyquist curve
`lti.zpplot(G1,G2,...)`	Zero-pole plot
`lti.rlplot(G1,G2,...)`	Root locus plot

You can use the same notation for continuous-time and discrete-time systems—just replace s with q above. For MIMO systems, the plots appear in a subplot array with `ny` rows and `nu` columns. It is possible to set the most common properties of standard plots such as `Xlim`, `Ylim`, `fontsize`, `linewidth`, and `axis`. Also, you can specify the color (or color order for multiple LTI object inputs) using the property `col`, which is a vector with one letter color abbreviations such as `b` for blue, `k` for black, and `r` for red.

Example

Create some continuous and discrete state-space models:

```
mc=ss([1 1;0 1],[0;1],[1 0],0)
              / 1  1 \        / 0 \
  d/dt x(t) = \ 0  1 / x(t) + \ 1 / u(t)

y(t) = (1  0) x(t) + (0) u(t)


md=ss([1 1;0 1],[0;1],[1 0],0,2)
           / 1  1 \        / 0 \
```

```
  x[k+1] = \ 0  1 / x[k] + \ 1 / u[k]

y[k] = (1  0) x[k] + (0) u[k]


ms=ss([1 1;0 1],[0;1],[1 0],0,[0 0;0 1],1,2)
           / 1  1 \        / 0 \
  x[k+1] = \ 0  1 / x[k] + \ 1 / u[k] + v[k]

y[k] = (1  0) x[k] + (0) u[k] + e[k]


                /0          0\
  Q = Cov(v) =  \0          1/

R = Cov(e) = 1

msum=ss('sum')
x[k+1] = 1 x[k] + 1 u[k]

y[k] = 1 x[k] + 1 u[k]
```

See Also `tf`

Purpose Compute the balanced realization for an SS object.

Syntax

```
sys2=balreal(sys1)
```

Description A balanced realization is defined as a state-space realization having the same diagonal controllability and observability Gramians

```
gram(sys1,'o')=gram(sys1,'c') = diagonal
```

The transfer function is unchanged. You can use balanced realizations to avoid numerical problems and for model approximation purposes as done in `modred`. Balanced realizations are also useful for model reduction.

The `balreal` function uses the following algorithm:

1. Solve the Lyaponov function $AP+PA^T+BB^T = 0$ for P using `P=gram(s,'c')`.
2. Solve the following Lyaponov function $A^TQ+QA+C^TC = 0$ for Q using `Q=gram(s,'o')`.
3. Compute the factorization $Q = R^TR$.
4. Compute the SVD $RPR^T = U\Sigma^2U^T$.
5. Compute the transformation matrix $T = \Sigma^{-1/2}U^TR$.
6. The balanced realization is given by $(A\_b,B\_b,C\_b,D\_b) = (TAT^{-1},TB,CT^{-1},D)$.

To avoid numerical problems using the overloaded operators, you should compute the balanced realization `s=balreal(s)` after each operation. The Signals and Systems Lab does not do this automatically, because it destroys the structure of the states, which is sometimes something you do not want.

Example Generate a random third-order transfer function, convert it to (observer-canonical) state-space form, and compute its balanced realization. Finally, check the Gramians:

```
m=rand(ss([2 1 0 1]))
                /  -0.47  1 \        /  -0.48 \
  d/dt x(t) = \ -0.065  0 / x(t) + \ -0.065 / u(t)

y(t) = (1  0) x(t) + (1) u(t)


mbal=balreal(m)
                /  -0.4    0.19 \        / 0.71 \
  d/dt x(t) = \ -0.19  -0.071 / x(t) + \ 0.14 / u(t)

y(t) = (-0.71  0.14) x(t) + (1) u(t)
```

```
gram(mbal,'o')
ans =
     0.6286   6.1e-017
   6.1e-017     0.1292
gram(mbal,'c')
ans =
     0.6286   3.4e-018
   3.4e-018     0.1292
```

See Also `ss`, `ss.gram`, `ss.minreal`, `ss.modred`

Purpose

Convert a continuous-time state-space model to discrete time.

Syntax

```
md=c2d(mc,fs,method)
```

Description

Sampling continuous-time models involves an assumption on what happens in between the sampling instants. You can assume that the signal is piecewise constant or piecewise linear and get slightly different results. A further alternative is the bilinear transformation, which guarantees that poles and zeros in the left half plane are mapped to the interior of the unit circle.

The method is a string describing the assumption on intersample behavior:

`{'ZOH'}`	Zero-order hold, assuming piecewise constant input
`'FOH'`	First-order hold, assuming piecewise linear input
`'bilinear'`	s=2/T (z-1)/(z+1)

Example

Compute a second-order Butterworth filter in continuous time. Then sample it using a zero-order hold (piecewise constant signal in each sampling interval) and bilinear transformation. Compare the frequency response of all three filters:

```
mc=ss(tf(1,[1 1 1]))
              / -1  -1 \        / 1 \
  d/dt x(t) = \  1   0 / x(t) + \ 0 / u(t)

y(t) = (0  1) x(t) + (0) u(t)


md=c2d(mc,0.1)
           / -0.0076  -0.0054 \        / 0.0054 \
  x[k+1] = \  0.0054  -0.0022 / x[k] + \      1 / u[k]

y[k] = (0  1) x[k] + (0) u[k]
```

See Also

`ss`, `ss.d2c`, `tf.c2d`

Purpose

Compute the controllability matrix of a system on SS form.

Syntax

```
C=ctrb(m)
```

Description

You can use the controllability matrix to find out whether a system is controllable or not. If `C` has full column rank, then the system is controllable.

Example

Display the controllability matrix for a random second-order model on controllability and observability form, respectively:

```
G=rand(tf([2 1]))

                s^2
   Y(s) = --------------- U(s)
          s^2+0.63*s+0.17

m=ss(G,'o')
              / -0.63  1 \        / -0.63 \
  d/dt x(t) = \ -0.17  0 / x(t) + \ -0.17 / u(t)

y(t) = (1  0) x(t) + (1) u(t)


ctrb(m)
ans =
    -0.6349     0.2311
    -0.1720     0.1092
m=ss(G,'c')
              / -0.63  -0.17 \        / 1 \
  d/dt x(t) = \     1      0 / x(t) + \ 0 / u(t)

y(t) = (-0.63  -0.17) x(t) + (1) u(t)


ctrb(m)
ans =
          1    -0.6349
          0          1
```

See Also

`ss`, `ss.obsv`

Purpose Convert a discrete-time state-space model to continuous time.

Syntax `mc=d2c(m,fs,method)`

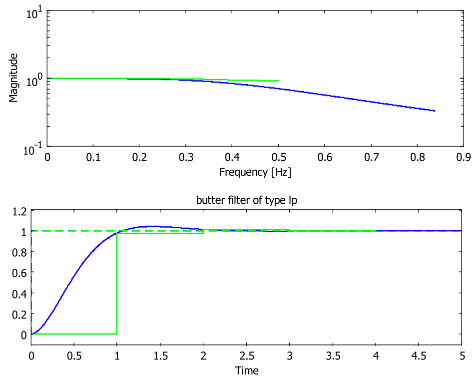
Description This is the inverse function of `c2d`. See `ss.c2d` for more information.

`method` is a string describing the assumption on intersample behavior.

`{'ZOH'}`	Zero-order hold, piecewise constant input assumed
`'FOH'`	First-order hold, piecewise linear input assumed
`'bilinear'`	s=2/T (z-1)/(z+1)

Example Compute a second-order Butterworth filter with a sampling frequency of 2. Then, compute the continuous-time filter and compare their transfer functions.

```
Gctf=getfilter(2,0.5,'fs',NaN);
Gc=ss(Gctf)
                 / -4.4  -9.9 \         / 1 \
  d/dt x(t) = \    1     0 / x(t) + \ 0 / u(t)

y(t) = (0  9.9) x(t) + (0) u(t)


Gd=c2d(Gc,1,'zoh')
             / -0.15  -0.38 \         / 0.039 \
  x[k+1] = \ 0.039  0.021 / x[k] + \ 0.099 / u[k]

y[k] = (0  9.9) x[k] + (0) u[k]


Gc=d2c(Gd,'zoh');
subplot(2,1,1), plot(Gc,Gd);
subplot(2,1,2), plot(step(Gc,5),step(Gd,5))
```



See Also ss, ss.c2d

Purpose Solve the discrete-time stationary Riccati equation

Syntax `P=dlqe(A,B,C,Q,R)`

Description The discrete-time stationary Riccati equation is defined as

$$P_p = AP_pA^T - AP_pC^T(CP_pC^T + R)^{-1}CP_pA' + Q,$$
$$P_f = P_p - P_pC^T(CP_pC^T + R)^{-1}CP_p,$$
$$K = AP_pC^T(CP_pC^T + R)^{-1}$$

Here, `A`, `B`, and `C` are the state-space matrices of the model, and `Q` and `R` are the covariance matrices of the noise sources (see `ss`).

K is the stationary Kalman gain, P_p is the stationary covariance matrix for the prediction errors and P_f the corresponding covariance matrix for filtering errors. This Riccati equation solver uses a simple iterative algorithm.

Example Get the example of a motion model in 1D, and compute its stationary solution to the Riccati solution for predicted and filtered covariance as well as the stationary Kalman gain:

```
m=exlti('motion1D')
             / 1  1 \
  x[k+1] = \ 0  1 / x[k] + v[k]

y[k] = (1  0) x[k] + e[k]


                   /0.25        0.5\
  Q = Cov(v) =  \ 0.5           1/

R = Cov(e) = 1

[K,Pf,Pp]=dlqe(m)
K =
     0.7500
     0.5000
Pf =
     3.0000     2.0000
     2.0000     2.0000
Pp =
     0.7500     0.5000
     0.5000     1.0000
```

See Also `ss`, `ss.lqe`

Purpose Estimate a linear state space model from data

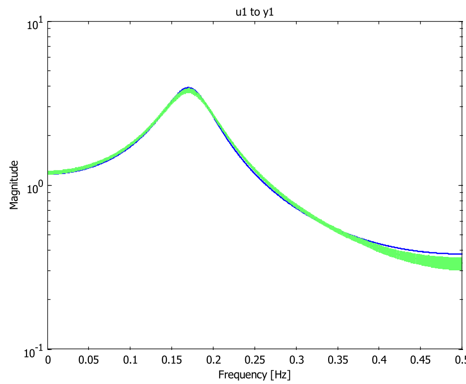
Syntax `mhat=estimate(m,z)`

Description A state-space model with structure as specified in `m` is estimated from the data in `z`. The input and output dimensions of `m` and `z` must be the same. The implementation is based on the `tf.estimate` function, so essentially the code is

```
sys=ss(estimate(tf(s),sig,varargin{:}))
```

Example Generate a random state-space model, simulate data, and estimate a model with the same structure:

```
G=rand(ss([2 1 0 1],1))
           / 0.79  1 \        / 0.85 \
  x[k+1] = \ -0.7  0 / x[k] + \ -0.7 / u[k]

y[k] = (1  0) x[k] + (1) u[k]


y=simulate(G,getsignal('prbs',100))+0.2*randn(100,1);
Ghat=estimate(G,y)
           / 0.77  -0.68 \        / 1 \
  x[k+1] = \    1      0 / x[k] + \ 0 / u[k]

y[k] = (0.89  -0.71) x[k] + (0.99) u[k]


           / 0.77  -0.68 \        / 1 \
  x[k+1] = \    1      0 / x[k] + \ 0 / u[k]

y[k] = (0.89  -0.71) x[k] + (0.99) u[k]


plot(G,Ghat,'conf',90)
```



See Also `ss`, `tf.estimate`

Purpose

Compute the controllability (observability) Gramian.

Syntax

```
P=gram(m)
```

Description

The controllability Gramian P is defined as the solution to $P = APA' + BB'$

The observability Gramian is defined analogously as the solution to $Q = A'PA + C'C$, and you obtain it by `Q=gram(m,'o');`

Example

Take an arbitrary second-order transfer function on state-space form and check its Gramians. Then compute a balance realization and verify that they are equal.

```
m=rand(ss([2 1 0 1]))
                 /   -0.47  1 \        /   -0.48 \
  d/dt x(t) = \ -0.065  0 / x(t) + \ -0.065 / u(t)

y(t) = (1  0) x(t) + (1) u(t)


gram(m,'o')
ans =
     1.0620          0
          0    16.2184
gram(m,'c')
ans =
     0.3182     0.0327
     0.0327     0.0046
mb=balreal(m)
                 /   -0.4    0.19 \        / 0.71 \
  d/dt x(t) = \ -0.19  -0.071 / x(t) + \ 0.14 / u(t)

y(t) = (-0.71  0.14) x(t) + (1) u(t)


gram(mb,'o')
ans =
     0.6286   6.1e-017
   6.1e-017     0.1292
gram(mb,'c')
ans =
     0.6286   3.4e-018
   3.4e-018     0.1292
```

The input Gramian is well balanced for state-space models on controller canonical form as in this example. To balance both the controllability and observability Gramians, transform the model to a balanced realization.

See Also

`ss`, `ss.balreal`, `ss.minreal`, `ss.modred`, `ss.obsv`, `ss.ctrb`

Purpose Simulate an impulse response.

Syntax `y=impulse(G,T)`

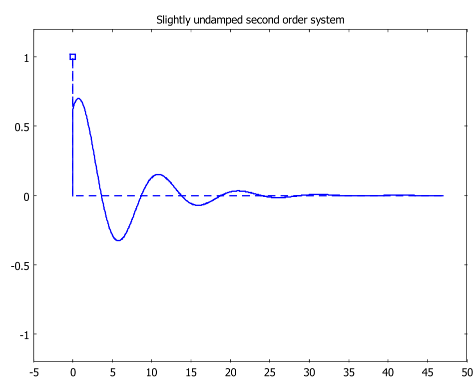
Description Available input arguments:

ARGUMENT	DESCRIPTION
G	SS object
T	Simulation time. By default, it is estimated from the dominating pole
y	SIG object

The impulse response is computed by formulas from system theory rather than simulated using numerical integration routines. To achieve the latter simulation, use a call like `y=simulate(G,getsignal('impulse'))`.

Example Simulate an impulse response of a second-order lightly damped system:

```
G=ss(exlti('tf2c'))
                  / -0.3   -0.41 \         / 1 \
  d/dt x(t) = \     1       0 / x(t) + \ 0 / u(t)

y(t) = (0.62  0.41) x(t) + (0) u(t)


y=impulse(G);
plot(y)
```



See Also `ss`, `ss.step`, `ss.simulate`

Purpose Estimate the state of a state-space model using the Kalman filter (KF).

Syntax `[x,V]=kalman(m,z,Property1,Value1,...)`

Description For a linear state-space model defined by

$$\begin{aligned} x_{k+1} &= A_k x_k + B_{u,k} u_k + B_{u,k} v_k, \\ y_k &= C_k x_k + D_k u_k + e_k, \\ &\mathrm{Cov}(x_0) = P_0,\ \ E(x_0) = \hat{x}_{1|0}, \\ \mathrm{Cov}(v_k) &= Q_k, \mathrm{Cov}(e_k) = R_k, \mathrm{Cov}(v_k, e_k) = 0. \end{aligned}$$

the optimal linear filter is given by the Kalman filter (KF) recursions

$$\begin{aligned} \hat{x}_{k+1|k} &= A_k \hat{x}_{k|k} + B_{u,k} u_k, \\ P_{k+1|k} &= A_k P_{k|k} A_k^T + B_{v,k} Q_k B_{v,k}^T, \\ \hat{x}_{k|k} &= \hat{x}_{k|k-1} + P_{k|k-1} C_k^T (C_{k|k-1} C_k^T + R_k)^{-1} (y_k - C_k \hat{x}_{k|k-1} - D_{u,k} u_k), \\ P_{k|k} &= P_{k|k-1} - P_{k|k-1} C_k^T (C_{k|k-1} C_k^T + R_k)^{-1} C_{k|k-1}. \end{aligned}$$

The inputs to the KF are the SS object and a SIG object containing the observations y_k and possible an input u_k, and the outputs are the state estimate $x_{k|k}$ and its covariance matrix $P_{k|k}$. There is also a possibility to predict future states $x_{k+m|k}$, $P_{k+m|k}$ with the m-step ahead predictor, or to compute the smoothed estimate using the complete observation record $x_{k|N}$, $P_{k|N}$. The corresponding output estimate and covariance are also computed. All these quantities are packed into a SIG object, where also the signal labels inherited from the model are assigned.

The arguments are as follows:

- `m` is a SS object defining the model matrices A, B, C, D, Q, R.
- `z` is a SIG object with measurements y and inputs u if applicable. The state field is not used by the KF.
- `x` is a SIG object with state estimates. `xhat=x.x` and signal estimate `yhat=x.y`.
- `V` is the normalized sum of squared innovations, which should be a sequence of `chi2dist(nx)` variables when the model is correct.

The optional parameters are summarized in the table below.

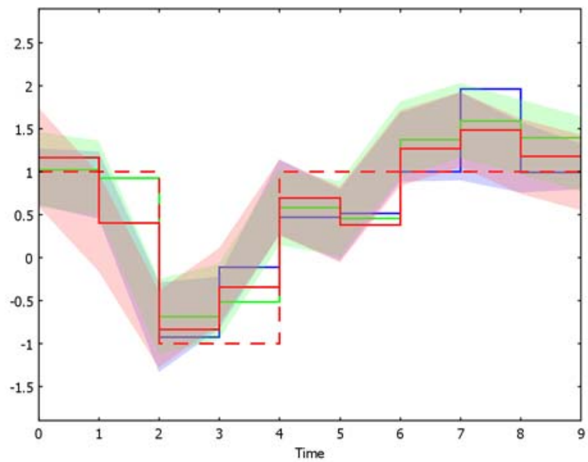
TABLE 2-22: OPTIONAL PARAMETERS FOR THE KALMAN FUNCTION

PROPERTY	VALUE	DESCRIPTION
alg	{1},2,3,4	Type of implementation:
		1 stationary KF.
		2, time-varying KF.
		3, square root filter.
		4, fixed interval KF smoother Rauch-Tung-Striebel.
		5, sliding window KF, delivering xhat(t|y(t-k+1:t)), where k is the length of the sliding window.
k	k>0 {0}	Prediction horizon: 0 for filter (default), 1 for one-step ahead predictor, generally k>0 gives xhat(t+k|t) and y(t+k|t) for alg=1,2. In case alg=5, k=L is the size of the sliding window.
P0	{[]}	Initial covariance matrix. Scalar value scales identity matrix. Empty matrix gives a large identity matrix.
x0	{[]}	Initial state matrix. Empty matrix gives a zero vector.
Q	{[]}	Process noise covariance (overrides the value in m.Q). Scalar value scales m.Q.
R	{[]}	Measurement noise covariance (overrides the value in m). Scalar value scales m.R.

Example

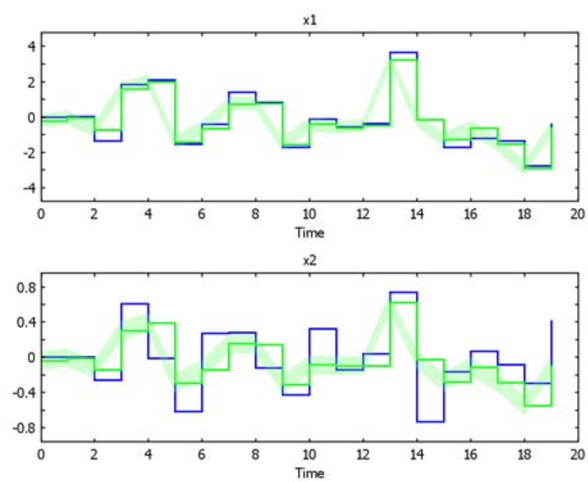
Simulate a random state space model, and use the KF to estimate the state from noisy outputs.

```
m=rand(ss([2 1 1 1],1));
m.R=0.1*m.R;
m.Q=0.1*m.Q;
u=getsignal('prbs',10,2);
z=simulate(m,u);
x1=kalman(m,z,'alg',1);
x2=kalman(m,z,'alg',2);
plot(z,x1,x2,'conf',90)
```



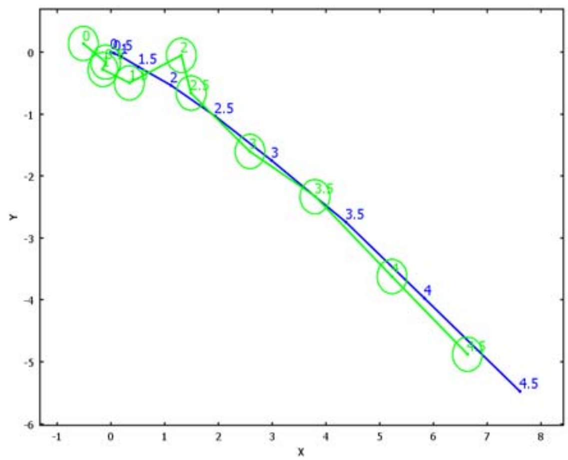
Simulate a random time series, and estimate the state.

```
m=rand(ss([2 0 1 1],1));
m.R=0.1*m.R;
z=simulate(m,20);
x=kalman(m,z);
xplot(z,x,'conf',90)
```



Simulate a constant acceleration motion model with noisy position estimates, then estimate the state vector and plot the position estimate with uncertainty ellipsoids.

```
m=exlti('ca2D');
m.R=10*m.R;
z=simulate(m,10);
xhat=kalman(m,z);
xplot2(z,xhat,'conf',90,[1 2]);
```



See Also `nl.ekf`, `sig.xplot`, `sig.xplot2`

Purpose Solve the continuous-time stationary Riccati equation.

Syntax

```
[K,P]=lqe(m)
```

Description The continuous-time stationary Riccati equation is defined as

$$AP + PA^T - PC^TR^{-1}CP + Q = 0$$
$$K = PC^TR^{-1}$$

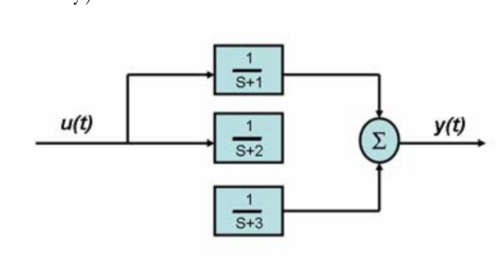
P corresponds to the solution, and K is the stationary Kalman gain.

See Also `ss.dlqe`

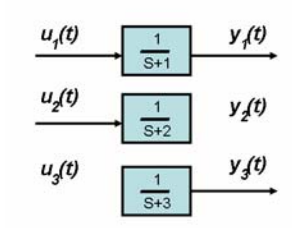
Purpose Compute a minimal realization of a system on SS form.

Syntax `[mmin,T,uind,yind]=minreal(m)`

Description Model reduction basically cancels common zeros and poles and removes states that are either not observable or not controllable. The following figure illustrates a simple case where the last two states are not needed to model the input-output dynamics (though these states still may contain important information on internal stability).



For MIMO system, a minimal realization also removes inputs that are not related to the outputs, and vice versa. The following figure illustrates one such example:



`T` is the transformation matrix such that `xnew=T*x`.

`uind` and `yind` are the removed inputs and outputs, respectively.

The algorithm works as follows:

1 Call `ss.modred` with an automatic choice of model order, so the truncation is done where all eigenvalues of the balanced `A` matrix are zero up to a numerical uncertainty.

2 Screen the columns of the `B` and `D` matrices for all-zero vectors, corresponding to unused inputs, and remove such cases.

3 Screen the `C` and D matrices for zero columns, and remove them.

Example

Define the SISO system in the first of the previous examples, and compute its minimal realization:

```
G=ss(diag([-1 -2 -3]),[1;1;0],[1 0 1],0)
              / -1   0   0 \        / 1 \
  d/dt x(t) = |  0  -2   0 | x(t) + | 1 | u(t)
              \  0   0  -3 /        \ 0 /

y(t) = (1  0  1) x(t) + (0) u(t)


minreal(G)
d/dt x(t) = -1 x(t) + 1 u(t)

y(t) = 1 x(t) + 0 u(t)
```

Note that the transfer function contains both poles and zeros, at –2 and –3, respectively.

The illustrative MIMO example above is defined and reduced below.

```
G=ss(diag([-1 -2 -3]),diag([1;1;0]),diag([1 0 1]),zeros(3))
              / -1   0   0 \        / 1  0  0 \
  d/dt x(t) = |  0  -2   0 | x(t) + | 0  1  0 | u(t)
              \  0   0  -3 /        \ 0  0  0 /

          /1  0  0\          /0  0  0\
  y(t) = | 0  0  0 | x(t) + | 0  0  0 | u(t)
          \0  0  1/          \0  0  0/


minreal(G)
d/dt x(t) = -1 x(t) + 1 u(t)

y(t) = 1 x(t) + 0 u(t)
```

See Also

`ss`, `ss.modred`, `ss.balreal`, `ss.gram`

Purpose

Compute a reduced-order model using a balanced realization.

Syntax

```
[mred,T]=modred(m,n);
```

Description

The function first calls `balreal` to get a balanced realization, where the eigenvalues of the `A` matrix are sorted in order. This model is then truncated at order `n`. `T` is the transformation matrix such that `xnew=T*x`.

`n` is the order of the filter after model reduction. If `n` is empty or zero, the model order is chosen automatically to cancel common poles and zeros. For SISO systems, `modred` and `minreal` are then equivalent.

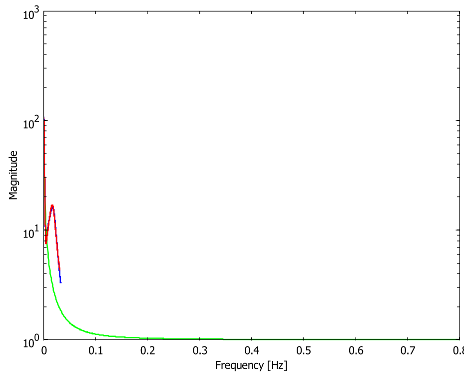
The algorithm works as follows:

1. First call `balreal` to get a balanced state-space realization, where the eigenvalues of the A matrix are sorted in order.
2. Then truncate this model at model order n. If n is empty or zero, the model order is chosen automatically, based on the eigenvalues in A.

Example

Compute a random state-space model of order 12. Then, approximate this with a second-order system and a system with an automatically chosen model order.

```
m=rand(ss([12 1 0 1]));
mred2=modred(m,2);
mredauto=modred(m)
              / 0.00034  -0.0057   0.0012  -0.0019  -0.00044
              |    0.01    0.012  -0.0029    0.022     0.005
              |  -0.023    0.017   -0.013    0.095    0.0037
  d/dt x(t) = | 0.00091    0.022    -0.15   -0.056    -0.068
              | -0.0038    0.026    0.054    0.061    -0.051
              \  0.0027   -0.044    -0.19    -0.32       2.3

  0.0016 \        /  0.37 \
  -0.017 |        |   0.7 |
   -0.05 |        |   0.8 |
    0.35 | x(t) + |  0.64 | u(t)
    -1.7 |        | -0.38 |
    -1.4 /        \   1.7 /

y(t) = (-0.16  0.58  0.33  -0.65  -0.33  1.7) x(t) + (1)

u(t)


plot(m,mred2,mredauto);
```



See Also ss, ss.balreal, ss.gram, ss.minreal

Purpose Compute the observability matrix of a system on SS form.

Syntax `O=obsv(m)`

Description You can use the observability matrix to find out whether a system is observable or not. If `O` has full column rank, then the system is observable.

Example Display the observability matrix for a random second-order model on controllability and observability form, respectively:

```
G=rand(tf([2 1]))

                 s^2
   Y(s) = --------------- U(s)
          s^2+0.63*s+0.17

m=ss(G,'o')
              / -0.63  1 \        / -0.63 \
  d/dt x(t) = \ -0.17  0 / x(t) + \ -0.17 / u(t)

y(t) = (1  0) x(t) + (1) u(t)


obsv(m)
ans =
          1          0
    -0.6349          1
m=ss(G,'c')
              / -0.63  -0.17 \        / 1 \
  d/dt x(t) = \     1      0 / x(t) + \ 0 / u(t)

y(t) = (-0.63  -0.17) x(t) + (1) u(t)


obsv(m)
ans =
    -0.6349    -0.1720
     0.2311     0.1092
```

See Also `ss`, `ss.ctrb`

Purpose Simulate a signal from an SS object.

Syntax `z=simulate(G,N,Property1,Value1,...)`

Description This function is the overloaded plot function for SS objects.

The second input argument is one of the following:

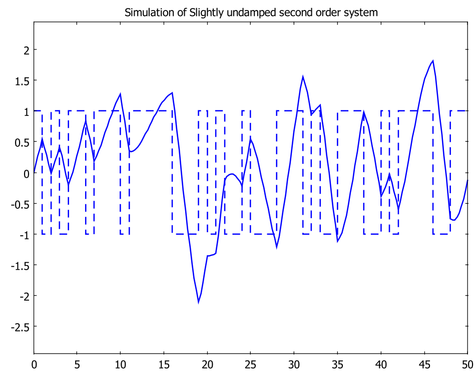
INPUT ARGUMENT	DESCRIPTION
u	Input to input-output models
N	Number of data to simulate.
	u is white noise for input-output models

Optional parameters:

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
MC	{10}	Number of Monte Carlo simulations
T	{0}	Simulation time. The default (for T=0) is computed from the dominating pole.

Example Simulate a PRBS signal through a lightly damped system on state-space form, and plot the result:

```
G=ss(exlti('tf2c'))
                 / -0.3  -0.41 \         / 1 \
  d/dt x(t) = \    1       0 / x(t) + \ 0 / u(t)

y(t) = (0.62  0.41) x(t) + (0) u(t)


u=getsignal('cprbs',50);
y=simulate(G,u);
plot(y)
```



See Also `tf.simulate`, `ss.step`, `ss.impulse`

Purpose

Convert SS models to covariance functions.

Syntax

`c=ss2covf(m,Property1,Value1,...)`	Explicit call
`c=covf(m,Property1,Value1,...)`	Implicit call

Description

The theoretical covariance function for a state-space model is computed. The covariance function is defined for stochastic processes, so this function is not applicable to transfer function part of the state-space model. That is, the deterministic input dynamics is not considered for the covariance function.

PROPERTY	VALUE	DESCRIPTION
taumax	{30}	Maximum lag for which the covariance function is computed
MC	{100}	Number of Monte Carlo simulations to compute the confidence bound (0 means no bound)

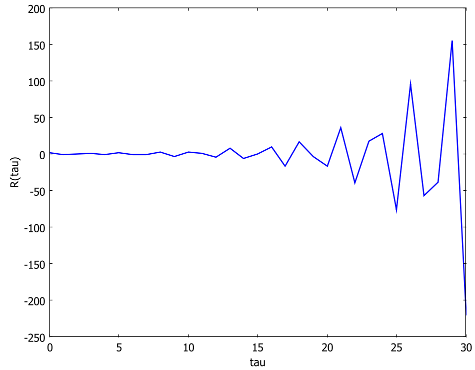
The following steps describe the algorithm:

1. `R(0) = C * Pibar * C' + R`
2. `R(tau) = C * A^tau * Pibar * C' + C * A^(tau-1) * S,`
   `tau=1,2,...,taumax`, where `Pibar` is the controllability Gramian (see `gram`).

Example

Create a random stochastic SS model and compute its covariance function:

```
m=rand(ss([4 0 1 1]))
                /    -1.9   1   0   0 \
                |    -1.7   0   1   0 |
  d/dt x(t) =   |   -0.24   0   0   1 | x(t) + v(t)
                \  -0.098   0   0   0 /

y(t) = (1   0   0   0) x(t) + e(t)


                 /   7.7         12         1.2       -0.27\
                |     12         18         1.8       -0.41 |
  Q = Cov(v) =  |    1.2        1.8        0.19      -0.042 |
                 \-0.27       -0.41      -0.042      0.0095/

R = Cov(e) = 1

c=covf(m);
plot(c)
```



See Also `sig.sig2covf`, `covf`

Purpose Compute frequency-domain response of an LTI object.

Syntax

`Hf=ss2freq(lti,Property1,Value1,...)` Explicit call
`Hf=freq(lti,Property1,Value1,...)` Implicit call

Description The frequency function $H(f)$ is the Fourier transform of the input-output transfer function if the SS object contains an input u (TF part of the SS object). Otherwise, it is the frequency function for the noise model (ARMA part of the SS object).

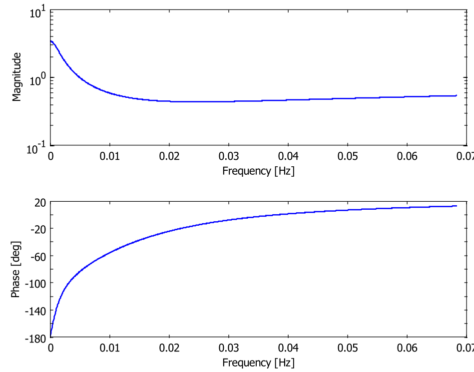
This conversion supports MIMO systems, and evaluates $H(i\omega) = C\,(i\omega I - A)^{-1}B + D$ for continuous-time systems and $H(e^{i\omega}) = C(e^{i\omega}I - A)^{-1}B + D$ for discrete-time systems, respectively, for each input-output channel.

The available properties are:

PROPERTY	VALUE	DESCRIPTION
MC	{30}	Number of Monte Carlo simulations used in lti2cov
N	{1024}	Number of frequency grid points
f	{}	Frequency grid (overrides N)
fmax		Maximum frequency. Default is fs/2 for discrete-time systems, and 8 times the dominating poles bandwidth for continuous-time systems

Example Create a random stochastic SS model and compute its frequency function:

```
m=rand(ss([3 1 0 1]))
                 /    -1.5   1   0 \         /   -0.6 \
    d/dt x(t) = |    -0.3   0   1 | x(t) + | -0.13 | u(t)
                 \ 0.0028   0   0 /         \ 0.012 /

y(t) = (1  0  0) x(t) + (1) u(t)


Hf=freq(m);      % Implicit call
Hf=ss2freq(m);   % Explicit call
bode(Hf);
```



See Also `freq`, `tf.tf2freq`

Purpose Convert a linear state-space model into an NL model object.

Syntax `mnl=ss.ss2nl(mss)`

Description A linear state-space model represented by a set of matrices is converted to symbolic inline functions for dynamics and observations, respectively. All other fields (labels, sampling frequency, description, name, and so on) are kept.

Example Generate a random state-space model and convert it to an NL object:

```
mss=rand(ss([2 2 2 2]))
              / -0.63  1 \        / 0.92  0.75 \
  d/dt x(t) = \ -0.17  0 / x(t) + \ 0.81  0.84 / u(t) + v(t)

          /   1     0\          /   1     1\
  y(t) =  \0.75  0.81/  x(t) +  \0.31  0.26/  u(t) + e(t)


                /  3.6      -0.39\
  Q = Cov(v) =  \-0.39      0.059/

               / 1          0\
  R = Cov(e) = \ 0          1/

ss2nl(mss)
NL object
 dx/dt = [-0.6348615080768729 1 ; -0.1719648566020535
0]*x(1:2,:)+[0.9175267265890872 0.748133741866277 ;
0.812135496471976 0.8437165228041728]*u(1:2,:) +
N([0;0],[3.62,-0.387;-0.387,0.0591])
    y = [1 0 ; 0.7546923726749087 0.8096911178933771]*x(1:2,:)+[1
1 ; 0.30656439000410485 0.26369928274838383]*u(1:2,:) +
N([0;0],[1,0;0,1])
   x0' = [0         0]
```

See Also `nl.nl2ss`, `nl`

Purpose Compute the spectrum from a stochastic state-space model as an SS object.

Syntax

```
Phi=ss2spec(m,Property1,Value1,...)  Explicit call
Phi=spec(m,Property1,Value1,...)     Implicit call
```

Description The spectrum is computed from the stochastic part of the SS object. That is, any possible input part is neglected.

PROPERTY	VALUE	DESCRIPTION
MC	{30}	Number of Monte Carlo simulations used in lti2cov
N	{1024}	Number of frequency grid points
f	{}	Frequency grid (overrides N)
fmax		Maximum frequency. Default is fs/2 for discrete-time systems, and 8 times the dominating poles bandwidth for continuous-time systems

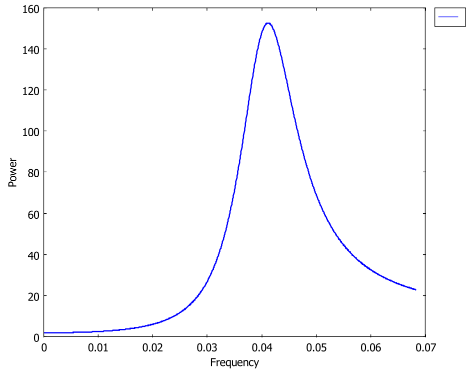
The spectrum is computed in two steps:

1 The transfer function from process noise to output is computed as $H(i\omega) = C(i\omega I + A)^{-1}Q^{1/2}$.

2 The spectrum is then $\Phi(i\omega) = H(i\omega)H(i\omega)^T + R$.

Examples Generate a random SS model, and compute its corresponding spectrum:

```
m=rand(ss([4 0 1 1]))
              /    -1.9  1  0  0 \
              |    -1.7  0  1  0 |
  d/dt x(t) = |   -0.24  0  0  1 | x(t) + v(t)
              \  -0.098  0  0  0 /

y(t) = (1  0  0  0) x(t) + e(t)


                /  7.7         12        1.2       -0.27\
               |    12         18        1.8       -0.41 |
  Q = Cov(v) = |   1.2        1.8       0.19      -0.042 |
                \-0.27      -0.41     -0.042      0.0095/

R = Cov(e) = 1

Phi=spec(m);      % Implicit call
Phi=ss2spec(m);   % Explicit call
plot(Phi);
```



See Also `spec`, `spec.plot`

Purpose

Convert a SS object state-space model to a transfer function TF.

Synopsis

```
sys=ss2tf(s0)          Explicit call
sys=tf(s0)             Implicit call
```

Description

This SS method calls the function `ss2tf`. In contrast to that function, this method works for MIMO systems and uncertain systems. Each input-output channel is treated separately. Any stochastic part of the SS object is neglected.

Example

Convert a SISO state-space model to a transfer function:

```
m1=rand(ss([3 1 0 1]))
                /  -1.6  1  0 \        /  0.5 \
  d/dt x(t) = | -0.96  0  1 | x(t) + |  1.1 | u(t)
                \ -0.19  0  0 /        \ 0.54 /

y(t) = (1  0  0) x(t) + (1) u(t)


G1=ss2tf(m1)  % Explicit call

          s^3+2.1*s^2+2*s+0.73
   Y(s) = ---------------------- U(s)
         s^3+1.6*s^2+0.96*s+0.19

G1=tf(m1)     % Implicit call

          s^3+2.1*s^2+2*s+0.73
   Y(s) = ---------------------- U(s)
         s^3+1.6*s^2+0.96*s+0.19
```

Conversion of stochastic MIMO model:

```
m2=rand(ss([2 2 2 2]))
              / -2  1 \        /  -1.8  0.11 \
  d/dt x(t) = \ -1  0 / x(t) + \ -0.97  0.28 / u(t) + v(t)

          /   1     0\          /    1     1\
  y(t) =  \0.83  0.93/  x(t) +  \0.091  0.85/  u(t) + e(t)


                /5.4        3.4\
  Q = Cov(v) =  \3.4        2.2/

               / 1          0\
  R = Cov(e) = \ 0          1/

G2=ss2tf(m2)  % Explicit call
```

```
          s^2+0.22*s+0.072
Y1(s) = ---------------- U1(s)
             s^2+2*s+1

          s^2+2.1*s+1.3
Y1(s) = ------------- U2(s)
            s^2+2*s+1

          0.091*s^2-2.2*s-0.79
Y2(s) = -------------------- U1(s)
               s^2+2*s+1

          0.85*s^2+2.1*s+1.5
Y2(s) = ------------------ U2(s)
              s^2+2*s+1
```

See Also `ss`, `tf`, `tf2ss`

Purpose Convert a SS object state-space model to a transfer function TF.

Syntax

```
sys=ss2tf(s0)          Explicit call
sys=tf(s0)             Implicit call
```

Description This SS method calls the function `ss2tf`. In contrast to that function, this method works for MIMO systems and uncertain systems. Each input-output channel is treated separately. Any stochastic part of the SS object is neglected.

Example Convert a SISO state-space model to a transfer function:

```
m1=rand(ss([3 1 0 1]))
              /  -1.6  1  0 \       /  0.5 \
  d/dt x(t) = | -0.96  0  1 | x(t) + |  1.1 | u(t)
              \ -0.19  0  0 /       \ 0.54 /

y(t) = (1  0  0) x(t) + (1) u(t)


G1=ss2tf(m1)  % Explicit call

           s^3+2.1*s^2+2*s+0.73
   Y(s) = ----------------------- U(s)
          s^3+1.6*s^2+0.96*s+0.19

G1=tf(m1)     % Implicit call

           s^3+2.1*s^2+2*s+0.73
   Y(s) = ----------------------- U(s)
          s^3+1.6*s^2+0.96*s+0.19
```

Conversion of stochastic MIMO model:

```
m2=rand(ss([2 2 2 2]))
              / -2  1 \       /  -1.8  0.11 \
  d/dt x(t) = \ -1  0 / x(t) + \ -0.97  0.28 / u(t) + v(t)

          /   1    0\          /    1     1\
  y(t) =  \0.83  0.93/  x(t) +  \0.091  0.85/  u(t) + e(t)


                /5.4       3.4\
  Q = Cov(v) =  \3.4       2.2/

               / 1         0\
  R = Cov(e) = \ 0         1/

G2=ss2tf(m2)  % Explicit call
```

```
         s^2+0.22*s+0.072
Y1(s) = ---------------- U1(s)
            s^2+2*s+1

         s^2+2.1*s+1.3
Y1(s) = ------------- U2(s)
           s^2+2*s+1

         0.091*s^2-2.2*s-0.79
Y2(s) = -------------------- U1(s)
              s^2+2*s+1

         0.85*s^2+2.1*s+1.5
Y2(s) = ------------------ U2(s)
             s^2+2*s+1
```

See Also `ss`, `tf`, `tf2ss`

Purpose

Create LaTeX code from a state-space SS object.

Syntax

```
texcode=tex(s,Property1,Value1,...)
```

Description

The output is TeX code that you can paste into any LaTeX document. Alternatively, the code is put into a file, which is input by reference into the document.

Filters (freeware or shareware) for other word processors are available:

- TexPoint: freeware for Microsoft Powerpoint
- LaImport: FrameMaker
- TeX2Word: Microsoft Word
- LaTeX2rtf: RTF documents
- TeX4ht: HTML or XML hypertext documents

The properties are:

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`filename`	`{''}`	Name of the .tex file (none for `''`)
`decimals`	`{1}`	Number of decimals
`env`	`{'eqnarray*'}`	TeX environment, `''` means no env

Example

Generate a random model and its LaTeX code:

```
m=rand(ss([2 1 0 1]))
              / -0.63  1 \        /  1.1 \
  d/dt x(t) = \ -0.17  0 / x(t) + \ 0.54 / u(t)

y(t) = (1  0) x(t) + (1) u(t)


tex(m,'decimals',3)
ans =
  \begin{eqnarray*}
  \dot{x}(t) &=&
  \left[
  \begin{array}{rr}
  -0.635 & 1.000 \\
  -0.172 & 0.000
  \end{array}
  \right]
  x(t)
   +
  \left[
  \begin{array}{r}
  1.080 \\
```

```
0.545
\end{array}
\right]
u(t)
\\
y(t) &=&
\left[
\begin{array}{rr}
1.000 & 0.000
\end{array}
\right]
x(t)
 +
\begin{array}{r}
1.000
\end{array}
u(t)
\end{eqnarray*}
```

Importing the LaTeX code to FrameMaker produces the following printout:

$$\dot{x}(t) \quad = \left[\begin{array}{rr} -0.635 & 1.000 \\ -0.172 & 0.000 \end{array}\right] x(t) + \left[\begin{array}{r} 1.080 \\ 0.545 \end{array}\right] u(t)$$

$$y(t) \quad = \left[\begin{array}{rr} 1.000 & 0.000 \end{array}\right] x(t) + \begin{array}{r} 1.000 \end{array} u(t)$$

See Also `tf.tex`, `textable`, `texmatrix`

Purpose

Computes the zeros, poles and gains of an SS object.

Syntax

`[z,p,k,zMC,pMC,kMC]=zpk(s)`

Description

The following list describes the output arguments:

OUTPUT ARGUMENT	DESCRIPTION
p	A row vector with poles
z	A (ny x nb x nu) matrix with zeros
k	A (ny x nu) matrix with gains
`zMC,pMC,kMC`	The corresponding Monte Carlo arrays, where the first index corresponds to the MC samples

Example

The zeros and poles of a SISO system:

```
m1=rand(ss([3 1 0 1]))
              /  -1.6  1  0 \        / 0.66 \
  d/dt x(t) = | -0.96  0  1 | x(t) + | 0.69 | u(t)
              \ -0.19  0  0 /        \ 0.19 /

y(t) = (1  0  0) x(t) + (1) u(t)


[z,p,k]=zpk(m1)
z =
    -1.0078    -0.7226    -0.5340
p =
   -0.5972 +   0.3341i   -0.5972 -   0.3341i   -0.4147 +
0i
k =
  1
```

The zeros and poles of a SISO system:

```
m2=rand(ss([2 2 0 2]))
              / -2  1 \       /  -1.8  0.11 \
  d/dt x(t) = \ -1  0 / x(t) + \ -0.97  0.28 / u(t)

          /   1     0\          /   1     1\
  y(t) =  \0.67  0.66/  x(t) +  \0.83  0.93/  u(t)


[z,p,k]=zpk(m2)
z(:,:,1) =
    -0.1109 +    0.2443i    -0.1109 -    0.2443i
     0.1079 +    0.4193i     0.1079 -    0.4193i
z(:,:,2) =
    -1.0712 +    0.4189i    -1.0712 -    0.4189i
```

```
   -1.1563 +    0.4836i    -1.1563 -    0.4836i
p =
   -1.0161 +    0.1042i    -1.0161 -    0.1042i
k =
         1          1
    0.8258     0.9252
```

See Also `ss`, `tf`, `tf.zpk`

Purpose Convert state-space model to a transfer function.

Syntax

```
[b,a]=ss2tf(A,B,C,D)
```

Description The state-space model characterized by the `A`, `B`, `C`, and `D` matrices is transformed into a transfer function represented on polynomial `b,a` form.

This function is limited to SISO systems. Use the SS method `ss.ss2tf` for MIMO systems.

Example Compute the transfer function of a discrete-time double summator:

```
A=[1 1;0 1]; B=[0;1]; C=[1 0]; D=0;
[b,a]=ss2tf(A,B,C,D)
b =
  0  0  1
a =
   1  -2   1
```

See Also `tf.tf2ss`, `ss.ss2tf`, `tf2ss`

Purpose Simulate a step response.

Syntax `y=step(G,T)`

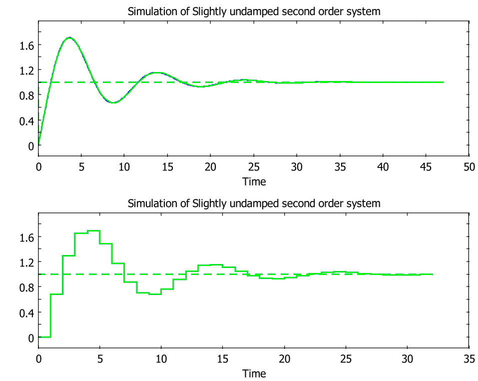
Description Available input arguments:

ARGUMENT	DESCRIPTION
G	SS object
T	Simulation time. By default, it is estimated from the dominating pole
y	SIG object

The step response is computed by formulas from system theory rather than simulated using numerical integration routines. You achieve the latter simulation with a call like `y=simulate(G,getsignal('cstep'))`.

Example Simulate a step response of a second-order lightly damped system:

```
Gc=ss(exlti('tf2c'))
              / -0.3  -0.41 \        / 1 \
  d/dt x(t) = \    1      0 / x(t) + \ 0 / u(t)

y(t) = (0.62  0.41) x(t) + (0) u(t)


yc=step(Gc);
ycsim=simulate(Gc,getsignal('cstep',yc.t(end)))
SIG object with continuous time input-output state space data
  Name:       Simulation of Slightly undamped second order system
  Sizes:       N = 201,  ny = 1, nu = 1, nx = 2
subplot(2,1,1)
plot(yc,ycsim)
Gd=ss(exlti('tf2d'))
           / 1.4  -0.74 \        / 1 \
  x[k+1] = \   1      0 / x[k] + \ 0 / u[k]

y[k] = (0.68  -0.34) x[k] + (0) u[k]


yd=step(Gd);
ydsim=simulate(Gd,getsignal('step',length(yd.y)))
SIG object with discrete time (fs = 1) input-output state space
data
  Name:       Simulation of Slightly undamped second order system
  Sizes:       N = 33,  ny = 1, nu = 1, nx = 2
subplot(2,1,2)
staircase(yd,ydsim)
```



See Also `tf.step`

Purpose The Student t distribution.

Syntax `X=tdist(n)`

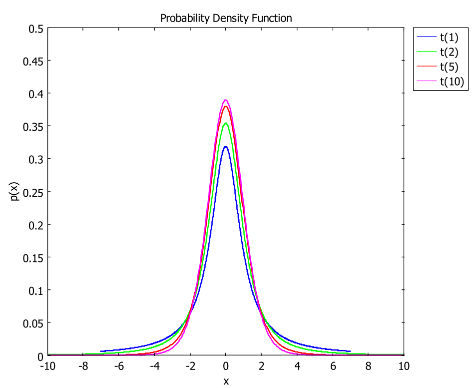
Description The probability density function of the t distribution, and its first two moments, are given by

$$p(x;n) = \frac{\Gamma((n+1)/2)}{\sqrt{n\pi}\Gamma(n/2)}(1+x^2/n)^{-(n+1)/2},$$
$$E(X) = 0,$$
$$\mathrm{Var}(X) = \frac{n}{n-2}, \quad n > 2.$$

`n` must be a positive integer. This is a child of `pdfclass`, and all of its methods apply to this distribution, in particular `pdfclass.estimate` and the plot functions.

Example Illustration of some sample distributions:

```
n=[1 2 5 10];
for i=1:4; X{i}=tdist(n(i)); end
plot(X{:})
axis([-10 10 0 0.5])
```



See Also `pdfclass`

Purpose

Create LaTeX code for a matrix.

Syntax

```
texcode=texmatrix(A,Property1,Value1,....)
```

Description

The output is TeX code that you can paste into any LaTeX document. Alternatively, the code is put into a file, which you then can input by reference into a document.

Filters (freeware or shareware) for other word processors are available:

- TexPoint: freeware for Microsoft Powerpoint
- LaImport: FrameMaker
- TeX2Word: Microsoft Word
- LaTeX2rtf: RTF documents
- TeX4ht: HTML or XML hypertext documents

The properties are:

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`filename`	`{''}`	Name of the .tex file (none for `''`)
`decimals`	`{1}`	Number of decimals
`env`	`{'eqnarray*'}`	TeX environment, `''` means no env

Example

Generate TeX code for a random matrix with three decimals:

```
A=randn(4);
texmatrix(A,'decimals',2)
ans =
  \begin{eqnarray*}
  \left[
  \begin{array}{rrrr}
  -0.43 & -0.67 & 0.97 & -1.37 \\
  -0.67 & 0.47 & 0.18 & 1.84 \\
  -0.35 & -0.63 & 1.28 & 0.52 \\
  -0.51 & 0.48 & 0.73 & -0.29
  \end{array}
  \right]
  \end{eqnarray*}
```

The output when you imported this code into a FrameMaker document is:

$$\left[\begin{array}{rrrr} -0.43 & -0.67 & 0.97 & -1.37 \\ -0.67 & 0.47 & 0.18 & 1.84 \\ -0.35 & -0.63 & 1.28 & 0.52 \\ -0.51 & 0.48 & 0.73 & -0.29 \end{array}\right]$$

See Also `textable`

Purpose Create LaTeX code for a matrix in tabular form.

Syntax

```
texcode=textable(A,Property1,Value1,....)
```

Description The output is TeX code that you can paste into any LaTeX document. Alternatively, the code is put into a file, which you can input by reference into a document.

Filters (freeware or shareware) for other word processors are available:

- TexPoint: freeware for Microsoft Powerpoint
- LaImport: FrameMaker
- TeX2Word: Microsoft Word
- LaTeX2rtf: RTF documents
- TeX4ht: HTML or XML hypertext documents

The properties are:

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`filename`	`{''}`	Name of the .tex file (none for `''`)
`decimals`	`{1}`	Number of decimals
`xlabel`	`{''}`	Array with strings of column labels
`ylabel`	`{''}`	Array with strings of row labels
`title`	`{''}`	String with table title

Example Generate a table for the chi-square distribution:

```
d=1:6;
p=0:0.1:1;
for m=1:length(d)
   xlabel{m}=['d=',num2str(d(m))];
   for n=1:length(p)
      ylabel{n}=['p=',num2str(p(n))];
      A(n,m)=erfinv(chi2dist(d(m)),p(n));
   end
end
textable(A,'ylabel',ylabel,'xlabel',xlabel,'title','h for
P(chi2(d)<h)=p','filename',chi2table')
```

It looks as follows in FrameMaker:

h for P(chi2(d)<h)=p

	d=1	d=2	d=3	d=4	d=5	d=6
p=0.1	0.0	0.2	0.6	1.1	1.6	2.2
p=0.2	0.1	0.4	1.0	1.6	2.3	3.1
p=0.3	0.2	0.7	1.4	2.2	3.0	3.8
p=0.4	0.3	1.0	1.9	2.7	3.6	4.6
p=0.5	0.5	1.4	2.3	3.3	4.3	5.3
p=0.6	0.7	1.8	2.9	4.0	5.1	6.2
p=0.7	1.1	2.4	3.6	4.9	6.0	7.2
p=0.8	1.7	3.2	4.6	5.9	7.3	8.5
p=0.9	2.7	4.5	6.2	7.7	9.2	10.6
p=1	7.0	10.5	13.4	16.0	18.4	20.7

See Also `texmatrix`

Purpose

The transfer function (TF) model object.

Syntax

Different ways to construct a TF object:

`TF([na nb nk])`	Empty SISO structure (fs=0 by convention)
`TF([na nb nk nu ny])`	Empty MIMO structure (fs=0 by convention)
`TF([na nb nk],fs)`	Empty structure with sampling frequency
`TF(b,a)`	Polynomial definition
`TF(b,a,fs)`	Stochastic input-output model

The following special constructors are also available:

`TF('unit')`	Defines the unit system y=u
`TF('delay')`	Defines the unit delay system y(t)=u(t-1)
`TF('sum')`	Defines the summator (integrator approx.) y(t)=y(t-1)+u(t)
`TF('int')`	Defines the integrator G(s)=1/s
`TF('s')`	Laplace operator
`TF('q',fs)`	Time shift operator
`TF('z',fs)`	z transform operator (same as q)

Description

The entered numerator `b` and denominator `a` polynomials can be arbitrary vectors of arbitrary lengths. However, it is good practice to fill up with zeros to equal length. TF otherwise fills up with zeros *from the left*. This corresponds to polynomials in descending powers of s and z, respectively.

The following conventions apply:

- The sampling frequency is given in Hertz for discrete-time systems, and is by convention `NaN` for continuous-time systems (default if `fs` is omitted).
- Coefficient $a(1)$ preceding $y(t)$ is always one.
- $b(1)$ is nonzero.
- If `b` and `a` are specified of different length, the shorter one is extended with zeros from the right. Use `b,a` of equal size to avoid problems!
- `nk` is the relative degree, so `nk>=0` for causal systems, and `nk<0` for noncausal systems.

That is, the stored difference equation (or differential equation for continuous time) is

$$y(t) + a(1)y(t-1) + \ldots + a(na)y(t-na) = \\ b(1)u(t-nk) + b(2)u(t-nk-1) + \ldots + b(nb)u(t-nk-nb+1)$$

For MIMO systems, the SISO transfer function from u_i to y_j is given by `b(j,:,i)` and `a`. Note that `a` and `nk` are the same for all inputs and outputs, in order to be consistent with (unstructured) state-space models.

Overloaded functions (methods) include in short:

- A display function (`display`) used whenever you request a workspace printout.
- A plot function (`ltiplot`) invoked when you type `plot(G)`, `bode(G)`, and other plot functions.
- Simulation functions (`simulate, impulse, step`) to produce a `SIG` object.
- An estimation function (`estimate`) to produce a model from a `SIG` object.
- A filter function (`kalman`) invoked when you type `filter(G,y)` for estimating and predicting system state.
- Operators for basic model operations such as `+`, `-`, `*`, `/`, and `feedback`.

The following section provides a more detailed presentation of the methods.

Overloaded operators:

`plus`	+G	Unitary plus
`uminus`	-G	Unitary minus
`plus`	G1+G2	Parallel connection with summation at output
`minus`	G1-G2	Parallel connection with difference at output
`power`	G.^n	Repeated multiplication
`mpower`	G^n	Repeated multiplication
`inv`	inv(G)	Inverse of square systems
`eq`	G1==G2	Test for equality
`mrdivide`	1/G	Right inverse of a system
`mldivide`	G\1	Left inverse of a system
`mtimes`	G1*G2	Series connection of two models
`feedback`	feedback(G1,G2)	Feedback connection of two systems
`diag`	diag(G1,G2,...)	Append independent models
`append`	append(G1,G2)	Append independent models
`ctranspose`	G.'	Reverse the inputs with outputs

`transpose`	G'	Reverse the inputs with outputs
`horzcat`	[G1 G2]	Horizontal concatenation
`vertcat`	[G1;G2]	Vertical concatenation
`arrayread`	G(i,j)	Pick out subsystems by indexing

These operators do not have a separate reference page.Certain methods require an input to be feasible (such as `plus`, `mtimes`, and `feedback`). For stochastic models, inversion as done in `inv`, for instance, is impossible to define.

Overloaded model conversions (SS to SS):

`tf.c2d`	Convert a continuous-time TF to a discrete-time TF
`tf.d2c`	Convert a discrete-time TF to a continuous-time TF
`tf.minreal`	Compute a minimal realization by removing common zeros and poles

Methods to transform to other object types:

`tf.tf2freq`	Compute frequency-domain response G(f)
`tf.zpk`	Compute the zeros, poles, and gain
`tf.tf2ss`	Compute the state-space model
`tf.impulse`	Compute the analytic impulse response
`tf.step`	Compute the analytic step response
`tf.simulate`	Simulate a signal y=G(u) from an LTI state-space model

Filter functions are also overloaded as methods to the TF object:

`tf.filter`	Standard causal filtering
`tf.filtfilt`	Noncausal and zero-phase forward-backward filtering
`tf.ncfilter`	Noncausal stable filtering of arbitrary transfer function

In contrast to the corresponding methods, these apply to MIMO transfer functions.

You can view an LTI object of all kinds in different ways using the following plot methods of the LTI object and all inherited model objects:

- Bode diagram of amplitude and phase as a function of frequency f of $G(f)$. There are options that you can use to plot only the amplitude, only the phase, or both.
- Nyquist curve, where the plot shows $G(f)$ as a complex function.

- Pole-zero plots, which show the poles and zeros of $G(s)$ in the complex plane.
- Root locus for $G(s)$, which is a plot of the poles of the closed-loop system, using a constant feedback K, as a function of K. For SISO TF objects, the poles are the roots of the equation $A(s) + KB(s) = 0$. For MIMO state-space models, the closed loop poles are determined by the eigenvalues to the matrix $A - B(1/kI + D)^{-1}C$.

`lti.bode(G1,G2,...)`	Bode plot of amplitude and phase curves
`lti.bodeamp(G1,G2,...)`	Bode diagram of phase curve
`lti.bodephase(G1,G2,...)`	Bode diagram of phase curve
`lti.nyquist(G1,G2,...)`	Nyquist curve
`lti.zpplot(G1,G2,...)`	Zero-pole plot
`lti.rlplot(G1,G2,...)`	Root locus plot

You can use the same notation for continuous and discrete-time systems—just replace s with q above. For MIMO systems, the plots appear in a subplot array with `ny` rows and `nu` columns. It is possible to set the most common properties of standard plots such as `Xlim`, `Ylim`, `fontsize`, `linewidth`, and `axis`. Also, you can specify the color (or color order for multiple LTI object inputs) using the property `col`, which is a vector with one letter color abbreviations such as `b` for blue, `k` for black, and `r` for red.

Example

Create some continuous and discrete transfer functions:

```
Gc=tf([1 0 0],[1 1 1])

            s^2
   Y(s) = ------- U(s)
          s^2+s+1

Gc=tf(1,[1 1 1])        % Note convention!

             1
   Y(s) = ------- U(s)
          s^2+s+1

Gd=tf(1,[1 1 1],2)

             1
   Y(z) = ------- U(z)
          z^2+z+1
```

See Also ss

Purpose Convert continuous-time TF to discrete-time TF.

Syntax `Gd=c2d(Gc,fs,method)`

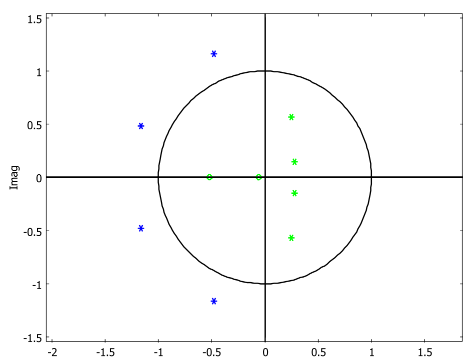
Description Sampling is characterized by the sampling frequency and assumption on intersample behavior.

INPUT PARAMETER	VALUE/ DEFAULT	DESCRIPTION
`fs`	`{1}`	Sampling frequency
`method`	`'bilinear'`	String describing the assumption on intersample behavior

The function only supports bilinear interpolation is supported. For more options, use the SS method `ss.c2d`.

Example Discretize a fourth-order Butterworth filter:

```
Gc=getfilter(4,0.2,'fs',NaN);
Gd=c2d(Gc,1)

         0.052*z^3+0.28*z^2+0.15*z+0.0073
   Y(z) = -------------------------------- U(z)
          z^4-1*z^3+0.75*z^2-0.26*z+0.037

zpplot(Gc,Gd)
```



See Also `ss.c2d`, `tf.d2c`

Purpose

Convert discrete-time TF to continuous-time TF.

Syntax

`Gc=d2c(Gd,method)`

Description

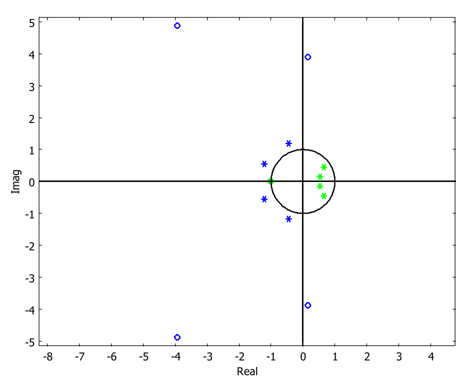
Sampling is characterized by the sampling frequency and assumption on intersample behavior. The function only supports bilinear interpolation. For more options, use the SS method `ss.d2c`.

INPUT PARAMETER	VALUE/ DEFAULT	DESCRIPTION
`fs`	`{1}`	Sampling frequency
`method`	`'bilinear'`	String describing the assumption on intersample behavior

Example

Convert a discretized fourth-order Butterworth filter to an equivalent continuous-time form:

```
Gd=getfilter(4,0.2)

          0.0048*z^4+0.019*z^3+0.029*z^2+0.019*z+0.0048
   Y(z) = --------------------------------------------- U(z)
                 z^4-2.4*z^3+2.3*z^2-1.1*z+0.19

Gc=d2c(Gd)

          0.0048*s^4+0.036*s^3+0.25*s^2+0.51*s+2.9
   Y(s) = ---------------------------------------- U(s)
               s^4+3.3*s^3+5.6*s^2+5.5*s+2.9

zpplot(Gc,Gd)
```



See Also

`ss.d2c`, `tf.c2d`

Purpose

Estimate a transfer-function model from data in a SIG object.

Syntax

`Gout=estimate(Gin,z,Property1,Value1,...)`

Description

A TF model with structure as specified in `Gin` is estimated from the signal `z` using a two-step least squares (LS) algorithm:

1. Estimate a high-order FIR model using LS and a user-provided set of input-output data as a SIG object. This step calls `arx.estimate`.
2. In the second step, simulate the high-order FIR model without noise using the same input as available in the data.
3. Then estimate the low-order ARX model using LS, using `arx.estimate` again, this time with the sought number of poles and zeros.
4. Finally, convert the ARX model into the corresponding TF model with the same `b` and `a` polynomials.
5. For uncertainty representation, one of the following schemes are applied:
   - **e** If Monte Carlo realizations of the output signal are provided in the SIG object, repeat the preceding four steps for each of them, and create the cell array `sysMC`.
   - **f** Otherwise, the algorithm assumes that
   - The true system is contained in the high-order FIR model
   - The estimate can be considered Gaussian distributed

   The latter is true for Gaussian noise and asymptotically otherwise. Monte Carlo simulations are used to convert the Gaussian high-order estimate to a sample-based representation of the non-Gaussian distribution of the low-order ARX estimate. More precisely, the uncertainty is then represented by taking random parameter vectors from the high-order FIR model and repeating steps 2 to 4 above to obtain the cell array `sysMC`.

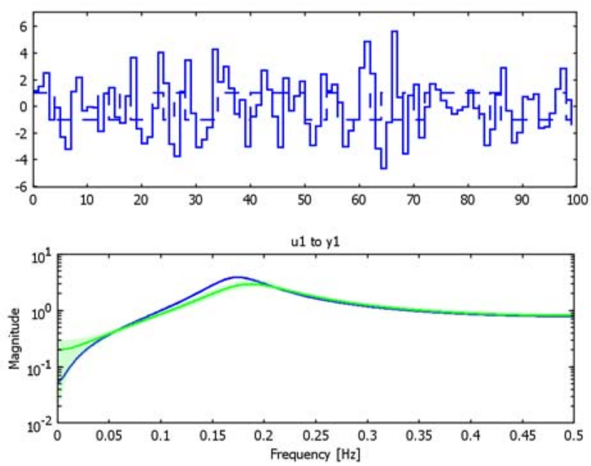
PROPERTY	VALUE/DEFAULT	DESCRIPTION
`MC`	`{30}`	Number of Monte Carlo simulations
`nfir`	`min([10*na,50])`	FIR order in the first step

Example

Generate a random TF model, simulate a PRBS signal, add noise, and identify the TF model of the same structure from the noisy data:

```
N=100;
Gstruc=tf([2 2 1 1 1]);
Gstruc.fs=1;
```

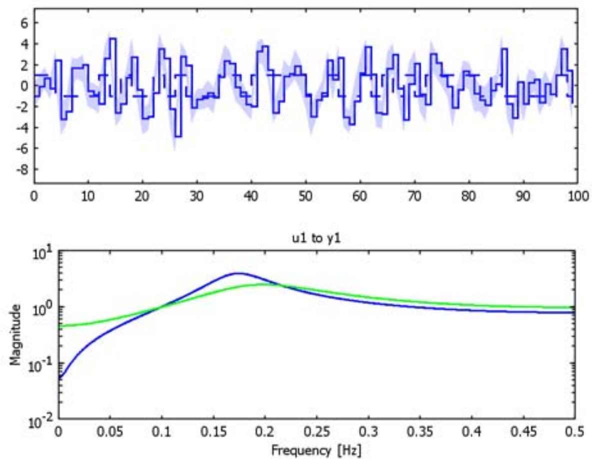
```
G0=rand(Gstruc)

                z-0.95
   Y(z) = -------------- U(z)
          z^2-0.79*z+0.7

u=getsignal('prbs',N);
y=simulate(G0,u);
yn=y+1*randn(N,1);
Ghat=estimate(Gstruc,yn)

               1*z-0.82
   Y(z) = --------------- U(z)
          z^2-0.65*z+0.62

subplot(2,1,1), plot(yn)
subplot(2,1,2), bodeamp(G0,Ghat)
```



Same thing, this time with Monte Carlo realizations of the signal.

```
y.MC=30;
yn=y+1*ndist(0,1);
Ghat=estimate(Gstruc,yn)

              1.2*z-0.75
   Y(z) = --------------- U(z)
          z^2-0.52*z+0.49

subplot(2,1,1), plot(yn,'conf',90)
```

```
subplot(2,1,2), bodeamp(G0,Ghat)
```



See Also `arx`, `arx.estimate`, `tf`

Purpose Filtering operation as a TF method.

Syntax `y=filter(G,u)`

Description The low-level `filter` function is used internally.

Basically, this filtering operation does the following steps:

```
ytmp=filter(G.b,G.a,u.y);
y=sig(ytmp,u.fs);
```

Note that the sampling frequency of the input SIG object has precedence to the one specified in `G`. `G.fs` is used only if `u.fs=NaN`.

The `filter` method, unlike the `filter` function, works for MIMO TF and SIG objects.

Monte-Carlo filtering is performed according to the following precedence rules:

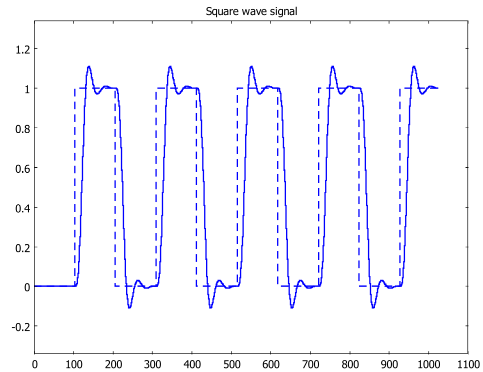
1. If the signal contains MC data (`u.MC>0`), then `y` gets the same number of Monte Carlo samples, each one corresponding to a filtering to one input realization.
2. Otherwise, if the TF object is uncertain, then the input `u` is filtered through `G.MC` Monte Carlo realizations of `G`.

Example The following examples show how to use `filter` for different types of systems:

- SISO filtering
- SISO filtering with MC data
- MIMO filtering
- MIMO filtering with MC data

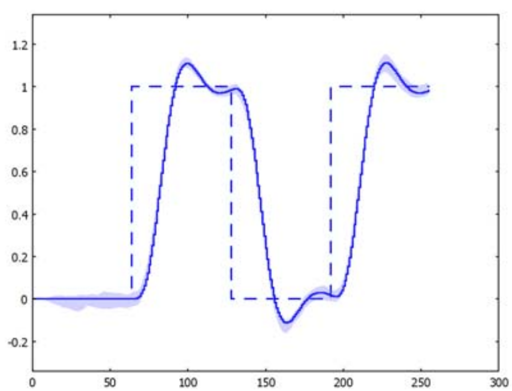
First, low-pass filter a square wave:

```
G=getfilter(4,0.05);
u=getsignal('square');
y=filter(G,u);
staircase(y)
```



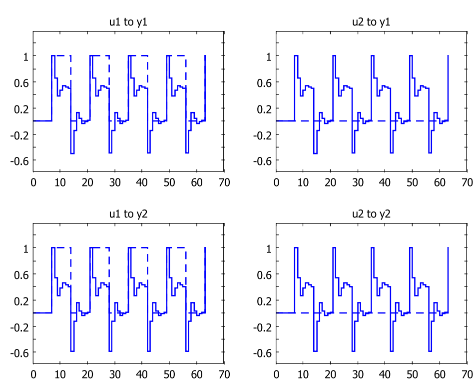
Now, add 20 different noise realizations to the square wave and repeat the filtering. Monte Carlo data then becomes available in the output SIG object, and you can illustrate them with confidence band (as the following example shows) or scatter plots.

```
MC=20;
u=getsignal('square',256,128);
fs=u.fs;
u=u.y;
uMC=repmat(u',MC,1)+0.1*randn(MC,length(u));
u=sig(u,fs,[],[],uMC);
y=filter(G,u);
staircase(y,'conf',90)
```



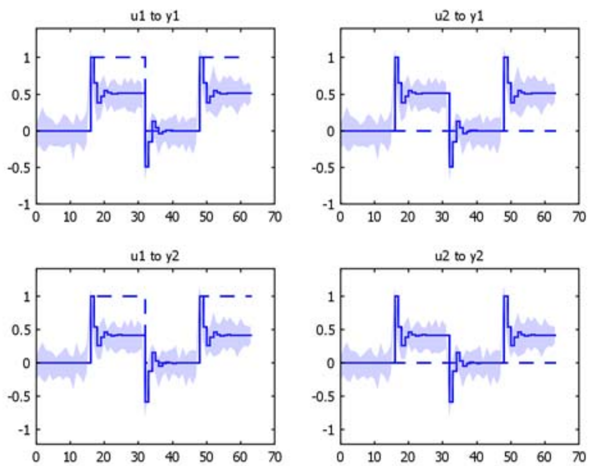
Generate a random MIMO system and a 2D input signal by extending a square wave with a zero signal:

```
Gstruc=tf([2 2 0 2 2]);
Gstruc.fs=1;
G2=rand(Gstruc)

               z(z-0.36)
   Y1(z) = ---------------- U1(z)
            z^2-0.018*z+0.27

               z(z-0.36)
   Y1(z) = ---------------- U2(z)
            z^2-0.018*z+0.27

               z(z-0.48)
   Y2(z) = ---------------- U1(z)
            z^2-0.018*z+0.27

               z(z-0.58)
   Y2(z) = ---------------- U2(z)
            z^2-0.018*z+0.27

u=getsignal('square',64);
u2=[u sig(zeros(size(u),1))]
SIG object with discrete time (fs = 1) time series
  Name:         Square wave signal
  Description: Example getsignal('square',64,13)
  Sizes:        N = 64,  ny = 2
  MC is set to: 30
  #MC samples:  0
y2=filter(G2,u2);
staircase(y2)
```



Again, you can add noise realizations to the signal, and `filter` then generates Monte Carlo data.

```
u=getsignal('square',64,32);
u=u.y;
u2=[u zeros(size(u))];
utmp(1,:,:)=u2;
u2MC=repmat(utmp,MC,1,1)+0.1*randn(MC,64,2);
u2n=sig(u2,1,[],[],u2MC);
y2=filter(G2,u2n);
staircase(y2,'conf',90)
```



See Also `filtfilt`, `tf.filtfilt`, `tf.ncfilter`

Purpose Noncausal implementation of a filter as a TF method.

Syntax `y=filtfilt(G,u)`

Description The low-level `filtfilt` function is used internally. The `filfilt` method, unlike the `filtfilt` function, works for MIMO TF and SIG objects.

Basically, the following is performed:

```
x=filter(b,a,u);
xr=x(end:-1:1);
yr=filter(b,a,xr);
y=yr(end:-1:1);
```

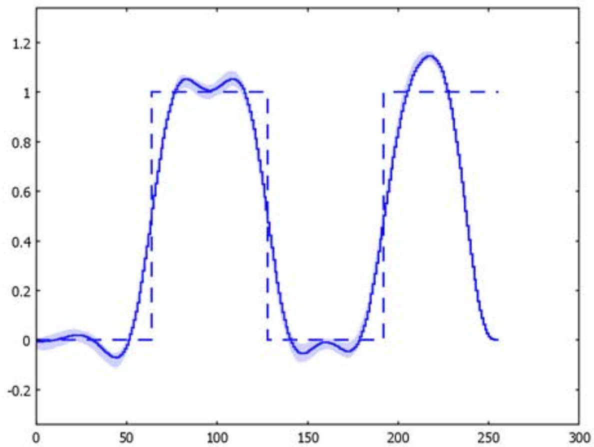
Note that the sampling frequency of the input SIG object has precedence to the one specified in `G`.

Monte-Carlo filtering is performed according to the following precedence rules:

1. If the signal contains MC data (`u.MC>0`), then `y` gets the same number of Monte Carlo samples, each one corresponding to a filtering to one input realization.
2. Otherwise, if the TF object is uncertain, then the input `u` is filtered through `G.MC` Monte Carlo realizations of `G`.

Example Low-pass filtering of a square wave:

```
G=getfilter(4,0.05);
MC=20;
u=getsignal('square',256,128);
fs=u.fs;
u=u.y;
uMC=repmat(u',MC,1)+0.1*randn(MC,length(u));
u=sig(u,fs,[],[],uMC);
y=filtfilt(G,u);
staircase(y,'conf',90)
```



See `tf.filter` for more examples.

See Also `filtfilt`, `tf.filter`, `tf.ncfilter`

Purpose Generate the impulse/pulse response of a TF object.

Syntax `y=impulse(G,T)`

Description The arguments are as follows:

ARGUMENT	DESCRIPTION
G	TF object
T	Simulation length (number of samples N or time).
	Default it is estimated from dominating pole using T=timeconstant(G)
y	Output SIG object

For SISO transfer functions, the function basically performs the following:

```
u=getsignal('impulse',T);
y=simulate(G,u);
```

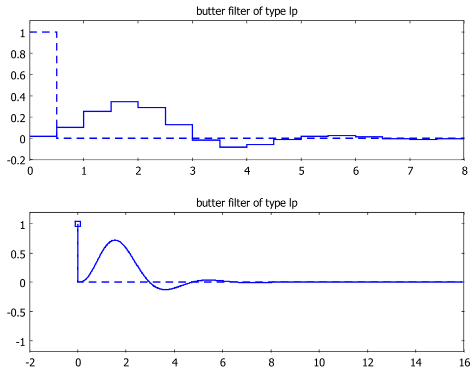
For the MIMO case, the input is repeated `nu` times. If the impulse response for a particular input-output channel is desired, do `impulse(G(yind,uind),T);`

For discrete-time TF objects, the pulse function is used instead.

The function adds the phrase "Impulse response of" to the name field of `G`, if nonempty.

Example Compute the impulse response of a discrete-time and continuous-time Butterworth filter, respectively:

```
Gd=getfilter(4,0.3,'fs',2);
yd=impulse(Gd);
Gc=getfilter(4,0.3,'fs',NaN);
yc=impulse(Gc);
subplot(2,1,1), staircase(yd)
subplot(2,1,2), plot(yc)
```



See Also `tf`, `tf.step`, `tf.simulate`, `getsignal`

Purpose Cancel common zeros and poles in a TF object

Syntax `G=minreal(G)`

Description The function systematically searches for common roots in all numerator polynomials `b(j,:,i)` and denominator polynomial a and cancel these. Many operations on TF objects lead to transfer functions with many zeros and poles in common, which should be cancelled out. This is not done automatically, however.

The function works for MIMO, but in contrast to `ss.minreal` it does not cancel input and output dimensions that do not contribute to the input-output dynamics.

Example Feedback is a typical operation which leads to an overparameterized transfer function.

```
s=tf('s');
G=1/(s^2+s+2);
Gc=feedback(G, 1)

                  s^2+s+2
   Y(s) = --------------------- U(s)
          s^4+2*s^3+6*s^2+5*s+6

minreal(Gc)

              1
   Y(s) = --------- U(s)
          s^2+1*s+3
```

See Also `ss.minreal`,`tf.zpk`

Purpose Stable noncausal filtering operation as a TF method

Syntax `y=ncfilter(G,u)`

Description Discrete-time transfer function and signal are presumed. The low-level filter function `Illustrate a RARX object graphically in the frequency domain using surf.` is used internally. The filter method, unlike f`Illustrate a RARX object graphically in the frequency domain using surf.`, works for MIMO TF and SIG objects.

Basically, the following is performed:

```
ytmp=ncfilter(G.b,G.a,u.y);
y=sig(ytmp,u.fs);
```

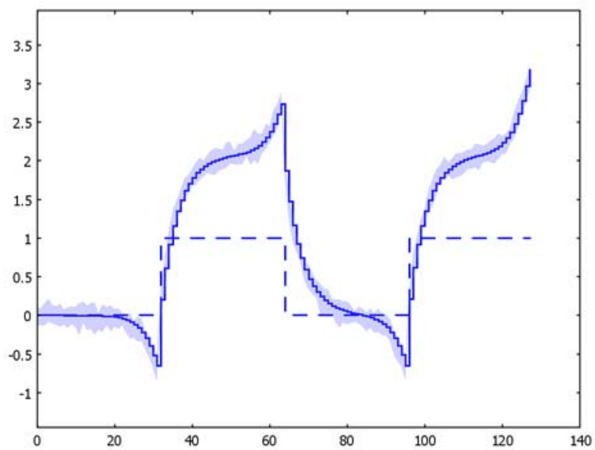
Note that the sampling frequency of the input SIG object has precedence to the one specified in `G`. `G.fs` is used only if `u.fs=NaN`.

Monte-Carlo filtering is performed according to the following precedence rules:

1 If the signal contains MC data (`u.MC>0`), then `y` gets the same number of Monte Carlo samples, each one corresponding to a filtering to one input realization.
2 Otherwise, if the TF object is uncertain, then the input `u` is filtered through `G.MC` Monte Carlo realizations of `G`.

Example Create a filter object with poles and zeros both inside and outside the unit circle, and filter a square wave:

```
G=tf(poly([0.2 0.5 1.2 2]),poly([0.4 0.8 1.4 1.8]),1);
MC=20;
u=getsignal('square',128,64);
fs=u.fs;
u=u.y;
uMC=repmat(u',MC,1)+0.1*randn(MC,length(u));
u=sig(u,fs,[],[],uMC);
y=ncfilter(G,u);
staircase(y,'conf',90)
```



See Also `filtfilt`, `tf.filter`, `tf.filtfilt`

Purpose Simulate a TF object using a SIG object as input.

Syntax `[z,xf]=simulate(G,u,Property1,Value1,...)`

Description `G` is the TF object, `z` the returned simulated SIG object, and `u` is the input-signal object with presumingly the same sampling interval as the model (for instance, use `u=sig(uvec,fs))`. If there are inconsistent sampling intervals, the one in the SIG object has precedence.

`xf` is the final state, and you can forward it as an argument `'xi'` for simulation over segments.

The simulation algorithm basically works as follows:

1. For discrete-time systems, the low-level filter functions is applied to each input-output MIMO channel.
2. For continuous-time systems, the simulation is based on fast sampling. The algorithm computes the time constant `Tc` of the system using the `timeconstant` function, which is based on the dominating stable pole. The continuous-time system is then resampled 200 times faster using `c2d(G,200/Tc)`. For signals with discontinuities, these are first located. These can be either steps or impulses (see the SIG object constructor for information about how these work). The input is then segmented between the boundaries defined by the discontinuities, and a separate sampling and simulation is done in each segment, where the filter state is saved and used in the next segment.

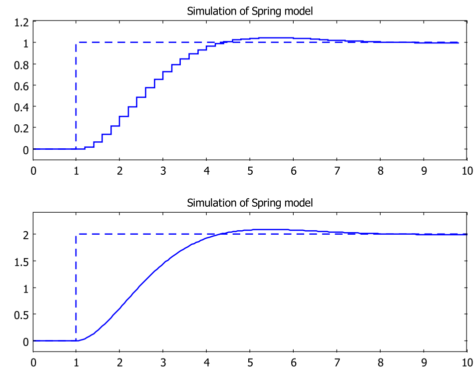
For nonuniformly sampled inputs, only continuous models apply.

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`MC`	Default value inherited from model or signal	Number of Monte Carlo simulations
`xi`	`zeros(nx,1)`	Initial state as a nx=max([na nb-1]) vector

Example Simulate a spring model in continuous time and discrete time, respectively.

```
Gc=exlti('tf3c');
fs=5;
Gd=c2d(Gc,fs);
ud=[getsignal('zeros',fs);getsignal('ones',9*fs)];
ud.fs=fs;
yd=simulate(Gd,ud);
umat=[0 0 2 2]';
t=[0 1 1 10];
```

```
u=sig(umat,t);
y=simulate(Gc,u);
subplot(2,1,1), staircase(yd)
subplot(2,1,2), plot(y)
```



See Also `tf.filter`, `ss.simulate`

Purpose Generates the step response of a TF object.

Syntax `y=step(G,T)`

Description The arguments are as follows:

ARGUMENT	DESCRIPTION
G	TF object
T	Simulation length (number of samples N or time).
	Default it is estimated from dominating pole using T=timeconstant(G)
y	Output SIG object

For SISO transfer functions, the function basically performs the following:

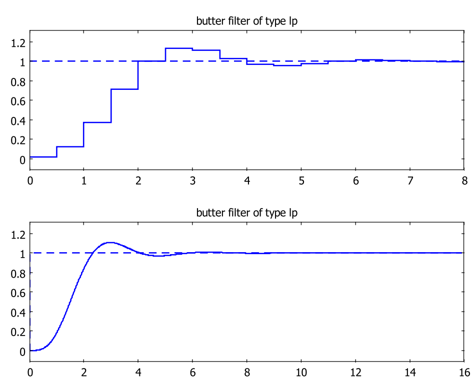
```
u=getsignal('step',T);
y=simulate(G,u);
```

For the MIMO case, the input is repeated `nu` times. If you want the step response for a particular input-output channel, do `step(G(yind,uind),T);`

The function adds the phrase "Step response of" to the name field of `G`, if nonempty.

Example Compute the step response of a discrete-time and continuous-time Butterworth filter, respectively.

```
Gd=getfilter(4,0.3,'fs',2);
yd=step(Gd);
Gc=getfilter(4,0.3,'fs',NaN);
yc=step(Gc);
subplot(2,1,1), staircase(yd)
subplot(2,1,2), plot(yc)
```



See Also `tf`, `tf.impulse`, `tf.simulate`, `getsignal`

Purpose Create LaTeX code from transfer function TF object.

Syntax

```
texcode=tex(s,Property1,Value1,...)
```

Description The output is TeX code that you can paste into any LaTeX document. Alternatively, the code is put into a file, which you can input by reference into a document.

Filters (freeware or shareware) for other word processors are available:

- TexPoint: freeware for Microsoft Powerpoint
- LaImport: FrameMaker
- TeX2Word: Microsoft Word
- LaTeX2rtf: RTF documents
- TeX4ht: HTML or XML hypertext documents

The properties are:

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`filename`	`{''}`	Name of the .tex file (none for '')
`format`	`{'%11.2g'}`	Numeric format, see sprintf
`env`	`{'eqnarray*'}`	Tex environment, '' means no env

Example Generate a random model and its LaTeX code:

```
m=rand(tf([2 1 1]))

                 s
   Y(s) = --------------- U(s)
          s^2+0.63*s+0.17

tex(m)
ans =
  \begin{eqnarray*}
   Y(z) &=& \frac{z^1 (1)}{z^2+0.63\cdot z+0.17}  U(z)
  \end{eqnarray*}
```

Importing the LaTeX code to a FrameMaker file produces the following printout:

$$Y(z) \quad = \frac{z^1(1)}{z^2 + 0.63 \cdot z + 0.17} U(z)$$

See Also `ss.tex`, `texmatrix`, `textable`

Purpose

Compute frequency-domain response of a TF object.

Syntax

```
Hf=tf2freq(G,Property1,Value1,...)   Explicit call
Hf=freq(G,Property1,Value1,...)      Implicit call
```

Description

The frequency function $H(f)$ is the Fourier transform of the input-output transfer function TF object. This conversion supports MIMO systems, and evaluates $H(i\omega) = b(i2\pi f)/a(i2\pi f)$ for continuous-time systems and $H(e^{i\omega}) = b(ei^{2\pi f})/a(e^{i2\pi f})$ for discrete-time systems, respectively, for each input-output channel.

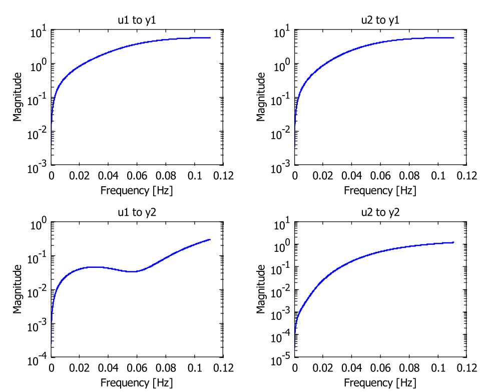
Properties:

PROPERTY	VALUE	DESCRIPTION
MC	{30}	Number of Monte Carlo simulations used in lti2cov
N	{1024}	Number of frequency grid points
f	{}	Frequency grid (overrides N)
fmax		Maximum frequency. Default is fs/2 for discrete-time systems, and 8 times the dominating poles bandwidth for continuous-time systems

Example

Create a random stochastic TF model and compute its frequency function:

```
G=rand(tf([6 6 0 2 2]));
Gf=tf2freq(G); % Explicit call
Gf=freq(G);    % Implicit call
bodeamp(Gf)
```



See Also

`freq`, `ss.ss2freq`

Purpose Convert a transfer-function (TF) object to a state-space (SS) object.

Syntax

`sys=tf2ss(G,form)`	Explicit call
`sys=ss(G,form)`	Implicit call

Description `G` is the TF object, `sys` is the returned SS object, and `form` is described below:

FORM	DESCRIPTION
`'observer' or 'o'`	For observability form
`'controller' or 'c'`	For controllability form

Unlike the `tf2ss` function, this method works for MIMO systems. The observability form works straightforwardly for SIMO systems, and the controllability form works for MISO systems. For MIMO systems, there is no simple standard form. Here, a simple append is used, which leads to a nonminimal realization. For instance, for the observability form, one SIMO realization is found for each output, and these are then appended. `ss.minreal` can be used to decrease the model order, but then the structure is lost.

Example Transform a SISO system, a MISO system, and a MIMO system, respectively:

```
G1=rand(tf([2 2 0 1 1]))

               s(s+0.99)
   Y(s) = --------------- U(s)
            s^2+0.63*s+0.17

m1=tf2ss(G1,'o')
              / -0.63  1 \        /  0.36 \
  d/dt x(t) = \ -0.17  0 / x(t) + \ -0.17 / u(t)

y(t) = (1  0) x(t) + (1) u(t)


G2=rand(tf([2 2 0 2 1]));
m2=tf2ss(G2,'o')
              /  -1.3  1 \        / -0.73  -0.25 \
  d/dt x(t) = \ -0.52  0 / x(t) + \ -0.52  -0.52 / u(t)

y(t) = (1  0) x(t) + (1  1) u(t)


G3=rand(tf([2 2 0 2 2]));
m3=tf2ss(G3,'o')
              /  -2.8  1      0  0 \        /  -2.6   -1.5 \
              | -0.67  0      0  0 |        | -0.67  -0.67 |
```

```
d/dt x(t) = |     0  0    -2.8  1 | x(t) + |  -1.6   -2.4 |
            \     0  0   -0.67  0 /        \ -0.67  -0.67 /


u(t)

        /1  0  0  0\          /1  1\
y(t) =  \0  0  1  0/  x(t) +  \1  1/  u(t)
```

See Also `tf2ss`, `ss.minreal`, `ss.ss2tf`

Purpose

Set parameters in a TF object to stochastic distributions.

Syntax

`Gu=uncertain(G,c,X,MC)`

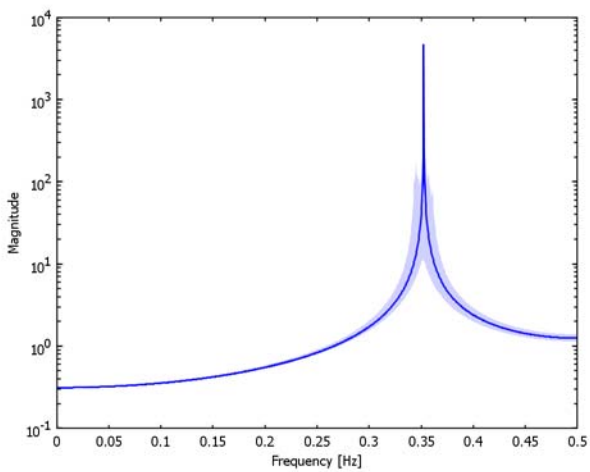
Description

The method `uncertain` is available for setting parameters in a transfer function to a probability density function. Any of the ones available in the Signals and Systems Lab can be used, or you can construct it yourself.

Example

The following example illustrates how the parameter $a(2)$, affecting the pole angle of a resonant system, is changed from 1.2 from construction to a uniform distribution. The nominal system in the result gets `a(2)=E(udist(1.1,1.3)=1.2`, that is, it is unchanged. The Monte Carlo samples of the system in the field `Gu.sysMC` get random values of $a(2)$ taken from this distribution. This uncertainty is propagated to the default plot method (Bode amplitude) as well as to all other subsequent model operations and visualization tools.

```
G=tf(1,[1 1.2 1],1);
Gu=uncertain(G,'a(2)',udist(1.1,1.3),100)

                1
   Y(z) = ----------- U(z)
          z^2+1.2*z+1

plot(Gu)
```



See Also

`tf`, `tf.estimate`, `lti.bode`, `lti.nyquist`, `lti.zpplot`, `lti.rlplot`

Purpose Compute the zeros, poles, and gain.

Syntax `[z,p,k,zMC,pMC,kMC]=zpk(s)`

Description The following list describes the output arguments:

ARGUMENT	DESCRIPTION
p	A row vector with poles
z	A (ny x nb x nu) matrix with zeros
k	A (ny x nu) matrix with gains
zMC,pMC,kMC	The corresponding Monte Carlo arrays

Example Zeros, poles, and gains for second-order MIMO system:

```
G=rand(tf([2 2 0 2 2]))

                 s(s+0.99)
   Y1(s) = --------------- U1(s)
              s^2+0.63*s+0.17

                  s(s+1)
   Y1(s) = --------------- U2(s)
              s^2+0.63*s+0.17

                 s(s+0.72)
   Y2(s) = --------------- U1(s)
              s^2+0.63*s+0.17

                 s(s+0.53)
   Y2(s) = --------------- U2(s)
              s^2+0.63*s+0.17

[z,p,k]=zpk(G)
z(:,:,1) =
          0    -0.9920
          0    -0.7226
z(:,:,2) =
          0    -1.0078
          0    -0.5340
p =
    -0.3174 +    0.2668i    -0.3174 -    0.2668i
k =
  1  1
  1  1
```

See Also `ss.zpk`

Purpose Convert transfer function to state-space model.

Syntax `[A,B,C,D]=tf2ss(b,a)`

Description The transfer function represented on polynomial `b,a` form is transformed into an state-space model characterized by the `A,B,C,D` matrices, where `form` denotes the state-space representation:

FORM	DESCRIPTION
`'controller'`	Controller canonical form
`'observer'`	Observer canonical form

This function is limited to SISO systems. Use the TF method `tf2ss` for MIMO systems.

Example The following example creates the `a` and `b` polynomials for a transfer function and then converts it to a state-space model:

```
a=poly([1 -0.5 -0.3+0.2i -0.3-0.2i])
a =
          1     0.1000    -0.6700    -0.3650    -0.0650
b=poly([1 -0.7 ])
b =
          1    -0.3000    -0.7000
[A,B,C,D]=tf2ss(b,a)
A =
    -0.1000     0.6700     0.3650     0.0650
          1          0          0          0
          0          1          0          0
          0          0          1          0
B =
  1
  0
  0
  0
C =
    -0.4000    -0.0300     0.3650     0.0650
D =
1
```

See Also `tf.tf2ss`, `ss2tf`

Purpose The Time-Frequency Description (TFD) object.

Syntax The following ways to create a TFD object are available:

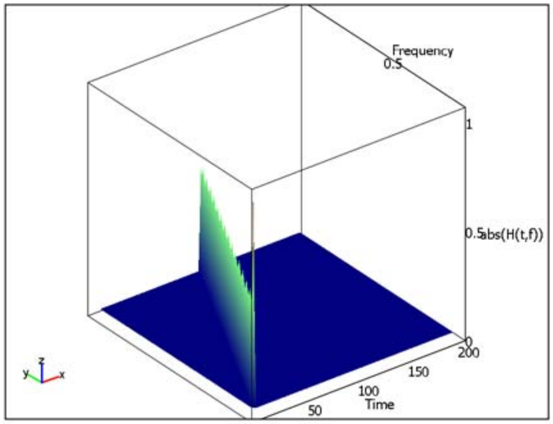
`Yt=tfd`	Empty object
`Yt=tfd(Y,t,f)`	Direct construction
`Yt=tfd(mt)`	Conversion from RARX model
`Yt=tfd(z)`	Estimation from signal object, equivalent to `Yt=estimate(tfd,z)` and `Yt=sig2tfd(z)`

Description The TFD object contains the following fields:

TFD	TIME FREQUENCY DESCRIPTION
tfd.E	Energy in each time frequency bin
tfd.f	Frequency
tfd.t	Time

Example Directly define a time-frequency description of a chirp-like signal:

```
Nf=100;
Nt=200;
E=zeros(Nf,Nt);
t=1:Nt;
f=(1:Nf)/Nf;
E(1:100,1:100)=eye(100);
E(2:100,1:99)=0.5*eye(99);
E(1:100,2:101)=0.5*eye(100);
Yt=tfd(E,t,f);
Yt.fs=4;
surf(Yt,'histeq','off')
```



See Also `tfdplot`, `rarx.rarx2tfd`

Purpose

Estimate a Time-Frequency Description (TFD) of a signal.

Syntax

Use the following calls to estimate a TFD of a signal:

`Yt=estimate(tfd,y,Property1,Value1,...)`	Explicit call
`Yt=tfd(y,Property1,Value1,...)`	Implicit call
`Yt=sig2tfd(y,Property1,Value1,...)`	Direct low-level call

Description

The function computes the periodogram over segments of the signal. A window is applied on each segment, and the segments can overlap each other. The output is a TFD object with fields `E` (energy), `t` (time), and `f` (frequency). The computations are similar to the Welch spectral estimate (in fact, taking the average over time of a TFD `mean(Yt.E')` yields the Welch spectral estimate). In the same way, the smoothed signal energy is computed by `mean(Yt.E)`.

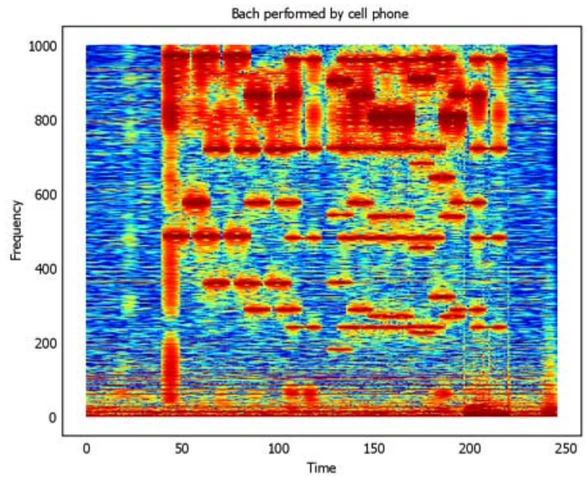
Optional parameters:

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
`S`	`{max(N/ 25,128)}`	Segment length in samples. The larger S, the better frequency resolution but the worse time resolution
`overlap`	`{90[%]}`	Overlap of each segment in percent, 0<=overlap<100
`fs`	`{2}`	Sampling frequency, scales the frequency axis f
`win`	`{'hamming'}`	Data window on each segment, see getwindow for options

Example

Load a piece of music performed by a cellular phone and display its TFD. The individual notes in the chords are visible as red regions.

```
load bach
sig2tfd(y,'S',2000)
```



See Also `tfd`, `tfdplot`, `getwindow`

Purpose Illustrate a time-frequency description (TFD).

Syntax `tfdplot(tfd,Property1,Value1,...)`

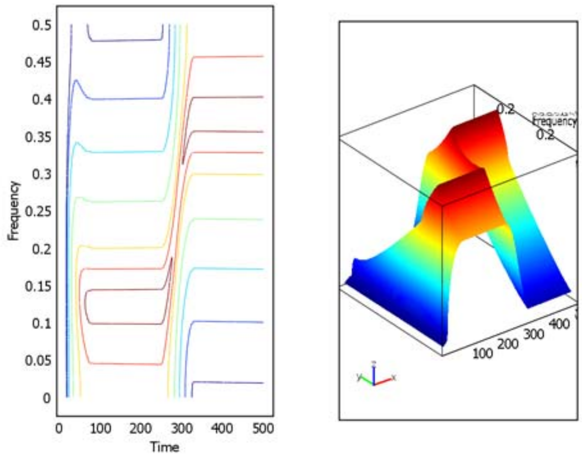
Description Different 3D views of the TFD object are possible. Note that you can only plot one TFD object at the time. The function is called from `ltv2tfd`, `sig2tfd`, and `ltvplot`.

TABLE 2-23: TFDPLOT PROPERTIES

PROPERTY	VALUE/{DEFAULT}	DESCRIPTION
axis	{gca}	Axis handle where plot is added
view	{'contour'}	TFD as contour plot
	'surf'	TFD as surf plot
	'mesh'	TFD as mesh plot
	'image'	TFD as an image
histeq	{'on'} | 'off'	Histogram equalization for energy (z) values
fontsize	{14}	Font size

Example Compute the TFD of an example AR(2) LTV object, and illustrate with contour and surf plots:

```
m=exltv('ar2');
tfd=ltv2tfd(m);
subplot(1,2,1), tfdplot(tfd,'view','contour')
subplot(1,2,2), tfdplot(tfd,'view','surf')
```



See Also `ltv2tfd`, `sig2tfd`, `ltvplot`, `histeq`

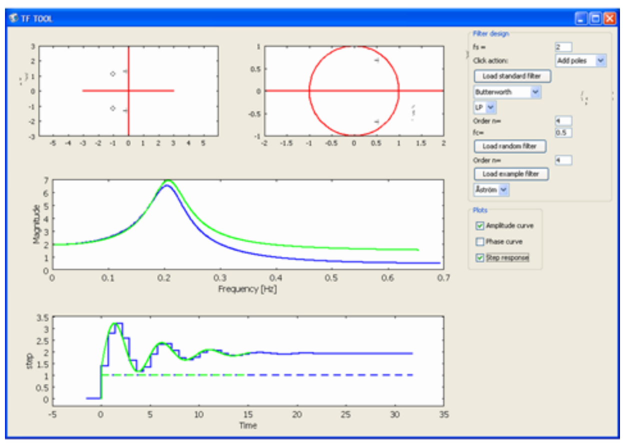
Purpose Visualization of transfer function properties.

Syntax `tftool`

Description The `tftool` graphical user interface provides interactive tools for designing a transfer function or filter by positioning the poles and zeros. You can also visualize the properties of a system described by a transfer function using the following plots:

- Bode amplitude plot
- Bode phase plot
- Impulse response
- Step response

Example The following plot shows `tftool` with the Bode amplitude plot and the step-response plot active:



Purpose The uniform distribution.

Syntax `X=udist(a,b)`

Description The probability density function of the uniform distribution, and its first two moments, are given by

$$p(x;a,b) = \frac{1}{b-a}, \quad a < x < b,$$

$$E(X) = \frac{a+b}{2},$$

$$\mathrm{Var}(X) = \frac{(b-a)^2}{12}.$$

`n` must be a positive integer. This is a child of `pdfclass`, and all of its methods apply to this distribution, in particular `pdfclass.estimate` and the plot functions.

The linearity property of the uniform distribution is implemented symbolically.

Example Illustration of the linearity property:

```
U=udist(1,2)
U(1,2)
U1=U+2
U(3,4)
U2=1+2*U
U(3,5)
```

See Also `pdfclass`

INDEX