Signal Processing Toolbox™
Reference

R2012b

MATLAB®

How to Contact MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Signal Processing Toolbox™ Reference

© COPYRIGHT 1988–2012 by The MathWorks, Inc.

Trademarks

Patents

Contents

By Category

2

Functions — Alphabetical List

3

Index

1

Function Reference

Signal Processing Functions in MATLAB

conv	Convolution and polynomial multiplication
conv2	2-D convolution
corrcoef	Correlation coefficients
cov	Covariance matrix
cplxpair	Sort complex numbers into complex conjugate pairs
deconv	Deconvolution and polynomial division
fft	Discrete Fourier transform
fft2	2-D discrete Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
filter2	2-D digital filter
freqspace	Frequency spacing for frequency response
ifft	Inverse discrete Fourier transform
ifft2	2-D inverse discrete Fourier transform
unwrap	Correct phase angles to produce smoother phase plots

Digital Filters

Discrete-Time Filters

dfilt.cascade	Cascade of discrete-time filters
dfilt.delay	Delay filter
dfilt.df1	Discrete-time, direct-form I filter
dfilt.df1sos	Discrete-time, second-order section, direct-form I filter
dfilt.df1t	Discrete-time, direct-form I transposed filter
dfilt.df1tsos	Discrete-time, second-order section, direct-form I transposed filter
dfilt.df2	Discrete-time, direct-form II filter
dfilt.df2sos	Discrete-time, second-order section, direct-form II filter

dfilt.df2t	Discrete-time, direct-form II transposed filter
dfilt.df2tsos	Discrete-time, second-order section, direct-form II transposed filter
dfilt.dfasymfir	Discrete-time, direct-form antisymmetric FIR filter
dfilt.dffir	Discrete-time, direct-form, FIR filter
dfilt.dffirt	Discrete-time, direct-form FIR transposed filter
dfilt.dfsymfir	Discrete-time, direct-form symmetric FIR filter
dfilt.fftfir	Discrete-time, overlap-add, FIR filter
dfilt.latticeallpass	Discrete-time, lattice allpass filter
dfilt.latticear	Discrete-time, lattice, autoregressive filter
dfilt.latticearma	Discrete-time, lattice, autoregressive, moving-average filter
dfilt.latticemamax	Discrete-time, lattice, moving-average filter
dfilt.latticemamin	Discrete-time, lattice, moving-average filter
dfilt.parallel	Discrete-time, parallel structure filter
dfilt.scalar	Discrete-time, scalar filter
dfilt.statespace	Discrete-time, state-space filter

FIR Filter Design

cfirpm	Complex and nonlinear-phase equiripple FIR filter design
fir1	Window-based finite impulse response filter design
fir2	Frequency sampling-based finite impulse response filter design
fircls	Constrained least square, FIR multiband filter design
fircls1	Constrained least square, lowpass and highpass, linear phase, FIR filter design
firls	Least square linear-phase FIR filter design
firpm	Parks-McClellan optimal FIR filter design
firpmord	Parks-McClellan optimal FIR filter order estimation
intfilt	Interpolation FIR filter design
kaiserord	Kaiser window FIR filter design estimation parameters
sgolay	Savitzky-Golay filter design

Communications Filters

firrcos	Raised cosine FIR filter design
gaussfir	Gaussian FIR pulse-shaping filter

IIR Digital Filter Design

butter	Butterworth filter design
cheby1	Chebyshev Type I filter design (passband ripple)
cheby2	Chebyshev Type II filter design (stopband ripple)
ellip	Elliptic filter design
maxflat	Generalized digital Butterworth filter design
yulewalk	Recursive digital filter design

IIR Filter Order Estimation

buttord	Butterworth filter order and cutoff frequency
cheb1ord	Chebyshev Type I filter order
cheb2ord	Chebyshev Type II filter order
ellipord	Minimum order for elliptic filters

Filter Analysis

abs	Absolute value (magnitude)
angle	Phase angle
filternorm	2-norm or infinity-norm of digital filter
freqz	Frequency response of digital filter
fvtool	Open Filter Visualization Tool
grpdelay	Average filter delay (group delay)
impz	Impulse response of digital filter

Filter Implementation

Filter Specification Objects -- Response Types

fdesign	Filter specification object
fdesign.arbmag	Arbitrary response magnitude filter specification object
fdesign.bandpass	Bandpass filter specification object
fdesign.bandstop	Bandstop filter specification object
fdesign.differentiator	Differentiator filter specification object
fdesign.highpass	Highpass filter specification object
fdesign.hilbert	Hilbert filter specification object
fdesign.lowpass	Lowpass filter specification
fdesign.pulseshaping	Pulse-shaping filter specification object

Filter Specification Objects -- Design Methods

design	Apply design method to filter specification object
designmethods	Methods available for designing filter from specification object
designopts	Valid input arguments and values for specification object and method
equiripple	Equiripple single-rate FIR filter from specification object
kaiserwin	Kaiser window filter from specification object
validstructures	Structures for specification object with design method
window (filter design method)	FIR filter using windowed impulse response

Analog Filters

Analog Lowpass Filter Prototypes (p. 1-10)	Prototyping functions for analog lowpass filters
Analog Filter Design (p. 1-10)	Design functions for analog filters
Filter Analysis (p. 1-11)	Analysis functions for analog filters
Analog Filter Transformation (p. 1-11)	Transformation functions for analog filters
Filter Discretization (p. 1-11)	Discretization functions for analog filters

Analog Lowpass Filter Prototypes

besselap	Bessel analog lowpass filter prototype
buttap	Butterworth filter prototype
cheb1ap	Chebyshev Type I analog lowpass filter prototype
cheb2ap	Chebyshev Type II analog lowpass filter prototype
ellipap	Elliptic analog lowpass filter prototype

Analog Filter Design

besself	Bessel analog filter design
butter	Butterworth filter design
cheby1	Chebyshev Type I filter design (passband ripple)

cheby2   Chebyshev Type II filter design
                                (stopband ripple)

ellip   Elliptic filter design

Filter Analysis

abs   Absolute value (magnitude)

freqs   Frequency response of analog filters

Analog Filter Transformation

lp2bp   Transform lowpass analog filters to
                                bandpass

lp2bs   Transform lowpass analog filters to
                                bandstop

lp2hp   Transform lowpass analog filters to
                                highpass

lp2lp   Change cutoff frequency for lowpass
                                analog filter

Filter Discretization

bilinear   Bilinear transformation method for
                                analog-to-digital filter conversion

impinvar   Impulse invariance method for
                                analog-to-digital filter conversion

Linear Systems

latc2tf	Convert lattice filter parameters to transfer function form
polyscale	Scale roots of polynomial
polystab	Stabilize polynomial
residuez	z-transform partial-fraction expansion
sos2ss	Convert digital filter second-order section parameters to state-space form
sos2tf	Convert digital filter second-order section data to transfer function form
sos2zp	Convert digital filter second-order section parameters to zero-pole-gain form
ss2sos	Convert digital filter state-space parameters to second-order sections form
ss2tf	Convert state-space filter parameters to transfer function form
ss2zp	Convert state-space filter parameters to zero-pole-gain form
tf2latc	Convert transfer function filter parameters to lattice filter form
tf2sos	Convert digital filter transfer function data to second-order sections form
tf2ss	Convert transfer function filter parameters to state-space form

tf2zp	Convert transfer function filter parameters to zero-pole-gain form
tf2zpk	Convert transfer function filter parameters to zero-pole-gain form
zp2sos	Convert zero-pole-gain filter parameters to second-order sections form
zp2ss	Convert zero-pole-gain filter parameters to state-space form
zp2tf	Convert zero-pole-gain filter parameters to transfer function form

Windows

barthannwin	Modified Bartlett-Hann window
bartlett	Bartlett window
blackman	Blackman window
blackmanharris	Minimum 4-term Blackman-Harris window
bohmanwin	Bohman window
chebwin	Chebyshev window
dpss	Discrete prolate spheroidal (Slepian) sequences
dpssclear	Remove discrete prolate spheroidal sequences from database
dpssdir	Discrete prolate spheroidal sequences database directory
dpssload	Load discrete prolate spheroidal sequences from database
flattopwin	Flat Top weighted window
gausswin	Gaussian window
hamming	Hamming window
hann	Hann (Hanning) window
kaiser	Kaiser window
nuttallwin	Nuttall-defined minimum 4-term Blackman-Harris window
parzenwin	Parzen (de la Valle-Poussin) window
rectwin	Rectangular window
sigwin	Signal processing window object
taylorwin	Taylor window
triang	Triangular window
tukeywin	Tukey (tapered cosine) window

window   Window function gateway

wvtool   Open Window Visualization Tool

Transforms

bitrevorder	Permute data into bit-reversed order
czt	Chirp *z*-transform
dct	Discrete cosine transform (DCT)
dftmtx	Discrete Fourier transform matrix
digitrevorder	Permute input into digit-reversed order
fwht	Fast Walsh–Hadamard transform
goertzel	Discrete Fourier transform with second-order Goertzel algorithm
hilbert	Discrete-time analytic signal using Hilbert transform
idct	Inverse discrete cosine transform
ifwht	Inverse Fast Walsh–Hadamard transform

Cepstral Analysis

cceps	Complex cepstral analysis
icceps	Inverse complex cepstrum
rceps	Real cepstrum and minimum phase reconstruction

Statistical Signal Processing

corrmtx	Data matrix for autocorrelation matrix estimation
cpsd	Cross power spectral density
dspdata	DSP data parameter information
dspdata.psd	Power spectral density
mscohere	Magnitude squared coherence
pburg	PSD using Burg method
pcov	PSD using covariance method
peig	Pseudospectrum using eigenvector method
periodogram	PSD using periodogram
pmcov	PSD using modified covariance method
pmtm	PSD using multitaper method (MTM)
pmusic	Pseudospectrum using MUSIC algorithm
pwelch	PSD using Welch's method
pyulear	PSD using Yule-Walker AR method
rooteig	Frequency and power content using eigenvector method
rootmusic	Root MUSIC algorithm
spectrogram	Spectrogram using short-time Fourier transform
spectrum	Spectral estimation
tfestimate	Transfer function estimate
xcorr	Cross-correlation

xcorr2   2–D cross-correlation

xcov   Cross-covariance

Parametric Modeling

arburg	Autoregressive (AR) all-pole model parameters estimated using Burg method
arcov	Estimate AR model parameters using covariance method
armcov	Estimate AR model parameters using modified covariance method
aryule	Estimate autoregressive (AR) all-pole model using Yule-Walker method
invfreqs	Identify continuous-time filter parameters from frequency response data
invfreqz	Identify discrete-time filter parameters from frequency response data
prony	Prony method for filter design
stmcb	Compute linear model using Steiglitz-McBride iteration

Linear Prediction

ac2poly	Convert autocorrelation sequence to prediction polynomial
ac2rc	Convert autocorrelation sequence to reflection coefficients
aryule	Estimate autoregressive (AR) all-pole model using Yule-Walker method
is2rc	Convert inverse sine parameters to reflection coefficients
lar2rc	Convert log area ratio parameters to reflection coefficients
levinson	Levinson-Durbin recursion
lpc	Linear prediction filter coefficients
lsf2poly	Convert line spectral frequencies to prediction filter coefficients
poly2ac	Convert prediction filter polynomial to autocorrelation sequence
poly2lsf	Convert prediction filter coefficients to line spectral frequencies
poly2rc	Convert prediction filter polynomial to reflection coefficients
rc2ac	Convert reflection coefficients to autocorrelation sequence
rc2is	Convert reflection coefficients to inverse sine parameters
rc2lar	Convert reflection coefficients to log area ratio parameters
rc2poly	Convert reflection coefficients to prediction filter polynomial

Multirate Signal Processing

decimate	Decimation — decrease sampling rate
downsample	Decrease sampling rate by integer factor
interp	Interpolation — increase sampling rate by integer factor
resample	Change sampling rate by rational factor
upfirdn	Upsample, apply FIR filter, and downsample
upsample	Increase sampling rate by integer factor

Waveform Generation

chirp	Swept-frequency cosine
diric	Dirichlet or periodic sinc function
gauspuls	Gaussian-modulated sinusoidal pulse
gmonopuls	Gaussian monopulse
pulstran	Pulse train
rectpuls	Sampled aperiodic rectangle
sawtooth	Sawtooth or triangle wave
sinc	Sinc vector or matrix
square	Square wave
tripuls	Sampled aperiodic triangle
vco	Voltage controlled oscillator

Specialized Operations

buffer	Buffer signal vector into matrix of data frames
cell2sos	Convert second-order sections cell array to matrix
db	Convert energy or power measurements to decibels
db2mag	Convert decibels (dB) to magnitude
db2pow	Convert decibels (dB) to power
demod	Demodulation for communications simulation
eqtflength	Equalize lengths of transfer function's numerator and denominator
mag2db	Convert magnitude to decibels (dB)
marcumq	Generalized Marcum Q function
modulate	Modulation for communications simulation
pow2db	Convert power to decibels (dB)
seqperiod	Compute period of sequence
sos2cell	Convert second-order sections matrix to cell array
strips	Strip plot
udecode	Decode 2-level quantized integer inputs to floating-point outputs
uencode	Quantize and encode floating-point inputs to integer outputs

GUIs

fdatool	Open Filter Design and Analysis Tool
filterbuilder	GUI-based filter design
fvtool	Open Filter Visualization Tool
sptool	Open interactive digital signal processing tool
wintool	Open Window Design and Analysis Tool
wvtool	Open Window Visualization Tool

Signal Statistics

peak2peak	Maximum-to-minimum difference
peak2rms	Peak-magnitude-to-RMS ratio
rms	Root-mean-square level
rssq	Root-sum-of-squares level

Measurements of Bilevel Waveforms

dutycycle	Duty cycle of pulse waveform
falltime	Fall time of negative-going bilevel waveform transitions
midcross	Mid-reference level crossing for bilevel waveform
overshoot	Overshoot metrics of bilevel waveform transitions
pulseperiod	Period of bilevel pulse
pulsesep	Separation between bilevel waveform pulses
pulsewidth	Bilevel waveform pulse width
risetime	Rise time of positive-going bilevel waveform transitions
settlingtime	Settling time for bilevel waveform
slewrate	Slew rate of bilevel waveform
statelevels	State-level estimation for bilevel waveform with histogram method
undershoot	Undershoot metrics of bilevel waveform transitions

By Category

Windows

sigwin.barthannwin	Construct Bartlett-Hanning window object
sigwin.bartlett	Construct Bartlett window object
sigwin.blackman	Construct Blackman window object
sigwin.blackmanharris	Construct Blackman–Harris window object
sigwin.bohmanwin	Construct Bohman window object
sigwin.chebwin	Construct Dolph-Chebyshev window object
sigwin.flattopwin	Construct flat top window object
sigwin.gausswin	Construct Gaussian window object
sigwin.hamming	Construct Hamming window object
sigwin.hann	Construct Hann (Hanning) window object
sigwin.kaiser	Construct Kaiser window object
sigwin.nuttallwin	Construct Nuttall defined 4–term Blackman-Harris window object
sigwin.parzenwin	Construct Parzen window object
sigwin.rectwin	Construct rectangular window object
sigwin.taylorwin	Construct Taylor window object
sigwin.triang	Construct triangular window object
sigwin.tukeywin	Construct Tukey window object

3

Functions — Alphabetical List

abs

Purpose   Absolute value (magnitude)

Description   abs is a MATLAB function.

Examples   Calculate the magnitude of the DFT of a sequence:

```
t = (0:99)/100;                      % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);   % Signal
y = fft(x);                          % DFT of x
m = abs(y);                          % Magnitude
m=m(1:51);            % Unique magnitudes
```

Plot the magnitude:

```
f=0:50;     % Frequency vector
plot(f,m);
ylabel('Magnitude'); xlabel('Hz');
```

Purpose
Convert autocorrelation sequence to prediction polynomial

Syntax
```
a = ac2poly(r)
[a,efinal] = ac2poly(r)
```

Description
`a = ac2poly(r)` finds the linear prediction, FIR filter polynomial `a` corresponding to the autocorrelation sequence `r`. `a` is the same length as `r`, and `a(1) = 1`. The prediction filter polynomial represents the coefficients of the prediction filter whose output produces a signal whose autocorrelation sequence is approximately the same as the given autocorrelation sequence `r`.

`[a,efinal] = ac2poly(r)` returns the final prediction error `efinal`, determined by running the filter for `length(r)` steps.

Tips
You can apply this function to real or complex data.

Examples
Consider the autocorrelation sequence:

```
r = [5.0000 -1.5450 -3.9547 3.9331 1.4681 -4.7500];
```

The corresponding prediction filter polynomial is

```
[a,efinal] = ac2poly(r)
a =
    1.0000  0.6147  0.9898  0.0004  0.0034 -0.0077
efinal =
    0.1791
```

References
[1] Kay, S.M. *Modern Spectral Estimation.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

See Also
ac2rc | poly2ac | rc2poly

ac2rc

Purpose	Convert autocorrelation sequence to reflection coefficients
Syntax	`[k,r0] = ac2rc(r)`
Description	`[k,r0] = ac2rc(r)` finds the reflection coefficients k corresponding to the autocorrelation sequence r. r0 contains the zero-lag autocorrelation. If r is a matrix where the columns are separate channels of autocorrelation sequences, r0 contains the zero-lag autocorrelation coefficient for each channel. These reflection coefficients can be used to specify the lattice prediction filter that produces a sequence with approximately the same autocorrelation sequence as the given sequence r.
Tips	You can apply this function to real or complex data.
References	[1] Kay, S.M. *Modern Spectral Estimation.* Englewood Cliffs, NJ: Prentice-Hall, 1988.
See Also	ac2poly | poly2rc | rc2ac

Purpose   Phase angle

Description   angle is a MATLAB function.

Signal-specific Example   Calculate the phase of the FFT of a sequence.

```
t = (0:99)/100;                       % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);  % Signal
y = fft(x);                           % Compute DFT of x
p = unwrap(angle(y));                 % Phase
```

Plot the phase:

```
f = (0:length(y)-1)'/length(y)*100;   % Frequency vector
plot(f,p)
```

arburg

Purpose	Autoregressive (AR) all-pole model parameters estimated using Burg method

Syntax	`ar_coeffs = arburg(data,order)`
	`[ar_coeffs,NoiseVariance] = arburg(data,order)`
	`[ar_coeffs,NoiseVariance,reflect_coeffs] = arburg(data,order)`

Description	`ar_coeffs = arburg(data,order)` returns the AR coefficients for the input `data` and model `order`. The elements of `ar_coeffs` are normalized by `ar_coeffs(1)`. The model `order` requires an integer value less than the length of the input `data`.

Description `ar_coeffs = arburg(data,order)` returns the AR coefficients for the input `data` and model `order`. The elements of `ar_coeffs` are normalized by `ar_coeffs(1)`. The model `order` requires an integer value less than the length of the input `data`.

[ar_coeffs,NoiseVariance] = arburg(data,order) returns the estimated variance `NoiseVariance` of the white noise input.

[ar_coeffs,NoiseVariance,reflect_coeffs] = arburg(data,order) returns the reflection coefficients `reflect_coeffs`.

Definitions

AR(p) Model

In an AR model of order p, the current output is a linear combination of the past p outputs plus a white noise input. The weights on the p past outputs minimize the mean-square prediction error of the autoregression. If *y[n]* is the current value of the output and *x[n]* is a zero mean white noise input, the AR(p) model is:

$$y[n] + \sum_{k=1}^{p} a(k)y[n-k] = x[n]$$

Reflection Coefficients

The *reflection coefficients* are the partial autocorrelation coefficients scaled by (–1). The reflection coefficients indicate the time dependence between *y[n]* and *y[n-k]* after subtracting the prediction based on the intervening *k-1* time steps.

Examples   Generate AR(4) process and estimate coefficients:

```
A=[1 -2.7607 3.8106 -2.6535 0.9238];
% AR(4) coefficients
y=filter(1,A,0.2*randn(1024,1));
% Filter a white noise input to create AR(4) process
ar_coeffs=arburg(y,4);
%compare the results in ar_coeffs to the vector A.
```

Estimate input noise variance for AR(4) model:

```
A=[1 -2.7607 3.8106 -2.6535 0.9238];
% Generate noise standard deviations
% Seed random number generator for reproducible results
rng default;
noise_stdz=rand(50,1)+0.5;
for j=1:50
y=filter(1,A,noise_stdz(j)*randn(1024,1));
[ar_coeffs,NoiseVariance(j)]=arburg(y,4);
end
%Compare actual vs. estimated variances
plot(noise_stdz.^2,NoiseVariance,'k*');
xlabel('Input Noise Variance');
ylabel('Estimated Noise Variance');
```
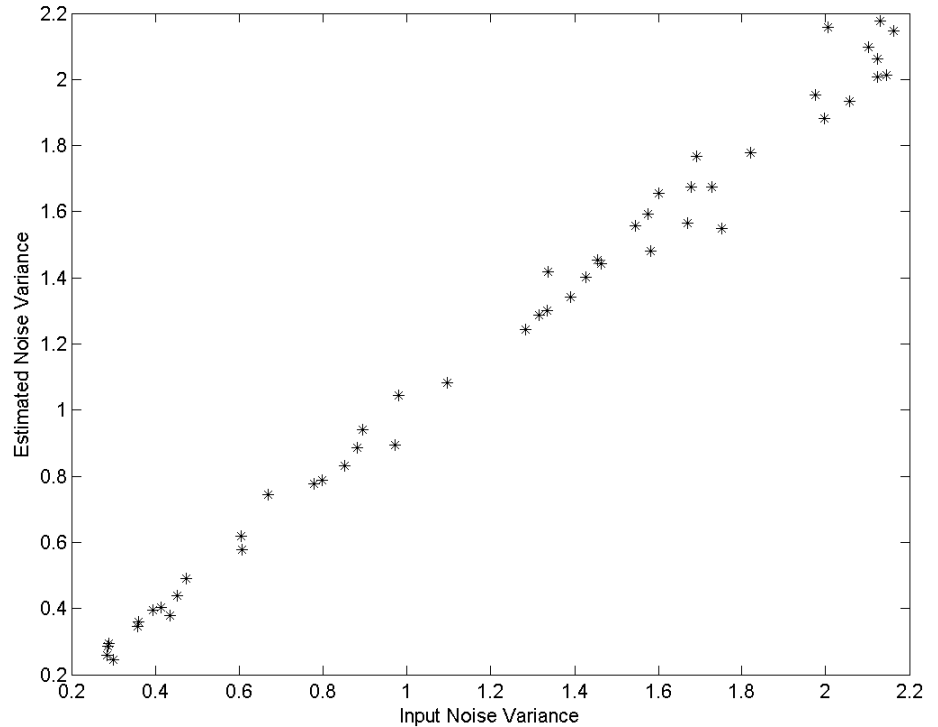
Algorithms   The Burg method estimates the reflection coefficients and uses the reflection coefficients to estimate the AR coefficients recursively. You can find the recursion and lattice filter relations describing the update of the forward and backward prediction errors in [1].

References   [1] Kay, S.M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice Hall, 1988, pp. 228–230.

See Also   arcov | armcov | aryule | levinson | lpc

How To   • "Parametric Modeling"

Purpose   Estimate AR model parameters using covariance method

Syntax   ```
a = arcov(x,p)
[a,e] = arcov(x,p)
```

**Description**     `a = arcov(x,p)` uses the covariance method to fit a `p`th order autoregressive (AR) model to the input signal, `x`, which is assumed to be the output of an AR system driven by white noise. This method minimizes the forward prediction error in the least-squares sense. The vector `a` contains the normalized estimate of the AR system parameters, $A(z)$, in descending powers of $z$. Let $y(n)$ be a wide-sense stationary random process obtained by filtering a white noise input with variance

$e$ with the system function $A(z)$. If $P_y(e^{j\omega})$ is the power spectral density of $y(n)$, then:

$$P_y(e^{j\omega}) = \frac{e}{|A(e^{j\omega})|^2} = \frac{e}{|1 + \sum_{k=1}^{P} a(k)e^{-j\omega k}|^2}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order `p` is important.

`[a,e] = arcov(x,p)` returns the variance estimate, `e`, of the white noise input to the AR model.

**See Also**     arburg | armcov | aryule | lpc | pcov | prony

**Purpose**     Estimate AR model parameters using modified covariance method

**Syntax**      a = armcov(x,p)
                [a,e] = armcov(x,p)

**Description**   a = armcov(x,p) uses the modified covariance method to fit a pth
                order autoregressive (AR) model to the input signal, x, which is
                assumed to be the output of an AR system driven by white noise. This
                method minimizes the forward and backward prediction errors in the
                least-squares sense. The vector a contains the normalized estimate of
                the AR system parameters, $A(z)$, in descending powers of $z$. Let $y(n)$ be
                a wide-sense stationary random process obtained by filtering a white

                noise input with variance $e$ with the system function $A(z)$. If $P_y(e^{j\omega})$ is
                the power spectral density of $y(n)$:

$$P_y(e^{j\omega}) = \frac{e}{|A(e^{j\omega})|^2} = \frac{e}{|1 + \sum_{k=1}^{P} a(k)e^{-j\omega k}|^2}$$

                Because the method characterizes the input data using an all-pole
                model, the correct choice of the model order p is important.

                [a,e] = armcov(x,p) returns the variance estimate, e, of the white
                noise input to the AR model.

**See Also**    arburg | arcov | aryule | lpc | pmcov | prony

# aryule

**Purpose**        Estimate autoregressive (AR) all-pole model using Yule-Walker method

**Syntax**         ```
ar_coeffs = aryule(data,order)
[ar_coeffs,NoiseVariance] = aryule(data,order)
[ar_coeffs,NoiseVariance,reflect_coeffs] = aryule(data,order)
```

Description   `ar_coeffs = aryule(data,order)` returns the AR coefficients for
the input `data` and model `order`. The elements of `ar_coeffs` are
normalized by `ar_coeffs(1)`. `order` is a positive integer that cannot
exceed the length of the input data.

`[ar_coeffs,NoiseVariance] = aryule(data,order)` returns the
estimated variance `NoiseVariance` of the white noise input.

`[ar_coeffs,NoiseVariance,reflect_coeffs] =
aryule(data,order)` returns the reflection coefficients
`reflect_coeffs`.

Definitions   **AR(p) Model**

In an AR model of order p, the current output is a linear combination
of the past p outputs plus a white noise input. The weights on the
p past outputs minimize the mean-square prediction error of the
autoregression. If *y[n]* is the current value of the output and *x[n]* is a
zero-mean white noise input, the AR(p) model is:

$$\sum_{k=0}^{p} a[k]y[n-k] = x[n]$$

Reflection Coefficients

The reflection coefficients are the partial autocorrelation coefficients
scaled by (–1). The reflection coefficients indicate the time dependence
between *y[n]* and *y[n-k]* after subtracting the prediction based on the
intervening *k-1* time steps.

Examples   Create an AR(4) process and estimate the coefficients:

```
A=[1 -2.7607 3.8106 -2.6535 0.9238];
% AR(4) coefficients
y=filter(1,A,0.2*randn(1024,1));
%filter a white noise input to create AR(4) process
ar_coeffs=aryule(y,4);
%compare the results in ar_coeffs to the vector A.
```

Estimate model order using decay of reflection coefficients:

```
rng default;
y=filter(1,[1 -0.75 0.5],0.2*randn(1024,1));
%create AR(2) process
[ar_coeffs,NoiseVariance,reflect_coeffs]=aryule(y,10);
% Fit AR(10) model
stem(reflect_coeffs); axis([-0.05 10.5 -1 1]);
title('Reflection Coefficients by Lag'); xlabel('Lag');
ylabel('Reflection Coefficent');
```

Reflection Coefficients by Lag
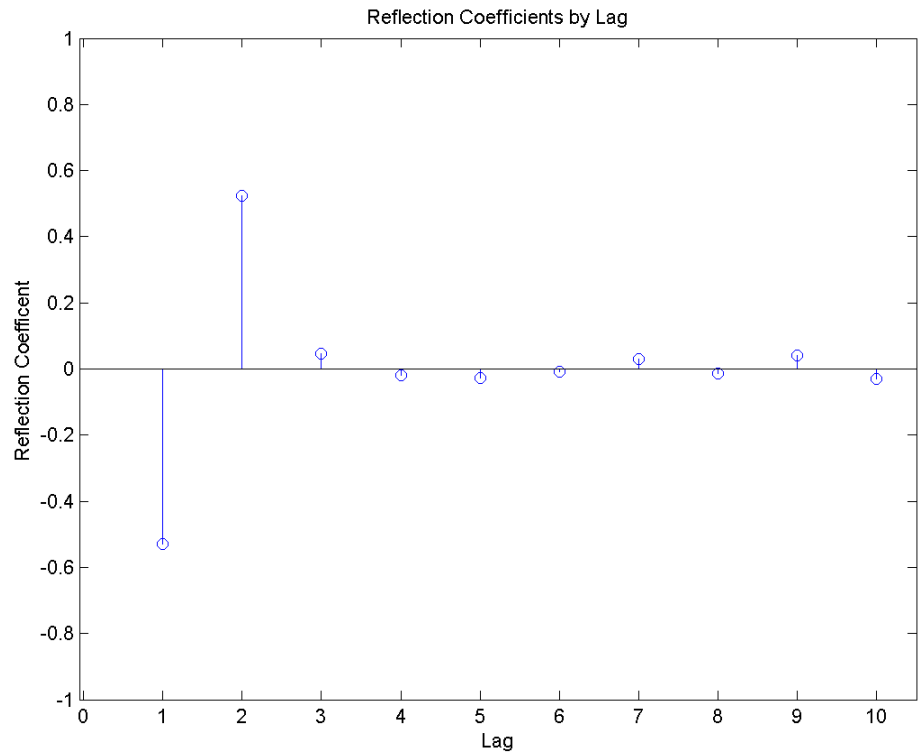
The reflection coefficients decay to zero after lag 2, which indicates that an AR(10) model significantly overestimates the time dependence in the data.

Algorithms   `aryule` uses the Levinson-Durbin recursions on the biased estimate of the sample autocorrelation sequence to compute the coefficients.

References   Monson,H. *Statistical Digital Signal Processing and Modeling*, John Wiley & Sons, 1996.

See Also   arburg | arcov | armcov | levinson | lpc

How To   • "Parametric Modeling"

barthannwin

|	
Purpose	Modified Bartlett-Hann window
Syntax	`w = barthannwin(L)`
Description	`w = barthannwin(L)` returns an L-point modified Bartlett-Hann window in the column vector w. Like Bartlett, Hann, and Hamming windows, this window has a mainlobe at the origin and asymptotically decaying sidelobes on both sides. It is a linear combination of weighted Bartlett and Hann windows with near sidelobes lower than both Bartlett and Hann and with far sidelobes lower than both Bartlett and Hamming windows. The mainlobe width of the modified Bartlett-Hann window is not increased relative to either Bartlett or Hann window mainlobes.

Note The Hann window is also called the Hanning window.

Examples	Create a 64-point Bartlett-Hann window and display the result using WVTool:

```
L=64;
wvtool(barthannwin(L))
```
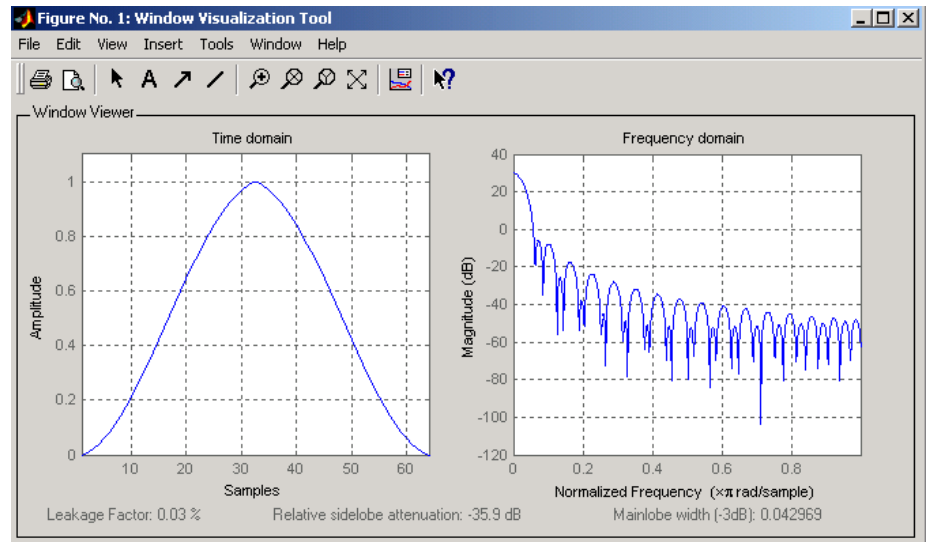
Algorithms   The equation for computing the coefficients of a Modified
Bartlett-Hanning window is

$$w(n) = 0.62 - 0.48 \left| \left(\frac{n}{N} - 0.5 \right) \right| + 0.38 \cos \left(2\pi \left(\frac{n}{N} - 0.5 \right) \right)$$

where $0 \le n \le N$ and the window length is $L = N + 1$.

References   [1] Ha, Y.H., and J.A. Pearce. "A New Window and Comparison to
Standard Windows." *IEEE® Transactions on Acoustics, Speech, and
Signal Processing.* Vol. 37, No. 2, (February 1999). pp. 298-301.

[2] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal
Processing.* Upper Saddle River, NJ: Prentice-Hall, 1999, p. 468.

See Also   bartlett | blackmanharris | bohmanwin | nuttallwin | parzenwin |
rectwin | triang | window | wintool | wvtool

bartlett

Purpose   Bartlett window

Syntax   w = bartlett(L)

Description w = bartlett(L) returns an L-point Bartlett window in the column
vector w, where L must be a positive integer. The coefficients of a
Bartlett window are computed as follows:

$$w(n) = \begin{cases} \dfrac{2n}{N}, & 0 \le n \le \dfrac{N}{2} \\ 2 - \dfrac{2n}{N}, & \dfrac{N}{2} \le n \le N \end{cases}$$

The window length $L = N + 1$.

The Bartlett window is very similar to a triangular window as returned
by the triang function. The Bartlett window always ends with zeros
at samples 1 and n, however, while the triangular window is nonzero
at those points. For L odd, the center L-2 points of bartlett(L) are
equivalent to triang(L-2).

Note If you specify a one-point window (set L=1), the value 1 is returned.

Examples   Create a 64-point Bartlett window and display the result using WVTool:

```
L=64;
wvtool(bartlett(L))
```
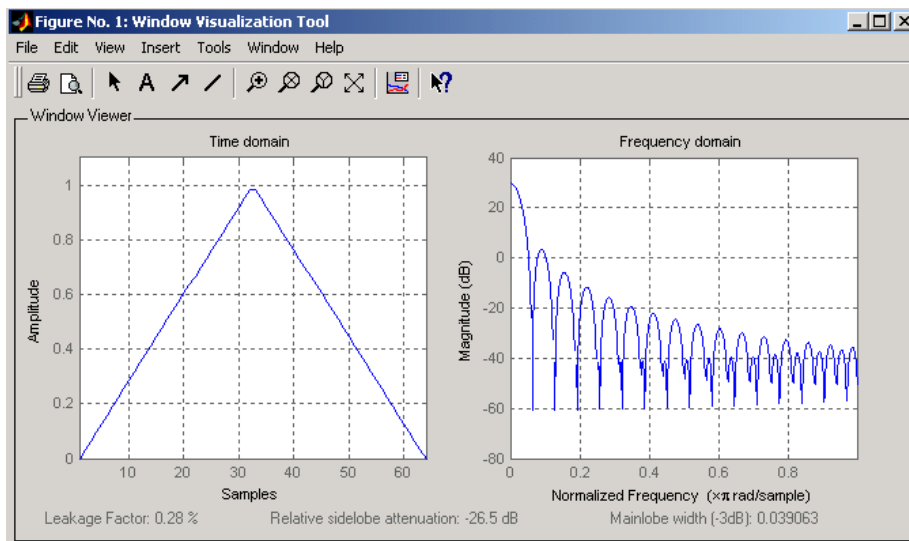
References   [1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing.* Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

See Also   barthannwin | blackmanharris | bohmanwin | nuttallwin | parzenwin | rectwin | triang | window | wintool | wvtool

besselap

Purpose	Bessel analog lowpass filter prototype
Syntax	`[z,p,k] = besselap(n)`

Description   `[z,p,k] = besselap(n)` returns the poles and gain of an order n
Bessel analog lowpass filter prototype. n must be less than or equal to
25. The function returns the poles in the length n column vector p and
the gain in scalar k. z is an empty matrix because there are no zeros.
The transfer function is

$$H(s) = \frac{k}{(s - p(1))(s - p(2))\cdots(s - p(n))}$$

besselap normalizes the poles and gain so that at low frequency and
high frequency the Bessel prototype is asymptotically equivalent to the
Butterworth prototype of the same order [1]. The magnitude of the filter
is less than $\sqrt{1/2}$ at the unity cutoff frequency $\Omega_c = 1$.

Analog Bessel filters are characterized by a group delay that is
maximally flat at zero frequency and almost constant throughout the
passband. The group delay at zero frequency is

$$\left(\frac{(2n)!}{2^n n!} \right)^{1/n}$$

Algorithms   besselap finds the filter roots from a lookup table constructed using
Symbolic Math Toolbox™ software.

References   [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal
Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 228-230.

See Also   besself | buttap | cheb1ap | cheb2ap | ellipap

Purpose   Bessel analog filter design

Syntax
```
[b,a] = besself(n,Wo)
[z,p,k] = besself(...)
[A,B,C,D] = besself(...)
```

Description besself designs lowpass, analog Bessel filters, which are characterized by almost constant group delay across the entire passband, thus preserving the wave shape of filtered signals in the passband. besself does not support the design of digital Bessel filters.

[b,a] = besself(n,Wo) designs an order n lowpass analog Bessel filter, where Wo is the frequency up to which the filter's group delay is approximately constant. Larger values of the filter order (n) produce a group delay that better approximates a constant up to frequency Wo.

besself returns the filter coefficients in the length n+1 row vectors b and a, with coefficients in descending powers of *s*, derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \ldots + b(n+1)}{s^n + a(2)s^{n-1} + \ldots + a(n+1)}$$

[z,p,k] = besself(...) returns the zeros and poles in length n or 2*n column vectors z and p and the gain in the scalar k.

[A,B,C,D] = besself(...) returns the filter design in state-space form, where A, B, C, and D are

$$x = Ax + Bu$$
$$y = Cx + Du$$

and *u* is the input, *x* is the state vector, and *y* is the output.

Examples   Design a fifth-order analog lowpass Bessel filter with an approximate constant group delay up to 10,000 rad/s and plot the frequency response of the filter using freqs:

besself

```
[b,a] = besself(5,10000);
freqs(b,a)                    % Plot frequency response
```
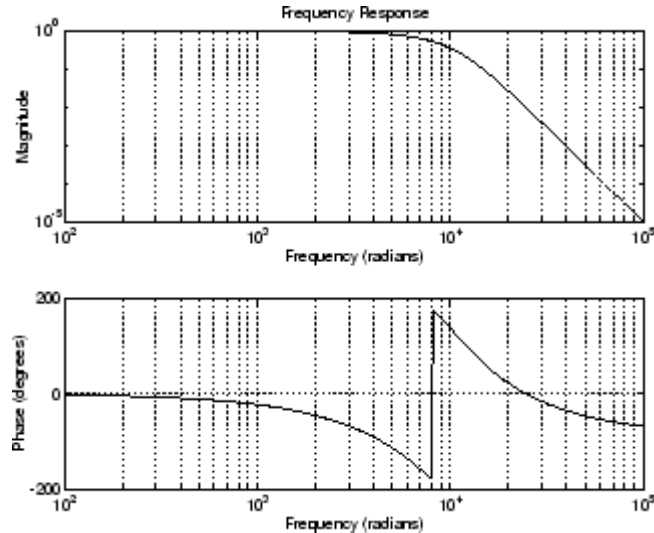

Limitations   Lowpass Bessel filters have a monotonically decreasing magnitude response, as do lowpass Butterworth filters. Compared to the Butterworth, Chebyshev, and elliptic filters, the Bessel filter has the slowest rolloff and requires the highest order to meet an attenuation specification.

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function coefficient form is the least accurate; numerical problems can arise for filter orders as low as 15.

Algorithms   besself performs a four-step algorithm:

1 It finds lowpass analog prototype poles, zeros, and gain using the besselap function.

2 It converts the poles, zeros, and gain into state-space form.

3 It transforms the lowpass prototype into a lowpass filter that meets the design specifications.

4 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also   besselap | butter | cheby1 | cheby2 | ellip

bilinear

Purpose   Bilinear transformation method for analog-to-digital filter conversion

Syntax
```
[zd,pd,kd] = bilinear(z,p,k,fs)
[zd,pd,kd] = bilinear(z,p,k,fs,fp)
[numd,dend] = bilinear(num,den,fs)
[numd,dend] = bilinear(num,den,fs,fp)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs,fp)
```

Description   The *bilinear transformation* is a mathematical mapping of variables. In digital filtering, it is a standard method of mapping the *s* or analog plane into the *z* or digital plane. It transforms analog filters, designed using classical filter design techniques, into their discrete equivalents.

The bilinear transformation maps the *s*-plane into the *z*-plane by

$$H(z) = H(s)\big|_{s=2f_z \frac{z-1}{z+1}}$$

This transformation maps the $j\Omega$ axis (from $\Omega = -\infty$ to $+\infty$) repeatedly around the unit circle (e^{jw}, from $\omega = -\pi$ to π) by

$$\omega = 2\tan^{-1}\left(\frac{\Omega}{2f_s}\right)$$

`bilinear` can accept an optional parameter `Fp` that specifies prewarping. `fp`, in hertz, indicates a "match" frequency, that is, a frequency for which the frequency responses before and after mapping match exactly. In prewarped mode, the bilinear transformation maps the *s*-plane into the *z*-plane with

$$H(z) = H(s)\big|_{s=\frac{2\pi f_p}{\tan\left(\pi\frac{f_p}{f_z}\right)}\frac{(z-1)}{(z+1)}}$$

With the prewarping option, `bilinear` maps the $j\Omega$ axis (from $\Omega = -\infty$ to $+\infty$) repeatedly around the unit circle (e^{jw}, from $\omega = -\pi$ to π) by

$$\omega = 2\tan^{-1}\left(\frac{\Omega \tan\left(\pi\dfrac{f_p}{f_s}\right)}{2\pi f_p}\right)$$

In prewarped mode, `bilinear` matches the frequency $2\pi f_p$ (in radians per second) in the *s*-plane to the normalized frequency $2\pi f_p/f_s$ (in radians per second) in the *z*-plane.

The `bilinear` function works with three different linear system representations: zero-pole-gain, transfer function, and state-space form.

Zero-Pole-Gain

`[zd,pd,kd] = bilinear(z,p,k,fs)` and

`[zd,pd,kd] = bilinear(z,p,k,fs,fp)` convert the *s*-domain transfer function specified by z, p, and k to a discrete equivalent. Inputs z and p are column vectors containing the zeros and poles, k is a scalar gain, and fs is the sampling frequency in hertz. `bilinear` returns the discrete equivalent in column vectors zd and pd and scalar kd. The optional match frequency, fp is in hertz and is used for prewarping.

Transfer Function

`[numd,dend] = bilinear(num,den,fs)` and

`[numd,dend] = bilinear(num,den,fs,fp)` convert an *s*-domain transfer function given by num and den to a discrete equivalent. Row vectors num and den specify the coefficients of the numerator and denominator, respectively, in descending powers of *s*. Let *B(s)* be the numerator polynomial and *A(s)* be the denominator polynomial. The transfer function is:

$$\frac{B(s)}{A(s)} = \frac{B(1)s^n + \cdots + B(n)s + B(n+1)}{A(1)s^m + \cdots + A(m)s + A(m+1)}$$

fs is the sampling frequency in hertz. `bilinear` returns the discrete equivalent in row vectors numd and dend in descending powers of *z*

(ascending powers of z^{-1}). fp is the optional match frequency, in hertz, for prewarping.

State-Space

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs)` and

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs,fp)` convert the continuous-time state-space system in matrices A, B, C, D

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

to the discrete-time system:

$$x[n+1] = A_d x[n] + B_d u[n]$$
$$y[n] \quad = C_d x[n] + D_d u[n]$$

fs is the sampling frequency in hertz. bilinear returns the discrete equivalent in matrices Ad, Bd, Cd, Dd. The optional match frequency, fp is in hertz and is used for prewarping.

Algorithms　　bilinear uses one of two algorithms depending on the format of the input linear system you supply. One algorithm works on the zero-pole-gain format and the other on the state-space format. For transfer function representations, bilinear converts to state-space form, performs the transformation, and converts the resulting state-space system back to transfer function form.

Zero-Pole-Gain Algorithm

For a system in zero-pole-gain form, bilinear performs four steps:

1 If fp is present, it prewarps:

```
fp = 2*pi*fp;
fs = fp/tan(fp/fs/2)
```

otherwise, fs = 2*fs.

2 It strips any zeros at ±∞ using

```
z = z(finite(z));
```

3 It transforms the zeros, poles, and gain using

```
pd = (1+p/fs)./(1-p/fs);    % Do bilinear transformation
zd = (1+z/fs)./(1-z/fs);
kd = real(k*prod(fs-z)./prod(fs-p));
```

4 It adds extra zeros at -1 so the resulting system has equivalent numerator and denominator order.

State-Space Algorithm

For a system in state-space form, `bilinear` performs two steps:

1 If `fp` is present, let

$$\lambda = \frac{\pi f_p}{\tan(\pi f_p / f_s)}$$

If `fp` is not present, let λ=*fs*.

2 Compute `Ad`, `Bd`, `Cd`, and `Dd` in terms of `A`, `B`, `C`, and `D` using

$$Ad = (I - A\tfrac{1}{2\lambda})^{-1}(I + A\tfrac{1}{2\lambda})$$
$$Bd = \tfrac{1}{\sqrt{\lambda}}(I - A\tfrac{1}{2\lambda})^{-1}B$$
$$Cd = \tfrac{1}{\sqrt{\lambda}}C(I - A\tfrac{1}{2\lambda})^{-1}$$
$$Dd = \tfrac{1}{2\lambda}C(I - A\tfrac{1}{2\lambda})^{-1}B + D$$

Diagnostics   `bilinear` requires that the numerator order be no greater than the denominator order. If this is not the case, `bilinear` displays

```
Numerator cannot be higher order than denominator.
```

bilinear

For `bilinear` to distinguish between the zero-pole-gain and transfer function linear system formats, the first two input parameters must be vectors with the same orientation in these cases. If this is not the case, `bilinear` displays

```
First two arguments must have the same orientation.
```

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design.* New York: John Wiley & Sons, 1987. Pgs. 209-213.

[2] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing.* Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 450-454.

See Also   impinvar | lp2bp | lp2bs | lp2hp | lp2lp

Purpose   Permute data into bit-reversed order

Syntax   y = bitrevorder(x)
[y,i] = bitrevorder(x)

Description   bitrevorder is useful for pre-arranging filter coefficients so that bit-reversed ordering does not have to be performed as part of an fft or inverse FFT (ifft) computation. This can improve run-time efficiency for external applications or for Simulink blockset models. Both MATLAB fft and ifft functions process linear input and output.

Note Using bitrevorder is equivalent to using digitrevorder with radix base 2.

---

y = bitrevorder(x) returns the input data in bit-reversed order in vector or matrix y. The length of x must be an integer power of 2. If x is a matrix, the bit-reversal occurs on the first dimension of x with size greater than 1. y is the same size as x.

[y,i] = bitrevorder(x) returns the bit-reversed vector or matrix y and the bit-reversed indices i, such that y = x(i). Recall that MATLAB matrices use 1-based indexing, so the first index of y will be 1, not 0.

The following table shows the numbers 0 through 7, the corresponding bits and the bit-reversed numbers.

Linear Index	Bits	Bit- Reversed	Bit-Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1

bitrevorder

Linear Index	Bits	Bit- Reversed	Bit-Reversed Index
5	101	101	5
6	110	011	3
7	111	111	7

Examples   Obtain the bit-reversed ordered output of a vector:

```
x=[0:7]';                    % Create a column vector
[x,bitrevorder(x)]
% ans =
%     0     0
%     1     4
%     2     2
%     3     6
%     4     1
%     5     5
%     6     3
%     7     7
```

See Also   fft | digitrevorder | ifft

Purpose   Blackman window

Syntax   w = blackman(N)
                w = blackman(N,SFLAG)

Description w = blackman(N) returns the N-point symmetric Blackman window in
                the column vector w, where N is a positive integer.

                w = blackman(N,SFLAG) returns an N-point Blackman window using
                the window sampling specified by 'sflag', which can be either
                'periodic' or 'symmetric' (the default). The 'periodic' flag
                is useful for DFT/FFT purposes, such as in spectral analysis. The
                DFT/FFT contains an implicit periodic extension and the periodic flag
                enables a signal windowed with a periodic window to have perfect
                periodic extension. When 'periodic' is specified, blackman computes a
                length N+1 window and returns the first N points. When using windows
                for filter design, the 'symmetric' flag should be used.

                See "Definitions" on page 3-32 for a description of the difference between
                the symmetric and periodic windows.

                Note If you specify a one-point window (set N=1), the value 1 is returned.

Examples   Create a 64-point Blackman window and display the result using
                WVTool:

                L=64;
                wvtool(blackman(L))

blackman

Definitions

The following equation defines the Blackman window of length *N*:

$$w(n) = 0.42 - 0.5\cos(2\pi n / (N-1)) + 0.08\cos(4\pi n / (N-1)) \quad 0 \le n \le M-1$$

where *M* is *N/2* for *N* even and *(N+1)/2* for *N* odd.

In the **symmetric** case, the second half of the Blackman window *M* ≤ *n* ≤ *N-1* is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Blackman window in FIR filter design.

The **periodic** Blackman window is constructed by extending the desired window length by one sample to *N+1*, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a Blackman window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing.* Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

See Also   flattopwin | hamming | hann | window | wintool | wvtool

blackmanharris

Purpose   Minimum 4-term Blackman-Harris window

Syntax   `w = blackmanharris(N)`
                    `w = blackmanharris(N,SFLAG)`

Description   `w = blackmanharris(N)` returns an N-point symmetric 4-term Blackman-Harris window in the column vector `w`. The window is minimum in the sense that its maximum sidelobes are minimized.

`w = blackmanharris(N,SFLAG)` uses `SFLAG` window sampling. `SFLAG` can be `'symmetric'` or `'periodic'`. The default is `'symmetric'`. You can find the equations defining the symmetric and periodic windows in "Definitions" on page 3-35.

Examples   Create a 32-point symmetric Blackman-Harris window and display the result using WVTool:

```
N = 32;
wvtool(blackmanharris(N))
```
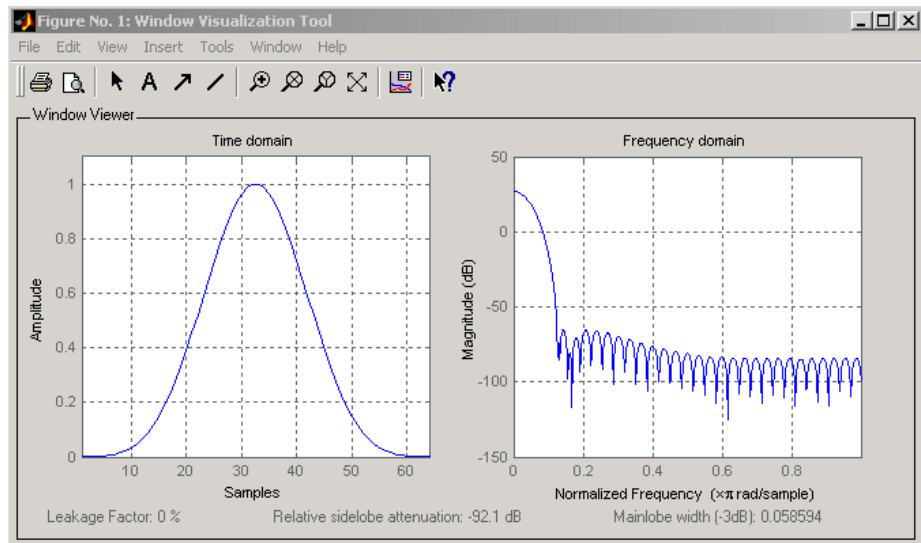
Definitions   The equation for the **symmetric** 4-term Blackman-harris window of length N is

$$w(n) = a_0 - a_1 \cos(\tfrac{2\pi n}{N-1}) + a_2 \cos(\tfrac{4\pi n}{N-1}) - a_3 \cos(\tfrac{6\pi n}{N-1}) \quad 0 \le n \le N-1$$

The equation for the **periodic** 4-term Blackman-harris window of length N is

$$w(n) = a_0 - a_1 \cos(\tfrac{2\pi n}{N}) + a_2 \cos(\tfrac{4\pi n}{N}) - a_3 \cos(\tfrac{6\pi n}{N}) \quad 0 \le n \le N-1$$

The periodic window is N-periodic.

The following table lists the coefficients:

Coefficient	Value
a0	0.35875
a1	0.48829
a2	0.14128
a3	0.01168

References   [1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE.* Vol. 66 (January 1978). pp. 51-84.

See Also   barthannwin | bartlett | bohmanwin | nuttallwin | parzenwin | rectwin | triang | window | wintool | wvtool

bohmanwin

Purpose   Bohman window

Syntax   w = bohmanwin(L)

Description   w = bohmanwin(L) returns an L-point Bohman window in column
                    vector w. A Bohman window is the convolution of two half-duration
                    cosine lobes. In the time domain, it is the product of a triangular
                    window and a single cycle of a cosine with a term added to set the first
                    derivative to zero at the boundary. Bohman windows fall off as $1/w^4$.

Examples   Compute a 64-point Bohman window and display the result using
                    WVTool:

                    L=64;
                    wvtool(bohmanwin(L))
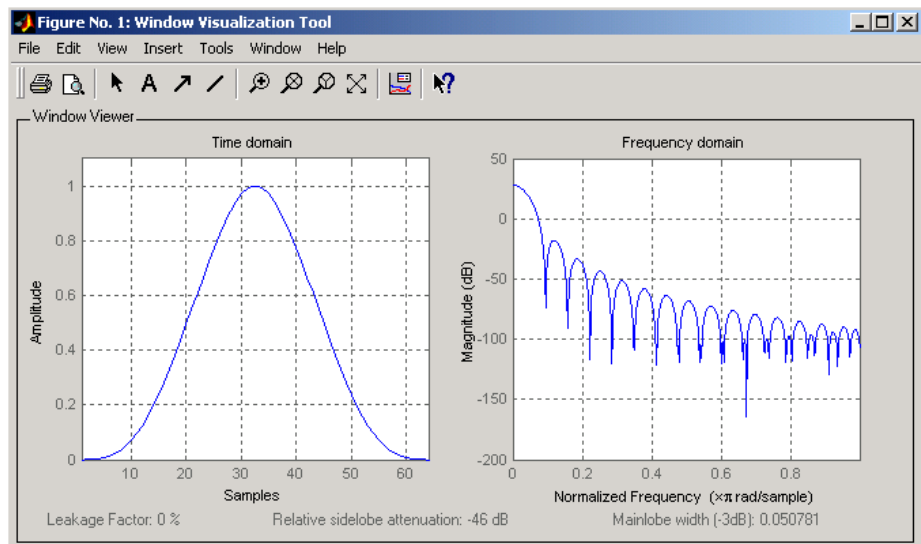
Algorithms   The equation for computing the coefficients of a Bohman window is

$$w(x) = (1 - |x|)\cos(\pi |x|) + \frac{1}{\pi}\sin(\pi |x|) \quad -1 \le x \le 1$$

where x is a length L vector of linearly spaced values generated using linspace. The first and last elements of the Bohman window are forced to be identically zero.

References   [1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978). p. 67.

See Also   barthannwin | bartlett | blackmanharris | nuttallwin | parzenwin | rectwin | triang | window | wintool | wvtool

buffer

Purpose	Buffer signal vector into matrix of data frames
Syntax	`y = buffer(x,n)`

Syntax
```
y = buffer(x,n)
y = buffer(x,n,p)
y = buffer(x,n,p,opt)
[y,z] = buffer(...)
[y,z,opt] = buffer(...)
```

Description `y = buffer(x,n)` partitions a length-L signal vector `x` into nonoverlapping data segments (frames) of length `n`. Each data frame occupies one column of matrix output `y`, which has `n` rows and `ceil(L/n)` columns. If `L` is not evenly divisible by `n`, the last column is zero-padded to length `n`.

`y = buffer(x,n,p)` overlaps or underlaps successive frames in the output matrix by `p` samples:

• For `0 < p < n` (overlap), `buffer` repeats the final `p` samples of each frame at the beginning of the following frame. For example, if `x = 1:30` and `n = 7`, an overlap of `p = 3` looks like this.

The first frame starts with `p` zeros (the default initial condition), and the number of columns in `y` is `ceil(L/(n-p))`.

• For `p < 0` (underlap), `buffer` skips `p` samples between consecutive frames. For example, if `x = 1:30` and `n = 7`, a buffer with underlap of `p = -3` looks like this.

```
y =
     1    11    21
     2    12    22
     3    13    23
     4    14    24
     5    15    25
     6    16    26
     7    17    27
```


The number of columns in y is ceil(L/(n-p)).

y = buffer(x,n,p,opt) specifies a vector of samples to precede x(1) in an overlapping buffer, or the number of initial samples to skip in an underlapping buffer:

• For 0 < p < n (overlap), opt specifies a length-p vector to insert before x(1) in the buffer. This vector can be considered an *initial condition*, which is needed when the current buffering operation is one in a sequence of consecutive buffering operations. To maintain the desired frame overlap from one buffer to the next, opt should contain the final p samples of the previous buffer in the sequence. See "Continuous Buffering" on page 3-42 below.

  By default, opt is zeros(p,1) for an overlapping buffer. Set opt to 'nodelay' to skip the initial condition and begin filling the buffer immediately with x(1). In this case, L must be length(p) or longer. For example, if x = 1:30 and n = 7, a buffer with overlap of p = 3 looks like this.

• For p < 0 (underlap), opt is an integer value in the range [0,-p] specifying the number of initial input samples, x(1:opt), to skip before adding samples to the buffer. The first value in the buffer

is therefore x(opt+1). By default, opt is zero for an underlapping buffer.

This option is especially useful when the current buffering operation is one in a sequence of consecutive buffering operations. To maintain the desired frame underlap from one buffer to the next, opt should equal the difference between the total number of points to skip between frames (p) and the number of points that were *available* to be skipped in the previous input to buffer. If the previous input had fewer than p points that could be skipped after filling the final frame of that buffer, the remaining opt points need to be removed from the first frame of the current buffer. See "Continuous Buffering" on page 3-42 for an example of how this works in practice.

[y,z] = buffer(...) partitions the length-L signal vector x into frames of length n, and outputs only the *full* frames in y. If y is an overlapping buffer, it has n rows and m columns, where

```
m = floor(L/(n-p))                  % When length(opt) = p
```

or

```
m = floor((L-n)/(n-p))+1            % When opt = 'nodelay'
```

If y is an underlapping buffer, it has n rows and m columns, where

```
m = floor((L-opt)/(n-p)) + (rem((L-opt),(n-p)) >= n)
```

If the number of samples in the input vector (after the appropriate overlapping or underlapping operations) exceeds the number of places available in the n-by-m buffer, the remaining samples in x are output in vector z, which for an overlapping buffer has length

```
length(z) = L - m*(n-p)             % When length(opt) = p
```

or

```
length(z) = L - ((m-1)*(n-p)+n)    % When opt = 'nodelay'
```

and for an underlapping buffer has length

```
length(z) = (L-opt) - m*(n-p)
```

Output z shares the same orientation (row or column) as x. If there are no remaining samples in the input after the buffer with the specified overlap or underlap is filled, z is an empty vector.

`[y,z,opt] = buffer(...)` returns the last p samples of a overlapping buffer in output opt. In an underlapping buffer, opt is the difference between the total number of points to skip between frames (-p) and the number of points in x that were *available* to be skipped after filling the last frame:

- For 0 < p < n (overlap), opt (as an output) contains the final p samples in the last frame of the buffer. This vector can be used as the *initial condition* for a subsequent buffering operation in a sequence of consecutive buffering operations. This allows the desired frame overlap to be maintained from one buffer to the next. See "Continuous Buffering" on page 3-42 below.

- For p < 0 (underlap), opt (as an output) is the difference between the total number of points to skip between frames (-p) and the number of points in x that were *available* to be skipped after filling the last frame.

```
opt = m*(n-p) + opt - L        % z is the empty vector.
```

where opt on the right-hand side is the input argument to buffer, and opt on the left-hand side is the output argument. Here m is the number of columns in the buffer, which is

```
m = floor((L-opt)/(n-p)) + (rem((L-opt),(n-p))>=n)
```

Note that for an underlapping buffer output opt is always zero when output z contains data.

The opt output for an underlapping buffer is especially useful when the current buffering operation is one in a sequence of consecutive

buffering operations. The `opt` output from each buffering operation specifies the number of samples that need to be skipped at the start of the next buffering operation to maintain the desired frame underlap from one buffer to the next. If fewer than `p` points were available to be skipped after filling the final frame of the current buffer, the remaining `opt` points need to be removed from the first frame of the next buffer.

In a sequence of buffering operations, the `opt` output from each operation should be used as the `opt` input to the subsequent buffering operation. This ensures that the desired frame overlap or underlap is maintained from buffer to buffer, as well as from frame to frame within the same buffer. See "Continuous Buffering" on page 3-42 below for an example of how this works in practice.

Continuous Buffering

In a continuous buffering operation, the vector input to the `buffer` function represents one frame in a sequence of frames that make up a discrete signal. These signal frames can originate in a frame-based data acquisition process, or within a frame-based algorithm like the FFT.

As an example, you might acquire data from an A/D card in frames of 64 samples. In the simplest case, you could rebuffer the data into frames of 16 samples; `buffer` with `n = 16` creates a buffer of four frames from each 64-element input frame. The result is that the signal of frame size 64 has been converted to a signal of frame size 16; no samples were added or removed.

In the general case where the original signal frame size, `L`, is not equally divisible by the new frame size, `n`, the overflow from the last frame needs to be captured and recycled into the following buffer. You can do this by iteratively calling `buffer` on input `x` with the two-output-argument syntax:

```
[y,z] = buffer([z;x],n)     % x is a column vector.
[y,z] = buffer([z,x],n)     % x is a row vector.
```

This simply captures any buffer overflow in `z`, and prepends the data to the subsequent input in the next call to `buffer`. Again, the input signal,

x, of frame size L, has been converted to a signal of frame size n without any insertion or deletion of samples.

Note that continuous buffering cannot be done with the single-output syntax y = buffer(...), because the last frame of y in this case is zero padded, which adds new samples to the signal.

Continuous buffering in the presence of overlap and underlap is handled with the opt parameter, which is used as both an input and output to buffer. The following two examples demonstrate how the opt parameter should be used.

Examples

Example 1: Continuous Overlapping Buffers

First create a buffer containing 100 frames, each with 11 samples:

```
data = buffer(1:1100,11);      % 11 samples per frame
```

Imagine that the frames (columns) in the matrix called data are the sequential outputs of a data acquisition board sampling a physical signal: data(:,1) is the first D/A output, containing the first 11 signal samples; data(:,2) is the second output, containing the next 11 signal samples, and so on.

You want to rebuffer this signal from the acquired frame size of 11 to a frame size of 4 with an overlap of 1. To do this, you will repeatedly call buffer to operate on each successive input frame, using the opt parameter to maintain consistency in the overlap from one buffer to the next.

Set the buffer parameters:

```
n = 4;          % New frame size
p = 1;          % Overlap
opt = -5;       % Value of y(1)
z = [];         % Initialize the carry-over vector.
```

Now repeatedly call buffer, each time passing in a new signal frame from data. Note that overflow samples (returned in z) are carried over and prepended to the input in the subsequent call to buffer:

```
for i=1:size(data,2),        % Loop over each source
                             %   frame (column)
   x = data(:,i);            % Single frame of D/A output
   [y,z,opt] = buffer([z;x],n,p,opt);
   disp(y);                  % Display the buffer of data.
   pause
end
```

Here's what happens during the first four iterations.

Iteration	Input frame [z;x]'	opt (input)	opt (output)	Output buffer (y)	Overflow (z)
i=1	[1:11]	−5	9	$\begin{bmatrix} -5 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$	[10 11]
i=2	[10 11 12:22]	9	21	$\begin{bmatrix} 9 & 12 & 15 & 18 \\ 10 & 13 & 16 & 19 \\ 11 & 14 & 17 & 20 \\ 12 & 15 & 18 & 21 \end{bmatrix}$	[22]
i=3	[22 23:33]	21	33	$\begin{bmatrix} 21 & 24 & 27 & 30 \\ 22 & 25 & 28 & 31 \\ 23 & 26 & 29 & 32 \\ 24 & 27 & 30 & 33 \end{bmatrix}$	[]
i=4	[34:44]	33	42	$\begin{bmatrix} 33 & 36 & 39 \\ 34 & 37 & 40 \\ 35 & 38 & 41 \\ 36 & 39 & 42 \end{bmatrix}$	[43 44]

Note that the size of the output matrix, y, can vary by a single column from one iteration to the next. This is typical for buffering operations with overlap or underlap.

Example 2: Continuous Underlapping Buffers

Again create a buffer containing 100 frames, each with 11 samples:

```
data = buffer(1:1100,11);  % 11 samples per frame
```

Again, imagine that data(:,1) is the first D/A output, containing the first 11 signal samples; data(:,2) is the second output, containing the next 11 signal samples, and so on.

You want to rebuffer this signal from the acquired frame size of 11 to a frame size of 4 with an underlap of 2. To do this, you will repeatedly call buffer to operate on each successive input frame, using the opt parameter to maintain consistency in the underlap from one buffer to the next.

Set the buffer parameters:

```
n = 4;     % New frame size
p = -2;    % Underlap
opt = 1;   % Skip the first input element, x(1).
z = [];    % Initialize the carry-over vector.
```

Now repeatedly call buffer, each time passing in a new signal frame from data. Note that overflow samples (returned in z) are carried over and prepended to the input in the subsequent call to buffer:

```
for i=1:size(data,2),       % Loop over each source
                            %    frame (column)
  x = data(:,i);            % Single frame of D/A output
  [y,z,opt] = buffer([z;x],n,p,opt);
  disp(y);                  % Display the buffer of data
  pause
end
```

Here's what happens during the first three iterations.

buffer

Iteration	Input frame [z;x]'	opt (input)	opt (output)	Output buffer (y)	Overflow (z)
i=1	[1:11]	1	2	$\begin{bmatrix} 1 & - \end{bmatrix}$ }skip $\begin{bmatrix} 2 & 8 \\ 3 & 9 \\ 4 & 10 \\ 5 & 11 \end{bmatrix}$ $\begin{bmatrix} 6 & - \\ 7 & - \end{bmatrix}$ }skip	[]
i=2	[12:22]	2	0	$\begin{bmatrix} 12 \\ 13 \end{bmatrix}$ }skip $\begin{bmatrix} 14 \\ 15 \\ 16 \\ 17 \end{bmatrix}$ $\begin{bmatrix} 18 \\ 19 \end{bmatrix}$ }skip	[20 21 22]
i=3	[20 21 22 23:33]	0	0	$\begin{bmatrix} - & - \end{bmatrix}$ }skip $\begin{bmatrix} 20 & 26 \\ 21 & 27 \\ 22 & 28 \\ 23 & 29 \end{bmatrix}$ $\begin{bmatrix} 24 & 30 \\ 25 & 31 \end{bmatrix}$ }skip	[32 33]

Diagnostics Error messages are displayed when $p \geq n$ or length(opt)≠length(p) in an overlapping buffer case:

```
Frame overlap P must be less than the buffer size N.
Initial conditions must be specified as a length-P vector.
```

See Also reshape

Purpose   Butterworth filter prototype

Syntax   `[z,p,k] = buttap(n)`

Description `[z,p,k] = buttap(n)` returns the poles and gain of an order `n` Butterworth analog lowpass filter prototype. The function returns the poles in the length `n` column vector `p` and the gain in scalar `k`. `z` is an empty matrix because there are no zeros. The transfer function is

$$H(s) = \frac{z(s)}{p(s)} = \frac{k}{(s - p(1))(s - p(2))\cdots(s - p(n))}$$

Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall. In the lowpass case, the first 2n-1 derivatives of the squared magnitude response are zero at $\omega = 0$. The squared magnitude response function is

$$|H(\omega)|^2 = \frac{1}{1 + (\omega / \omega_0)^{2n}}$$

corresponding to a transfer function with poles equally spaced around a circle in the left half plane. The magnitude response at the cutoff angular frequency ω_0 is always $1/\sqrt{2}$ regardless of the filter order. `buttap` sets ω_0 to 1 for a normalized result.

Algorithms
```
z = [];
p = exp(sqrt(-1)*(pi*(1:2:2*n-1)/(2*n)+pi/2)).';
k = real(prod(-p));
```

References   [1] Parks, T.W., and C.S. Burrus. *Digital Filter Design.* New York: John Wiley & Sons, 1987. Chapter 7.

See Also   besselap | butter | cheb1ap | cheb2ap | ellipap

butter

Purpose

Butterworth filter design

Syntax

```
[z,p,k] = butter(n,Wn)
[z,p,k] = butter(n,Wn,'ftype')
[b,a] = butter(n,Wn)
[b,a] = butter(n,Wn,'ftype')
[A,B,C,D] = butter(n,Wn)
[A,B,C,D] = butter(n,Wn,'ftype')
[z,p,k] = butter(n,Wn,'s')
[z,p,k] = butter(n,Wn,'ftype','s')
[b,a] = butter(n,Wn,'s')
[b,a] = butter(n,Wn,'ftype','s')
[A,B,C,D] = butter(n,Wn,'s')
[A,B,C,D] = butter(n,Wn,'ftype','s')
```

Description

butter designs lowpass, bandpass, highpass, and bandstop digital and analog Butterworth filters. Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall.

Butterworth filters sacrifice rolloff steepness for monotonicity in the pass- and stopbands. Unless the smoothness of the Butterworth filter is needed, an elliptic or Chebyshev filter can generally provide steeper rolloff characteristics with a lower filter order.

Digital Domain

[z,p,k] = butter(n,Wn) designs an order n lowpass digital Butterworth filter with normalized cutoff frequency Wn. It returns the zeros and poles in length n column vectors z and p, and the gain in the scalar k.

[z,p,k] = butter(n,Wn,'ftype') designs a highpass, lowpass, or bandstop filter, where the string 'ftype' is one of the following:

- 'high' for a highpass digital filter with normalized cutoff frequency Wn

- 'low' for a lowpass digital filter with normalized cutoff frequency Wn

- `'stop'` for an order `2*n` bandstop digital filter if `Wn` is a two-element vector, `Wn = [w1 w2]`. The stopband is `w1 < ω < w2`.

Cutoff frequency is that frequency where the magnitude response of the filter is $\sqrt{1/2}$. For `butter`, the normalized cutoff frequency `Wn` must be a number between 0 and 1, where 1 corresponds to the Nyquist frequency, π radians per sample.

If `Wn` is a two-element vector, `Wn = [w1 w2]`, `butter` returns an order `2*n` digital bandpass filter with passband `w1 < ω < w2`.

With different numbers of output arguments, `butter` directly obtains other realizations of the filter. To obtain the transfer function form, use two output arguments as shown below.

Note See "Limitations" on page 3-52 below for information about numerical issues that affect forming the transfer function.

`[b,a] = butter(n,Wn)` designs an order `n` lowpass digital Butterworth filter with normalized cutoff frequency `Wn`. It returns the filter coefficients in length `n+1` row vectors `b` and `a`, with coefficients in descending powers of *z*.

$$H(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \ldots + a(n+1)z^{-n}}$$

`[b,a] = butter(n,Wn,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = butter(n,Wn)` or

`[A,B,C,D] = butter(n,Wn,'ftype')` where A, B, C, and D are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

and *u* is the input, *x* is the state vector, and *y* is the output.

Analog Domain

`[z,p,k] = butter(n,Wn,'s')` designs an order `n` lowpass analog Butterworth filter with angular cutoff frequency `Wn` rad/s. It returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`. `butter`'s angular cutoff frequency `Wn` must be greater than 0 rad/s.

If `Wn` is a two-element vector with `w1 < w2`, `butter(n,Wn,'s')` returns an order `2*n` bandpass analog filter with passband $w1 < \omega < w2$.

`[z,p,k] = butter(n,Wn,'`*ftype*`','s')` designs a highpass, lowpass, or bandstop filter using the *ftype* values described above.

With different numbers of output arguments, `butter` directly obtains other realizations of the analog filter. To obtain the transfer function form, use two output arguments as shown below:

`[b,a] = butter(n,Wn,'s')` designs an order `n` lowpass analog Butterworth filter with angular cutoff frequency `Wn` rad/s. It returns the filter coefficients in the length `n+1` row vectors `b` and `a`, in descending powers of *s,* derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \ldots + b(n+1)}{s^n + a(2)s^{n-1} + \ldots + a(n+1)}$$

`[b,a] = butter(n,Wn,'`*ftype*`','s')` designs a highpass, lowpass, or bandstop filter using the *ftype* values described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = butter(n,Wn,'s')` or

`[A,B,C,D] = butter(n,Wn,'`*ftype*`','s')` where A, B, C, and D are

$$x = Ax + Bu$$
$$y = Cx + Du$$

and u is the input, x is the state vector, and y is the output.

Examples   **Highpass Filter**

For data sampled at 1000 Hz, design a 9th-order highpass Butterworth filter with cutoff frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[z,p,k] = butter(9,300/500,'high');
[sos,g] = zp2sos(z,p,k);       % Convert to SOS form
Hd = dfilt.df2tsos(sos,g);     % Create a dfilt object
h = fvtool(Hd);                % Plot magnitude response
set(h,'Analysis','freq')       % Display frequency response
```

butter

Limitations   In general, you should use the [z,p,k] syntax to design IIR filters. To analyze or implement your filter, you can then use the [z,p,k] output with zp2sos and an sos dfilt structure. For higher order filters (possibly starting as low as order 8), numerical problems due to roundoff errors may occur when forming the transfer function using the [b,a] syntax. The following example illustrates this limitation:

```
n = 6; Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer Function design
[b,a] = butter(n,Wn,ftype);
h1=dfilt.df2(b,a);        % This is an unstable filter.

% Zero-Pole-Gain design
[z, p, k] = butter(n,Wn,ftype);
[sos,g]=zp2sos(z,p,k);
h2=dfilt.df2sos(sos,g);

% Plot and compare the results
hfvt=fvtool(h1,h2,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```
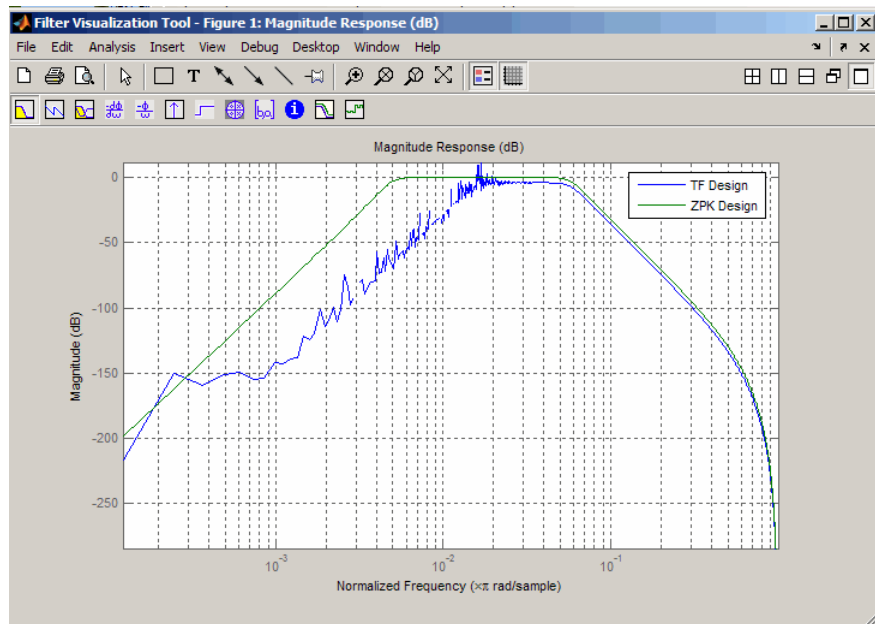
Algorithms   butter uses a five-step algorithm:

1 It finds the lowpass analog prototype poles, zeros, and gain using the buttap function.

2 It converts the poles, zeros, and gain into state-space form.

3 It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.

4 For digital filter design, butter uses bilinear to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at Wn or w1 and w2.

5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also   besself | buttap | buttord | cheby1 | cheby2 | ellip | maxflat

Purpose   Butterworth filter order and cutoff frequency

Syntax   `[n,Wn] = buttord(Wp,Ws,Rp,Rs)`
`[n,Wn] = buttord(Wp,Ws,Rp,Rs,'s')`

Description   buttord calculates the minimum order of a digital or analog Butterworth filter required to meet a set of filter design specifications.

Digital Domain

`[n,Wn] = buttord(Wp,Ws,Rp,Rs)` returns the lowest order, n, of the digital Butterworth filter with no more than Rp dB of passband ripple and at least Rs dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies, Wn, is also returned. Use the output arguments n and Wn in butter.

Choose the input arguments to specify the stopband and passband according to the following table.

Description of Stopband and Passband Filter Parameters

Parameter	Description
Wp	Passband corner frequency Wp, the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π radians per sample.
Ws	Stopband corner frequency Ws, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
Rp	Passband ripple in decibels.
Rs	Stopband attenuation in decibels. This value is the number of decibels the stopband is down from the passband.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	Wp < Ws, both scalars	(Ws,1)	(0,Wp)
Highpass	Wp > Ws, both scalars	(0,Ws)	(Wp,1)
Bandpass	The interval specified by Ws contains the one specified by Wp (Ws(1) < Wp(1) < Wp(2) < Ws(2)).	(0,Ws(1)) and (Ws(2),1)	(Wp(1),Wp(2))
Bandstop	The interval specified by Wp contains the one specified by Ws (Wp(1) < Ws(1) < Ws(2) < Wp(2)).	(Ws(1),Ws(2))	(0,Wp(1)) and (Wp(2),1)

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

Analog Domain

[n,Wn] = buttord(Wp,Ws,Rp,Rs,'s') finds the minimum order n and cutoff frequencies Wn for an analog Butterworth filter. You specify the frequencies Wp and Ws similar those described in the Description of Stopband and Passband Filter Parameters on page 3-55 table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use buttord for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications on page 3-56 table above.

Examples   **Example 1**

For data sampled at 1000 Hz, design a lowpass filter with no more than 3 dB of ripple in the passband from 0 to 40 Hz, and at least 60 dB of attenuation in the stopband. Plot the filter's frequency response.

```
Wp = 40/500; Ws = 150/500;
[n,Wn] = buttord(Wp,Ws,3,60);
% Returns n = 5; Wn=0.0810;
[b,a] = butter(n,Wn);
freqz(b,a,512,1000);
title('n=5 Butterworth Lowpass Filter')
```
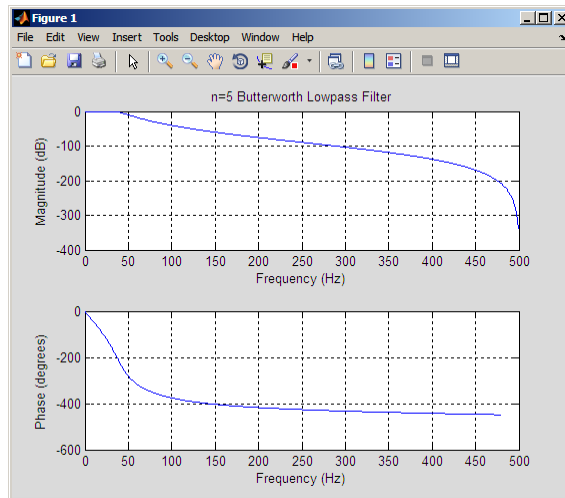

Example 2

Design a bandpass filter with a passband from 60 to 200 Hz with at most 3 dB of passband ripple and at least 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Wn] = buttord(Wp,Ws,Rp,Rs);
```

buttord

```
% Returns n =16; Wn =[0.1198 0.4005];
[b,a] = butter(n,Wn);
freqz(b,a,128,1000)
title('n=16 Butterworth Bandpass Filter')
```
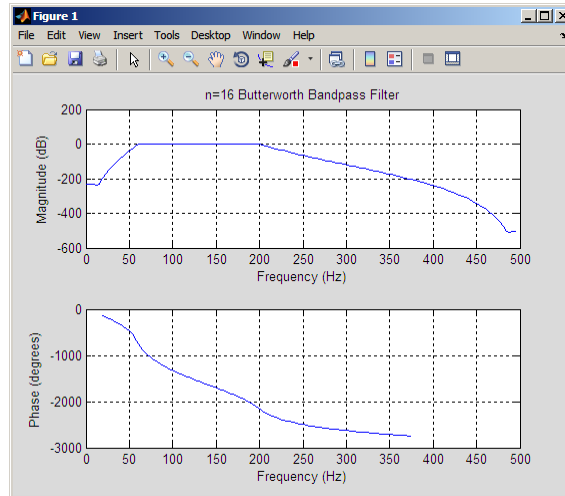

Algorithms buttord's order prediction formula is described in [1]. It operates in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before estimating the order and natural frequency, and then converts back to the *z*-domain.

buttord initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for lowpass and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

References [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 227.

See Also butter | cheb1ord | cheb2ord | ellipord | kaiserord

Purpose   Complex cepstral analysis

Syntax   xhat = cceps(x)
                [xhat,nd] = cceps(x)
                [xhat,nd,xhat1] = cceps(x)
                [...] = cceps(x,n)

Description Cepstral analysis is a nonlinear signal processing technique that is
                applied most commonly in speech processing and homomorphic filtering
                [1].

Note cceps only works on real data.

---

xhat = cceps(x) returns the complex cepstrum of the real data
sequence x using the Fourier transform. The input is altered, by the
application of a linear phase term, to have no phase discontinuity at ±π
radians. That is, it is circularly shifted (after zero padding) by some
samples, if necessary, to have zero phase at π radians.

[xhat,nd] = cceps(x) returns the number of samples nd of (circular)
delay added to x prior to finding the complex cepstrum.

[xhat,nd,xhat1] = cceps(x) returns a second complex cepstrum
xhat1 computed using an alternative factorization algorithm[1][2]. This
method can be applied only to finite duration signals. See the Algorithm
section below for a comparison of the Fourier and factorization methods
of computing the complex cepstrum.

[...] = cceps(x,n) zero pads x to length n and returns the length
n complex cepstrum of x.

Algorithms cceps is an implementation of algorithm 7.1 in [3]. A lengthy Fortran
                program reduces to these three lines of MATLAB code, which compose
                the core of cceps:

                ```
                h = fft(x);
                ```

```
logh = log(abs(h)) + sqrt(-1)*rcunwrap(angle(h));
y = real(ifft(logh));
```

Note rcunwrap in the above code segment is a special version of unwrap that subtracts a straight line from the phase. rcunwrap is a local function within cceps and is not available for use from the MATLAB command line.

The following table lists the pros and cons of the Fourier and factorization algorithms.

Algorithm	Pros	Cons
Fourier	Can be used for any signal.	Requires phase unwrapping. Output is aliased.
Factorization	Does not require phase unwrapping. No aliasing	Can be used only for short duration signals. Input signal must have an all-zero Z-transform with no zeros on the unit circle.

In general, you cannot use the results of these two algorithms to verify each other. You can use them to verify each other only when the first element of the input data is positive, the Z-transform of the data sequence has only zeros, all of these zeros are inside the unit circle, and the input data sequence is long (or padded with zeros).

Examples   The following example uses cceps to show an echo.

```
Fs = 100;
t = 0:1/Fs:1.27;

% 45Hz sine sampled at 100Hz
s1 = sin(2*pi*45*t);
```

```
% Add an echo with half the amplitude and 0.2 second later
s2 = s1 + 0.5*[zeros(1,20) s1(1:108)];

c = cceps(s2);
plot(t,c)
```

Notice the echo at 0.2 second.
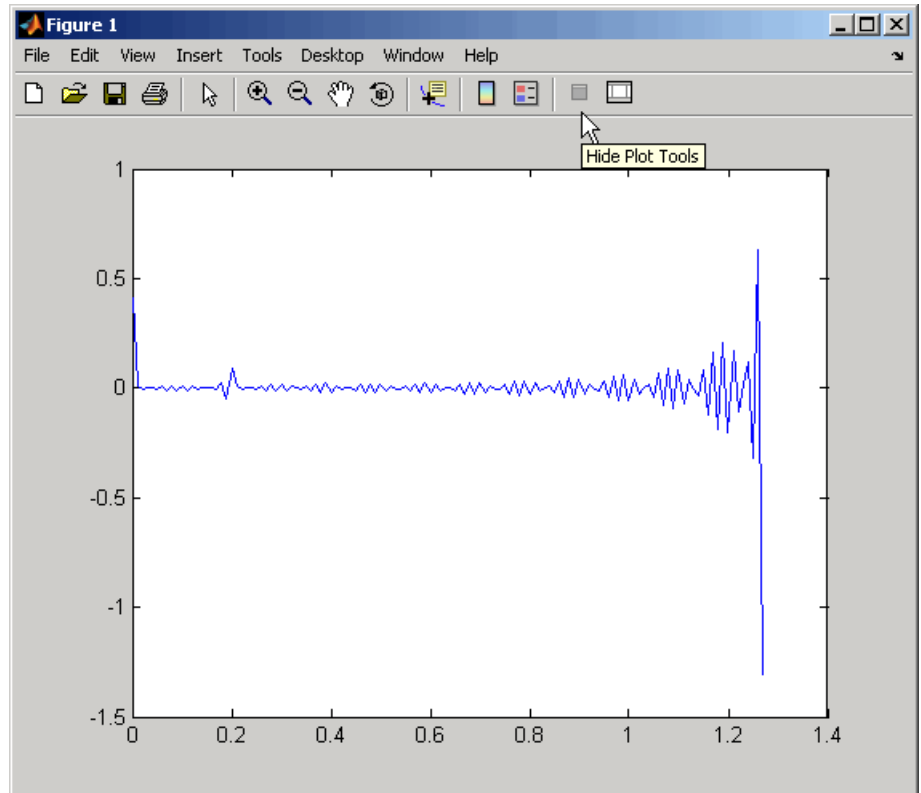
References   [1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing.* Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 788-789.

cceps

[2] Steiglitz, K., and B. Dickinson. "Computation of the complex cepstrum by factorization of the Z-transform" in *Proc. Int. Conf. ASSP.* 1977, pp. 723–726.

[3] *IEEE Programs for Digital Signal Processing.* IEEE Press. New York: John Wiley & Sons, 1979.

See Also   icceps | hilbert | rceps | unwrap

Purpose   Modulo-N circular convolution

Syntax   c = cconv(a,b,n)
                 c = cconv(gpuArrayA,gpuArrayB,n)

Description Circular convolution is used to convolve two discrete Fourier transform
                 (DFT) sequences. For very long sequences, circular convolution may
                 be faster than linear convolution.

                 c = cconv(a,b,n) circularly convolves vectors a and b. n is
                 the length of the resulting vector. If you omit n, it defaults to
                 length(a)+length(B)-1. When n = length(a)+length(B)-1,
                 the circular convolution is equivalent to the linear convolution
                 computed with conv. You can also use cconv to compute the circular
                 cross-correlation of two sequences (see the example below).

                 c = cconv(gpuArrayA,gpuArrayB,n) returns the circular convolution
                 of the input vectors of class gpuArray. See "Using gpuArray"
                 for details on gpuArray objects. Using cconv with gpuArray
                 objects requires Parallel Computing Toolbox™ software and a
                 CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See
                 http://www.mathworks.com/products/parallel-computing/requirements.html
                 for details. The output vector, c, is a gpuArray object. See "Circular
                 Convolution using the GPU" on page 3-64 for an example of using the
                 GPU to compute the circular convolution.

Examples   The following example calculates a modulo-4 circular convolution.

```
a = [2 1 2 1];
b = [1 2 3 4];
c = cconv(a,b,4)
% Returns
% c =
%    14    16    14    16
```

                 The following example compares a circular correlation, where n uses the
                 default value, and a linear convolution. The resulting norm is a value

that is virtually zero, which shows that the two convolutions produce virtually the same result.

```
a = [1 2 -1 1];
b = [1 1 2 1 2 2 1 1];
c = cconv(a,b)      % Circular convolution
cref = conv(a,b)    % Linear convolution
norm(c-cref)
```

The following example uses cconv to compute the circular cross-correlation of two sequences. The result is compared to the cross-correlation computed using xcorr.

```
a = [1 2 2 1]+1i;
b = [1 3 4 1]-2*1i;
c = cconv(a,conj(fliplr(b)),7); % Compute using cconv
cref = xcorr(a,b);        % Compute using xcorr
norm(c-cref)
```

Circular Convolution using the GPU

The following example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See http://www.mathworks.com/products/parallel-computing/requirements.html for details.

Create two signals consisting of a 1 kHz sine wave in additive white Gaussian noise. The sampling rate is 10 kHz

```
Fs = 1e4;
t = 0:1/Fs:10-(1/Fs);
x = cos(2*pi*1e3*t)+randn(size(t));
y = sin(2*pi*1e3*t)+randn(size(t));
```

Put x and y on the GPU using gpuArray. Obtain the circular convolution using the GPU.

```
x = gpuArray(x);
y = gpuArray(y);
cirC = cconv(x,y,length(x)+length(y)-1);
```

Compare the result to the linear convolution of x and y.

```
linC = conv(x,y);
norm(linC-cirC,2)
```

Return the circular convolution, cirC, to the MATLAB workspace using gather.

```
cirC = gather(cirC);
```

References   [1] Orfanidis, S.J., *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1996. pp. 524–529.

See Also   conv

cell2sos

Purpose	Convert second-order sections cell array to matrix
Syntax	`m = cell2sos(c)`

Description   `m = cell2sos(c)` changes a 1-by-*L* cell array `c` consisting of 1-by-2 cell arrays into an *L*-by-6 second-order section matrix `m`. Matrix `m` takes the same form as the matrix generated by `tf2sos`. You can use m = cell2sos(c) to invert the results of c = sos2cell(m).

`c` must be a cell array of the form

```
c = { {b1 a1} {b2 a2} ... {bL aL} }
```

where both `bi` and `ai` are row vectors of at most length 3, and i = 1, 2, ..., *L*. The resulting matrix `m` is given by

```
m = [b1 a1;b2 a2; ... ;bL aL]
```

See Also	`sos2cell` | `tf2sos`

Purpose	Complex and nonlinear-phase equiripple FIR filter design

Syntax

```
b = cfirpm(n,f,@fresp)
b = cfirpm(n,f,@fresp,w)
b = cfirpm(n,f,a)
b = cfirpm(n,f,a,w)
b = cfirpm(...,'sym')
b = cfirpm(...,'skip_stage2')
b = cfirpm(..., 'debug')
b = cfirpm(...,{lgrid})
[b,delta] = cfirpm(...)
[b,delta,opt] = cfirpm(...)
```

Description cfirpm allows arbitrary frequency-domain constraints to be specified for the design of a possibly complex FIR filter. The Chebyshev (or minimax) filter error is optimized, producing equiripple FIR filter designs.

b = cfirpm(n,f,@*fresp*) returns a length n+1 FIR filter with the best approximation to the desired frequency response as returned by function *fresp*, which is called by its function handle (@*fresp*). f is a vector of frequency band edge pairs, specified in the range -1 and 1, where 1 corresponds to the normalized Nyquist frequency. The frequencies must be in increasing order, and f must have even length. The frequency bands span f(k) to f(k+1) for k odd; the intervals f(k+1) to f(k+2) for k odd are "transition bands" or "don't care" regions during optimization.

Predefined fresp frequency response functions are included for a number of common filter designs, as described below. For all of the predefined frequency response functions, the symmetry option '*sym*' defaults to 'even' if no negative frequencies are contained in f and d = 0; otherwise '*sym*' defaults to 'none'. (See the '*sym*' option below for details.) For all of the predefined frequency response functions, d specifies a group-delay offset such that the filter response has a group delay of n/2+d in units of the sample interval. Negative values create less delay; positive values create more delay. By default d = 0:

• @lowpass, @highpass, @allpass, @bandpass, @bandstop

These functions share a common syntax, exemplified below by the string `'lowpass'`.

`b = cfirpm(n,f,@lowpass,...)` and

`b = cfirpm(n,f,{@lowpass,d},...)` design a linear-phase (n/2+d delay) filter.

Note For `@bandpass` filters, the first element in the frequency vector must be less than or equal to zero and the last element must be greater than or equal to zero.

- `@multiband` designs a linear-phase frequency response filter with arbitrary band amplitudes.

  `b = cfirpm(n,f,{@multiband,a},...)` and

  `b = cfirpm(n,f,{@multiband,a,d},...)` specify vector `a` containing the desired amplitudes at the band edges in `f`. The desired amplitude at frequencies between pairs of points `f(k)` and `f(k+1)` for `k` odd is the line segment connecting the points `(f(k),a(k))` and `(f(k+1),a(k+1))`.

- `@differentiator` designs a linear-phase differentiator. For these designs, zero-frequency must be in a transition band, and band weighting is set to be inversely proportional to frequency.

  `b = cfirpm(n,f,{@differentiator,fs},...)` and

  `b = cfirpm(n,f,{@differentiator,fs,d},...)` specify the sample rate `fs` used to determine the slope of the differentiator response. If omitted, `fs` defaults to 1.

- `@hilbfilt` designs a linear-phase Hilbert transform filter response. For Hilbert designs, zero-frequency must be in a transition band.

  `b = cfirpm(n,f,@hilbfilt,...)` and

  `b = cfirpm(N,F,{@hilbfilt,d},...)` design a linear-phase (n/2+d delay) Hilbert transform filter.

- `@invsinc` designs a linear-phase inverse-sinc filter response.

  `b = cfirpm(n,f,{@invsinc,a},...)` and

  `b = cfirpm(n,f,{@invsinc,a,d},...)` specify gain `a` for the sinc-function, computed as sinc(a*g), where g contains the optimization grid frequencies normalized to the range [-1,1]. By default, `a=1`. The group-delay offset is `d`, such that the filter response will have a group delay of N/2 + `d` in units of the sample interval, where N is the filter order. Negative values create less delay and positive values create more delay. By default, `d=0`.

`b = cfirpm(n,f,@`*fresp*`,w)` uses the real, non-negative weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f`, so there is exactly one weight per band.

`b = cfirpm(n,f,a)` is a synonym for
`b = cfirpm(n,f,{@multiband,a})`.

`b = cfirpm(n,f,a,w)` applies an optional set of positive weights, one per band, for use during optimization. If `w` is not specified, the weights are set to unity.

`b = cfirpm(...,'`*sym*`')` imposes a symmetry constraint on the impulse response of the design, where '*sym*' may be one of the following:

- `'none'` indicates no symmetry constraint. This is the default if any negative band edge frequencies are passed, or if *fresp* does not supply a default.

- `'even'` indicates a real and even impulse response. This is the default for highpass, lowpass, allpass, bandpass, bandstop, invsinc, and multiband designs.

- `'odd'` indicates a real and odd impulse response. This is the default for Hilbert and differentiator designs.

- `'real'` indicates conjugate symmetry for the frequency response

If any '*sym*' option other than `'none'` is specified, the band edges should be specified only over positive frequencies; the negative frequency region is filled in from symmetry. If a '*sym*' option is not

specified, the *fresp* function is queried for a default setting. Any user-supplied *fresp* function should return a valid '*sym*' string when it is passed the string 'defaults' as the filter order N.

b = cfirpm(...,'skip_stage2') disables the second-stage optimization algorithm, which executes only when cfirpm determines that an optimal solution has not been reached by the standard firpm error-exchange. Disabling this algorithm may increase the speed of computation, but may incur a reduction in accuracy. By default, the second-stage optimization is enabled.

b = cfirpm(..., '*debug*') enables the display of intermediate results during the filter design, where '*debug*' may be one of 'trace', 'plots', 'both', or 'off'. By default it is set to 'off'.

b = cfirpm(...,{lgrid}) uses the integer lgrid to control the density of the frequency grid, which has roughly 2^nextpow2(lgrid*n) frequency points. The default value for lgrid is 25. Note that the {lgrid} argument must be a 1-by-1 cell array.

Any combination of the '*sym*', 'skip_stage2', '*debug*', and {lgrid} options may be specified.

[b,delta] = cfirpm(...) returns the maximum ripple height delta.

[b,delta,opt] = cfirpm(...)   returns a structure opt of optional results computed by cfirpm and contains the following fields.

Field	Description
opt.fgrid	Frequency grid vector used for the filter design optimization
opt.des	Desired frequency response for each point in opt.fgrid
opt.wt	Weighting for each point in opt.fgrid
opt.H	Actual frequency response for each point in opt.fgrid
opt.error	Error at each point in opt.fgrid

Field	Description
`opt.iextr`	Vector of indices into `opt.fgrid` for extremal frequencies
`opt.fextr`	Vector of extremal frequencies

User-definable functions may be used, instead of the predefined frequency response functions for @*fresp*. The function is called from within `cfirpm` using the following syntax

```
[dh,dw] = fresp(n,f,gf,w,p1,p2,...)
```

where:

- `n` is the filter order.

- `f` is the vector of frequency band edges that appear monotonically between -1 and 1, where 1 corresponds to the Nyquist frequency.

- `gf` is a vector of grid points that have been linearly interpolated over each specified frequency band by `cfirpm`. `gf` determines the frequency grid at which the response function must be evaluated. This is the same data returned by `cfirpm` in the `fgrid` field of the `opt` structure.

- `w` is a vector of real, positive weights, one per band, used during optimization. `w` is optional in the call to `cfirpm`; if not specified, it is set to unity weighting before being passed to *fresp*.

- `dh` and `dw` are the desired complex frequency response and band weight vectors, respectively, evaluated at each frequency in grid `gf`.

- `p1`, `p2`, ..., are optional parameters that may be passed to *fresp*.

Additionally, a preliminary call is made to *fresp* to determine the default symmetry property '*sym*'. This call is made using the syntax:

```
sym = fresp('defaults',{n,f,[],w,p1,p2,...})
```

The arguments may be used in determining an appropriate symmetry default as necessary. The function private/lowpass.m may be useful as a template for generating new frequency response functions.

Examples

Example 1

Design a 31-tap, linear-phase, lowpass filter:

```
b = cfirpm(30,[-1 -0.5 -0.4 0.7 0.8 1],@lowpass);
fvtool(b,1)
```

Click the **Magnitude and Phase Response** button.
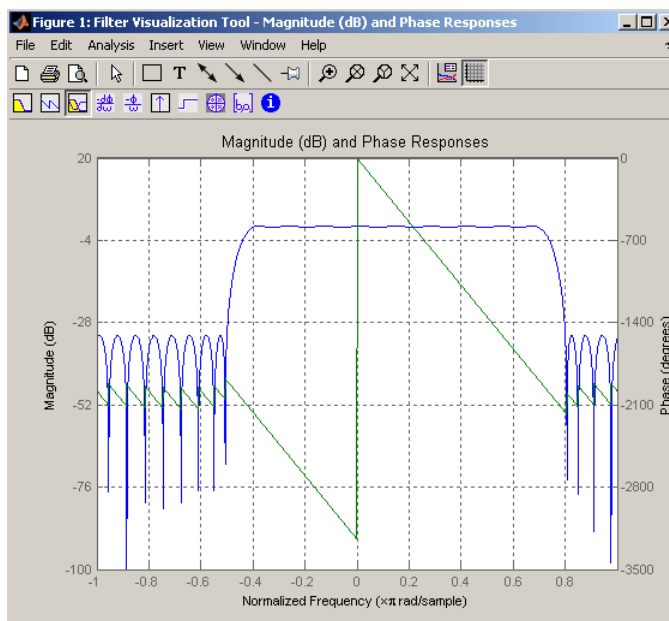
Example 2

Design a nonlinear-phase allpass FIR filter:

```
n = 22;                    % Filter order
```

```
f = [-1 1];                    % Frequency band edges
w = [1 1];                     % Weights for optimization
gf = linspace(-1,1,256);       % Grid of frequency points
d = exp(-1i*pi*gf*n/2 + 1i*pi*pi*sign(gf).*gf.*gf*(4/pi));
                   % Desired frequency response
```

Vector d now contains the complex frequency response that we desire
for the FIR filter computed by cfirpm.

Now compute the FIR filter that best approximates this response:

```
b = cfirpm(n,f,'allpass',w,'real');  % Approximation
freqz(b,1,256,'whole');
subplot(2,1,1); hold on               % Overlay response
plot(pi*(gf+1),20*log10(abs(fftshift(d))),'r--')
subplot(2,1,2); hold on
plot(pi*(gf+1),unwrap(angle(fftshift(d)))*180/pi,'r--')
legend('Approximation','Desired')
```

cfirpm
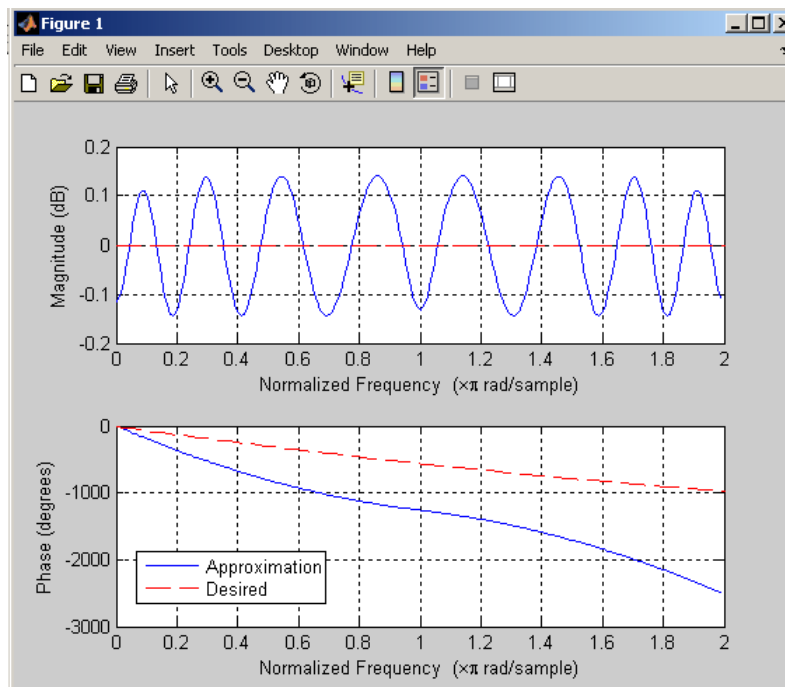
Algorithms   An extended version of the Remez exchange method is implemented for the complex case. This exchange method obtains the optimal filter when the equiripple nature of the filter is restricted to have n+2 extremals. When it does not converge, the algorithm switches to an ascent-descent algorithm that takes over to finish the convergence to the optimal solution. See the references for further details.

References   [1] Karam, L.J., and J.H. McClellan. "Complex Chebyshev Approximation for FIR Filter Design." *IEEE Trans. on Circuits and Systems II,*March 1995. Pgs. 207-216.

[2] Karam, L.J. *Design of Complex Digital FIR Filters in the Chebyshev Sense,* Ph.D. Thesis, Georgia Institute of Technology, March 1995.

[3] Demjanjov, V.F., and V.N. Malozemov. *Introduction to Minimax,* New York: John Wiley & Sons, 1974.

See Also   fir1 | fir2 | firls | firpm | function_handle

cheb1ap

Purpose   Chebyshev Type I analog lowpass filter prototype

Syntax   [z,p,k] = cheb1ap(n,Rp)

Description   [z,p,k] = cheb1ap(n,Rp) returns the poles and gain of an order n
                    Chebyshev Type I analog lowpass filter prototype with Rp dB of ripple
                    in the passband. The function returns the poles in the length n column
                    vector p and the gain in scalar k. z is an empty matrix, because there
                    are no zeros. The transfer function is

$$H(s) = \frac{z(s)}{p(s)} = \frac{k}{(s - p(1))(s - p(2))\ldots(s - p(n))}$$

Chebyshev Type I filters are equiripple in the passband and monotonic
in the stopband. The poles are evenly spaced about an ellipse in the left
half plane. The Chebyshev Type I passband edge angular frequency ω_0
is set to 1.0 for a normalized result. This is the frequency at which the
passband ends and the filter has magnitude response of $10^{-Rp/20}$.

References   [1] Parks, T.W., and C.S. Burrus. *Digital Filter Design,* New York: John
                    Wiley & Sons, 1987. Chapter 7.

See Also   besselap | buttap | cheby1 | cheb2ap | ellipap

Purpose   Chebyshev Type I filter order

Syntax   [n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
                  [n,Wp] = cheb1ord(Wp,Ws,Rp,Rs,'s')

Description   cheb1ord calculates the minimum order of a digital or analog Chebyshev
                 Type I filter required to meet a set of filter design specifications.

Digital Domain

[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs) returns the lowest order n of the
Chebyshev Type I filter that loses no more than Rp dB in the passband
and has at least Rs dB of attenuation in the stopband. The scalar (or
vector) of corresponding cutoff frequencies Wp, is also returned. Use the
output arguments n and Wp with the cheby1 function.

Choose the input arguments to specify the stopband and passband
according to the following table.

Description of Stopband and Passband Filter Parameters

Parameter	Description
Wp	Passband corner frequency Wp, the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π radians per sample.
Ws	Stopband corner frequency Ws, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
Rp	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
Rs	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	Wp < Ws, both scalars	(Ws,1)	(0,Wp)
Highpass	Wp > Ws, both scalars	(0,Ws)	(Wp,1)
Bandpass	The interval specified by Ws contains the one specified by Wp (Ws(1) < Wp(1) < Wp(2) < Ws(2)).	(0,Ws(1)) and (Ws(2),1)	(Wp(1),Wp(2))
Bandstop	The interval specified by Wp contains the one specified by Ws (Wp(1) < Ws(1) < Ws(2) < Wp(2)).	(0,Wp(1)) and (Wp(2),1)	(Ws(1),Ws(2))

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

Analog Domain

[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs,'s') finds the minimum order n and cutoff frequencies Wp for an analog Chebyshev Type I filter. You specify the frequencies Wp and Ws similar to those described in the Description of Stopband and Passband Filter Parameters on page 3-77 table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use cheb1ord for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications on page 3-78 table above.

Examples For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz and at least 60 dB of ripple in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

```
Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
% Returns n = 4 Wp =0.0800
[b,a] = cheby1(n,Rp,Wp);
freqz(b,a,512,1000);
title('n=4 Chebyshev Type I Lowpass Filter')
```
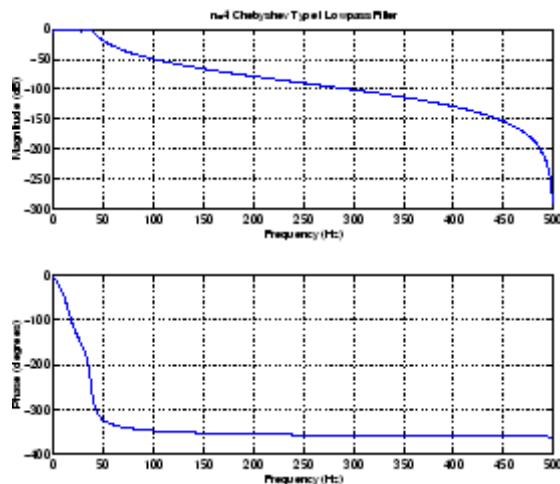

Next design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
% Returns n =7 Wp =[0.1200    0.4000]
[b,a] = cheby1(n,Rp,Wp);
```

```
freqz(b,a,512,1000);
title('n=7 Chebyshev Type I Bandpass Filter')
```
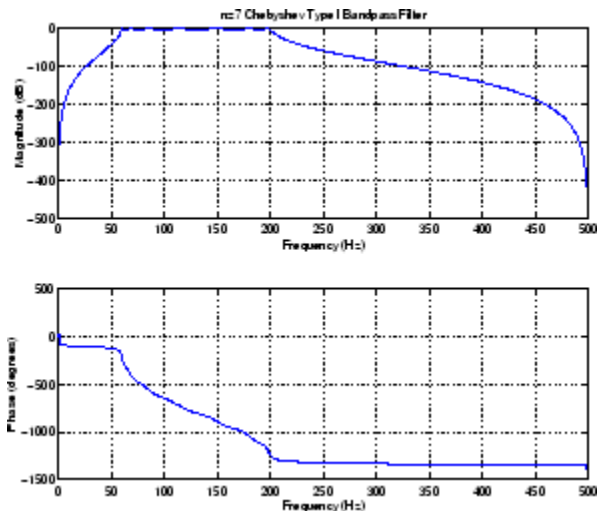

Algorithms   cheb1ord uses the Chebyshev lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before the order and natural frequency estimation process, and then converts them back to the *z*-domain.

cheb1ord initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for low- or highpass filters) or to -1 and 1 rad/s (for bandpass or bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

References   [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

See Also   buttord | cheby1 | cheb2ord | ellipord | kaiserord

Purpose   Chebyshev Type II analog lowpass filter prototype

Syntax   `[z,p,k] = cheb2ap(n,Rs)`

Description   `[z,p,k] = cheb2ap(n,Rs)` finds the zeros, poles, and gain of an order `n`
                   Chebyshev Type II analog lowpass filter prototype with stopband ripple
                   `Rs` dB down from the passband peak value. `cheb2ap` returns the zeros
                   and poles in length `n` column vectors `z` and `p` and the gain in scalar `k`. If
                   `n` is odd, `z` is length `n-1`. The transfer function is

$$H(s) = \frac{z(s)}{p(s)} = k\frac{(s-z(1))(s-z(2))\cdots(s-z(n))}{(s-p(1))(s-p(2))\cdots(s-p(n))}$$

                   Chebyshev Type II filters are monotonic in the passband and equiripple
                   in the stopband. The pole locations are the inverse of the pole locations
                   of `cheb1ap`, whose poles are evenly spaced about an ellipse in the left
                   half plane. The Chebyshev Type II stopband edge angular frequency ω_0
                   is set to 1 for a normalized result. This is the frequency at which the
                   stopband begins and the filter has magnitude response of $10^{-Rs/20}$.

Algorithms   Chebyshev Type II filters are sometimes called *inverse Chebyshev* filters
                   because of their relationship to Chebyshev Type I filters. The `cheb2ap`
                   function is a modification of the Chebyshev Type I prototype algorithm:

                   1 `cheb2ap` replaces the frequency variable ω with $1/\omega$, turning the
                         lowpass filter into a highpass filter while preserving the performance
                         at $\omega = 1$.

                   2 `cheb2ap` subtracts the filter transfer function from unity.

References   [1] Parks, T.W., and C.S. Burrus. *Digital Filter Design,* New York:
                   John Wiley & Sons, 1987. Chapter 7.

See Also   `besselap` | `buttap` | `cheb1ap` | `cheby2` | `ellipap`

cheb2ord

Purpose	Chebyshev Type II filter order
Syntax	[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs) [n,Ws] = cheb2ord(Wp,Ws,Rp,Rs,'s')
Description	cheb2ord calculates the minimum order of a digital or analog Chebyshev Type II filter required to meet a set of filter design specifications.

Digital Domain

[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs) returns the lowest order n of the Chebyshev Type II filter that loses no more than Rp dB in the passband and has at least Rs dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies Ws, is also returned. Use the output arguments n and Ws in cheby2.

Choose the input arguments to specify the stopband and passband according to the following table.

Description of Stopband and Passband Filter Parameters

Parameter	Description
Wp	Passband corner frequency Wp, the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π radians per sample.
Ws	Stopband corner frequency Ws, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
Rp	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
Rs	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	Wp < Ws, both scalars	(Ws,1)	(0,Wp)
Highpass	Wp > Ws, both scalars	(0,Ws)	(Wp,1)
Bandpass	The interval specified by Ws contains the one specified by Wp (Ws(1) < Wp(1) < Wp(2) < Ws(2)).	(0,Ws(1)) and (Ws(2),1)	(Wp(1),Wp(2))
Bandstop	The interval specified by Wp contains the one specified by Ws (Wp(1) < Ws(1) < Ws(2) < Wp(2)).	(0,Wp(1)) and (Wp(2),1)	(Ws(1),Ws(2))

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

Analog Domain

[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs,'s') finds the minimum order n and cutoff frequencies Ws for an analog Chebyshev Type II filter. You specify the frequencies Wp and Ws similar to those described in the Description of Stopband and Passband Filter Parameters on page 3-82 table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use cheb2ord for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications on page 3-83 table above.

Examples

Example 1

For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz, and at least 60 dB

of attenuation in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

```
Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
% Returns n =4 Ws =0.3000
[b,a] = cheby2(n,Rs,Ws);
freqz(b,a,512,1000);
title('n=4 Chebyshev Type II Lowpass Filter')
```
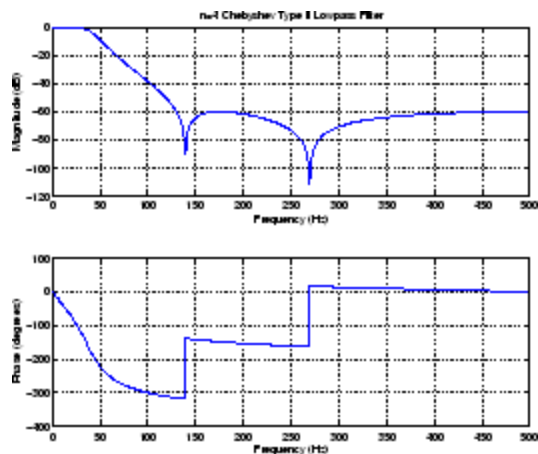

Example 2

Next design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
% Returns n =7 Ws =[0.1000 0.5000]
[b,a] = cheby2(n,Rs,Ws);
freqz(b,a,512,1000)
```

```
title('n=7 Chebyshev Type II Bandpass Filter')
```


n=7 Chebyshev Type II Bandpass Filter
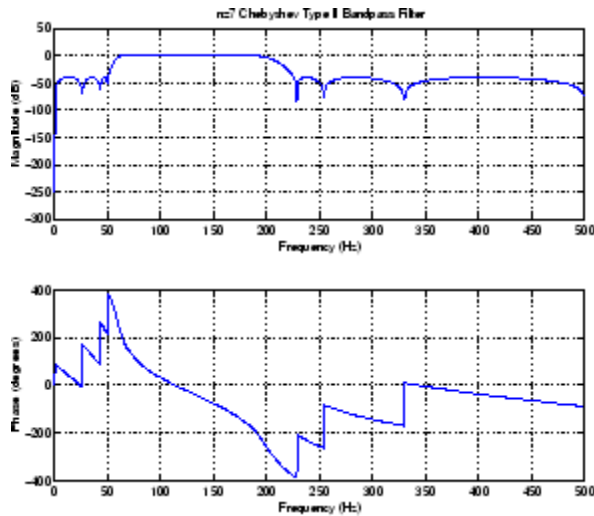
Algorithms   cheb2ord uses the Chebyshev lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before the order and natural frequency estimation process, and then converts them back to the *z*-domain.

cheb2ord initially develops a lowpass filter prototype by transforming the stopband frequencies of the desired filter to 1 rad/s (for low- and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the passband specification.

References   [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

See Also   buttord | cheb1ord | cheby2 | ellipord | kaiserord

chebwin

Purpose	Chebyshev window

Syntax   `w = chebwin(L,r)`

Description   `w = chebwin(L,r)` returns the column vector w containing the length L Chebyshev window whose Fourier transform sidelobe magnitude is r dB below the mainlobe magnitude. The default value for r is 100.0 dB.

Note If you specify a one-point window (set L=1), the value 1 is returned.

Examples   Create a 64-point Chebyshev window with 100 dB of sidelobe attenuation and display the result using WVTool:

```
L=64;
wvtool(chebwin(L))
```
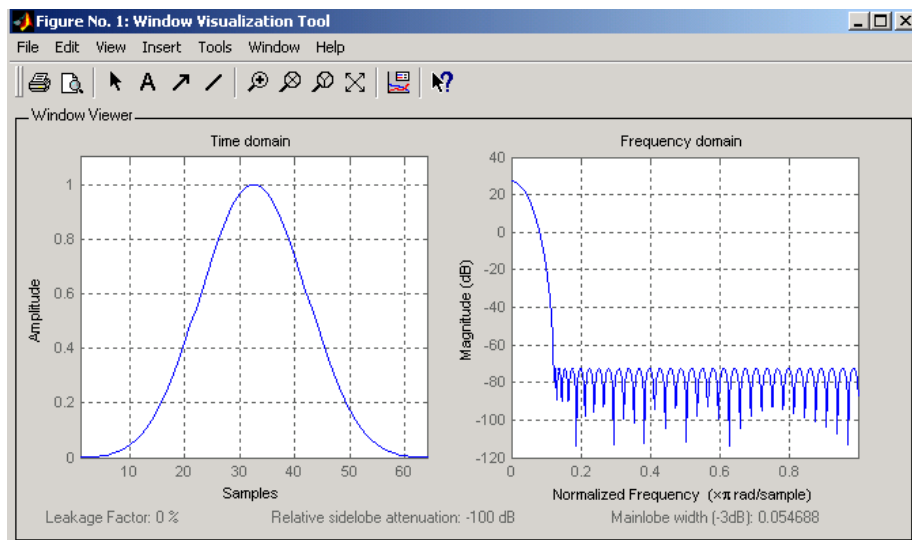
Algorithms   An artifact of the equiripple design method used in chebwin is the presence of impulses at the endpoints of the time-domain response. This is due to the constant-level sidelobes in the frequency domain. The magnitude of the impulses are on the order of the size of the spectral sidelobes. If the sidelobes are large, the effect at the endpoints may be significant. For more information on this effect, see [2].

References   [1] *IEEE Programs for Digital Signal Processing.* IEEE Press. New York: John Wiley & Sons, 1979. Program 5.2.

[2] Harris, Fredric J. *Multirate Signal Processing for Communication Systems,* New Jersey: Prentice Hall PTR, 2004, pp. 60-64.

See Also   gausswin | kaiser | tukeywin | window | wintool | wvtool

cheby1

Purpose   Chebyshev Type I filter design (passband ripple)

Syntax
```
[z,p,k] = cheby1(n,R,Wp)
[z,p,k] = cheby1(n,R,Wp,'ftype')
[b,a] = cheby1(n,R,Wp)
[b,a] = cheby1(n,R,Wp,'ftype')
[A,B,C,D] = cheby1(n,R,Wp)
[A,B,C,D] = cheby1(n,R,Wp,'ftype')
[z,p,k] = cheby1(n,R,Wp,'s')
[z,p,k] = cheby1(n,R,Wp,'ftype','s')
[b,a] = cheby1(n,R,Wp,'s')
[b,a] = cheby1(n,R,Wp, 'ftype','s')
[A,B,C,D] = cheby1(n,R,Wp,'s')
[A,B,C,D] = cheby1(n,R,Wp,'ftype','s')
```

Description   cheby1 designs lowpass, bandpass, highpass, and bandstop digital and
analog Chebyshev Type I filters. Chebyshev Type I filters are equiripple
in the passband and monotonic in the stopband. Type I filters roll off
faster than type II filters, but at the expense of greater deviation from
unity in the passband.

Digital Domain

[z,p,k] = cheby1(n,R,Wp) designs an order n Chebyshev lowpass
digital Chebyshev filter with normalized passband edge frequency Wp
and R dB of peak-to-peak ripple in the passband. It returns the zeros
and poles in length n column vectors z and p and the gain in the scalar k.

[z,p,k] = cheby1(n,R,Wp,'ftype') designs a highpass, lowpass, or
bandstop filter, where the string 'ftype' is one of the following:

- 'high' for a highpass digital filter with normalized passband edge
  frequency Wp

- 'low' for a lowpass digital filter with normalized passband edge
  frequency Wp

- 'stop' for an order 2*n bandstop digital filter if Wp is a two-element
  vector, Wp = [w1 w2]. The stopband is w1 < ω < w2.

Normalized passband edge frequency is the frequency at which the magnitude response of the filter is equal to -R dB. For cheby1, the normalized passband edge frequency Wp is a number between 0 and 1, where 1 corresponds to half the sample rate, π radians per sample. Smaller values of passband ripple R lead to wider transition widths (shallower rolloff characteristics).

If Wp is a two-element vector, Wp = [w1 w2], cheby1 returns an order 2*n bandpass filter with passband w1 < ω < w2.

With different numbers of output arguments, cheby1 directly obtains other realizations of the filter. To obtain the transfer function form, use two output arguments as shown below.

Note See "Limitations" on page 3-92 for information about numerical issues that affect forming the transfer function.

[b,a] = cheby1(n,R,Wp) designs an order n Chebyshev lowpass digital Chebyshev filter with normalized passband edge frequency Wp and R dB of peak-to-peak ripple in the passband. It returns the filter coefficients in the length n+1 row vectors b and a, with coefficients in descending powers of *z*.

$$H(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \ldots + a(n+1)z^{-n}}$$

[b,a] = cheby1(n,R,Wp,'*ftype*') designs a highpass, lowpass, or bandstop filter, where the string '*ftype*' is 'high', 'low', or 'stop', as described above.

To obtain state-space form, use four output arguments as shown below:

[A,B,C,D] = cheby1(n,R,Wp) or

[A,B,C,D] = cheby1(n,R,Wp,'*ftype*') where A, B, C, and D are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

and u is the input, x is the state vector, and y is the output.

Analog Domain

`[z,p,k] = cheby1(n,R,Wp,'s')` designs an order `n` lowpass analog Chebyshev Type I filter with angular passband edge frequency `Wp` rad/s. It returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`.

Angular passband edge frequency is the frequency at which the magnitude response of the filter is `-R` dB. For `cheby1`, the angular passband edge frequency `Wp` must be greater than 0 rad/s.

If `Wp` is a two-element vector `Wp = [w1 w2]` with `w1 < w2`, then `cheby1(n,R,Wp,'s')` returns an order `2*n` bandpass analog filter with passband `w1 < ω< w2`.

`[z,p,k] = cheby1(n,R,Wp,'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is `'high'`, `'low'`, or `'stop'`, as described above.

You can supply different numbers of output arguments for `cheby1` to directly obtain other realizations of the analog filter. To obtain the transfer function form, use two output arguments as shown below.

`[b,a] = cheby1(n,R,Wp,'s')` designs an order `n` lowpass analog Chebyshev Type I filter with angular passband edge frequency `Wp` rad/s. It returns the filter coefficients in length `n+1` row vectors `b` and `a`, in descending powers of *s*, derived from the transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \ldots + b(n+1)}{s^n + a(2)s^{n-1} + \ldots + a(n+1)}$$

`[b,a] = cheby1(n,R,Wp, 'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = cheby1(n,R,Wp,'s')` or

`[A,B,C,D] = cheby1(n,R,Wp,'`*ftype*`','s')` where A, B, C, and D are defined as

$$x = Ax + Bu$$
$$y = Cx + Du$$

and *u* is the input, *x* is the state vector, and *y* is the output.

Examples   **Lowpass Filter**

For data sampled at 1000 Hz, design a 9th-order lowpass Chebyshev Type I filter with 0.5 dB of ripple in the passband and a passband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[z,p,k] = cheby1(9,0.5,300/500);
[sos,g] = zp2sos(z,p,k);   % Convert to SOS form
Hd = dfilt.df2tsos(sos,g);  % Create a dfilt object
h = fvtool(Hd)       % Plot magnitude response
set(h,'Analysis','freq')   % Display frequency response
```

The frequency response of the filter is

```
freqz(b,a,512,1000)
```
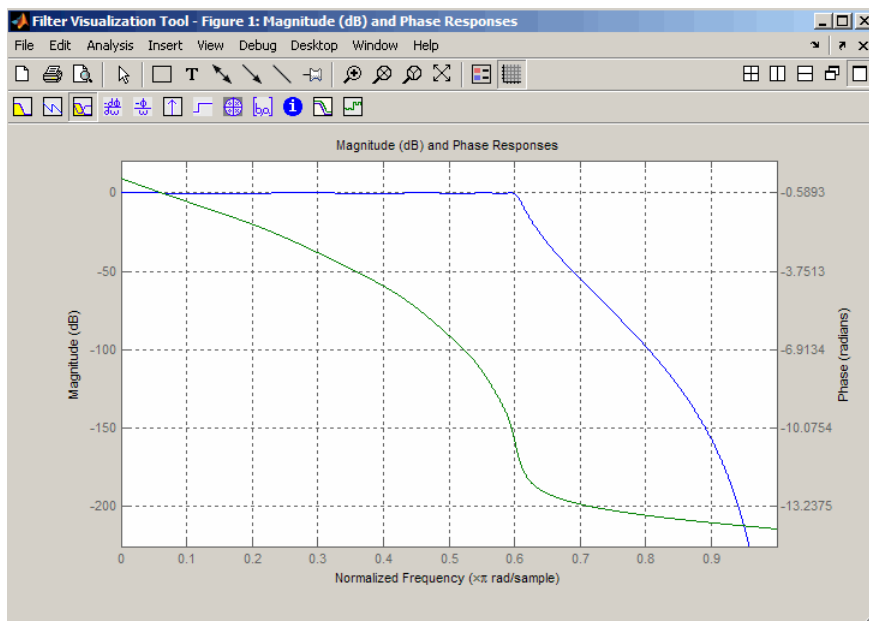
Limitations   In general, you should use the [z,p,k] syntax to design IIR filters. To analyze or implement your filter, you can then use the [z,p,k] output with zp2sos and an sos dfilt structure. For higher order filters (possibly starting as low as order 8), numerical problems due to roundoff errors may occur when forming the transfer function using the [b,a] syntax. The following example illustrates this limitation:

```
n = 6;
r = 0.1;
Wn = ([2.5e6 29e6]/500e6);
ftype = 'bandpass';

% Transfer Function design
[b,a] = cheby1(n,r,Wn,ftype);
h1=dfilt.df2(b,a);        % This is an unstable filter.
```

```
% Zero-Pole-Gain design
[z, p, k] = cheby1(n,r, Wn,ftype);
[sos,g]=zp2sos(z,p,k);
h2=dfilt.df2sos(sos,g);

% Plot and compare the results
hfvt=fvtool(h1,h2,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```
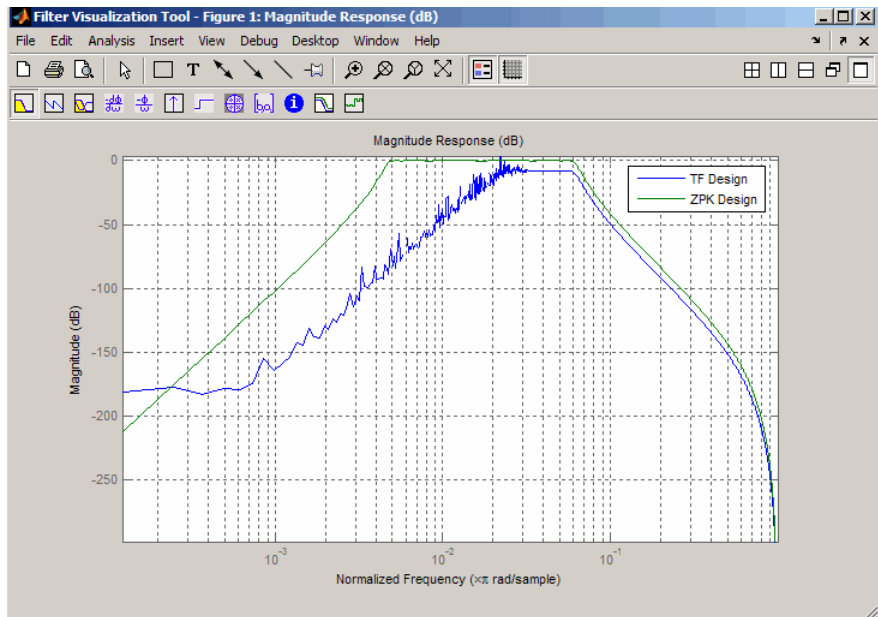

Algorithms   cheby1 uses a five-step algorithm:

1 It finds the lowpass analog prototype poles, zeros, and gain using the cheb1ap function.

2 It converts the poles, zeros, and gain into state-space form.

3 It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.

4 For digital filter design, `cheby1` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wp` or `w1` and `w2`.

5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also   `besself` | `butter` | `cheb1ap` | `cheb1ord` | `cheby2` | `ellip`

Purpose   Chebyshev Type II filter design (stopband ripple)

Syntax   ```
[z,p,k] = cheby2(n,R,Wst)
[z,p,k] = cheby2(n,R,Wst,'ftype')
[b,a] = cheby2(n,R,Wst)
[b,a] = cheby2(n,R,Wst,'ftype')
[A,B,C,D] = cheby2(n,R,Wst)
[A,B,C,D] = cheby2(n,R,Wst,'ftype')
[z,p,k] = cheby2(n,R,Wst,'s')
[z,p,k] = cheby2(n,R,Wst,'ftype','s')
[b,a] = cheby2(n,R,Wst,'s')
[b,a] = cheby2(n,R,Wst,'ftype','s')
[A,B,C,D] = cheby2(n,R,Wst,'s')
[A,B,C,D] = cheby2(n,R,Wst,'ftype','s')
```

**Description**    cheby2 designs lowpass, highpass, bandpass, and bandstop digital
and analog Chebyshev Type II filters. Chebyshev Type II filters are
monotonic in the passband and equiripple in the stopband. Type II
filters do not roll off as fast as type I filters, but are free of passband
ripple.

### Digital Domain

[z,p,k] = cheby2(n,R,Wst) designs an order n lowpass digital
Chebyshev Type II filter with normalized stopband edge frequency
Wst and stopband ripple R dB down from the peak passband value. It
returns the zeros and poles in length n column vectors z and p and the
gain in the scalar k.

*Normalized stopband edge frequency* is the beginning of the stopband,
where the magnitude response of the filter is equal to -R dB. For
cheby2, the normalized stopband edge frequency Wst is a number
between 0 and 1, where 1 corresponds to half the sample rate. Larger
values of stopband attenuation R lead to wider transition widths
(shallower rolloff characteristics).

If Wst is a two-element vector, Wst = [w1 w2], cheby2 returns an order
2*n bandpass filter with passband w1 < ω < w2.

[z,p,k] = cheby2(n,R,Wst,'*ftype*')designs a highpass, lowpass, or bandstop filter, where the string '*ftype*' is one of the following:

- 'high' for a highpass digital filter with normalized stopband edge frequency Wst

- 'low' for a lowpass digital filter with normalized stopband edge frequency Wst

- 'stop' for an order 2*n bandstop digital filter if Wst is a two-element vector, Wst = [w1 w2]. The stopband is w1 < ω < w2.

With different numbers of output arguments, cheby2 directly obtains other realizations of the filter. To obtain the transfer function form, use two output arguments as shown below.

---

**Note** See "Limitations" on page 3-99 below for information about numerical issues that affect forming the transfer function.

---

[b,a] = cheby2(n,R,Wst) designs an order n lowpass digital Chebyshev Type II filter with normalized stopband edge frequency Wst and stopband ripple R dB down from the peak passband value. It returns the filter coefficients in the length n+1 row vectors b and a, with coefficients in descending powers of *z*.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

[b,a] = cheby2(n,R,Wst,'*ftype*') designs a highpass, lowpass, or bandstop filter, where the string '*ftype*' is 'high', 'low', or 'stop', as described above.

To obtain state-space form, use four output arguments as shown below.

[A,B,C,D] = cheby2(n,R,Wst) or

[A,B,C,D] = cheby2(n,R,Wst,'*ftype*') where A, B, C, and D are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] \quad = Cx[n] + Du[n]$$

and $u$ is the input, $x$ is the state vector, and $y$ is the output.

### Analog Domain

`[z,p,k] = cheby2(n,R,Wst,'s')` designs an order n lowpass analog Chebyshev Type II filter with angular stopband edge frequency Wst rad/s.. It returns the zeros and poles in length n or 2*n column vectors z and p and the gain in the scalar k.

*Angular stopband edge frequency* is the frequency at which the magnitude response of the filter is equal to -R dB. For cheby2, the angular stopband edge frequency Wst must be greater than 0 rad/s.

If Wst is a two-element vector Wst = [w1 w2] with w1 < w2, then `cheby2(n,R,Wst,'s')` returns an order 2*n bandpass analog filter with passband w1 < ω< w2.

`[z,p,k] = cheby2(n,R,Wst,'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string '*ftype*' is 'high', 'low', or 'stop', as described above.

With different numbers of output arguments, cheby2 directly obtains other realizations of the analog filter. To obtain the transfer function form, use two output arguments as shown below:

`[b,a] = cheby2(n,R,Wst,'s')` designs an order n lowpass analog Chebyshev Type II filter with angular stopband edge frequency Wst rad/s.. It returns the filter coefficients in the length n+1 row vectors b and a, with coefficients in descending powers of *s*, derived from the transfer function.

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

`[b,a] = cheby2(n,R,Wst,'ftype','s')` designs a highpass , lowpass, or bandstop filter, where the string '*ftype*' is 'high', 'low', or 'stop', as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = cheby2(n,R,Wst,'s')` or

`[A,B,C,D] = cheby2(n,R,Wst,'ftype','s')` where A, B, C, and D are

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

and $u$ is the input, $x$ is the state vector, and $y$ is the output.

## Examples

### Lowpass Filter

For data sampled at 1000 Hz, design a ninth-order lowpass Chebyshev Type II filter with stopband attenuation 20 dB down from the passband and a stopband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[z,p,k] = cheby2(9,20,300/500);
[sos,g] = zp2sos(z,p,k);   % Convert to SOS form
Hd = dfilt.df2tsos(sos,g);  % Create a dfilt object
h = fvtool(Hd);       % Plot magnitude response
set(h,'Analysis','freq')   % Display frequency response
```

Limitations   In general, you should use the [z,p,k] syntax to design IIR filters.
To analyze or implement your filter, you can then use the [z,p,k]
output with zp2sos and an sos dfilt structure. For higher order filters
(possibly starting as low as order 8), numerical problems due to roundoff
errors may occur when forming the transfer function using the [b,a]
syntax. The following example illustrates this limitation:

```
n = 6;
r = 80;
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer Function design
[b,a] = cheby2(n,r,Wn,ftype);
h1=dfilt.df2(b,a);      % This is an unstable filter.
```

```
% Zero-Pole-Gain design
[z, p, k] = cheby2(n,r,Wn,ftype);
[sos,g]=zp2sos(z,p,k);
h2=dfilt.df2sos(sos,g);
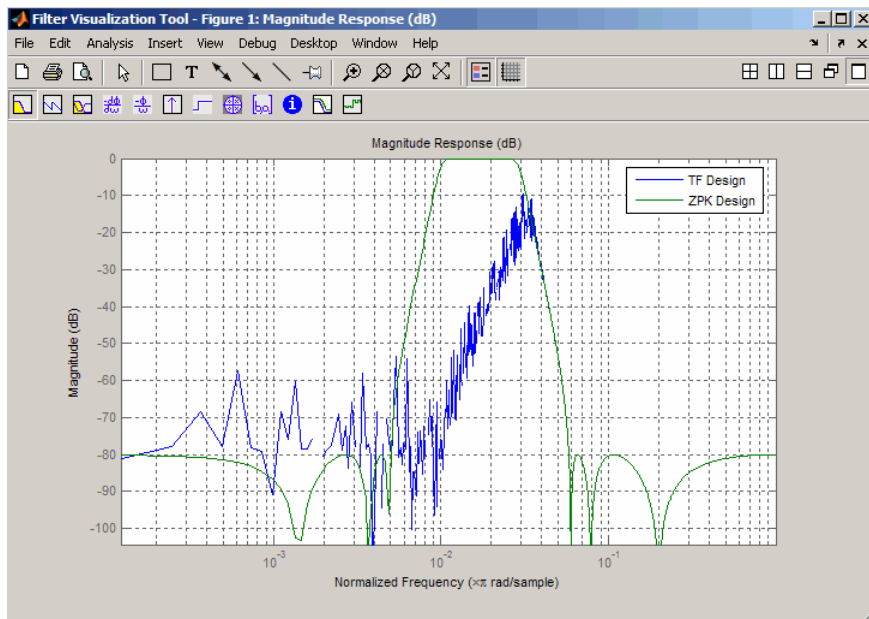
% Plot and compare the results
hfvt=fvtool(h1,h2,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```


Algorithms   cheby2 uses a five-step algorithm:

1 It finds the lowpass analog prototype poles, zeros, and gain using the cheb2ap function.

2 It converts poles, zeros, and gain into state-space form.

3 It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.

4 For digital filter design, cheby2 uses bilinear to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at Wst or w1 and w2.

5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also   besself | butter | cheb2ap | cheb1ord | cheby1 | ellip

chirp

Purpose   Swept-frequency cosine

Syntax   `y = chirp(t,f0,t1,f1)`
                     `y = chirp(t,f0,t1,f1,'`*method*`')`
                     `y = chirp(t,f0,t1,f1,'`*method*`',phi)`
                     `y = chirp(t,f0,t1,f1,'quadratic',phi,'`*shape*`')`

Description   `y = chirp(t,f0,t1,f1)` generates samples of a linear swept-frequency
                     cosine signal at the time instances defined in array `t`, where `f0` is
                     the instantaneous frequency at time 0, and `f1` is the instantaneous
                     frequency at time `t1`. `f0` and `f1` are both in hertz. If unspecified, `f0` is e^{-6}
                     for logarithmic chirp and 0 for all other methods, `t1` is 1, and `f1` is `100`.

                     `y = chirp(t,f0,t1,f1,'`*method*`')` specifies alternative sweep method
                     options, where *method* can be:

                     - `linear`, which specifies an instantaneous frequency sweep $f_i(t)$ given
                       by

                       $$f_i(t) = f_0 + \beta t$$

                       where

                       $$\beta = (f_1 - f_0)/t_1$$

                       and the default value for f_0 is 0. ß ensures that the desired frequency
                       breakpoint f_1 at time t_1 is maintained.

                     - `quadratic`, which specifies an instantaneous frequency sweep $f_i(t)$
                       given by

                       $$f_i(t) = f_0 + \beta t^2$$

                       where

                       $$\beta = (f_1 - f_0)/t_1^2$$

and the default value for f_0 is 0. If $f_0 > f_1$ (downsweep), the default shape is convex. If $f_0 < f_1$ (upsweep), the default shape is concave.

- `logarithmic` specifies an instantaneous frequency sweep $f_i(t)$ given by

$$f_i(t) = f_0 \times \beta^t$$

where

$$\beta = \left(\frac{f_1}{f_0} \right)^{\frac{1}{t_1}}$$

and the default value for f_0 is 1e^{-6}. Both an upsweep ($f_1 > f_0$) and a downsweep ($f_0 > f_1$) of frequency is possible.

Each of the above methods can be entered as `'li'`, `'q'`, and `'lo'`, respectively.

`y = chirp(t,f0,t1,f1,'`*method*`',phi)` allows an initial phase `phi` to be specified in degrees. If unspecified, `phi` is `0`. Default values are substituted for empty or omitted trailing input arguments.

`y = chirp(t,f0,t1,f1,'quadratic',phi,'`*shape*`')` specifies the `shape` of the quadratic swept-frequency signal's spectrogram. `shape` is either `concave` or `convex`, which describes the shape of the parabola in the positive frequency axis. If `shape` is omitted, the default is convex for downsweep ($f_0 > f_1$) and is concave for upsweep ($f_0 < f_1$).

Convex downsweep
shape

Concave upsweep
shape

Examples   **Example 1**

Compute the spectrogram of a chirp with linear instantaneous
frequency deviation:

```
t = 0:0.001:2;            % 2 secs @ 1kHz sample rate
y = chirp(t,0,1,150);     % Start @ DC,
                          %   cross 150Hz at t=1 sec
spectrogram(y,256,250,256,1E3,'yaxis')
```

Example 2

Compute the spectrogram of a chirp with quadratic instantaneous frequency deviation:

```
% –2 secs @ 1kHz sample rate
t = -2:0.001:2;
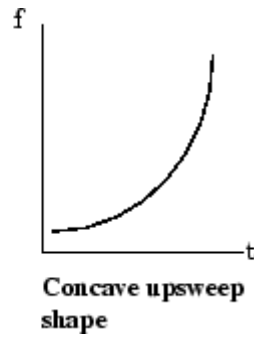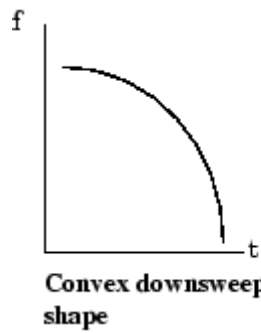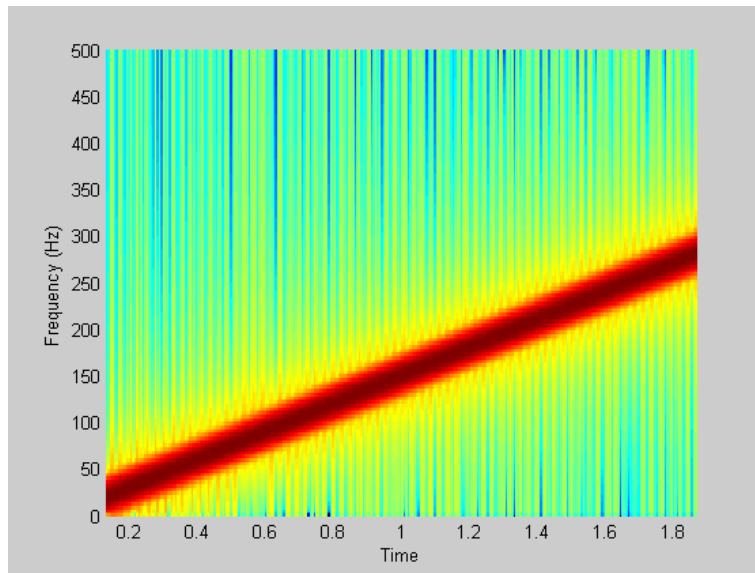
% Start @ 100Hz, cross 200Hz at t=1 sec
y = chirp(t,100,1,200,'quadratic');

spectrogram(y,128,120,128,1E3,'yaxis')
```

Example 3

Compute the spectrogram of a convex quadratic chirp:

```
t = -1:0.001:1;         % +/-1 second @ 1kHz sample rate
fo = 100; f1 = 400;     % Start at 100Hz, go up to 400Hz
y = chirp(t,fo,1,f1,'q',[],'convex');
spectrogram(y,256,200,256,1000,'yaxis')
```

Example 4

Compute the spectrogram of a concave quadratic chirp:

```
t = 0:0.001:1;      % 1 second @ 1kHz sample rate
fo = 100; f1 = 25;  % Start at 100Hz, go down to 25Hz
y = chirp(t,fo,1,f1,'q',[],'concave');
spectrogram(y,hanning(256),128,256,1000,'yaxis')
```

Example 5

Compute the spectrogram of a logarithmic chirp:

```
t = 0:0.001:10;      % 10 seconds @ 1kHz sample rate
fo = 10; f1 = 400;   % Start at 10Hz, go up to 400Hz
y = chirp(t,fo,10,f1,'logarithmic');
spectrogram(y,256,200,256,1000,'yaxis')
```

See Also   cos | diric | gauspuls | pulstran | rectpuls | sawtooth | sin | sinc | square | tripuls

convmtx

Purpose	Convolution matrix
Syntax	`A = convmtx(c,n)` `A = convmtx(r,n)`
Description	A *convolution matrix* is a matrix, formed from a vector, whose inner product with another vector is the convolution of the two vectors. `A = convmtx(c,n)` where `c` is a length `m` column vector returns a matrix `A` of size `(m+n-1)-by-n`. The product of `A` and another column vector `x` of length `n` is the convolution of `c` with `x`. `A = convmtx(r,n)` where `r` is a length `m` row vector returns a matrix `A` of size `n-by-(m+n-1)`. The product of `A` and another row vector `x` of length `n` is the convolution of `r` with `x`.

Examples	Generate a simple convolution matrix:
	
	``h = [1 2 3 2 1];
	convmtx(h,7);```
	
	Note that `convmtx` handles edge conditions by zero padding.
	
	In practice, it is more efficient to compute convolution using
	
	```y = conv(c,x);```
	
	than by using a convolution matrix.
	
	``n = length(x);
	y = convmtx(c,n)*x;```

Algorithms	`convmtx` uses the function `toeplitz` to generate the convolution matrix.
See Also	`conv` | `convn` | `conv2` | `dftmtx`

Purpose   Data matrix for autocorrelation matrix estimation

Syntax   X = corrmtx(x,m)
                X = corrmtx(x,m,'*method*')
                [X,R] = corrmtx(...)

Description X = corrmtx(x,m) returns an (n+m)-by-(m+1) rectangular Toeplitz
                matrix X, such that X'X is a (biased) estimate of the autocorrelation
                matrix for the length n data vector x.

                X = corrmtx(x,m,'*method*') computes the matrix X according to the
                method specified by the string '*method*':

                • 'autocorrelation': (default) X is the (n+m)-by-(m+1) rectangular
                  Toeplitz matrix that generates an autocorrelation estimate for the
                  length n data vector x, derived using *prewindowed* and *postwindowed*
                  data, based on an mth order prediction error model.

                • 'prewindowed': X is the n-by-(m+1) rectangular Toeplitz matrix that
                  generates an autocorrelation estimate for the length n data vector x,
                  derived using *prewindowed* data, based on an mth order prediction
                  error model.

                • 'postwindowed': X is the n-by-(m+1) rectangular Toeplitz matrix that
                  generates an autocorrelation estimate for the length n data vector x,
                  derived using *postwindowed* data, based on an mth order prediction
                  error model.

                • 'covariance': X is the (n-m)-by-(m+1) rectangular Toeplitz matrix
                  that generates an autocorrelation estimate for the length n data
                  vector x, derived using *nonwindowed* data, based on an mth order
                  prediction error model.

                • 'modified': X is the 2(n-m)-by-(m+1) modified rectangular Toeplitz
                  matrix that generates an autocorrelation estimate for the length n
                  data vector x, derived using forward and backward prediction error
                  estimates, based on an mth order prediction error model.

                [X,R] = corrmtx(...) also returns the (m+1)-by-(m+1) autocorrelation
                matrix estimate R, calculated as X'*X.

corrmtx

Examples

```
n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
X=corrmtx(s,12,'mod');
```

Algorithms

The Toeplitz data matrix computed by `corrmtx` depends on the method you select. The matrix determined by the autocorrelation (default) method is given by the following matrix.

$$
X = \begin{bmatrix}
x(1) & \cdots & 0 \\
\vdots & \ddots & \vdots \\
\hline
x(m+1) & \cdots & x(1) \\
\vdots & \ddots & \vdots \\
x(n-m) & \cdots & x(m+1) \\
\vdots & \ddots & \vdots \\
x(n) & \cdots & x(n-m) \\
\hline
0 & \cdots & x(n)
\end{bmatrix}
$$

In this matrix, m is the same as the input argument `m` to `corrmtx`, and n is `length(x)`. Variations of this matrix are used to return the output X of `corrmtx` for each method:

- `'autocorrelation'`: (default) X = X, above.

- `'prewindowed'`: X is the n-by-(m+1) submatrix of X that is given by the portion of X above the lower gray line.

- `'postwindowed'`: X is the n-by-(m+1) submatrix of X that is given by the portion of X below the upper gray line.

- `'covariance'`: X is the (n-m)-by-(m+1) submatrix of X that is given by the portion of X between the two gray lines.

- `'modified'`: X is the 2(n-m)-by-(m+1) matrix X_{mod} shown below.

$$
X_{\mathrm{mod}} = \begin{bmatrix}
x(m+1) & \cdots & x(1) \\
\vdots & \ddots & \vdots \\
x(n-m) & \cdots & x(m+1) \\
\vdots & \ddots & \vdots \\
x(n) & \cdots & x(n-m) \\
x*(1) & \cdots & x*(m+1) \\
\vdots & \ddots & \vdots \\
x*(m+1) & \cdots & x*(n-m) \\
\vdots & \ddots & \vdots \\
x*(n-m) & \cdots & x*(n)
\end{bmatrix}
$$

References [1] Marple, S.L. *Digital Spectral Analysis,* Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 216-223.

See Also peig | pmusic | rooteig | rootmusic | xcorr

cpsd

Purpose   Cross power spectral density

Syntax   Pxy = cpsd(x,y)
                   Pxy = cpsd(x,y,window)
                   Pxy = cpsd(x,y,window,noverlap)
                   [Pxy,W] = cpsd(x,y,window,noverlap,nfft)
                   [Pxy,F] = cpsd(x,y,window,noverlap,nfft,fs)
                   [...] = cpsd(...,'twosided')
                   cpsd(...)

Description   Pxy = cpsd(x,y) estimates the cross power spectral density Pxy of
                   the discrete-time signals x and y using the Welch's averaged, modified
                   periodogram method of spectral estimation. The cross power spectral
                   density is the distribution of power per unit frequency and is defined as

$$P_{xy}(\omega) = \sum_{m=-\infty}^{\infty} R_{xy}(m)e^{-j\omega m}$$

The cross-correlation sequence is defined as

$$R_{xy}(m) = E\{x_{n+m}y^*_n\} = E\{x_n y^*_{n-m}\}$$

where x_n and y_n are jointly stationary random processes, $-\infty < n < \infty$,
and $E\{\cdot\}$ is the expected value operator.

For real x and y, cpsd returns a one-sided CPSD and for complex x or
y, it returns a two-sided CPSD.

cpsd uses the following default values:

Parameter	Description	Default Value
nfft	FFT length which determines the frequencies at which the PSD is estimated	Maximum of 256 or the next power of 2 greater than the length of each section of x or y
	For real x and y, the length of Pxy is (nfft/2+1) if nfft is even or (nfft+1)/2 if nfft is odd. For complex x or y, the length of Pxy is nfft.	
	If nfft is greater than the signal length, the data is zero-padded. If nfft is less than the signal length, the segment is wrapped using datawrap so that the length is equal to nfft.	
fs	Sampling frequency	1
window	Windowing function and number of samples to use for each section	Periodic Hamming window of length to obtain eight equal sections of x and y
noverlap	Number of samples by which the sections overlap	Value to obtain 50% overlap

Note You can use the empty matrix [] to specify the default value for any input argument except x or y. For example, Pxy = cpsd(x,y,[],[],128) uses a Hamming window, default noverlap to obtain 50% overlap, and the specified 128 nfft.

Pxy = cpsd(x,y,window) specifies a windowing function, divides x and y into overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for window, Pxy uses a Hamming window of that length. The x and y vectors are divided into eight equal sections of that length. If the signal cannot be sectioned evenly with 50% overlap, it is truncated.

Pxy = cpsd(x,y,window,noverlap) overlaps the sections of x by noverlap samples. noverlap must be an integer smaller than the length of window.

[Pxy,W] = cpsd(x,y,window,noverlap,nfft) uses the specified FFT length nfft in estimating the CPSD. It also returns W, which is the vector of normalized frequencies (in rad/sample) at which the CPSD is estimated. For real signals, the range of W is [0, pi] when nfft is even and [0, pi) when nfft is odd. For complex signals, the range of W is [0, 2*pi).

[Pxy,F] = cpsd(x,y,window,noverlap,nfft,fs) returns Pxy as a function of frequency and a vector F of frequencies at which the CPSD is estimated. fs is the sampling frequency in Hz. For real signals, the range of F is [0, fs/2] when nfft is even and [0, fs/2) when nfft is odd. For complex signals, the range of F is [0, fs).

[...] = cpsd(...,'twosided') returns the two-sided CPSD of real signals x and y. The length of the resulting Pxy is nfft and its range is [0, 2*pi] if you do not specify fs. If you specify fs, the range is [0,fs). Entering'onesided'for a real signal produces the default. You can place the 'onesided' or 'twosided' string in any position after the noverlap parameter.

cpsd(...) plots the CPSD versus frequency in the current figure window.

Examples   Generate two colored noise signals and plot their CPSD. Specify a length 1024 FFT and a 500 point triangular window with no overlap.

```
rng default;
h = fir1(30,0.2,rectwin(31));
```

```
h1 = ones(1,10)/sqrt(10);
r = randn(16384,1);
x = filter(h1,1,r);
y = filter(h,1,x);
cpsd(x,y,triang(500),250,1024)
```


Algorithms   cpsd uses Welch's averaged periodogram method. See the references
                  listed below.

References   [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal
                  Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 414-419.

cpsd

[2] Welch, P.D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." *IEEE Trans. Audio Electroacoust,* Vol. AU-15 (June 1967). Pgs. 70-73.

[3] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing,* Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 737.

See Also   dspdata | mscohere | pburg | pcov | peig | periodogram | pmcov | pmtm | pmusic | pwelch | pyulear | spectrum | tfestimate

Purpose   Chirp *z*-transform

Syntax   y = czt(x,m,w,a)
                 y = czt(x)

Description y = czt(x,m,w,a) returns the chirp *z*-transform of signal x. The chirp
                 z-transform is the *z*-transform of x along a spiral contour defined by w
                 and a. m is a scalar that specifies the length of the transform, w is the
                 ratio between points along the *z*-plane spiral contour of interest, and
                 scalar a is the complex starting point on that contour. The contour, a
                 spiral or "chirp" in the *z*-plane, is given by

                 z = a*(w.^-(0:m-1))

                 y = czt(x) uses the following default values:

                 • m = length(x)

                 • w = exp(-j*2*pi/m)

                 • a = 1

                 With these defaults, czt returns the *z*-transform of x at m equally
                 spaced points around the unit circle. This is equivalent to the discrete
                 Fourier transform of x, or fft(x). The empty matrix [] specifies the
                 default value for a parameter.

                 If x is a matrix, czt(x,m,w,a) transforms the columns of x.

Examples   Create a random vector x of length 1013 and compute its DFT using czt:

                 ```
                 rng default;
                 x = randn(1013,1);
                 y = czt(x);
                 ```

                 Use czt to zoom in on a narrow-band section (100 to 150 Hz) of a filter's
                 frequency response. First design the filter:

                 ```
                 h = fir1(30,125/500,rectwin(31));  % filter
                 ```

Establish frequency and CZT parameters:

```
fs = 1000; f1 = 100; f2 = 150;     % in hertz
m = 1024;
w = exp(-j*2*pi*(f2-f1)/(m*fs));
a = exp(j*2*pi*f1/fs);
```

Compute the frequency response of the filter using `fft` and `czt`:

```
y = fft(h,1000);
z = czt(h,m,w,a);
fy = (0:length(y)-1)'*1000/length(y);
fz = ((0:length(z)-1)'*(f2-f1)/length(z)) + f1;
subplot(211);
plot(fy(1:500),abs(y(1:500))); axis([1 500 0 1.2])
xlabel('Hz'); ylabel('Magnitude');
title('Magnitude Response using FFT')
subplot(212);
plot(fz,abs(z)); axis([f1 f2 0 1.2])
xlabel('Hz'); ylabel('Magnitude');
title('Magnitude Response using CZT ')
```

Algorithms   czt uses the next power-of-2 length FFT to perform a fast convolution when computing the *z*-transform on a specified chirp contour [1].

Diagnostics   If m, w, or a is not a scalar, czt gives the following error message:

Inputs M, W, and A must be scalars.

References   [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 393-399.

See Also   fft | freqz

db

Purpose	Convert energy or power measurements to decibels

Syntax

```
dboutput = db(X)
dboutput = db(X,SignalType)
dboutput = db(X,R)
dboutput = db(X,'voltage',R)
```

Description

dboutput = db(X) converts the elements of the vector or matrix X to decibels (dB). The elements of X are voltage measurements across a resistance of 1 ohm.

dboutput = db(X,SignalType) specifies the signal type represented by the elements of X as 'voltage' or 'power'. The entries are not case sensitive. The default value is 'voltage'. For voltage measurements, the resistance defaults to 1 ohm. If you specify SignalType as 'power', the elements of X must be nonnegative.

dboutput = db(X,R) specifies the resistance R for voltage measurements. You can specify a resistance only when the signal measurements are voltages.

dboutput = db(X,'voltage',R) specifies the resistance R for voltage measurements. This syntax is equivalent to db(X,R).

Input Arguments

X

Signal measurements. X must be a vector or matrix. If the elements of X are power measurements, all elements must be nonnegative.

SignalType

Type of signal measurements. Valid entries for SignalType are 'voltage' or 'power'. The entries are not case sensitive. If you specify SignalType as 'power', the elements of X must be nonnegative.

Default: 'voltage'

R

Resistive load in ohms. You can specify resistance only when the SignalType is 'voltage'.

Default: 1

Output Arguments

dboutput

The energy or power measurements in the input X in decibels. dboutput has the same dimensions as the input X.

If the input X contains voltage (energy) measurements, dboutput is:

$$dB = 10 \log_{10}(|X|^2 / R)$$

If the input X contains power measurements, dboutput is:

$$dB = 10 \log_{10}(X)$$

Examples

Convert voltage to decibels. Assume that the resistance is 2 ohms.

```
V = 1;
R = 2;
dboutput = db(V,2)
% equivalent to 10*log10(1/2)
```

Convert a vector of power measurements to decibels.

```
rng default
X = abs(randn(10,1));
dboutput = db(X,'power')
```

Alternatives

- mag2db — Converts magnitude measurements to decibels.

- pow2db — Converts power measurements to decibels.

See Also

db2mag | db2pow | mag2db | pow2db

db2mag

Purpose   Convert decibels (dB) to magnitude

Syntax   y = db2mag(ydb)

Description   y = db2mag(ydb) returns the corresponding magnitude y for a given decibel (dB) value ydb. The relationship between magnitude and decibels is ydb = $20*\log_{10}$(y).

See Also   mag2db

Purpose   Convert decibels (dB) to power

Syntax   y = db2pow(ydb)

Description   y = db2pow(ydb) returns the corresponding power value y for a given
                  decibel (dB) value ydb. The relationship between power and decibels
                  is ydb = $10*\log_{10}$(y).

See Also   pow2db

dct

Purpose

Discrete cosine transform (DCT)

Syntax

```
y = dct(x)
y = dct(x,n)
```

Description

`y = dct(x)` returns the unitary discrete cosine transform of x

$$y(k) = w(k) \sum_{n=1}^{N} x(n) \cos(\frac{\pi(2n-1)(k-1)}{2N}) \quad k = 1, 2, \ldots N$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{N}} & k = 1 \\ \sqrt{\dfrac{2}{N}} & 2 \le k \le N \end{cases}$$

N is the length of x, and x and y are the same size. If x is a matrix, `dct` transforms its columns. The series is indexed from $n = 1$ and $k = 1$ instead of the usual $n = 0$ and $k = 0$ because MATLAB vectors run from 1 to N instead of from 0 to N- 1.

`y = dct(x,n)` pads or truncates x to length n before transforming.

The DCT is closely related to the discrete Fourier transform. You can often reconstruct a sequence very accurately from only a few DCT coefficients, a useful property for applications requiring data reduction.

Examples

Find how many DCT coefficients represent 99% of the energy in a sequence:

```
x = (1:100) + 50*cos((1:100)*2*pi/40);
X = dct(x);
[XX,ind] = sort(abs(X)); ind = fliplr(ind);
i = 1;
while (norm([X(ind(1:i)) zeros(1,100-i)])/norm(X)<.99)
```

```
    i = i + 1;
end
% i = 3
```

References   [1] Jain, A.K. *Fundamentals of Digital Image Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[2] Pennebaker, W.B., and J.L. Mitchell. *JPEG Still Image Data Compression Standard,* New York, NY: Van Nostrand Reinhold, 1993. Chapter 4.

See Also   fft | idct | dct2 | idct2

decimate

Purpose	Decimation — decrease sampling rate
Syntax	`y = decimate(x,r)`
	`y = decimate(x,r,n)`
	`y = decimate(x,r,'fir')`
	`y = decimate(x,r,n,'fir')`

Description Decimation reduces the original sampling rate for a sequence to a lower rate, the opposite of interpolation. The decimation process filters the input data with a lowpass filter and then resamples the resulting smoothed signal at a lower rate.

`y = decimate(x,r)` reduces the sample rate of x by a factor r. The decimated vector y is r times shorter in length than the input vector x. By default, `decimate` employs an eighth-order lowpass Chebyshev Type I filter with a cutoff frequency of 0.8*(Fs/2)/r. It filters the input sequence in both the forward and reverse directions to remove all phase distortion, effectively doubling the filter order.

`y = decimate(x,r,n)` uses an order n Chebyshev filter. Orders above 13 are not recommended because of numerical instability. In this case, a warning is displayed.

Note For better results when r is greater than 13, you should break r into its factors and call `decimate` several times.

`y = decimate(x,r,'fir')` uses an order 30 FIR filter, instead of the Chebyshev IIR filter. Here `decimate` filters the input sequence in only one direction. This technique conserves memory and is useful for working with long sequences.

`y = decimate(x,r,n,'fir')` uses an order n FIR filter.

Examples Decimate a signal by a factor of four:

```
t = 0:.00025:1;                    % Time vector
```

```
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = decimate(x,4);
```

View the original and decimated signals:

```
stem(x(1:120)), axis([0 120 -2 2])    % Original signal
title('Original Signal')
figure
stem(y(1:30))                         % Decimated signal
title('Decimated Signal')
```


Algorithms   decimate uses decimation algorithms 8.2 and 8.3 from [1]:

1 It designs a lowpass filter. By default, decimate uses a Chebyshev
  Type I filter with normalized cutoff frequency 0.8/r and 0.05 dB of
  passband ripple. For the fir option, decimate designs a lowpass FIR
  filter with cutoff frequency 1/r using fir1.

2 For the FIR filter, decimate applies the filter to the input vector in
  one direction. In the IIR case, decimate applies the filter in forward
  and reverse directions with filtfilt.

3 decimate resamples the filtered data by selecting every rth point.

> **Note** Depending on the CPU and operating system of your computer, the decimate function may use a lower filter order. If the specified filter order will produce passband distortion, caused by roundoff errors accumulated from the convolutions needed to create the transfer function, the filter order is automatically reduced.

Diagnostics   If r is not an integer, decimate gives the following error message:

Resampling rate R must be an integer.

If n specifies an IIR filter with order greater than 13, decimate gives the following warning:

Warning: IIR filters above order 13 may be unreliable.

References   [1] *IEEE Programs for Digital Signal Processing,* IEEE Press. New York: John Wiley & Sons, 1979. Chapter 8.

See Also   cheby1 | downsample | filtfilt | fir1 | mfilt | interp | resample

Purpose	Demodulation for communications simulation
Syntax	x = demod(y,fc,fs,'*method*') x = demod(y,fc,fs,'*method*',opt) x = demod(y,fc,fs,'pwm','centered')
Description	demod performs demodulation, that is, it obtains the original signal from a modulated version of the signal. demod undoes the operation performed by modulate.

x = demod(y,fc,fs,'*method*') and

x = demod(y,fc,fs,'*method*',opt) demodulate the real carrier signal y with a carrier frequency fc and sampling frequency fs, using one of the options listed below for *method*. (Note that some methods accept an option, opt.)

Note Use demod and modulate in the Signal Processing Toolbox™ with real-valued signals to obtain real-valued outputs. demod and modulate are not intended to accept complex-valued inputs or produce complex-valued outputs.

Method	Description

amdsb-sc
 	
or
 	
am
	Amplitude demodulation, double sideband, suppressed carrier. Multiplies y by a sinusoid of frequency fc and applies a fifth-order Butterworth lowpass filter using filtfilt.
	
	x = y.*cos(2*pi*fc*t);
	
	[b,a] = butter(5,fc*2/fs);
	
	x = filtfilt(b,a,x);

amdsb-tc	Amplitude demodulation, double sideband, transmitted carrier. Multiplies y by a sinusoid of

Method	Description
	frequency fc, and applies a fifth-order Butterworth lowpass filter using filtfilt.
	`x = y.*cos(2*pi*fc*t);`
	`[b,a] = butter(5,fc*2/fs);`
	`x = filtfilt(b,a,x);`
	If you specify opt, demod subtracts scalar opt from x. The default value for opt is 0.
amssb	Amplitude demodulation, single sideband. Multiplies y by a sinusoid of frequency fc and applies a fifth-order Butterworth lowpass filter using filtfilt.
	`x = y.*cos(2*pi*fc*t);`
	`[b,a] = butter(5,fc*2/fs);`
	`x = filtfilt(b,a,x);`
fm	Frequency demodulation. Demodulates the FM waveform by modulating the Hilbert transform of y by a complex exponential of frequency -fc Hz and obtains the instantaneous frequency of the result.
pm	Phase demodulation. Demodulates the PM waveform by modulating the Hilbert transform of y by a complex exponential of frequency -fc Hz and obtains the instantaneous phase of the result.
ppm	Pulse-position demodulation. Finds the pulse positions of a pulse-position modulated signal y. For correct demodulation, the pulses cannot overlap. x is length length(t)*fc/fs.

Method	Description
pwm	Pulse-width demodulation. Finds the pulse widths of a pulse-width modulated signal y. demod returns in x a vector whose elements specify the width of each pulse in fractions of a period. The pulses in y should start at the beginning of each carrier period, that is, they should be left justified.
qam	Quadrature amplitude demodulation.
	[x1,x2] = demod(y,fc,fs,'qam') multiplies y by a cosine and a sine of frequency fc and applies a fifth-order Butterworth lowpass filter using filtfilt.
	x1 = y.*cos(2*pi*fc*t);
	x2 = y.*sin(2*pi*fc*t);
	[b,a] = butter(5,fc*2/fs);
	x1 = filtfilt(b,a,x1);
	x2 = filtfilt(b,a,x2);

The default method is 'am'. In all cases except 'ppm' and 'pwm', x is the same size as y.

If y is a matrix, demod demodulates its columns.

x = demod(y,fc,fs,'pwm','centered') finds the pulse widths assuming they are centered at the beginning of each period. x is length length(y)*fc/fs.

See Also modulate | vco | fskdemod | genqamdemod | mskdemod | pamdemod | pmdemod | qamdemod

design

Purpose	Apply design method to filter specification object

Syntax

```
H = design(D)
H = design(D,METHOD)
H = design(D,METHOD,PARAM1,VALUE1,PARAM2,VALUE2,...)
H = design(D,METHOD,OPTS)
Hs = design(D,...,'SystemObject',sysobjflag)
```

Description

H = design(D) uses the filter specifications object D to generate a filter H. When you do not provide a design method as an input argument, design uses a default design method. Use designmethods(D,'default') to see the default design method for your filter specifications object.

H = design(D,METHOD) forces the design method specified by the string METHOD. METHOD must be one of the strings returned by designmethods. Use designmethods(D,'default') to determine which algorithm is used by default.

The design method you provide as the designmethod input argument must be one of the methods returned by

```
designmethods(d)
```

To help you design filters more quickly, the input argument METHOD accepts a variety of special keywords that force design to behave in different ways. The following table presents the keywords you can use for METHOD and how design responds to the keyword.

Designmethod Keyword	Description of the design Response
'FIR'	Forces design to produce an FIR filter. When no FIR design method exists for object D, design returns an error.
'IIR'	Forces design to produce an IIR filter. When no IIR design method exists for object D, design returns an error.

Designmethod Keyword	Description of the design Response
`'ALLFIR'`	Produces filters from every applicable FIR design method for the specifications in D, one filter for each design method. As a result, design returns multiple filters in the output object.
`'ALLIIR'`	Produces filters from every applicable IIR design method for the specifications in D, one filter for each design method. As a result, design returns multiple filters in the output object.
`'ALL'`	Designs filters using all applicable design methods for the specifications object D. As a result, design returns multiple filters, one for each design method. design uses the design methods in the order that designmethods(D) returns them.

Keywords are not case sensitive

When design returns multiple filters in the output object, use indexing to see the individual filters. For example, to see the third filter in H, enter

H(3)

H = design(D,METHOD,PARAM1,VALUE1,PARAM2,VALUE2,...) specifies design-method options. Use help(D,METHOD) for complete information on which design-method-specific options are available. You can also use designopts(D,METHOD) for a less-detailed listing of the design-method-specific options.

H = design(D,METHOD,OPTS) specifies design-method options using the structure OPTS. OPTS is usually obtained from designopts and then specified as an input to design. Use help(D,METHOD) for more information on optional inputs.

Hs = design(D,...,'SystemObject',*sysobjflag*) uses the filter specifications object D to generate a filter System object Hs when

design

sysobjflag is `true`. To generate System objects, you must have the DSP System Toolbox™ product installed. When *sysobjflag* is `false`, the function generates a `dfilt` or `mfilt` object `H`, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for `dfilt` and `mfilt` objects. To check design methods for System objects, use `designmethods` with the `'SystemObject'`,*sysobjflag* syntax.

If you are specifying design-method-specific options using `OPTS`, you can also set `OPTS.SystemObject` to `true` instead of calling `design` with the `'SystemObject'`,*sysobjflag* syntax.

Examples   Design an FIR equiripple lowpass filter. The passband edge frequency is 0.2π radians/sample, and the stopband edge frequency is 0.25π radians/sample. The passband ripple is 0.5 dB, and the stopband attenuation is 40 dB.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,0.5,40);
H = design(D); % Uses the default equiripple method.
```

If you have the DSP System Toolbox software installed, you can design a minimum-phase FIR equripple filter. Design a minimum–phase filter and compare the pole-zero plots of the original and minimum-phase designs.

```
Hmin = design(D,'equiripple','MinPhase',true);
hfvt = fvtool([H Hmin],'analysis','polezero');
legend(hfvt,'Original Design','Minimum Phase Design');
```

Design a Butterworth lowpass filter. The passband edge frequency is 0.2π radians/sample, and the stopband edge frequency is 0.25π radians/sample. The passband ripple is 0.5 dB, and the stopband attenuation is 40 dB. Obtain help on the design options specific to the Butterworth design method. Design the filter with the "MatchExactly' option set to 'Passband'.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,0.5,40);
% Query design-method-specific options
```

```
help(D,'butter')
% Match passband exactly
H = design(D,'butter','MatchExactly','passband');
```

If you have the DSP System Toolbox software, you can specify the *P*-th norm scaling on the second-order sections. Use L-infinity norm scaling in the time domain.

```
H = design(D,'butter','MatchExactly','passband','SOSScaleNorm','linf')
```

If you have the DSP System Toolbox software, you can create a filter System object.

```
Hs = design(D,'SystemObject',true);
```

See Also   designmethods | designopts

designmethods

Purpose   Methods available for designing filter from specification object

Syntax
```
M = designmethods(D)
M = designmethods(D,'default')
M = designmethods(D,TYPE)
M = designmethods(D,'full')
Ms = designmethods(D,...,'SystemObject',sysobjflag)
```

Description   `M = designmethods(D)` returns the available design methods for the filter specification object, `D`, and the current value of the `Specification` property.

`M = designmethods(D,'default')` returns the default design method for the filter specification object `D` and the current value of the `Specification` property.

`M = designmethods(D,TYPE)` returns either the `TYPE` design methods that apply to D. `TYPE` can be either `'FIR'` or `'IIR'`.

`M = designmethods(D,'full')` returns the full name for each of the available design methods. For example, `designmethods` with the `'full'` argument returns `Butterworth` for the `butter` method.

`Ms = designmethods(D,...,'SystemObject',sysobjflag)` returns the available design methods for designing filter System objects when *sysobjflag* is `true`. To use System objects, you must have the DSP System Toolbox product installed. When *sysobjflag* is `false`, the function checks methods for creating `dfilt` and `mfilt` objects, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for `dfilt` and `mfilt` objects.

Examples   Construct a lowpass filter specification object and determine the valid design methods. Obtain detailed command line help on the Chebyshev type I design method.

```
D =fdesign.lowpass('Fp,Fst,Ap,Ast',500,600,0.5,60,1e4);
M = designmethods(D)
help(D,M{2})
```

The last line of the example is equivalent to `help(D,'cheby1')`.

If you have DSP System Toolbox software installed, use the `'SystemObject'`,*sysobjflag* syntax to return design methods for a filter System object:

```
Ms = designmethods(D,'SystemObject',true);
```

See Also   design | designopts | fdesign

designopts

Purpose	Valid input arguments and values for specification object and method
Syntax	OPTS = designopts(D,METHOD)
Description	OPTS = designopts(D,METHOD) returns a structure array with the default design parameters used by the design method METHOD. METHOD must be one of the strings returned by designmethods. Use help(D,METHOD) to get a description of the design parameters. If you have DSP System Toolbox software installed, OPTS has the SystemObject property if at least one of the structures available for that design method is supported by System objects. However, not all structures for that design method are supported by System objects.
Examples	Create a lowpass filter with a numerator and denominator order of 10 and a 3-dB frequency of 0.2π radians/sample. Obtain the default design parameters for a Butterworth design, and test whether the filter structure is a direct-form II biquad.

```
D = fdesign.lowpass('Nb,Na,F3dB',10,10,0.2);
OPTS = designopts(D,'butter');
if (OPTS.FilterStructure == 'df2sos')
    fprintf('The default filter structure is Direct-Form II\n');
    fprintf('with second-order sections.\n');
end
```

If you have DSP System Toolbox software installed, OPTS has the SystemObject property.

See Also	design	designmethods	fdesign	validstructures

Purpose   Discrete-time filter

Syntax   Hd = dfilt.*structure*(input1,...)
                 Hd = [dfilt.*structure*(input1,...),dfilt.*structure*(input1,...),
                   ...]

Description Hd = dfilt.*structure*(input1,...) returns a discrete-time filter, Hd,
                 of type *structure*. Each structure takes one or more inputs. If you
                 specify a dfilt.*structure* with no inputs, a default filter is created.

                 Note You must use a *structure* with dfilt.

                 Hd = [dfilt.*structure*(input1,...),dfilt.*structure*(input1,...),...]
                 returns a vector containing dfilt filters.

                 Structures

                 Structures for dfilt specify the type of filter structure. Available types
                 of structures for dfilt are shown below.

dfilt.structure	Description	Coefficient Mapping Support in realizemdl
dfilt.delay	Delay	Not supported
dfilt.df1	Direct-form I	Supported
dfilt.df1sos	Direct-form I, second-order sections	Supported
dfilt.df1t	Direct-form I transposed	Supported
dfilt.df1tsos	Direct-form I transposed, second-order sections	Supported
dfilt.df2	Direct-form II	Supported
dfilt.df2sos	Direct-form II, second-order sections	Supported

dfilt.structure	Description	Coefficient Mapping Support in realizemdl
dfilt.df2t	Direct-form II transposed	Supported
dfilt.df2tsos	Direct-form II transposed, second-order sections	Supported
dfilt.dffir	Direct-form FIR	Supported
dfilt.dffirt	Direct-form FIR transposed	Supported
dfilt.dfsymfir	Direct-form symmetric FIR	Supported
dfilt.dfasymfir	Direct-form antisymmetric FIR	Supported
dfilt.fftfir	Overlap-add FIR	Not supported
dfilt.latticeallpass	Lattice allpass	Supported
dfilt.latticear	Lattice autoregressive (AR)	Supported
dfilt.latticearma	Lattice autoregressive moving-average (ARMA)	Supported
dfilt.latticemamax	Lattice moving-average (MA) for maximum phase	Supported
dfilt.latticemamin	Lattice moving-average (MA) for minimum phase	Supported
dfilt.statespace	State-space	Supported.
dfilt.scalar	Scalar gain object	Supported
dfilt.cascade	Filters arranged in series	Supported
dfilt.parallel	Filters arranged in parallel	Supported

For more information on each structure, use the syntax `help diflt.structure` at the MATLAB prompt or refer to its reference page.

Methods

Methods provide ways of performing functions directly on your `dfilt` object without having to specify the filter parameters again. You can

apply these methods directly on the variable you assigned to your `dfilt` object.

For example, if you create a `dfilt` object, `Hd`, you can check whether it has linear phase with `islinphase(Hd)`, view its frequency response plot with `fvtool(Hd)`, or obtain its frequency response values with `h=freqz(Hd)`. You can use all of the methods below in this way.

Note If your variable is a 1-D array of `dfilt` filters, the method is applied to each object in the array. Only `freqz`, `grpdelay`, `impz`, `is*`, `order`, and `stepz` methods can be applied to arrays. The `zplane` method can be applied to an array only if it is used without outputs.

Some of the methods listed below have the same name as Signal Processing Toolbox functions and they behave similarly. This is called *overloading* of functions.

Available methods are:

Method	Description
addstage	Adds a stage to a `cascade` or `parallel` object, where a stage is a separate, modular filter. See `dfilt.cascade` and `dfilt.parallel`.
block	(Available only with DSP System Toolbox product) `block(Hd)` creates a DSP System Toolbox block of the `dfilt` object. The `block` method can specify these properties/values: `'Destination'` indicates where to place the block. `'Current'` places the block in the current Simulink model. `'New'` creates a new model. If you enter the name of an existing subsytem in your model, the block is added to that subsystem. Default value is `'current'`.

Method	Description
	'Blockname' assigns the entered string to the block name. Default name is 'filter'.
	'OverwriteBlock' indicates whether to overwrite the block generated by the block method ('on') and defined by Blockname. Default is 'off'.
	'MapStates' specifies initial conditions in the block ('on'). Default is 'off'. See "Using Filter States" on page 3-152.
	'Link2Obj' indicates whether to specify the filter by linking the block and the command line dfilt object ('on') or by inserting the coefficients directly into the block with no further connection between the command line dfilt object and the block ('off'). If you set 'link2obj' to 'on', the dfilt variable name is inserted in the block and changes to the dfilt object via the command line are reflected in the linked block. Default value is 'off'.
	'MapCoeffstoPorts' indicates whether to map the filter coefficients to constant blocks connected to the generated block. Default value is 'off'. Setting 'MapCoeffstoPorts' to 'on' turns on the mapping and enables the 'CoeffNames' property, which defines the constant block parameter names. 'CoeffNames' is a cell array of strings. Default values are {'Num'} for Direct form FIR filters, {'K'} for lattice filters, {'Num','Den'} for IIR filters, and {Num','Den','g'} for biquad filters. Variables, defined by 'CoeffNames', are created in the MATLAB workspace and have the same data type as the filter's 'Arithmetic' property.

Method	Description
	Any existing variable with the same name is overwritten. Note that you can use either `'Link2Obj'` or `'MapCoeffstoPorts'`, but not both simultaneously.
cascade	Returns the series combination of two `dfilt` objects. See `dfilt.cascade`.
coeffs	Returns the filter coefficients in a structure containing fields that use the same property names as those in the original `dfilt`.
convert	Converts a `dfilt` object from one filter structure to another filter structure.
fcfwrite	Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. If the DSP System Toolbox product is installed, the file can contain multirate filters (`mfilt`) or adaptive filters (`adaptfilt`). Default filename is `untitled.fcf`.
	`fcfwrite(Hd,filename)` writes to a disk file named `filename` in the current working directory. The `.fcf` extension is added automatically.
	`fcfwrite(...,fmt)` writes the coefficients in the format `fmt`, where valid `fmt` strings are:
	`'hex'` for hexadecimal
	`'dec'` for decimal
	`'bin'` for binary representation.
fftcoeffs	Returns the frequency-domain coefficients used when filtering with a `dfilt.fftfir`.

Method	Description
filter	Performs filtering using the dfilt object.
	y = filter(Hd,x) filters x using the Hd filter and returns the filtered data in y. See "Using Filter States" on page 3-152 for information on using initial conditions. If x is a matrix, each column is filtered as an independent channel. If x is a multidimensional array, filter operates on the first nonsingleton dimension.
	y = filter(Hd,x,dim) operates along the dimension dim. If x is a vector or matrix and dim is 1, every column of x is a channel. If dim is 2, every row is a channel.
firtype	Returns the type (1-4) of a linear phase FIR filter.
freqz	Plots the frequency response in fvtool. Note that unlike the freqz function, this dfilt freqz method has a default length of 8192.
grpdelay	Plots the group delay in fvtool.
impz	Plots the impulse response in fvtool.
impzlength	Returns the length of the impulse response.
info	Displays brief dfilt information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length. To display detailed information about the design method, options, etc, use info(Hd, 'long'). The default display is 'short'. For multistage filters (cascade and parallel), use info(Hd.Stage(x)), where x is the stage number, to see information about that stage.

Method	Description
isallpass	Returns a logical 1 (i.e., true) if the dfilt object in an allpass filter or a logical 0 (i.e., false) if it is not.
iscascade	Returns a logical 1 if the dfilt object is cascaded or a logical 0 if it is not.
isfir	Returns a logical 1 if the dfilt object has finite impulse response (FIR) or a logical 0 if it does not.
islinphase	Returns a logical 1 if the dfilt object is linear phase or a logical 0 if it is not.
ismaxphase	Returns a logical 1 if the dfilt object is maximum-phase or a logical 0 if it is not.
isminphase	Returns a logical 1 if the dfilt object is minimum-phase or a logical 0 if it is not.
isparallel	Returns a logical 1 if the dfilt object has parallel stages or a logical 0 if it does not.
isreal	Returns a logical 1 if the dfilt object has real-valued coefficients or a logical 0 if it does not.
isscalar	Returns a logical 1 if the dfilt object is a scalar or a logical 0 if it is not scalar.
issos	Returns a logical 1 if the dfilt object has second-order sections or a logical 0 if it does not.
isstable	Returns a logical 1 if the dfilt object is stable or a logical 0 if it are not.
nsections	Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using nsections returns the total number of sections in all the stages (a stage with a single section returns 1).

Method	Description
nstages	Returns the number of stages of the filter, where a stage is a separate, modular filter.
nstates	Returns the number of states for an object.
order	Returns the filter order. If Hd is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If Hd has multiple stages, the order is given by the number of delays needed for a minimum realization of the overall filter.
parallel	Returns the parallel combination of two dfilt filters. See dfilt.parallel.
phasez	Plots the phase response in fvtool.
realizemdl	(Available only with Simulink software.) realizemdl(Hd) creates a Simulink model containing a subsystem block realization of your dfilt. realizemdl(Hd,p1,v1,p2,v2,...) creates the block using the properties p1, p2,... and values v1, v2,.. specified. The following properties are available: 'Blockname' specifies the name of the block. The default value is 'Filter'. 'Destination' specifies whether to add the block to a current Simulink model, create a new model, or place the block in an existing subsystem in your model. Valid values are 'current', 'new', or the name of an existing subsytem in your model. Default value is 'current'.

Method	Description
	'OverwriteBlock' specifies whether to overwrite an existing block that was created by realizemdl or create a new block. Valid values are 'on' and 'off' and the default is 'off'. Note that only blocks created by realizemdl are overwritten.
	The following properties optimize the block structure. Specifying 'on' turns the optimization on and 'off' creates the block without optimization. The default for each of the following is 'on'.
	'OptimizeZeros' removes zero-gain blocks.
	'OptimizeOnes' replaces unity-gain blocks with a direct connection.
	'OptimizeNegOnes' replaces negative unity-gain blocks with a sign change at the nearest summation block.
	'OptimizeDelayChains' replaces cascaded chains of delay block with a single integer delay block set to the appropriate delay.
removestage	Removes a stage from a cascade or parallel dfilt. See dfilt.cascade and dfilt.parallel.
setstage	Overwrites a stage of a cascade or parallel dfilt. See dfilt.cascade and dfilt.parallel.

Method	Description
sos	Converts the dfilt to a second-order sections dfilt. If Hd has a single section, the returned filter has the same class.
	sos(Hd,flag) specifies the ordering of the second-order sections. If flag='UP', the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If flag='down', the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them.
	sos(Hd,flag,scale) specifies the scaling of the gain and the numerator coefficients of all second-order sections. scale can be 'none', 'inf' (infinity-norm) or 'two' (2-norm). Using infinity-norm scaling with up ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with down ordering minimizes the peak roundoff noise.
ss	Converts the dfilt to state-space. To see the separate A,B,C,D matrices for the state-space model, use [A,B,C,D]=ss(Hd).
stepz	Plots the step response in fvtool.
	stepz(Hd,n) computes the first n samples of the step response.
	stepz(Hd,n,Fs) separates the time samples by T = 1/Fs, where Fs is assumed to be in Hz.
sysobj	Converts the dfilt to a filter System object. See the reference page for a list of supported objects. To use this method, you must have DSP System Toolbox software installed.
tf	Converts the dfilt to a transfer function.

Method	Description
zerophase	Plots the zero-phase response in `fvtool`.
zpk	Converts the `dfilt` to zeros-pole-gain form.
zplane	Plots a pole-zero plot in `fvtool`.

For more information on each method, use the syntax `help diflt/`*method* at the MATLAB prompt.

Viewing Properties

As with any object, you can use `get` to view a `dfilt` properties. To see a specific property, use

```
get(Hd,'property')
```

To see all properties for an object, use

```
get(Hd)
```

Changing Properties

To set specific properties, use

```
set(Hd,'property1',value,'property2',value,...)
```

Note that you must use single quotation marks around the property name.

Alternatively, you can get or set a property value with `Object.property`:

```
b = [0.05 0.9 0.05];
Hd = dfilt.dffir(b);
% Lowpass direct-form I FIR filter
Hd.arithmetic % get arithmetic property
% returns double
Hd.arithmetic = 'single';
% Set arithmetic property to single precision
```

Copying an Object

To create a copy of an object, use the `copy` method.

```
H2 = copy(Hd)
```

Note Using the syntax `H2 = Hd` copies only the object handle and does not create a new object.

Converting Between Filter Structures

To change the filter structure of a `dfilt` object `Hd`, use

```
Hd2=convert(Hd,'structure_string');
```

where `structure_string` is any valid structure name in single quotation marks. If `Hd` is a `cascade` or `parallel` structure, each of its stages is converted to the new structure.

Using Filter States

Two properties control the filter states:

- `states` — stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions. For `df1`, `df1t`, `df1sos` and `df1tsos` structures, `states` returns a `filtstate` object.

- `PersistentMemory` — controls whether filter `states` are saved. The default value is `'false'`, which causes the initial conditions to be reset to zero before filtering and turns off the display of `states` information. Setting `PersistentMemory` to `'true'` allows the filter to use your initial conditions or to reuse the final conditions of a previous filtering operation as the initial conditions of the next filtering operation. It also displays information about the filter `states`.

> **Note** If you set states and want to use them for filtering, you must set PersistentMemory to 'true' before you use the filter.

Examples   Create a direct-form I filter and use a method to see if it is stable.

```
[b,a] = butter(8,0.25);
Hd = dfilt.df1(b,a)
```

If a dfilt's numerator values do not fit on a single line, a description of the vector is displayed. To see the specific numerator values for this example, use

```
get(Hd,'numerator')
```

or alternatively

```
Hd.numerator
```

Refer to the reference pages for each structure for more examples.

See Also   dfilt.cascade | dfilt.df1 | dfilt.df1t | dfilt.df2 | dfilt.df2t | dfilt.dfasymfir | dfilt.dffir | dfilt.dffirt | dfilt.dfsymfir | dfilt.latticeallpass | dfilt.latticear | dfilt.latticearma | dfilt.latticemamax | dfilt.latticemamin | dfilt.parallel | dfilt.statespace | filter | freqz | grpdelay | impz | step | tf | zpk | zplane

dfilt.cascade

Purpose	Cascade of discrete-time filters
Syntax	Hd = dfilt.cascade(Hd1,Hd2,...)
Description	Hd = dfilt.cascade(Hd1,Hd2,...) returns a discrete-time filter, Hd, of type cascade, which is a serial interconnection of two or more dfilt filters, Hd1, Hd2, etc. Each filter in a cascade is a separate stage.

To add a filter (Hd1) to the end of an existing cascade (Hd), use

addstage(Hd,Hd1)

and to reorder the filters in a cascade, use the stage indices to indicate the desired ordering, such as.

Hd.stage = Hd.stage([1,3,2]);

You can also use the nondot notation format for calling a cascade:

cascade(Hd1,Hd2,...)

Examples	Cascade a lowpass filter and a highpass filter to produce a bandpass filter:

```
[b1,a1]=butter(8,0.6);          % Lowpass
[b2,a2]=butter(8,0.4,'high');   % Highpass
H1=dfilt.df2t(b1,a1);
H2=dfilt.df2t(b2,a2);
Hcas=dfilt.cascade(H1,H2)       % Bandpass-passband .4-.6
```

To view details of the first stage, use

info(Hcas.Stage(1))

To view the states of a stage, use

```
Hcas.stage(1).states
```

You can display states for individual stages only.

See Also   dfilt | dfilt.parallel | dfilt.scalar

dfilt.delay

Purpose   Delay filter

Syntax   Hd = dfilt.delay
              Hd = dfilt.delay(latency)

Description   Hd = dfilt.delay returns a discrete-time filter, Hd, of type delay, which adds a single delay to any signal filtered with Hd. The filtered signal has its values shifted by one sample.

Hd = dfilt.delay(latency) returns a discrete-time filter, Hd, of type delay, which adds the number of delay units specified in latency to any signal filtered with Hd. The filtered signal has its values shifted by the latency number of samples. The values that appear before the shifted signal are the filter states.

Examples   Create a delay filter with a latency of 4 and filter a simple signal to view the impact of applying a delay.

```
h = dfilt.delay(4)
h =
     FilterStructure: 'Delay'
             Latency: 4
     PersistentMemory: false
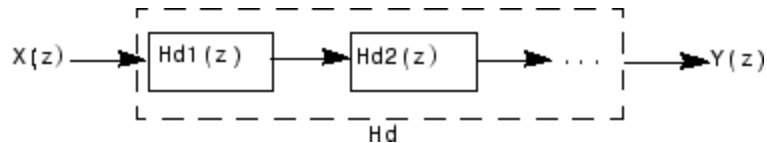
sig = 1:7      % Create some simple signal data
sig =
     1    2    3    4    5    6    7

states = h.states    % Filter states before filtering
states =
     0
     0
     0
     0

filter(h,sig)        % Filter using the delay filter
ans =
```

```
        0      0      0      0      1      2      3

states=h.states       % Filter states after filtering
states =
     4
     5
     6
     7
```

See Also   dfilt

dfilt.df1

Purpose	Discrete-time, direct-form I filter
Syntax	`Hd = dfilt.df1(b,a)` `Hd = dfilt.df1`
Description	`Hd = dfilt.df1(b,a)` returns a discrete-time, direct-form I filter, Hd, with numerator coefficients b and denominator coefficients a. The filter states for this object are stored in a `filtstates` object. `Hd = dfilt.df1` returns a default, discrete-time, direct-form I filter, Hd, with b=1 and a=1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.

**df1
(Direct-form I)**

Image of direct form one filter diagram

To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states      % Where Hd is the dfilt.df1 object and
double (Hs)         % Hs is the filtstates object
```

The vector is

$$\begin{bmatrix} zb(1) \\ zb(2) \\ \dots \\ zb(n) \\ za(1) \\ za(2) \\ \dots \\ za(n) \end{bmatrix}$$

Examples   Create a direct-form I discrete-time filter with coefficients from a fourth-order lowpass Butterworth design

```
[b,a] = butter(4,.5);
Hd = dfilt.df1(b,a)
```

See Also   dfilt | dfilt.df1t | dfilt.df2 | dfilt.df2t

Purpose Discrete-time, second-order section, direct-form I filter

Syntax
```
Hd = dfilt.df1sos(s)
Hd = dfilt.df1sos(b1,a1,b2,a2,...)
Hd = dfilt.df1sos(...,g)
Hd = dfilt.df1sos
```

Description Hd = dfilt.df1sos(s) returns a discrete-time, second-order section, direct-form I filter, Hd, with coefficients given in the s matrix. The filter states for this object are stored in a filtstates object.

Hd = dfilt.df1sos(b1,a1,b2,a2,...) returns a discrete-time, second-order section, direct-form I filter, Hd, with coefficients for the first section given in the b1 and a1 vectors, for the second section given in the b2 and a2 vectors, etc.

Hd = dfilt.df1sos(...,g) includes a gain vector g. The elements of g are the gains for each section. The maximum length of g is the number of sections plus one. If g is not specified, all gains default to one.

Hd = dfilt.df1sos returns a default, discrete-time, second-order section, direct-form I filter, Hd. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator a(1) cannot be 0.

---

df1sos
(Direct-form I, second-order sections)

To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states      % Where Hd is the dfilt.df1 object and
double (Hs)         % Hs is the filtstates object
```

The vector is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

For filters with more than one section, each section is a separate column in the matrix.

Examples   Specify a second-order sections, direct-form I discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code. The resulting filter has three sections.

```
[z,p,k] = ellip(6,1,60,.4);  % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);       % Convert to SOS
Hd = dfilt.df1sos(s,g)
```

See Also   dfilt | dfilt.df1tsos | dfilt.df2sos | dfilt.df2tsos

Purpose	Discrete-time, direct-form I transposed filter
Syntax	Hd = dfilt.df1t(b,a) Hd = dfilt.df1t
Description	Hd = dfilt.df1t(b,a) returns a discrete-time, direct-form I transposed filter, Hd, with numerator coefficients b and denominator coefficients a. The filter states for this object are stored in a filtstates object. Hd = dfilt.df1t returns a default, discrete-time, direct-form I transposed filter, Hd, with b=1 and a=1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator a(1) cannot be 0.

df1t
(Transposed Direct-form I)

To display the filter states, use this code to access the filtstates object.

```
Hs = Hd.states        % Where Hd is the dfilt.df1 object and
double (Hs)           % Hs is the filtstates object
```

The vector of states is:

$$
\begin{pmatrix}
zb(1) \\
zb(2) \\
\ldots \\
zb(M) \\
za(1) \\
za(2) \\
\ldots \\
za(N)
\end{pmatrix}
$$

Alternatively, you can access the states in the filtstates object:

```
b = [0.05 0.9 0.05];
Hd = dfilt.df1t(b,1);
Hd.States
% Returns
%  Numerator: [2x1 double]
%  Denominator: [0x1 double]
Hd.States.Numerator(1)=1; %Set zb(1) equal to 1.
```

Examples   Create a direct-form I transposed discrete-time filter with coefficients
from a fourth-order lowpass Butterworth design:

```
[b,a] = butter(4,.5);
Hd = dfilt.df1t(b,a)
```

See Also   dfilt | dfilt.df1 | dfilt.df2 | dfilt.df2t

dfilt.df1tsos

Purpose   Discrete-time, second-order section, direct-form I transposed filter

Syntax
```
Hd = dfilt.df1tsos(s)
Hd = dfilt.df1tsos(b1,a1,b2,a2,...)
Hd = dfilt.df1tsos(...,g)
Hd = dfilt.df1tsos
```

Description   Hd = dfilt.df1tsos(s) returns a discrete-time, second-order section, direct-form I, transposed filter, Hd, with coefficients given in the s matrix. The filter states for this object are stored in a filtstates object.

Hd = dfilt.df1tsos(b1,a1,b2,a2,...) returns a discrete-time, second-order section, direct-form I, tranposed filter, Hd, with coefficients for the first section given in the b1 and a1 vectors, for the second section given in the b2 and a2 vectors, etc.

Hd = dfilt.df1tsos(...,g) includes a gain vector g. The elements of g are the gains for each section. The maximum length of g is the number of sections plus one. If g is not specified, all gains default to one.

Hd = dfilt.df1tsos returns a default, discrete-time, second-order section, direct-form I, transposed filter, Hd. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator a(1) cannot be 0.

---

df1tsos
(Transposed Direct-form I, second-order sections)

To display the filter states, use this code to access the filtstates object.

```
Hs = Hd.states      % Where Hd is the dfilt.df1 object and
double (Hs)         % Hs is the filtstates object
```

The matrix is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

Examples   Specify a second-order sections, direct-form I, transposed discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);      % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);           % Convert to SOS
Hd = dfilt.df1tsos(s,g)
```

See Also   dfilt | dfilt.df1sos | dfilt.df2sos | dfilt.df2tsos

Purpose   Discrete-time, direct-form II filter

Syntax   Hd = dfilt.df2(b,a)
               Hd = dfilt.df2

Description   Hd = dfilt.df2(b,a) returns a discrete-time, direct-form II filter, Hd, with numerator coefficients b and denominator coefficients a.

Hd = dfilt.df2 returns a default, discrete-time, direct-form II filter, Hd, with b=1 and a=1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator a(1) cannot be 0.

---

df2
(Direct-form II)

The resulting filter states column vector is

$$
\begin{bmatrix}
z(1) \\
z(2) \\
\dots \\
z(n)
\end{bmatrix}
$$

Examples   Create a direct-form II discrete-time filter with coefficients from a fourth-order lowpass Butterworth design:

```
[b,a] = butter(4,.5);
Hd = dfilt.df2(b,a)
```

See Also   dfilt | dfilt.df1 | dfilt.df1t | dfilt.df2t

dfilt.df2sos

Purpose   Discrete-time, second-order section, direct-form II filter

Syntax   Hd = dfilt.df2sos(s)
                 Hd = dfilt.df2sos(b1,a1,b2,a2,...)
                 Hd = dfilt.df2sos(...,g)
                 Hd = dfilt.df2sos

Description Hd = dfilt.df2sos(s) returns a discrete-time, second-order section, direct-form II filter, Hd, with coefficients given in the s matrix.

Hd = dfilt.df2sos(b1,a1,b2,a2,...) returns a discrete-time, second-order section, direct-form II object, Hd, with coefficients for the first section given in the b1 and a1 vectors, for the second section given in the b2 and a2 vectors, etc.

Hd = dfilt.df2sos(...,g) includes a gain vector g. The elements of g are the gains for each section. The maximum length of g is the number of sections plus one. If g is not specified, all gains default to one.

Hd = dfilt.df2sos returns a default, discrete-time, second-order section, direct-form II filter, Hd. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator a(1) cannot be 0.

df2sos
(Direct-form II, second-order sections)

The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

For filters with more than one section, each section is a separate column in the vector.

dfilt.df2sos

Examples
Specify a second-order sections, direct-form II discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);    % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);         % Convert to SOS
Hd = dfilt.df2sos(s,g)
```

See Also
dfilt | dfilt.df1sos | dfilt.df1tsos | dfilt.df2tsos

Purpose Discrete-time, direct-form II transposed filter

Syntax Hd = dfilt.df2t(b,a)
Hd = dfilt.df2t

Description Hd = dfilt.df2t(b,a) returns a discrete-time, direct-form II transposed filter, Hd, with numerator coefficients b and denominator coefficients a.

Hd = dfilt.df2t returns a default, discrete-time, direct-form II transposed filter, Hd, with b=1 and a=1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator a(1) cannot be 0.

df2t
(Transposed Direct-form II)

The filter states of `dfilt.df2t` object can be extracted as a column vector with:

```
b =[1 2];
a =[1 -0.9];
Hd = dfilt.df2t(b,a);
FiltStates = double(Hd.States);
```

The resulting filter states column vector is

$$\begin{pmatrix} z(1) \\ z(2) \end{pmatrix}$$

Examples   Create a direct-form II transposed discrete-time filter with coefficients from a 4–th order lowpass Butterworth design:

```
[b,a] = butter(4,.5);
Hd = dfilt.df2t(b,a);
```

See Also   dfilt | dfilt.df1 | dfilt.df1t | dfilt.df2

Purpose   Discrete-time, second-order section, direct-form II transposed filter

Syntax
```
Hd = dfilt.df2sos(s)
Hd = dfilt.df2tsos(b1,a1,b2,a2,...)
Hd = dfilt.df2tsos(...,g)
Hd = dfilt.df2tso
```

Description   Hd = dfilt.df2sos(s) returns a discrete-time, second-order section, direct-form II, transposed filter, Hd, with coefficients given in the s matrix.

Hd = dfilt.df2tsos(b1,a1,b2,a2,...) returns a discrete-time, second-order section, direct-form II, tranposed filter, Hd, with coefficients for the first section given in the b1 and a1 vectors, for the second section given in the b2 and a2 vectors, etc.

Hd = dfilt.df2tsos(...,g) includes a gain vector g. The elements of g are the gains for each section. The maximum length of g is the number of sections plus one. If g is not specified, all gains default to one.

Hd = dfilt.df2tso returns a default, discrete-time, second-order section, direct-form II, transposed filter, Hd. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator a(1) cannot be 0.

---

dfilt.df2tsos

df2tsos
(Transposed Direct-form II, second-order sections)

The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

Examples

Specify a second-order sections, direct-form II, transposed discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);  % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);       % Convert to SOS
Hd = dfilt.df2tsos(s,g)
```

See Also

dfilt | dfilt.df1sos | dfilt.df1tsos | dfilt.df2sos

Purpose   Discrete-time, direct-form antisymmetric FIR filter

Syntax   Hd = dfilt.dfasymfir(b)
                   Hd = dfilt.dfasymfir

Description   Hd = dfilt.dfasymfir(b) returns a discrete-time, direct-form, antisymmetric FIR filter, Hd, with numerator coefficients b.

Hd = dfilt.dfasymfir returns a default, discrete-time, direct-form, antisymmetric FIR filter, Hd, with b=1. This filter passes the input through to the output unchanged.

Note Only the first half of vector b is used because the second half is assumed to be antisymmetric. In the figure below for an odd number of coefficients, b(3) = 0, b(4) = –b(2) and b(5) = –b(1), and in the next figure for an even number of coefficients, b(4) = –b(3), b(5) = –b(2), and b(6) = –b(1).

---

dfasymfir
(Antisymmetric FIR)
Even order
Odd number of coefficients, length(b) = 7

Note that antisymmetry is defined as
$b(i) == -b(end - i + 1)$
so that the middle coefficient is zero for odd length
$b((end+1)/2) = 0$

dfasymfir
(Antisymmetric FIR)
Even number of coefficients, length(b) = 6

b(i) == -b(end - i + 1)

The resulting filter states column vector for the odd number of coefficients example above is

$$
\begin{bmatrix}
z(1) \\
z(2) \\
z(3) \\
z(4) \\
z(5) \\
z(6)
\end{bmatrix}
$$

Examples   **Odd Order**

Create a Type 4 25th order highpass direct-form antisymmetric FIR filter structure for a `dfilt` object, `Hd`, with the following code:

```
Num_coeffs = firpm(25,[0 .4 .5 1],[0 0 1 1],'h');
Hd = dfilt.dfasymfir(Num_coeffs);
```

Even Order

Create a 44th order lowpass direct-form antisymmetric FIR differentiator filter structure for a `dfilt` object, `Hd`, with the following code:

```
Num_coeffs = firpm(44,[0 .3 .4 1],[0 .2 0 0],'differentiator');
Hd = dfilt.dfasymfir(Num_coeffs);
```

See Also   `dfilt` | `dfilt.dffir` | `dfilt.dffirt` | `dfilt.dfsymfir`

Purpose   Discrete-time, direct-form, FIR filter

Syntax   Hd = dfilt.dffir(b)
                   Hd = dfilt.dffir

Description   Hd = dfilt.dffir(b) returns a discrete-time, direct-form finite
                   impulse response (FIR) filter, Hd, with numerator coefficients, b.

                   Hd = dfilt.dffir returns a default, discrete-time, direct-form FIR
                   filter, Hd, with b=1. This filter passes the input through to the output
                   unchanged.

dffir
(Direct-form FIR = Tapped delay line)

The resulting filter states column vector is

$$\begin{pmatrix} z(1) \\ z(2) \end{pmatrix}$$

Examples   Create a direct-form FIR discrete-time filter with coefficients from a 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.dffir(b)
```

See Also   dfilt | dfilt.dfasymfir | dfilt.dffirt | dfilt.dfsymfir

Purpose   Discrete-time, direct-form FIR transposed filter

Syntax   Hd = dfilt.dffirt(b)
                Hd = dfilt.dffirt

Description   Hd = dfilt.dffirt(b) returns a discrete-time, direct-form FIR transposed filter, Hd, with numerator coefficients b.

Hd = dfilt.dffirt returns a default, discrete-time, direct-form FIR transposed filter, Hd, with b=1. This filter passes the input through to the output unchanged.

dffirt
(Transposed Direct-form FIR)

The resulting filter states column vector is

$$\begin{pmatrix} z(1) \\ z(2) \end{pmatrix}$$

dfilt.dffirt

Examples   Create a direct-form FIR transposed discrete-time filter with coefficients from a 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.dffirt(b)
```

See Also   dfilt | dfilt.dffir | dfilt.dfasymfir | dfilt.dfsymfir

Purpose   Discrete-time, direct-form symmetric FIR filter

Syntax   Hd = dfilt.dfsymfir(b)
Hd = dfilt.dfsymfir

Description   Hd = dfilt.dfsymfir(b) returns a discrete-time, direct-form symmetric FIR filter, Hd, with numerator coefficients b.

Hd = dfilt.dfsymfir returns a default, discrete-time, direct-form symmetric FIR filter, Hd, with b=1. This filter passes the input through to the output unchanged.

Note Only the first half of vector b is used because the second half is assumed to be symmetric. In the figure below for an odd number of coefficients, b(3) = 0, b(4) = b(2) and b(5) = b(1), and in the next figure for an even number of coefficients, b(4) = b(3), b(5) = b(2), and b(6) = b(1).

dfsymfir
(Symmetric FIR)
Even order
Odd number of coefficients, length(b) = 5
b(i) == b(end - i + 1)

dfsymfir
(Symmetric FIR)
Odd order
Even number of coefficients, length(b) = 6
b(i) == b(end - i + 1)

The resulting filter states column vector for the odd number of coefficients example above is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \\ z(4) \end{bmatrix}$$

Examples   **Odd Order**

Specify a fifth-order direct-form symmetric FIR filter structure for a dfilt object, Hd, with the following code:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
Hd = dfilt.dfsymfir(b)
```

Even Order

Specify a fourth-order direct-form symmetric FIR filter structure for a dfilt object, Hd, with the following code:

```
b = [-0.01 0.1 0.8 0.1 -0.01];
Hd = dfilt.dfsymfir(b)
```

See Also   dfilt | dfilt.dfasymfir | dfilt.dffir | dfilt.dffirt

Purpose   Discrete-time, overlap-add, FIR filter

Syntax   `Hd = dfilt.fftfir(b,len)`
                 `Hd = dfilt.fftfir(b)`
                 `Hd = dfilt.fftfir`

Description This object uses the overlap-add method of block FIR filtering, which is very efficient for streaming data.

`Hd = dfilt.fftfir(b,len)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`. The block length is the number of input points to use for each overlap-add computation.

`Hd = dfilt.fftfir(b)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len=100`.

`Hd = dfilt.fftfir` returns a default, discrete-time, FFT, FIR filter, `Hd`, with the numerator `b=1` and block length, `len=100`. This filter passes the input through to the output unchanged.

Note When you use a `dfilt.fftfir` object to filter, the input signal length must be an integer multiple of the object's block length, `len`. The resulting number of FFT points = (filter length + the block length - 1). The filter is most efficient if the number of FFT points is a power of 2.

The `fftfir` uses an overlap-add block processing algorithm, which is represented as follows,

where len is the block length and M is the length of the numerator-1, (length(b)-1), which is also the number of states. The output of each convolution is a block that is longer than the input block by a tail of (length(b)-1) samples. These tails overlap the next block and are added to it. The states reported by dfilt.fftfir are the tails of the final convolution.

Examples   Create an FFT FIR discrete-time filter with coefficients from a 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.fftfir(b)
```

To view the frequency domain coefficients used in the filtering, use the following command.

```
freq_coeffs = fftcoeffs(Hd);
```

See Also   dfilt | dfilt.dffir | dfilt.dfasymfir | dfilt.dffirt | dfilt.dfsymfir

Purpose   Discrete-time, lattice allpass filter

Syntax   ```
Hd = dfilt.latticeallpass(k)
Hd = dfilt.latticeallpass
```
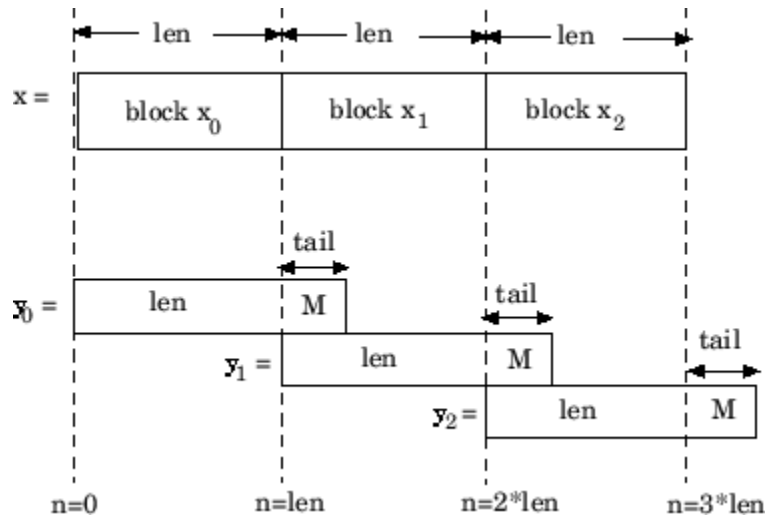
Description Hd = dfilt.latticeallpass(k) returns a discrete-time, lattice allpass filter, Hd, with lattice coefficients, k.

Hd = dfilt.latticeallpass returns a default, discrete-time, lattice allpass filter, Hd, with k=[ ]. This filter passes the input through to the output unchanged.

latticeallpass
(Lattice Allpass)
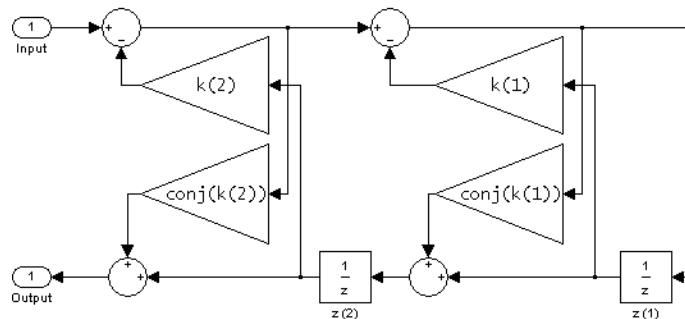
The resulting filter states column vector Hd.States is

$$\begin{pmatrix} z(1) \\ z(2) \end{pmatrix}$$

Examples   Form a third-order lattice allpass filter structure for a dfilt object, Hd, using the following lattice coefficients:

```
k = [.66 .7 .44];
Hd = dfilt.latticeallpass(k)
```

dfilt.latticeallpass

See Also dfilt | dfilt.latticear | dfilt.latticearma | dfilt.latticemamax | dfilt.latticemamin

Purpose   Discrete-time, lattice, autoregressive filter

Syntax   Hd = dfilt.latticear(k)
                     Hd = dfilt.latticear

Description   Hd = dfilt.latticear(k) returns a discrete-time, lattice
                     autoregressive filter, Hd, with lattice coefficients, k.

                     Hd = dfilt.latticear returns a default, discrete-time, lattice
                     autoregressive filter, Hd, with k=[ ]. This filter passes the input through
                     to the output unchanged.
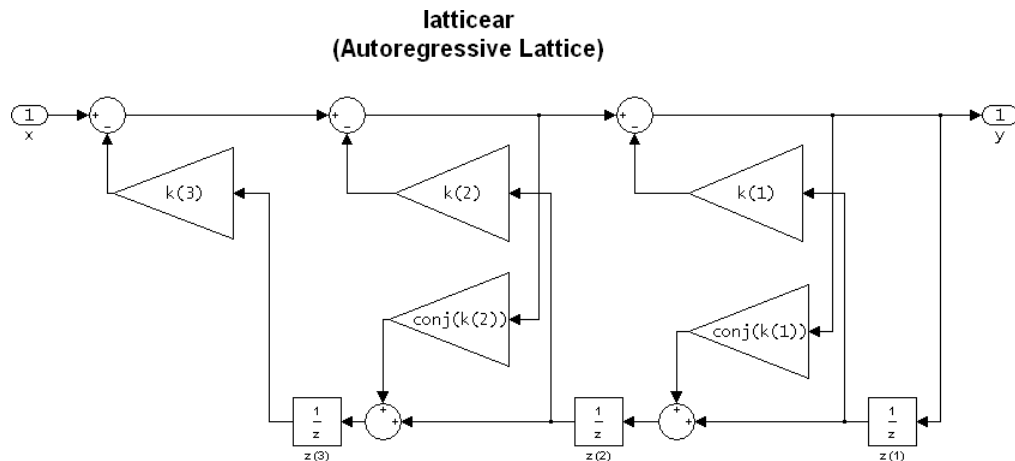
**latticear
(Autoregressive Lattice)**

The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

Examples   Form a third-order lattice autoregressive filter structure for a dfilt
                     object, Hd, using the following lattice coefficients:

```
k = [.66 .7 .44];
Hd = dfilt.latticear(k)
```

See Also   dfilt | dfilt.latticeallpass | dfilt.latticearma |
dfilt.latticemamax | dfilt.latticemamin

Purpose   Discrete-time, lattice, autoregressive, moving-average filter

Syntax   Hd = dfilt.latticearma(k,v)
                 Hd = dfilt.latticearma

Description Hd = dfilt.latticearma(k,v) returns a discrete-time, lattice
                 autoregressive, moving-average filter, Hd, with lattice coefficients, k
                 and ladder coefficients v.

                 Hd = dfilt.latticearma returns a default, discrete-time, lattice
                 autoregressive, moving-average filter, Hd, with k=[ ] and v=1. This filter
                 passes the input through to the output unchanged.

**latticearma
(Autogressive Moving-Average Lattice)**
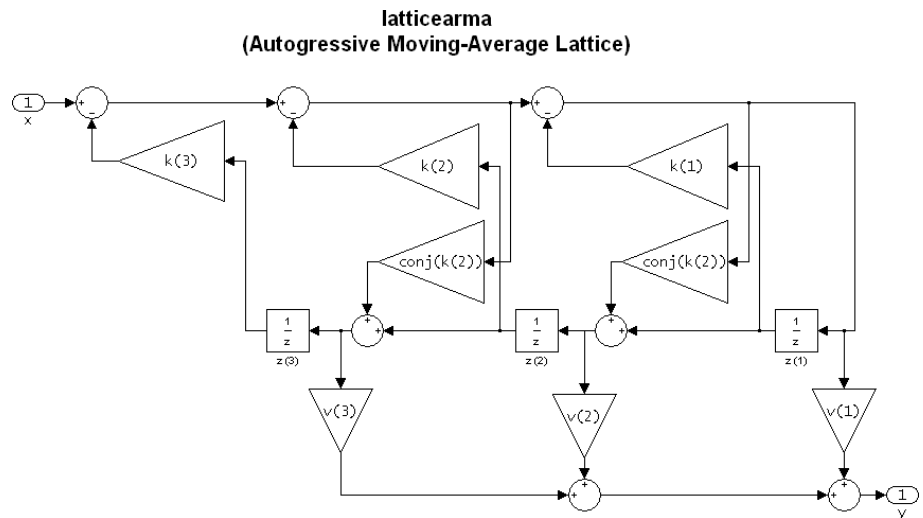
The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

dfilt.latticearma

Examples   Form a third-order lattice autoregressive, moving-average filter structure for a dfilt object, Hd, using the following lattice coefficients:

```
k = [.66 .7 .44];
Hd = dfilt.latticearma(k)
```

See Also   dfilt | dfilt.latticeallpass | dfilt.latticear | dfilt.latticemamax | dfilt.latticemamin

Purpose   Discrete-time, lattice, moving-average filter

Syntax   Hd = dfilt.latticemamax(k)
                     Hd = dfilt.latticemamax

Description   Hd = dfilt.latticemamax(k) returns a discrete-time, lattice, moving-average filter, Hd, with lattice coefficients k.

Note If the k coefficients define a maximum phase filter, the resulting filter in this structure is maximum phase. If your coefficients do not define a maximum phase filter, placing them in this structure does not produce a maximum phase filter.

Hd = dfilt.latticemamax returns a default discrete-time, lattice, moving-average filter, Hd, with k=[ ]. This filter passes the input through to the output unchanged.

latticemamax
(Moving-Average, Maximum Phase Lattice)

dfilt.latticemamax
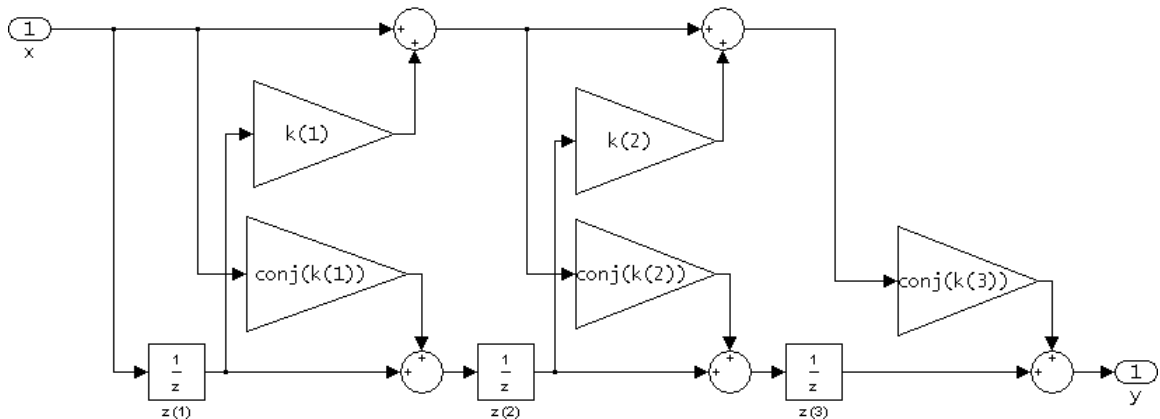
The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

Examples   Form a fourth-order lattice, moving-average, maximum phase filter structure for a dfilt object, Hd, using the following lattice coefficients:

```
k = [.66 .7 .44 .33];
Hd = dfilt.latticemamax(k)
```

See Also   dfilt | dfilt.latticeallpass | dfilt.latticear | dfilt.latticearma | dfilt.latticemamin

Purpose   Discrete-time, lattice, moving-average filter

Syntax   Hd = dfilt.latticemamin(k)
                Hd = dfilt.latticemamin

Description Hd = dfilt.latticemamin(k) returns a discrete-time, lattice,
                moving-average, minimum phase, filter, Hd, with lattice coefficients k.

Note If the k coefficients define a minimum phase filter, the resulting
filter in this structure is minimum phase. If your coefficients do not
define a minimum phase filter, placing them in this structure does not
produce a minimum phase filter.

Hd = dfilt.latticemamin returns a default discrete-time, lattice,
moving-average, minimum phase, filter, Hd, with k=[ ]. This filter
passes the input through to the output unchanged.

latticemamin
(Moving-Average, Minimum Phase Lattice)
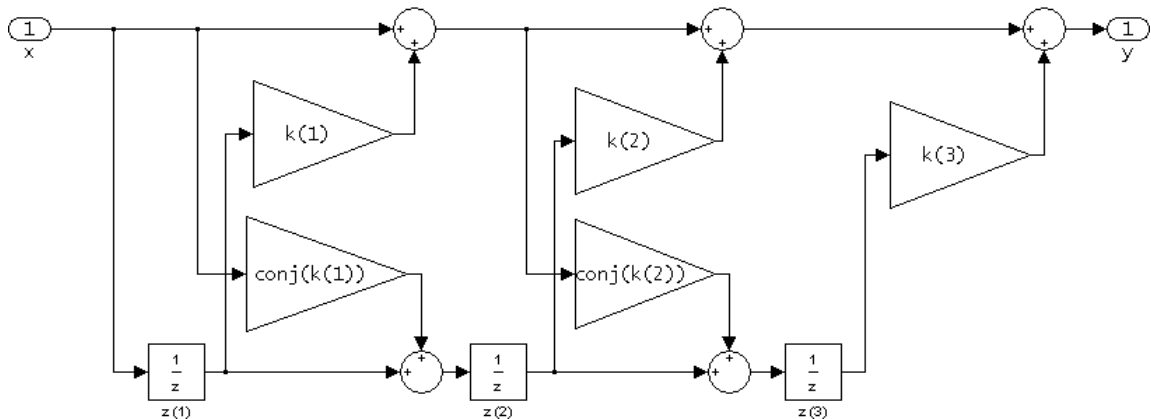
dfilt.latticemamin

The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

Examples   Form a third-order lattice, moving-average, minimum phase, filter structure for a dfilt object, Hd, using the following lattice coefficients.

```
k = [.66 .7 .44];
Hd = dfilt.latticemamin(k)
```

See Also   dfilt | dfilt.latticeallpass | dfilt.latticear | dfilt.latticearma | dfilt.latticemamax

Purpose   Discrete-time, parallel structure filter

Syntax   Hd = dfilt.parallel(Hd1,Hd2,...)

Description   Hd = dfilt.parallel(Hd1,Hd2,...) returns a discrete-time filter, Hd, which is a structure of two or more dfilt filters, Hd1, Hd2, etc. arranged in parallel. Each filter in a parallel structure is a separate stage. You can display states for individual stages only. To view the states of a stage use

Hd.stage(1).states

To append a filter (Hd1) onto an existing parallel filter (Hd), use

addstage(Hd,Hd1)

You can also use the nondot notation format for calling a parallel structure.

parallel(Hd1,Hd2,...)
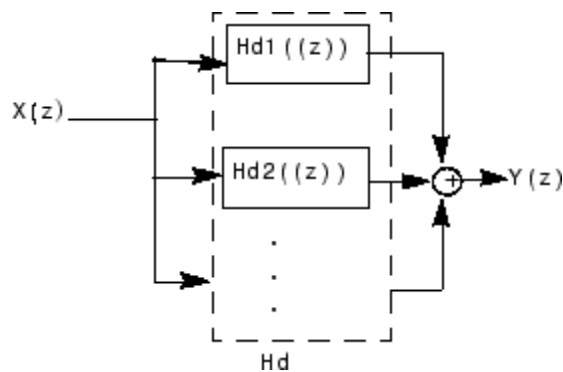
Examples   Using a parallel structure, create a coupled-allpass decomposition of a 7th order lowpass digital, elliptic filter with a normalized cutoff frequency of 0.5, 1 decibel of peak-to-peak ripple and a minimum stopband attenuation of 40 decibels.

```
k1 = [-0.0154    0.9846   -0.3048    0.5601];
Hd1 = dfilt.latticeallpass(k1);
k2 = [-0.1294    0.8341   -0.4165];
Hd2 = dfilt.latticeallpass(k2);
Hpar = parallel(Hd1 ,Hd2);
gain = dfilt.scalar(0.5);     % Normalize passband gain
Hcas = cascade(gain,Hpar);
```

For details on the stages of this filter, use

```
info(Hcas.Stage(1))
```

and

```
info(Hcas.Stage(2))
```

To view this filter, use

```
fvtool(Hcas)
```
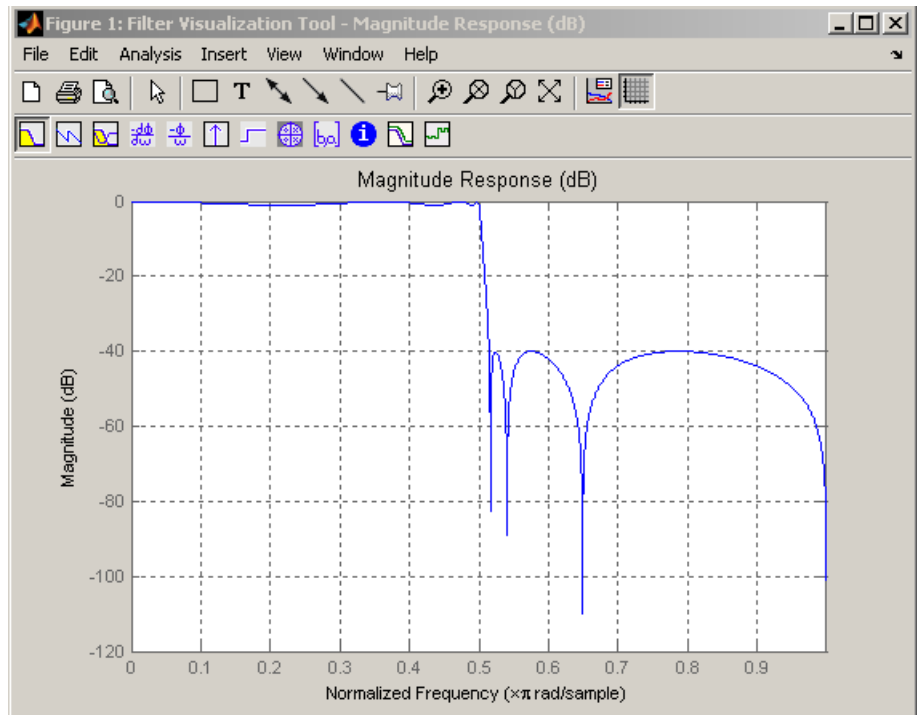
See Also   dfilt | dfilt.cascade

dfilt.scalar

Purpose	Discrete-time, scalar filter
Syntax	`Hd = dfilt.scalar(g)` `Hd = dfilt.scalar`
Description	`Hd = dfilt.scalar(g)` returns a discrete-time, scalar filter, `Hd`, with gain `g`, where `g` is a scalar. `Hd = dfilt.scalar` returns a default, discrete-time scalar gain filter, `Hd`, with gain `1`.
Examples	Create a direct-form I filter and a scalar object with a gain of 3 and cascade them together.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
Hd_filt = dfilt.df1(b,a);
Hd_gain = dfilt.scalar(3);
Hd_cascade = cascade(Hd_gain,Hd_filt);
hfvt = fvtool(Hd_filt,Hd_gain,Hd_cascade);
legend(hfvt,'Original Filter','Gain','Cascaded Filter',...
'location','southwest');
```
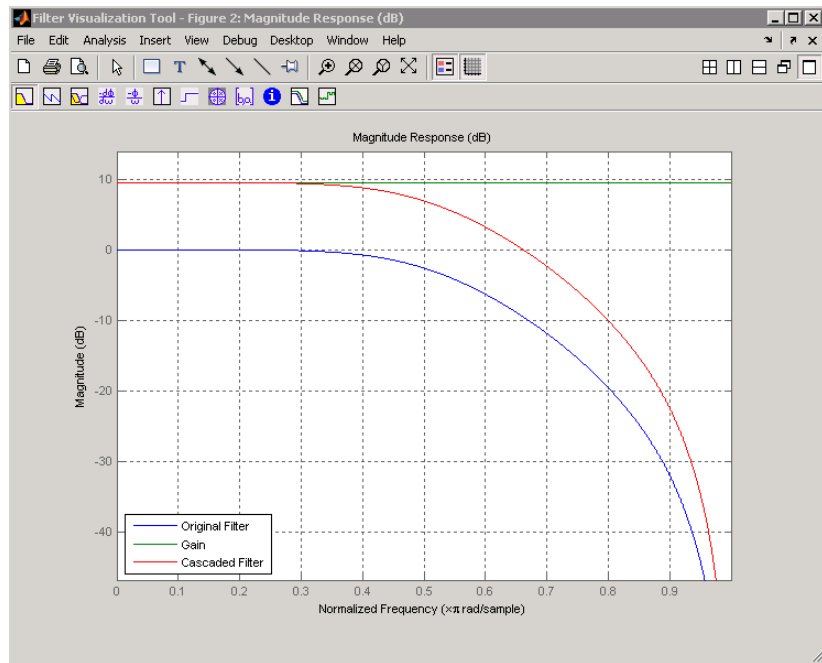
To view the stages of the cascaded filter, use

```
Hd.stage(1)
```

and

```
Hd.stage(2)
```

See Also   dfilt | dfilt.cascade

dfilt.statespace

Purpose   Discrete-time, state-space filter

Syntax   Hd = dfilt.statespace(A,B,C,D)
                 Hd = dfilt.statespace

Description Hd = dfilt.statespace(A,B,C,D) returns a discrete-time state-space filter, Hd, with rectangular arrays A, B, C, and D.

A, B, C, and D are from the matrix or state-space form of a filter's difference equations

$$x(n+1) = Ax(n) + Bu(n)$$
$$y(n) = Cx(n) + Du(n)$$

where $x(n)$ is the vector states at time n, $u(n)$ is the input at time n, y is the output at time n, A is the state-transition matrix, B is the input-to-state transmission matrix, C is the state-to-output transmission matrix, and D is the input-to-ouput transmission matrix. For single-channel systems, A is an m-by-m matrix where m is the order of the filter, B is a column vector, C is a row vector, and D is a scalar.

Hd = dfilt.statespace returns a default, discrete-time state-space filter, Hd, with A=[ ], B=[ ], C=[ ], and D=1. This filter passes the input through to the output unchanged.

Statespace
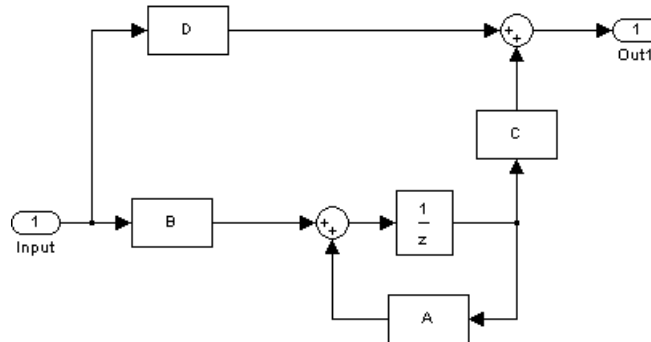
The resulting filter states column vector has the same number of rows as the number of rows of A or B.

Examples   Create a second-order, state-space filter structure from a second-order, lowpass Butterworth design.

```
[A,B,C,D] = butter(2,0.5);
Hd = dfilt.statespace(A,B,C,D)
```

See Also   dfilt

dftmtx

Purpose   Discrete Fourier transform matrix

Syntax   `A = dftmtx(n)`

Description   A *discrete Fourier transform matrix* is a complex matrix of values around the unit circle, whose matrix product with a vector computes the discrete Fourier transform of the vector.

`A = dftmtx(n)` returns the n-by-n complex matrix A that, when multiplied into a length n column vector x.

`y = A*x`

computes the discrete Fourier transform of x.

The inverse discrete Fourier transform matrix is

`Ai = conj(dftmtx(n))/n`

Examples   In practice, the discrete Fourier transform is computed more efficiently and uses less memory with an FFT algorithm

```
x = 1:256;
y1 = fft(x);
```

than by using the Fourier transform matrix.

```
n = length(x);
y2 = x*dftmtx(n);
norm(y1-y2)
```

Algorithms   dftmtx takes the FFT of the identity matrix to generate the transform matrix.

See Also   convmtx | fft

Purpose   Permute input into digit-reversed order

Syntax   y = digitrevorder(x,r)
                  [y,i] = digitrevorder(x,r)

Description   digitrevorder is useful for pre-ordering a vector of filter coefficients
                  for use in frequency-domain filtering algorithms, in which the fft
                  and ifft transforms are computed without digit-reversed ordering
                  for improved run-time efficiency.

                  y = digitrevorder(x,r) returns the input data in digit-reversed
                  order in vector or matrix y. The digit-reversal is computed using the
                  number system base (radix base) r, which can be any integer from 2 to
                  36. The length of x must be an integer power of r. If x is a matrix, the
                  digit reversal occurs on the first dimension of x with size greater than
                  1. y is the same size as x.

                  [y,i] = digitrevorder(x,r) returns the digit-reversed vector or
                  matrix y and the digit-reversed indices i, such that y = x(i). Recall
                  that MATLAB matrices use 1-based indexing, so the first index of y
                  will be 1, not 0.

                  The following table shows the numbers 0 through 15, the corresponding
                  digits and the digit-reversed numbers using radix base-4. The
                  corresponding radix base-2 bits and bit-reversed indices are also shown.

Linear Index	Base-4 Digits	Digit-Reversed	Digit-Reversed Index	Base-2 Bits	Base-2 Reversed (bitrevorder)	Bit-Reversed Index
0	00	00	0	0000	0000	0
1	01	10	4	0001	1000	8
2	02	20	8	0010	0100	4
3	03	30	12	0011	1100	12
4	10	01	1	0100	0010	2
5	11	11	5	0101	1010	10

digitrevorder

Linear Index	Base-4 Digits	Digit-Reversed	Digit-Reversed Index	Base-2 Bits	Base-2 Reversed (bitrevorder)	Bit-Reversed Index
6	12	21	9	0110	0110	6
7	13	31	13	0111	1110	14
8	20	02	2	1000	0001	1
9	21	12	6	1001	1001	9
10	22	22	10	1010	0101	5
11	23	32	14	1011	1101	13
12	30	03	3	1100	0011	3
13	31	13	7	1101	1011	11
14	32	23	11	1110	0111	7
15	33	33	15	1111	1111	15

Examples   Obtain the digit-reversed, radix base-3 ordered output of a vector
containing 9 values:

```
x=[0:8]';                    % Create a column vector
[x,digitrevorder(x,3)]
% ans =
%
%    0    0
%    1    3
%    2    6
%    3    1
%    4    4
%    5    7
%    6    2
%    7    5
%    8    8
```

See Also   bitrevorder | fft | ifft

Purpose Dirichlet or periodic sinc function

Syntax y = diric(x,n)

Description y = diric(x,n) returns a vector or array y the same size as x. The elements of y are the Dirichlet function of the elements of x. n must be a positive integer.

The Dirichlet function, or periodic sinc function, is

$$D(x) = \begin{cases} \dfrac{\sin(Nx/2)}{N\sin(x/2)} & x \neq 2\pi k, \quad k = 0, \pm 1, \pm 2, \pm 3, \ldots \\ (-1)^{k(N-1)} & x = 2\pi k, \quad k = 0, \pm 1, \pm 2, \pm 3, \ldots \end{cases}$$

for any nonzero integer n. This function has period 2π for n odd and period 4π for n even. Its peak value is 1, and its minimum value is -1 for n even. The magnitude of this function is (1/n) times the magnitude of the discrete-time Fourier transform of the n-point rectangular window.

Diagnostics If n is not a positive integer, diric gives the following error message:

Requires n to be a positive integer.

See Also cos | gauspuls | pulstran | rectpuls | sawtooth | sin | sinc | square | tripuls

downsample

Purpose	Decrease sampling rate by integer factor
Syntax	`y = downsample(x,n)` `y = downsample(x,n,phase)`
Description	`y = downsample(x,n)` decreases the sampling rate of x by keeping every n-th sample starting with the first sample. x can be a vector or a matrix. If x is a matrix, each column is considered a separate sequence. `y = downsample(x,n,phase)` specifies the number of samples by which to offset the downsampled sequence. phase must be an integer from 0 to n-1.
Examples	Decrease the sampling rate of a sequence by 3:

```
x = [1 2 3 4 5 6 7 8 9 10];
y = downsample(x,3)
% y =  1    4    7    10
```

Decrease the sampling rate of the sequence by 3 and add a phase offset of 2:

```
y = downsample(x,3,2)
% y = 3    6    9
```

Decrease the sampling rate of a matrix by 3:

```
x = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
y = downsample(x,3);
```

See Also   decimate | interp | interp1 | resample | spline | upfirdn | upsample

Purpose   Discrete prolate spheroidal (Slepian) sequences

Syntax   ```
                dps_seq = dpss(seq_length,time_halfbandwidth)
                [dps_seq,lambda] = dpss(seq_length,time_halfbandwidth)
                [...] = dpss(seq_length,time_halfbandwidth,num_seq)
                [...] = dpss(seq_length,time_halfbandwidth,'interp_method')
                [...] = dpss(...,Ni)
                [...] = dpss(...,'trace')
                ```

**Description**  dps_seq = dpss(seq_length,time_halfbandwidth) returns the first
                round(2*time_halfbandwidth) discrete prolate spheroidal (DPSS),
                or Slepian sequences of length seq_length. dps_seq is a matrix
                with seq_length rows and round(2*time_halfbandwidth) columns.
                time_halfbandwidth must be strictly less than seq_length/2.

                [dps_seq,lambda] = dpss(seq_length,time_halfbandwidth)
                returns the frequency-domain energy concentration ratios of the column
                vectors in dps_seq. The ratios represent the amount of energy in the
                passband *[-W,W]* to the total energy from *[-Fs/2, Fs/2]* where *Fs* is
                the sampling frequency. lambda is a column vector equal in length to
                the number of Slepian sequences.

                [...] = dpss(seq_length,time_halfbandwidth,num_seq) returns
                the first num_seq Slepian sequences with time half bandwidth product
                time_halfbandwidth ordered by their energy concentration ratios. If
                num_seq is a two-element vector, the returned Slepian sequences range
                from num_seq(1) to num_seq(2).

                [...] =
                dpss(seq_length,time_halfbandwidth,'interp_method') uses
                interpolation to compute the DPSSs from a user-created database of
                DPSSs. Create the database of DPSSs with dpsssave and ensure that
                the resulting dpss.mat file is in the MATLAB search path. Valid
                options for 'interp_method' are 'spline' and 'linear'.
                The interpolation method uses the Slepian sequences in the
                database with time half bandwidth product time_halfbandwidth
                and length closest to seq_length.

[...]  = dpss(...,Ni) interpolates from DPSSs of length Ni in the database dpss.mat.

[...]  = dpss(...,'trace') prints the method used to compute the DPSSs in the command window. Possible methods include: direct, spline interpolation, and linear interpolation.

## Definitions

### Discrete Prolate Spheroidal Sequences

The discrete prolate spheroidal or Slepian sequences derive from the following time-frequency concentration problem. For all finite-energy sequences $x[n]$ index limited to some set $[N_1, N_1 + N_2]$, which sequence maximizes the following ratio:

$$\lambda = \frac{\int_{-W}^{W} |X(f)|^2 df}{\int_{-Fs/2}^{Fs/2} |X(f)|^2 \, df}$$

where $Fs$ is the sampling frequency and $|W| < Fs/2$. Accordingly, this ratio determines which index-limited sequence has the largest proportion of its energy in the band *[-W,W]*. For index-limited sequences, the ratio must satisfy the inequality $0 < \lambda < 1$. The sequence maximizing the ratio is the first discrete prolate spheroidal or Slepian sequence. The second Slepian sequence maximizes the ratio and is orthogonal to the first Slepian sequence. The third Slepian sequence maximizes the ratio of integrals and is orthogonal to both the first and second Slepian sequences. Continuing in this way, the Slepian sequences form an orthogonal set of band limited sequences.

### Time Half Bandwidth Product

The time half bandwidth product is *NW* where *N* is the length of the sequence and *[-W,W]* is the effective bandwidth of the sequence. In constructing Slepian sequences, you choose the desired sequence length and bandwidth *2W*. Both the sequence length and bandwidth affect how many Slepian sequences have concentration ratios near one. As a rule,

there are *2NW-1* Slepian sequences with energy concentration ratios approximately equal to one. Beyond *2NW-1* Slepian sequences, the concentration ratios begin to approach zero. Common choices for the time half bandwidth product are: 2.5, 3, 3.5, and 4.

You can specify the bandwidth of the Slepian sequences in Hz by defining the time half bandwidth product as *NW/Fs* where *Fs* is the sampling frequency.

**Examples**     Construct a set of Slepian sequences:

```
seq_length = 512;
time_halfbandwidth = 2.5;
num_seq = 2*(2.5)-1;
%Obtain DPSSs
[dps_seq,lambda] = dpss(seq_length,time_halfbandwidth,num_seq);
% Plot the Slepian sequences
plot(dps_seq);
title('Slepian Sequences N=512, NW=2.5');
axis([O 512 -0.15 0.15]);
legend('1st','2nd','3rd','4th');
%Concentration ratios in lambda:
%1.0000    0.9998    0.9962    0.9521
```

References   Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications.* Cambridge: Cambridge University Press, 1993.

See Also   dpssclear | dpssload | dpsssave | spectrum.mtm

How To   • "Nonparametric Methods"

Purpose   Remove discrete prolate spheroidal sequences from database

Syntax   `dpssclear(n,nw)`

Description   `dpssclear(n,nw)` removes sequences with length `n` and time-bandwidth product `nw` from the DPSS MAT-file database `dpss.mat`.

See Also   dpss | dpssdir | dpssload | dpsssave

dpssdir

Purpose   Discrete prolate spheroidal sequences database directory

Syntax
```
dpssdir
dpssdir(n)
dpssdir(nw,'nw')
dpssdir(n,nw)
index = dpssdir
```

Description   dpssdir manages the database directory that contains the generated DPSS samples in the DPSS MAT-file database dpss.mat. Create the DPSS MAT-file database with dpsssave.

dpssdir lists the directory of saved sequences in dpss.mat.

dpssdir(n) lists the sequences saved with length n.

dpssdir(nw,'nw') lists the sequences saved with time-bandwidth product nw.

dpssdir(n,nw) lists the sequences saved with length n and time-bandwidth product nw.

index = dpssdir is a structure array describing the DPSS database. Pass n and nw options as for the no output case to get a filtered index.

See Also   dpss | dpssclear | dpssload | dpsssave

Purpose   Load discrete prolate spheroidal sequences from database

Syntax   `[e,v] = dpssload(n,nw)`

Description `[e,v] = dpssload(n,nw)` loads all sequences with length `n` and
time-bandwidth product `nw` in the columns of `e` and their corresponding
concentrations in vector `v` from the DPSS MAT-file database `dpss.mat`.
Create the `dpss.mat` file using `dpssave`.

See Also   dpss | dpssclear | dpssdir | dpsssave

dpsssave

Purpose	Discrete prolate spheroidal or Slepian sequence database
Syntax	dpsssave(time_halfbandwith,dps_seq,lambda) status = dpsssave(time_halfbandwith,dps_seq,lambda)

Description   dpsssave(time_halfbandwith,dps_seq,lambda) creates a
database of discrete prolate spheroidal (DPSS) or Slepian sequences
and saves the results in dpss.mat. The time half bandwidth
producttime_halfbandwith is a real-valued scalar determining
the frequency concentration of the Slepian sequences in dps_seq.
dps_seq is a *NxK* matrix of Slepian sequences where *N* is the length
of the sequences. lambda is a *1xK* vector containing the frequency
concentration ratios of the Slepian sequences in dps_seq.

If the database dpss.mat exists, subsequent calls to dpsssave append
the Slepian sequences to the existing file.

status = dpsssave(time_halfbandwith,dps_seq,lambda) returns a
1 if the database operation was successful or a 0 if unsuccessful.

Definitions   **Discrete Prolate Spheroidal Sequences**

The discrete prolate spheroidal or Slepian sequences derive from the
following time-frequency concentration problem. For all finite-energy

sequences $x[n]$ index limited to some set $[N_1, N_1 + N_2]$, which sequence
maximizes the following ratio:

$$\lambda = \frac{\displaystyle\int_{-W}^{W} |X(f)|^2 df}{\displaystyle\int_{-Fs/2}^{Fs/2} |X(f)|^2 \, df}$$

where *Fs* is the sampling frequency $|W| < Fs/2$. In other words, which
index-limited sequence has the largest proportion of its energy in the
band *[-W,W]*. For index-limited sequences, the ratio must satisfy the
inequality $0 < \lambda < 1$. The sequence maximizing the ratio is the first

discrete prolate spheroidal or Slepian sequence. The second Slepian sequence maximizes the ratio and is orthogonal to the first Slepian sequence. The third Slepian sequence maximizes the ratio of integrals and is orthogonal to both the first and second Slepian sequences. Continuing in this way, the Slepian sequences form an orthogonal set of band limited sequences.

Time Half Bandwidth Product

The time half bandwidth product is NW where N is the length of the sequence and $[-W,W]$ is the effective bandwidth of the sequence. In constructing Slepian sequences, you choose the desired sequence length and bandwidth $2W$. Both the sequence length and bandwidth affect how many Slepian sequences have concentration ratios near one. As a rule, there are $2NW-1$ Slepian sequences with energy concentration ratios approximately equal to one. Beyond $2NW-1$ Slepian sequences, the concentration ratios begin to approach zero. Common choices for the time half bandwidth product are: 2.5, 3, 3.5, and 4.

You can specify the bandwidth of the Slepian sequences in Hz by defining the time half bandwidth product as NW/Fs where Fs is the sampling frequency.

Examples   Create Slepian sequence database in current directory:

```
seq_length=512;
time_halfbandwidth=2.5;
num_seq=4;
[dps_seq,lambda]=dpss(seq_length,time_halfbandwidth);
% Create databased dpss.mat in current working directory
status=dpsssave(time_halfbandwidth,dps_seq,lambda);
% status should equal 1
```

References   Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications.* Cambridge: Cambridge University Press, 1993.

See Also   dpss | dpssclear | dpssdir | dpssload

dspdata

Purpose	DSP data parameter information

Syntax   Hs = dspdata.*dataobj*(input1,...)

Description   Hs = dspdata.*dataobj*(input1,...) returns a dspdata object Hs
of type *dataobj*. This object contains all the parameter information
needed for the specified type of *dataobj*. Each *dataobj* takes one or
more inputs, which are described on the individual reference pages. If
you do not specify any input values, the returned object has default
property values appropriate for the particular *dataobj* type.

> **Note** You must use a *dataobj* with dspdata.

Data Objects

A data object (*dataobj*) for dspdata specifies the type of data stored in
the object. Available *dataobj* types for dspdata are shown below.

dspdata.dataobj	Description
dspdata.msspectrum	Mean-square spectrum data (power)
dspdata.psd	Power spectral density data (power/frequency)
dspdata.pseudospectrum	Pseudospectrum data (power)

For more information on each *dataobj* type, use the syntax
help dspdata.*dataobj* at the MATLAB prompt or refer to its reference
page.

Methods

Methods provide ways of performing functions directly on your dspdata
object. You can apply these methods directly on the variable you
assigned to your dspdata object.

Method	Description
avgpower	Note that this method applies only to dspdata.psd objects.
	avgpower(Hs) computes the average power in a given frequency band. The technique uses a rectangle approximation of the integral of the Hs signal's power spectral density (PSD). If the signal is a matrix, the computation is done on each column. The average power is the total signal power and the SpectrumType property determines whether the total average power is contained in the one-sided or two-sided spectrum. For aa one-sided spectrum, the range is [0,pi] for even number of frequency points and [0,pi) for odd. For a two-sided spectrum the range is [0,2pi).
	avgpower(Hs,freqrange) specifies the frequency range over which to calculate the average power. freqrange is a two-element vector of the frequencies between which to calculate. If a frequency value does not match exactly the frequency in Hs, the next closest value is used. Note that the first frequency value in freqrange is included in the calculation and the second value is excluded.
centerdc	centerdc(Hs) or centerdc(Hs,true) shifts the data and frequency values so that the DC component is at the center of the spectrum. If the SpectrumType property is 'onesided', it is changed to 'twosided' and then the DC component is centered.
	centerdc(Hs,'false') shifts the data and frequency values so that the DC component is at the left edge of the spectrum.

Method	Description
findpeaks	findpeaks(Hs) finds local maxima or peaks. If no peaks are found, findpeaks returns an empty vector.
	[pks,frqs] = findpeaks(x) returns peaks values (pks) and the frequencies (frqs) at which the peaks occur.
	findpeaks(x,'minpeakheight',mph) returns only peaks greater than the minimum peak height mph, where mph is a real scalar. Default is -Inf.
	findpeaks(x,'minpeakdistance',mpd) returns only peaks separated by the minimum frequency units distance mpd, which is a positive integer. Setting the minimum peak distance ignores smaller peaks that may occur close to larger local peaks. Default is 1.
	findpeaks(x,'threshold',th) returns only peaks greater than their neighbors by at least the threshold th, which is a real, scalar value greater than or equal to 0. Default is 0.
	findpeaks(x,'npeaks',np) returns a maximum of np number of peaks. When np peaks are found, the search stops. Default is to return all peaks.
	findpeaks(x,'sortstr',str) specifies the sorting order, where str is 'ascend', 'descend' or 'none'. For 'ascend', the peaks are returned in order from smallest to largest, and vice versa for 'descend'. For 'none', the peaks are returned in the order in which they occur.

Method	Description
halfrange	halfrange(Hs) converts the Hs spectrum to a spectrum calculated over half the Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for dspdata.pseudospectrum objects. Note that the spectrum is assumed to be from a real signal (that is, halfrange uses half the data points regardless of whether the data is symmetric).
normalizefreq	normalizefreq(Hs) or normalizefreq(Hs,true) normalizes the frequency specifications in the Hs object to Fs so the frequencies are between 0 and 1. It also sets the NormalizedFrequency property to true. normalizefreq(Hs,false) converts the frequencies to linear frequencies. normalizefreq(Hs,false,Fs) sets a new sampling frequency Fs. This can be used only with false.
onesided	onesided(Hs) converts the Hs spectrum to a spectrum calculated over half the Nyquist interval and containing the total signal power. All associated properties affected by the new frequency range are adjusted automatically. This method is used for dspdata.psd and dspdata.msspectrum objects. Note that the spectrum is assumed to be from a real signal (that is, onesided uses half the data points regardless of whether the data is symmetric).

Method	Description
plot	Displays the data graphically in the current figure window.
	For a `dspdata.psd` object, it displays the power spectral density in dB/Hz.
	For a `dspdata.msspectrum` object, it displays the mean–square in dB.
	For a `dspdata.pseudospectrum` object, it displays the pseudospectrum in dB.
sfdr	This method applies only to `dspdata.msspectrum` objects.
	`sfdr(Hs)` computes the spurious-free dynamic range (SFDR) in dB of a mean square spectrum object `Hs`. SFDR is the usable range before spurious noise interferes with the signal.
	`[sfd,spur,frq] = sfdr(Hs)` returns the magnitude of the highest `spur` and the frequency `frq` at which it occurs.
	`sfdr(Hs,'minspurlevel',msl)` ignores spurs below the minimum spur level `msl`, which is a real scalar in dB.
	`sfdr(Hs,'minspurdistance',msd)` includes spurs only if they are separated by at least the minimum spur distance `msd`, which is a real, positive scalar in frequency units.

Method	Description
twosided	twosided(Hs) converts the Hs spectrum to a spectrum calculated over the whole Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for dspdata.psd and dspdata.msspectrum objects. Note that if your data is nonuniformly sampled, converting from onesided to twosided may produce incorrect results.
wholerange	wholerange(Hs) converts the Hs spectrum to a spectrum calculated over the whole Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for dspdata.pseudospectrum objects. Note that if your data is nonuniformly sampled, converting from half to wholerange may produce incorrect results.

For more information on each method, use the syntax help dspdata/*method* at the MATLAB prompt.

Plotting a dspdata Object

The plot method displays the dspdata object spectrum in a separate figure window.

```
plot(Hs)          % Plots an existing Hs object
```

Modifying a dspdata Object

After you create a dspdata object, you can use any of the methods in the table above to modify the object properties.

For example, to change the object from two-sided to one-sided, use

```
onesided(Hs)
```

The Hs object is modified.

Examples   See the msspectrum, psd, or pseudospectrum reference pages for specific examples.

See Also   dspdata.msspectrum | dspdata.psd | dspdata.pseudospectrum

dspdata.msspectrum

Purpose	Mean-square (power) spectrum

Syntax

```
Hmss = dspdata.msspectrum(Data)
Hmss = dspdata.msspectrum(Data,Frequencies)
Hmss = dspdata.msspectrum(...,'Fs',Fs)
Hmss = dspdata.msspectrum(...,'SpectrumType',SpectrumType)
Hmss = dspdata.msspectrum(...,'CenterDC',flag)
```

Description

The mean-squared spectrum (MSS) is intended for discrete spectra. Unlike the power spectral density (PSD), the peaks in the MSS reflect the power in the signal at a given frequency. The MSS of a signal is the Fourier transform of that signal's autocorrelation.

`Hmss = dspdata.msspectrum(Data)` uses the mean-square (power) spectrum data contained in `Data`, which can be in the form of a vector or a matrix, where each column is a separate set of data. Default values for other properties of the object are as follows:

Property	Default Value	Description
Name	`'Mean-square Spectrum'`	Read-only string
Frequencies	`[]` type `double`	Vector of frequencies at which the spectrum is evaluated. The range of this vector depends on the `SpectrumType` value. For a one-sided spectrum, the default range is [0, pi) or [0, `Fs`/2) for odd length, and [0, pi] or [0, `Fs`/2] for even length, if `Fs` is specified. For a two-sided spectrum, it is [0, 2pi) or [0, `Fs`).
		The length of the `Frequencies` vector must match the length of the columns of `Data`.
		If you do not specify `Frequencies`, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even.
		If `onesided` is selected and you specify `Frequencies`, the last frequency point is compared to the next-to-last point and to pi (or `Fs`/2, if `Fs` is specified). If the last point is closer to pi (or `Fs`/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd.
Fs	`'Normalized'`	Sampling frequency, which is `'Normalized'` if `NormalizedFrequency` is `true`. If `NormalizedFrequency` is `false` `Fs` defaults to 1 Hz.

dspdata.msspectrum

Property	Default Value	Description
SpectrumType	'Onesided'	Nyquist interval over which the spectral density is calculated. Valid values are 'Onesided' and 'Twosided'. See the onesided and twosided methods in dspdata for information on changing this property.
		The interval for Onesided is [0 pi) or [0 pi] depending on the number of FFT points, and for Twosided the interval is [0 2pi).
NormalizedFrequency	true	Whether the frequency is normalized (true) or not (false). This property is set automatically at construction time based on Fs. If Fs is specified, NormalizedFrequency is set to false. See the normalizefreq method in dspdata for information on changing this property.

Hmss = dspdata.msspectrum(Data,Frequencies) uses the mean–square spectrum data contained in Data and Frequencies vectors.

Hmss = dspdata.msspectrum(...,'Fs',Fs) uses the sampling frequency Fs. Specifying Fs uses a default set of linear frequencies (in Hz) based on Fs and sets NormalizedFrequency to false.

Hmss = dspdata.msspectrum(...,'SpectrumType',SpectrumType) uses the SpectrumType string to specify the interval over which the mean–square spectrum was calculated. For data that ranges from [0 pi) or [0 pi], set the SpectrumType to onesided; for data that ranges from [0 2pi), set the the SpectrumType to twosided.

Hmss = dspdata.msspectrum(...,'CenterDC',flag) uses the value of flag to indicate whether the zero-frequency (DC) component is centered. If flag is true, it indicates that the DC component is in

the center of the two-sided spectrum. Set the `flag` to `false` if the DC component is on the left edge of the spectrum.

Methods

Methods provide ways of performing functions directly on your `dspdata` object without having to specify the parameters again. You can apply a method directly on the variable you assigned to your `dspdata.msspectrum` object. You can use the following methods with a `dspdata.msspectrum` object.

- centerdc

- normalizefreq

- onesided

- plot

- sfdr

- twosided

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to true, use

```
Hmss = normalizefreq(Hs)
```

For detailed information on using the methods and plotting the spectrum, see the `dspdata` reference page.

Examples   In this example, we construct a mean-square spectrum data object from the one-sided PSD estimate of a signal. The signal consists of two sinusoids in additive noise.

```
Fs = 32e3;
t = 0:1/Fs:1-(1/Fs);
x = cos(2*pi*t*1.24e3)+cos(2*pi*t*10e3)+randn(size(t));
X = fft(x);
X=X(1:length(X)/2+1); %one-sided DFT
P = (abs(X)/length(x)).^2;     % Compute the mean-square power
P(2:end-1)=2*P(2:end-1); % Factor of two for one-sided estimate
```

dspdata.msspectrum

```
% at all frequencies except zero and the Nyquist
Hmss=dspdata.msspectrum(P,'Fs',Fs,'spectrumtype','onesided');
plot(Hmss);            % Plot the mean-square spectrum.
```


See Also dspdata.psd | dspdata.pseudospectrum | spectrum

Purpose   Power spectral density

Syntax   Hpsd = dspdata.psd(Data)
                Hpsd = dspdata.psd(Data,Frequencies)
                Hpsd = dspdata.psd(...,'Fs',Fs)
                Hpsd = dspdata.psd(...,'SpectrumType',SpectrumType)
                Hpsd = dspdata.psd(...,'CenterDC',flag)

Description The power spectral density (PSD) is intended for continuous spectra. The integral of the PSD over a given frequency band computes the average power in the signal over that frequency band. In contrast to the mean-squared spectrum, the peaks in this spectra do not reflect the power at a given frequency. See the avgpower method of dspdata for more information.

A one-sided PSD contains the total power of the signal in the frequency interval from DC to half of the Nyquist rate. A two-sided PSD contains the total power in the frequency interval from DC to the Nyquist rate.

Hpsd = dspdata.psd(Data) uses the power spectral density data contained in Data, which can be in the form of a vector or a matrix, where each column is a separate set of data. Default values for other properties of the object are shown below:

Property	Default Value	Description
Name	'Power Spectral Density'	Read-only string
Frequencies	[] type double	Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the SpectrumType value. For one-sided, the default range is [0, pi) or [0, Fs/2) for odd length, and [0, pi] or [0, Fs/2] for even length, if Fs is specified. For two-sided, it is [0, 2pi) or [0, Fs).

Property	Default Value	Description
		If you do not specify Frequencies, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even.
		If onesided is selected and you specify Frequencies, the last frequency point is compared to the next-to-last point and to pi (or Fs/2, if Fs is specified). If the last point is closer to pi (or Fs/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd.
		The length of the Frequencies vector must match the length of the columns of Data.
Fs	'Normalized'	Sampling frequency, which is 'Normalized' if NormalizedFrequency is true. If NormalizedFrequency is false Fs defaults to 1.

Property	Default Value	Description
SpectrumType	'Onesided'	Nyquist interval over which the power spectral density is calculated. Valid values are 'Onesided' and 'Twosided'. A one-sided PSD contains the total signal power in half the Nyquist interval. See the onesided and twosided methods in dspdata for information on changing this property. The range for half the Nyquist interval is [0 pi) or [0 pi] depending on the number of FFT points. For the whole Nyquist interval, the range is [0 2pi).
NormalizedFrequency	true	Whether the frequency is normalized (true) or not (false). This property is set automatically at construction time based on Fs. If Fs is specified, NormalizedFrequency is set to false. See the normalizefreq method in dspdata for information on changing this property.

Hpsd = dspdata.psd(Data,Frequencies) uses the power spectral density estimation data contained in Data and Frequencies vectors.

Hpsd = dspdata.psd(...,'Fs',Fs) uses the sampling frequency Fs. Specifying Fs uses a default set of linear frequencies (in Hz) based on Fs and sets NormalizedFrequency to false.

Hpsd = dspdata.psd(...,'SpectrumType',SpectrumType) uses the SpectrumType string to specify the interval over which the power spectral density was calculated. For data that ranges from [0 pi) or [0 pi], set the SpectrumType to onesided; for data that ranges from [0 2pi), set the SpectrumType to twosided.

Hpsd = dspdata.psd(...,'CenterDC',flag) uses the value of flag to indicate whether the zero-frequency (DC) component is centered. If

flag is `true`, it indicates that the DC component is in the center of the two-sided spectrum. Set the `flag` to `false` if the DC component is on the left edge of the spectrum.

Methods

Methods provide ways of performing functions directly on your `dspdata` object. You can apply a method directly on the variable you assigned to your `dspdata.psd` object. You can use the following methods with a `dspdata.psd` object.

- avgpower

- centerdc

- normalizefreq

- onesided

- plot

- twosided

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to true, use

```
Hpsd = normalizefreq(Hpsd)
```

For detailed information on using the methods and plotting the spectrum, see the `dspdata` reference page.

Examples   **Resolving Signal Components**

Estimate the power spectral density of a noisy sinusoidal signal with two frequency components and then store the results in a PSD data object and plot it.

```
Fs = 32e3;
t = 0:1/Fs:2.96;
x = cos(2*pi*t*1.24e3)+ cos(2*pi*t*10e3)+ randn(size(t));
nfft = 2^nextpow2(length(x));
Pxx = abs(fft(x,nfft)).^2/length(x)/Fs;
```

```
% Create a single-sided spectrum
Hpsd = dspdata.psd(Pxx(1:length(Pxx)/2),'Fs',Fs);
plot(Hpsd);
```



```
% Create a double-sided spectrum
Hpsd = dspdata.psd(Pxx,'Fs',Fs,'SpectrumType','twosided');
plot(Hpsd)
```

dspdata.psd

See Also   dspdata.msspectrum | dspdata.pseudospectrum | spectrum

Purpose   Pseudospectrum dspdata object

Syntax   Hps = dspdata.pseudospectrum(Data)
                Hps = dspdata.pseudospectrum(Data,Frequencies)
                Hps = dspdata.pseudospectrum(...,'Fs',Fs)
                Hps = dspdata.pseudospectrum...,'SpectrumRange',SpectrumRange)
                Hps = dspdata.pseudospectrum(...,'CenterDC',flag)

Description A pseudospectrum is an indicator of the presence of sinusoidal
                components in a signal.

                Hps = dspdata.pseudospectrum(Data) uses the pseudospsectrum
                data contained in Data, which can be in the form of a vector or a matrix,
                where each column is a separate set of data. Default values for other
                properties of the object are:

Property	Default Value	Description
Name	`'Pseudospectrum'`	Read-only string
Frequencies	`[]` type `double`	Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the `SpectrumRange` value. For half, the default range is [0, pi) or [0, `Fs`/2) for odd length, and [0, pi] or [0, `Fs`/2] for even length, if `Fs` is specified. For whole, it is [0, 2pi) or [0, `Fs`).
		If you do not specify `Frequencies`, a default vector is created. If half the Nyquist range is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even.
		If `half` the Nyquist range is selected and you specify `Frequencies`, the last frequency point is compared to the next-to-last point and to pi (or `Fs`/2, if `Fs` is specified). If the last point is closer to pi (or `Fs`/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd.
		The length of the `Frequencies` vector must match the length of the columns of `Data`.
Fs	`'Normalized'`	Sampling frequency, which is `'Normalized'` if `NormalizedFrequency` is `true`. If `NormalizedFrequency` is `false` `Fs` defaults to `1`.

Property	Default Value	Description
SpectrumRange	'Half'	Nyquist interval over which the pseudospectrum is calculated. Valid values are 'Half' and 'Whole'. See the half and whole methods in dspdata for information on changing this property. The interval for Half is [0 pi) or [0 pi] depending on the number of FFT points, and for Whole the interval is [0 2pi).
NormalizedFrequency	true	Whether the frequency is normalized (true) or not (false). This property is set automatically at construction time based on Fs. If Fs is specified, NormalizedFrequency is set to false. See the normalizefreq method in dspdata for information on changing this property.

Hps = dspdata.pseudospectrum(Data,Frequencies) uses the pseudospectrum estimation data contained in the Data and Frequencies vectors.

Hps = dspdata.pseudospectrum(...,'Fs',Fs) uses the sampling frequency Fs. Specifying Fs uses a default set of linear frequencies (in Hz) based on Fs and sets NormalizedFrequency to false.

Hps = dspdata.pseudospectrum...,'SpectrumRange',SpectrumRange) uses the SpectrumRange string to specify the interval over which the pseudospectrum was calculated. For data that ranges from [0 pi) or [0 pi], set the SpectrumRange to half; for data that ranges from [0 2pi), set the SpectrumRange to whole.

Hps = dspdata.pseudospectrum(...,'CenterDC',flag) uses the value of flag to indicate whether the zero-frequency (DC) component is centered. If flag is true, it indicates that the DC component is in the center of the whole Nyquist range spectrum. Set the flag to false if the DC component is on the left edge of the spectrum.

dspdata.pseudospectrum

Methods

Methods provide ways of performing functions directly on your dspdata object. You can apply a method directly on the variable you assigned to your dspdata.pseudospectrum object. You can use the following methods with a dspdata.pseudospectrum object.

- centerdc
- halfrange
- normalizefreq
- plot
- wholerange

For example, to normalize the frequency and set the NormalizedFrequency parameter to true, use

```
Hps = normalizefreq(Hps)
```

For detailed information on using the methods and plotting the pseudospectrum, see the dspdata reference page.

Examples **Storing and Plotting Pseudospectrum Data**

Use eigenanalysis to estimate the pseudospectrum of a noisy sinusoidal signal with two frequency components. Then store the results in a pseudospectrum data object and plot it.

```
Fs = 32e3;
t  = 0:1/Fs:2.96;
x = cos(2*pi*t*1.24e3) + cos(2*pi*t*10e3) + randn(size(t));
P = pmusic(x,4);
% Create data object
hps = dspdata.pseudospectrum(P,'Fs',Fs);
% Plot the pseudospectrum
plot(hps);
```

See Also   dspdata.msspectrum | dspdata.psd | spectrum

dspfwiz

Purpose   Open FDATool Realize Model panel to create Simulink filter block

Syntax   dspfwiz

Description

Note You must have the Simulink product installed to use this function.

dspfwiz opens FDATool with the Realize Model panel displayed.

Use other panels in FDATool to design your filter and then use the
Realize Model panel to create your filter as a subsystem block, which is
a combination of Sum, Gain, and Delay blocks, in a Simulink model.

If you also have the DSP System Toolbox software installed, you can
create a Digital Filter block instead of a subsystem block, by deselecting
the **Build model using basic elements** check box.

See Also   fdatool | dfilt

Purpose   Duty cycle of pulse waveform

Syntax
```
D = dutycycle(X)
D = dutycycle(X,FS)
D = dutycycle(X,T)
D = dutycycle(TAU,PRF)
[D,INITCROSS] = dutycycle(X,...)
[D,INITCROSS,FINALCROSS] = dutycycle(X,...)
[D,INITCROSS,FINALCROSS,NEXTCROSS] = dutycycle(X,...)
[D,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] =
dutycycle(X,...)
[D,INITCROSS,FINALCROSS,NEXTCROSS] = dutycycle(X,...,Name,
    Value)
dutycycle(X,...)
```

Description   `D = dutycycle(X)` returns the ratio of pulse width to pulse period for each positive-polarity pulse. `D` has length equal to the number of pulse periods in `X`. The sample instants of `X` correspond to the indices of `X`. To determine the transitions that define each pulse, `dutycycle` estimates the state levels of the input waveform by a histogram method. `dutycycle` identifies all regions, which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See "State-Level Tolerances" on page 3-255.

`D = dutycycle(X,FS)` specifies the sampling frequency, `FS`, in hertz as a positive scalar. The first sample instant of `X` corresponds to t=0.

`D = dutycycle(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

`D = dutycycle(TAU,PRF)` returns the ratio of pulse width to pulse period for a pulse width of `TAU` seconds and a pulse repetition frequency of `PRF`. The product of `TAU` and `PRF` must be less than or equal to 1.

`[D,INITCROSS] = dutycycle(X,...)` returns a vector, `INITCROSS`, whose elements correspond to the mid-crossings (mid-reference level

instants) of the initial transition of each pulse with a corresponding NEXTCROSS.

[D,INITCROSS,FINALCROSS] = dutycycle(X,...) returns a vector, FINALCROSS, whose elements correspond to the mid-crossings (mid-reference level instants) of the final transition of each pulse with a corresponding NEXTCROSS.

[D,INITCROSS,FINALCROSS,NEXTCROSS] = dutycycle(X,...) returns a vector, NEXTCROSS, whose elements correspond to the mid-crossings (mid-reference level instants) of the next detected transition for each pulse.

[D,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = dutycycle(X,...) returns the mid-reference level, MIDLEV. Because in a bilevel pulse waveform the state levels are constant, MIDLEV is a scalar.

[D,INITCROSS,FINALCROSS,NEXTCROSS] = dutycycle(X,...,Name,Value) returns the ratio of pulse width to pulse period with additional options specified by one or more Name,Value pair arguments.

dutycycle(X,...) plots the waveform, X, and marks the location of the mid-reference level instants and the associated reference levels. The state levels and associated lower and upper state boundaries are also plotted.

Input Arguments

X

Bilevel waveform. X is a real-valued row or column vector.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

TAU

Pulse width in seconds. The product of TAU and PRF must be less than or equal to 1.

PRF

Pulse repetition frequency in pulses/second. The product of TAU and PRF must be less than or equal to 1.

Name-Value Pair Arguments

MidPct

Mid-reference level as a percentage of the waveform amplitude.

> **Default:** 50

Polarity

Pulse polarity. Specify the polarity as `'positive'` or `'negative'`. If you specify `'positive'`, dutycycle looks for pulses with positive-going (positive polarity) initial transitions. If you specify `'negative'`, dutycycle looks for pulses with negative-going (negative polarity) initial transitions. See "Pulse Polarity" on page 3-253 for examples of positive and negative-polarity pulses.

> **Default:** `'positive'`

StateLevels

Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, dutycycle estimates the state levels from the input waveform using the histogram method.

Tolerance

dutycycle

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-255.

Default: 2

Output Arguments

D

Duty cycle. Duty cycle is the ratio of the pulse width to the pulse period. Because the pulse width cannot exceed the pulse period, $0 \le D \le 1$.

INITCROSS

Mid-reference level instant of initial transition. Because the duty cycle is defined as the ratio of pulse width to pulse period, initial transitions are only reported when dutycycle finds a corresponding NEXTCROSS.

FINALCROSS

Mid-reference level instant of final transition. The duty cycle is defined as the ratio of pulse width to pulse period. Thus, final transitions are only reported when dutycycle finds a corresponding NEXTCROSS.

NEXTCROSS

Mid-reference level instant of the first initial transition after the final transition of the preceding pulse.

MIDLEV

Mid-reference level. The waveform value that corresponds to the mid-reference level.

Definitions

Duty Cycle

The energy in a bilevel, or rectangular, pulse is equal to the product of the peak power, *Pt*, and the pulse width, τ. Devices to measure energy in a waveform operate on time scales longer than the duration of a single pulse. Therefore, it is common to measure the average power

$$P_{\mathrm{av}} = \frac{P_t \tau}{T},$$

where T is the pulse period.

The ratio of average power to peak power is the duty cycle:

$$D = \frac{P_t \tau / T}{P_t}$$

Pulse Polarity

If the pulse has a positive-going initial transition, the pulse has positive polarity. The following figure shows a positive polarity pulse.

Positive Polarity Pulse

Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has a negative-going initial transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.

Negative Polarity Pulse

Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

State-Level Tolerances

Each state level can have an associated lower- and upper-state boundary. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.

State Levels ± 2% Tolerance

Examples

Duty Cycle of Bilevel Waveform

Determine the duty cycle of a bilevel waveform. Use the vector indices as the sample instants.

```
load('pulseex.mat', 'x');
d = dutycycle(x);
```

Duty Cycle of Bilevel Waveform with Sampling Frequency

Determine the duty cycle of a bilevel waveform. The sampling frequency is 4 MHz.

```
load('pulseex.mat', 'x','t');
fs = 1/(t(2)-t(1));
d = dutycycle(x,fs);
```

Duty Cycle of Bilevel Waveform with Three Pulses

Create a pulse waveform with three pulses. The sampling frequency is 4 MHz. Determine the initial and final mid-reference level instants. Plot the result.

Even though there are three pulses, only two pulses have corresponding subsequent transitions.

```
load('pulseex.mat','x');
dt = 1/4e6;
ts = reshape(repmat(x(1:30),1,3),90,1);
t = 0:dt:(length(ts)*dt)-dt;
[d,initcross,finalcross,~,midlev] = dutycycle(ts,t);
plot(t,ts,'k'); hold on; grid on;
h0 = plot(initcross, midlev*ones(length(initcross)),'rx');
set(h0,'markersize',10,'linewidth',2.5);
h1 = plot(finalcross,midlev*ones(length(finalcross)),'bx');
set(h1,'markersize',10,'linewidth',2.5);
```

References   [1] Skolnik, M.I. *Introduction to Radar Systems*. New York, NY: McGraw-Hill, 1980.

[2] *IEEE Standard on Transitions, Pulses, and Related Waveforms*. IEEE Standard 181, 2003.

See Also   midcross | pulseperiod | pulsesep | pulsewidth

ellip

Purpose   Elliptic filter design

Syntax
```
[z,p,k] = ellip(n,Rp,Rs,Wp)
[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype')
[b,a] = ellip(n,Rp,Rs,Wp)
[b,a] = ellip(n,Rp,Rs,Wp,'ftype')
[A,B,C,D] = ellip(n,Rp,Rs,Wp)
[A,B,C,D] = ellip(n,Rp,Rs,Wp,'ftype')
[z,p,k] = ellip(n,Rp,Rs,Wp,'s')
[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype','s')
[b,a] = ellip(n,Rp,Rs,Wp,'s')
[b,a] = ellip(n,Rp,Rs,Wp,'ftype','s')
[A,B,C,D] = ellip(n,Rp,Rs,Wp,'s')
[A,B,C,D] = ellip(n,Rp,Rs,Wp,'ftype','s')
```

Description   ellip designs lowpass, bandpass, highpass, and bandstop digital and
analog elliptic filters. Elliptic filters offer steeper rolloff characteristics
than Butterworth or Chebyshev filters, but are equiripple in both the
pass- and stopbands. In general, elliptic filters meet given performance
specifications with the lowest order of any filter type.

Digital Domain

[z,p,k] = ellip(n,Rp,Rs,Wp) designs an order n lowpass digital
elliptic filter with normalized passband edge frequency Wp, Rp dB of
ripple in the passband, and a stopband Rs dB down from the peak value
in the passband. It returns the zeros and poles in length n column
vectors z and p and the gain in the scalar k.

The *normalized passband edge frequency* is the edge of the passband,
at which the magnitude response of the filter is -Rp dB. For ellip, the
normalized cutoff frequency Wp is a number between 0 and 1, where
1 corresponds to half the sampling frequency (Nyquist frequency).
Smaller values of passband ripple Rp and larger values of stopband
attenuation Rs both lead to wider transition widths (shallower rolloff
characteristics).

If Wp is a two-element vector, Wp = [w1 w2], ellip returns an order 2*n
bandpass filter with passband w1 < ω < w2.

[z,p,k] = ellip(n,Rp,Rs,Wp,'*ftype*') designs a highpass, lowpass, or bandstop filter, where the string '*ftype*' is one of the following:

- 'high' for a highpass digital filter with normalized passband edge frequency Wp

- 'low' for a lowpass digital filter with normalized passband edge frequency Wp

- 'stop' for an order 2*n bandstop digital filter if Wp is a two-element vector, Wp = [w1 w2]. The stopband is w1 < ω < w2.

With different numbers of output arguments, ellip directly obtains other realizations of the filter. To obtain the transfer function form, use two output arguments as shown below.

Note See "Limitations" on page 3-264 for information about numerical issues that affect forming the transfer function.

[b,a] = ellip(n,Rp,Rs,Wp) designs an order n lowpass digital elliptic filter with normalized passband edge frequency Wp, Rp dB of ripple in the passband, and a stopband Rs dB down from the peak value in the passband. It returns the filter coefficients in the length n+1 row vectors b and a, with coefficients in descending powers of *z*.

$$H(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \ldots + a(n+1)z^{-n}}$$

[b,a] = ellip(n,Rp,Rs,Wp,'*ftype*') designs a highpass, lowpass, or bandstop filter, where the string '*ftype*' is 'high', 'low', or 'stop', as described above.

To obtain state-space form, use four output arguments as shown below:

[A,B,C,D] = ellip(n,Rp,Rs,Wp) or

[A,B,C,D] = ellip(n,Rp,Rs,Wp,'*ftype*') where A, B, C, and D are

ellip

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

and u is the input, x is the state vector, and y is the output.

Analog Domain

`[z,p,k] = ellip(n,Rp,Rs,Wp,'s')` designs an order n lowpass analog elliptic filter with angular passband edge frequency Wp rad/s and returns the zeros and poles in length n or `2*n` column vectors z and p and the gain in the scalar k.

The *angular passband edge frequency* is the edge of the passband, at which the magnitude response of the filter is -Rp dB. For `ellip`, the angular passband edge frequency Wp must be greater than 0 rad/s.

If Wp is a two-element vector with w1 < w2, then ellip(n,Rp,Rs,Wp,'s') returns an order `2*n` bandpass analog filter with passband w1 < ω< w2.

`[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is `'high'`, `'low'`, or `'stop'`, as described above.

With different numbers of output arguments, ellip directly obtains other realizations of the analog filter. To obtain the transfer function form, use two output arguments as shown below:

`[b,a] = ellip(n,Rp,Rs,Wp,'s')` designs an order n lowpass analog elliptic filter with angular passband edge frequency Wp rad/s and returns the filter coefficients in the length n+1 row vectors b and a, in descending powers of *s*, derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \ldots + b(n+1)}{s^n + a(2)s^{n-1} + \ldots + a(n+1)}$$

`[b,a] = ellip(n,Rp,Rs,Wp,'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

[A,B,C,D] = ellip(n,Rp,Rs,Wp,'s') or

[A,B,C,D] = ellip(n,Rp,Rs,Wp,'*ftype*','s') where A, B, C, and D are

$$x = Ax + Bu$$
$$y = Cx + Du$$

and *u* is the input, *x* is the state vector, and *y* is the output.

Examples   **Lowpass Filter**

For data sampled at 1000 Hz, design a sixth-order lowpass elliptic filter with a passband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6, 3 dB of ripple in the passband, and 50 dB of attenuation in the stopband:

```
[z,p,k] = ellip(6,3,50,300/500);
[sos,g] = zp2sos(z,p,k);      % Convert to SOS form
Hd = dfilt.df2tsos(sos,g);    % Create a dfilt object
h = fvtool(Hd)                % Plot magnitude response
set(h,'Analysis','freq')      % Display frequency response
```

Magnitude (dB) and Phase Responses

Limitations   In general, you should use the [z,p,k] syntax to design IIR filters.
To analyze or implement your filter, you can then use the [z,p,k]
output with zp2sos and an sos dfilt structure. For higher order filters
(possibly starting as low as order 8), numerical problems due to roundoff
errors may occur when forming the transfer function using the [b,a]
syntax. The following example illustrates this limitation:

```
n = 6;
Rp = .1; Rs = 80;
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer Function design
[b,a] = ellip(n,Rp,Rs,Wn,ftype);
h1=dfilt.df2(b,a);     % This is an unstable filter.

% Zero-Pole-Gain design
```

```
[z, p, k] = ellip(n,Rp,Rs,Wn,ftype);
[sos,g]=zp2sos(z,p,k);
h2=dfilt.df2sos(sos,g);
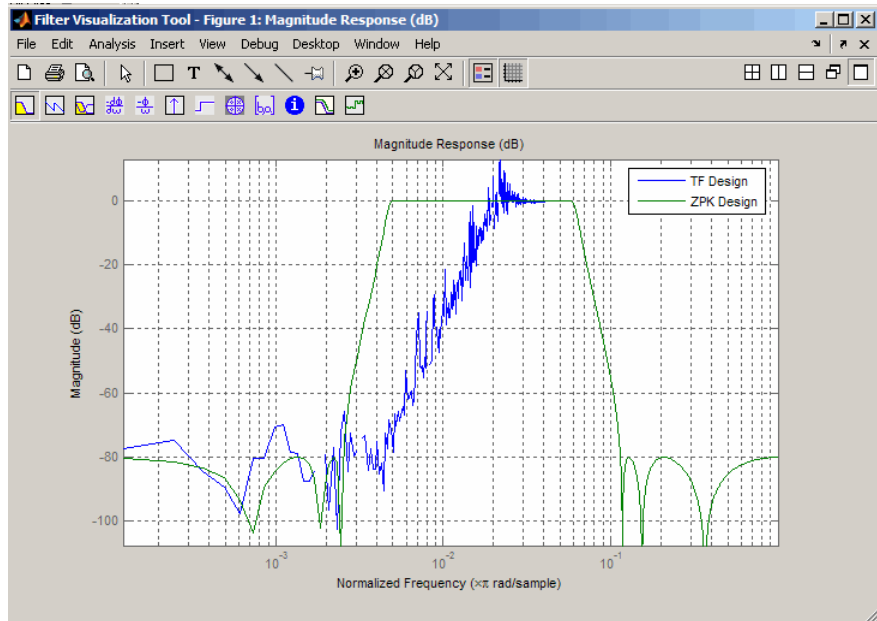
% Plot and compare the results
hfvt=fvtool(h1,h2,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```


Algorithms   The design of elliptic filters is the most difficult and computationally intensive of the Butterworth, Chebyshev Type I and II, and elliptic designs. ellip uses a five-step algorithm:

1 It finds the lowpass analog prototype poles, zeros, and gain using the ellipap function.

2 It converts the poles, zeros, and gain into state-space form.

3 It transforms the lowpass filter to a bandpass, highpass, or bandstop filter with the desired cutoff frequencies using a state-space transformation.

4 For digital filter design, `ellip` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wp` or `w1` and `w2`.

5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also   `besself | butter | cheby1 | cheby2 | ellipap | ellipord`

Purpose   Elliptic analog lowpass filter prototype

Syntax   `[z,p,k] = ellipap(n,Rp,Rs)`

Description   `[z,p,k] = ellipap(n,Rp,Rs)` returns the zeros, poles, and gain of an
order n elliptic analog lowpass filter prototype, with Rp dB of ripple in
the passband, and a stopband Rs dB down from the peak value in the
passband. The zeros and poles are returned in length n column vectors
z and p and the gain in scalar k. If n is odd, z is length n - 1. The
transfer function in factored zero-pole form is

$$H(s) = \frac{z(s)}{p(s)} = k\frac{(s-z_1)(s-z_2)\ldots(s-z_N)}{(s-p_1)(s-p_2)\ldots(s-p_M)}$$

Elliptic filters offer steeper rolloff characteristics than Butterworth and
Chebyshev filters, but they are equiripple in both the passband and the
stopband. Of the four classical filter types, elliptic filters usually meet a
given set of filter performance specifications with the lowest filter order.

`ellipap` sets the passband edge angular frequency ω_0 of the elliptic
filter to 1 for a normalized result. The *passband edge angular frequency*
is the frequency at which the passband ends and the filter has a
magnitude response of $10^{-Rp/20}$.

Algorithms   `ellipap` uses the algorithm outlined in [1]. It employs `ellipk` to
calculate the complete elliptic integral of the first kind and `ellipj` to
calculate Jacobi elliptic functions.

References   [1] Parks, T.W., and C.S. Burrus. *Digital Filter Design,* New York:
John Wiley & Sons, 1987. Chapter 7.

See Also   besselap | buttap | cheb1ap | cheb2ap | ellip

ellipord

Purpose   Minimum order for elliptic filters

Syntax   [n,Wp] = ellipord(Wp,Ws,Rp,Rs)
                  [n,Wp] = ellipord(Wp,Ws,Rp,Rs,'s')

Description   ellipord calculates the minimum order of a digital or analog elliptic filter required to meet a set of filter design specifications.

Digital Domain

[n,Wp] = ellipord(Wp,Ws,Rp,Rs) returns the lowest order n of the elliptic filter that loses no more than Rp dB in the passband and has at least Rs dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies Wp, is also returned. Use the output arguments n and Wp in ellip.

Choose the input arguments to specify the stopband and passband according to the following table.

Description of Stopband and Passband Filter Parameters

Parameter	Description
Wp	Passband corner frequency Wp, the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π radians per sample.
Ws	Stopband corner frequency Ws, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
Rp	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
Rs	Stopband attenuation, in decibels. This value is the number of decibels the stopband is attenuated with respect to the passband response.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	Wp < Ws, both scalars	(Ws,1)	(0,Wp)
Highpass	Wp > Ws, both scalars	(0,Ws)	(Wp,1)
Bandpass	The interval specified by Ws contains the one specified by Wp (Ws(1) < Wp(1) < Wp(2) < Ws(2)).	(0,Ws(1)) and (Ws(2),1)	(Wp(1),Wp(2))
Bandstop	The interval specified by Wp contains the one specified by Ws (Wp(1) < Ws(1) < Ws(2) < Wp(2)).	(0,Wp(1)) and (Wp(2),1)	(Ws(1),Ws(2))

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

Analog Domain

[n,Wp] = ellipord(Wp,Ws,Rp,Rs,'s') finds the minimum order n and cutoff frequencies Wp for an analog filter. You specify the frequencies Wp and Ws similar to those described in the Description of Stopband and Passband Filter Parameters on page 3-268 table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use ellipord for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications on page 3-269 table above.

ellipord

Examples   **Example 1**

For 1000 Hz data, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz and at least 60 dB of ripple in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

```
Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
% Returns n =4 Wp =0.0800
[b,a] = ellip(n,Rp,Rs,Wp);
freqz(b,a,512,1000);
title('n=4 Elliptic Lowpass Filter')
```


Example 2

Now design a bandpass filter with a passband from 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
```

```
Rp = 3; Rs = 40;
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
% Returns n =5 Wp =[0.1200    0.4000]
[b,a] = ellip(n,Rp,Rs,Wp);
freqz(b,a,512,1000);
title('n=5 Elliptic Bandpass Filter')
```


Algorithms   ellipord uses the elliptic lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both the analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before estimating the order and natural frequencies, and then converts them back to the *z*-domain.

ellipord initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for low- and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

ellipord

References   [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

See Also   buttord | cheb1ord | cheb2ord | ellip

Purpose	Equiripple single-rate FIR filter from specification object

Syntax

```
hd = design(d,'equiripple')
hd = design(d,'equiripple',designoption,value,designoption,
...value,...)
```

Description   `hd = design(d,'equiripple')` designs an equiripple FIR digital filter using the specifications supplied in the object d. Equiripple filter designs minimize the maximum ripple in the passbands and stopbands. `hd` is a `dfilt` object

`hd = design(d,'equiripple',designoption,value,designoption, ...value,...)` returns an equiripple FIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `equiripple`, refer to the command line help system. For example, to get specific information about using `equiripple` with d, the specification object, enter the following at the MATLAB prompt.

```
help(d,'equiripple')
```

Examples   First create a lowpass equiripple filter. Assume the data is sampled at 10,000 Hertz. The passband frequency is 500 Hertz with a stopband frequency of 700 Hz. The desired passband ripple is 1 dB with 60 dB of stopband attenuation.

```
Fs=10000;
Hd=fdesign.lowpass('Fp,Fst,Ap,Ast',500,700,1,60,10000);
d=design(Hd,'equiripple');
fvtool(d);
```

Displaying the filter in FVTool shows the equiripple nature of the filter.

The next example designs a lowpass equiripple filter with a direct-form transposed structure and density factor of 20 by specifying the FilterStructure and DensityFactor properties.

To set the design options for the filter, use the designopts method to obtain a structure array containing the current design options.

Change the fields of the structure array to specify your design options and invoke the design method with the structure array as an input argument.

```
% Use the same filter design as the previous example
Fs = 10000;
```

```
Hd = fdesign.lowpass('Fp,Fst,Ap,Ast',500,700,1,60,10000);
% Return the design options for the filter as a struct array
opts = designopts(Hd,'equiripple');
fieldnames(opts)
% Print out the filter structure- - direct-form FIR
opts.FilterStructure
% Change the filter structure to direct-form FIR transposed
opts.FilterStructure = 'dffirt';
% Change the filter density factor to 20
opts.DensityFactor = 20;
% Design the filter
d = design(Hd,'equiripple',opts)
```

An alternate way to design the preceding filter without using the structure array is:

```
Fs = 10000;
Hd = fdesign.lowpass('Fp,Fst,Ap,Ast',500,700,1,60,10000);
d = design(Hd,'equiripple','FilterStructure','dffirt','DensityFactor'
```

See Also   design | designmethods

eqtflength

Purpose   Equalize lengths of transfer function's numerator and denominator

Syntax
```
[b,a] = eqtflength(num,den)
[b,a,n,m] = eqtflength(num,den)
```

Description   `[b,a] = eqtflength(num,den)` modifies the vector `num` and/or the vector `den`, so that the resulting output vectors `b` and `a` have the same length. The input vectors `num` and `den` may have different lengths. The vector `num` represents the numerator polynomial of a given discrete-time transfer function, and the vector `den` represents its denominator. The resulting numerator `b` and denominator `a` represent the same discrete-time transfer function, but these vectors have the same length.

`[b,a,n,m] = eqtflength(num,den)` modifies the vectors as above and also returns the numerator order `n` and the denominator `m`, not including any trailing zeros.

Use `eqtflength` to obtain a numerator and denominator of equal length before applying transfer function conversion functions such as `tf2ss` and `tf2zp` to discrete-time models.

Examples
```
num = [1 0.5];
den = [1 0.75 0.6 0];
[b,a,n,m] = eqtflength(num,den);
```

Algorithms   `eqtflength(num,den)` appends zeros to either `num` or `den` as necessary. If both `num` and `den` have trailing zeros in common, these are removed.

See Also   `tf2ss` | `tf2zp`

Purpose   Fall time of negative-going bilevel waveform transitions

Syntax
```
F = falltime(X)
F = falltime(X,FS)
F = falltime(X,T)
[F,LT,UT] = falltime(...)
[F,LT,UT,LL,UL] = falltime(...)
[...] = falltime(...,Name,Value)
falltime(...)
```

Description   `F = falltime(X)` returns a vector, `F`, containing the time each transition of the bilevel waveform, `X`, takes to cross from the 90% to 10% reference levels. See "Percent Reference Levels" on page 3-281. To determine the transitions, `falltime` estimates the state levels of the input waveform by a histogram method. `falltime` identifies all regions, which cross the lower-state boundary of the high state and the upper-state boundary of the low state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See "State-Level Tolerances" on page 3-281. Because `falltime` uses interpolation, `F` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`F = falltime(X,FS)` specifies the sampling frequency in hertz. The sampling frequency determines the sample instants corresponding to the elements in `X`. The first sample instant in `X` corresponds to t=0. Because `falltime` uses interpolation, `F` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`F = falltime(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

`[F,LT,UT] = falltime(...)` returns vectors, `LT` and `UT`, whose elements correspond to the time instants where `X` crosses the lower and upper percent reference levels.

`[F,LT,UT,LL,UL] = falltime(...)` returns the levels, `LL` and `UL`, corresponding to the lower- and upper-percent reference levels.

falltime

[...] = falltime(...,Name,Value) returns the fall times with additional options specified by one or more Name,Value pair arguments.

falltime(...) plots the signal and darkens the regions of each transition where fall time is computed. The plot marks the lower and upper crossings and the associated reference levels. The state levels and the associated lower- and upper-state boundaries are also displayed.

Input Arguments

X

Bilevel waveform. X is a real-valued row or column vector.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

PctRefLevels

Reference levels as a percentage of the waveform amplitude. The low-state level is defined to be 0 percent. The high-state level is defined to be 100 percent. See "Percent Reference Levels" on page 3-281. 'PCTREFLEVELS' is a 2-element real row vector whose elements correspond to the lower- and upper-percent reference levels.

Default: [10 90]

StateLevels

Low and high-state levels. Specifies the levels to use for the low- and high-state levels as a 2-element real-valued row vector whose first and second elements correspond to the low- and high-state levels.

Tolerance

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-281.

Default: 2

Output Arguments

F

Fall times. `F` is a vector containing the duration of each negative-going transition. If you specify the sampling rate, `FS`, or the sampling instants, `T`, fall times are in seconds. If you do not specify a sampling rate, or sampling instants, fall times are in samples.

LT

Instants when negative-going transition crosses the lower-reference level. By default, the lower-reference level is the 10% reference level. You can change the default reference levels by specifying the `'PctRefLevels'` name-value pair.

UT

Instants when negative-going transition crosses the upper-reference level. By default, the upper reference level is the 90% reference level. You can change the default reference levels by specifying the `'PctRefLevels'` name-value pair.

LL

Lower-reference level in waveform amplitude units. `LL` is a vector containing the waveform values corresponding to the lower-reference level in each negative-going transition. By default, the lower-reference level is the 10% reference level. You can change the default reference levels by specifying the `'PctRefLevels'` name-value pair.

UL

Upper-reference level in waveform amplitude units. `LL` is a vector containing the waveform values corresponding to the upper-reference

level in each negative-going transition. By default, the upper-reference level is the 90% reference level. You can change the default reference levels by specifying the `'PctRefLevels'` name-value pair.

Definitions   **Negative-Going Transition**

A negative-going transition in a bilevel waveform is a transition from the high–state level to the low-state level. If the waveform is differentiable in the neighborhood of the transition, an equivalent definition is a transition with a negative first derivative. The following figure shows a negative-going transition.

X(t)

$$\frac{dX}{dt} < 0$$

In the preceding figure, the amplitude values of the waveform are not displayed because a negative-going transition does not depend on the actual waveform values. A negative-going transition is defined by the direction of the transition.

Percent Reference Levels

If S_1 is the low state, S_2 is the high state, and U is the *upper*-percent reference level. The waveform value corresponding to the upper percent reference level is

$$S_1 + \frac{U}{100}(S_2 - S_1)$$

If L is the *lower* percent reference level, the waveform value corresponding to the lower percent reference level is

$$S_1 + \frac{L}{100}(S_2 - S_1)$$

State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \tfrac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity (positive-going) bilevel waveform. The estimated state levels are indicated by a dashed red line.

State Levels ± 2% Tolerance

Examples   **Falltime in a Bilevel Waveform**

Determine the fall time in samples for a 2.3 V clock waveform.

Load the 2.3 V clock data. Determine the fall time in samples. Use the default [10 90] percent reference levels.

```
load('negtransitionex.mat', 'x');
F = falltime(x);
```

The fall time is less than 1, indicating that the transition occurred in a fraction of a sample.

Falltime with 20% and 80% Reference Levels

Determine the fall time in a 2.3 V clock waveform sampled at 4 MHz. Compute the fall time using the 20% and 80% reference levels.

Load the 2.3 V clock data with sampling instants. Plot the waveform.

```
load('negtransitionex.mat','x','t');
plot(t,x);
```

Determine the fall time using the 20% and 80% reference levels..

```
F = falltime(x,'PctRefLevels',[2O 80]);
```

Falltime, Reference-Level Instants, and Reference Levels

Determine the fall time, reference-level instants, and reference levels in a 2.3 V clock waveform sampled at 4 MHz.

Load the 2.3 V clock waveform along with the sampling instants.

```
load('negtransitionex.mat','x','t');
```

Determine the falltime, reference-level instants, and reference levels.

```
[F,LT,UT,LL,UL] = falltime(x,t);
```

Plot the waveform in microseconds with the upper and lower reference levels and reference level instants. Show that the fall time is the difference between the lower- and upper-reference level instants.

```
plot(t.*1e6,x);
```

falltime

```
xlabel('microseconds'); ylabel('Volts');
hold on; grid on;
plot(LT.*1e6,LL,'ro','markerfacecolor',[1 O O]);
plot(UT.*1e6,UL,'ro','markerfacecolor',[1 O O]);
fprintf('Rise time is %1.4f microseconds.\n',(LT-UT)*1e6)
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

See Also   risetime | slewrate | statelevels

Purpose   Open Filter Design and Analysis Tool

Syntax   fdatool

Description   fdatool opens the Filter Design and Analysis Tool (FDATool). Use
                   this tool to

- Design filters

- Analyze filters

- Modify existing filter designs

See "FDATool" and "Using FDATool" for detailed information about the
Filter Design and Analysis Tool.

Tips   The Filter Design and Analysis Tool provides more design methods than the SPTool Filter Designer, which will be removed in a future release. The Filter Design and Analysis Tool also integrates advanced filter design methods from the DSP System Toolbox software.

Note The Filter Design and Analysis Tool requires a screen resolution greater than 640 x 480.

See Also   fvtool | sptool | wvtool

Purpose   Filter specification object

Syntax   d = fdesign.*response*
                d = fdesign.*response*(spec)
                d = fdesign.*response*(...,Fs)
                d = fdesign.*response*(...,magunits)

Description **Filter Specification Objects**

d = fdesign.*response* returns a filter specification object d, of filter response *response*. To create filters from d, use one of the design methods listed in "Using Filter Design Methods with Specification Objects" on page 3-294

Note Several of the filter response types described below are only available if your installation includes the DSP System Toolbox. The DSP System Toolbox significantly expands the functionality available for the specification, design, and analysis of filters.

Here is how you design filters using fdesign.

1 Use fdesign.*response* to construct a filter specification object.

2 Use designmethods to determine which filter design methods work for your new filter specification object.

3 Use design to apply your filter design method from step 2 to your filter specification object to construct a filter object.

4 Use FVTool to inspect and analyze your filter object.

fdesign

> **Note** fdesign does not create filters. fdesign returns a filter
> specification object that contains the specifications for a filter, such as
> the passband cutoff or attenuation in the stopband. To design a filter
> hd from a filter specification object d, use d with a filter design method
> such as butter —hd = design(d,'butter').

For more guidance about using fdesign, refer to the examples in
"Getting Started". Alternatively, type the following at the MATLAB
prompt for more information:

```
help fdesign
```

response can be one of the entries in the following table that specify
the filter response desired, such as a bandstop filter or an interpolator.

fdesign Response String	Description
arbgrpdelay	fdesign.arbgrpdelay creates an object to specify allpass arbitrary group delay filters. Requires the DSP System Toolbox
arbmag	fdesign.arbmag creates an object to specify IIR filters that have arbitrary magnitude responses defined by the input arguments.
arbmagnphase	fdesign.arbmagnphase creates an object to specify IIR filters that have arbitrary magnitude and phase responses defined by the input arguments. Requires the DSP System Toolbox.
audioweighting	fdesign.audioweighting creates a filter specification object for audio weighting filters. The supported audio weighting types are: A, C, C-message, ITU-T 0.41, and ITU-R 468-4 weighting. Requires the DSP System Toolbox

fdesign Response String	Description
bandpass	`fdesign.bandpass` creates an object to specify bandpass filters.
bandstop	`fdesign.bandstop` creates an object to specify bandstop filters.
ciccomp	`fdesign.ciccomp` creates an object to specify filters that compensate for the CIC decimator or interpolator response curves. Requires the DSP System Toolbox.
comb	`fdesign.comb` creates an object to specify a notching or peaking comb filter. Requires the DSP System Toolbox.
decimator	`fdesign.decimator` creates an object to specify decimators. Requires the DSP System Toolbox
differentiator	`fdesign.differentiator` creates an object to specify an FIR differentiator filter.
fracdelay	`fdesign.fracdelay` creates an object to specify fractional delay filters. Requires the DSP System Toolbox.
halfband	`fdesign.halfband` creates an object to specify halfband filters. Requires the DSP System Toolbox.
highpass	`fdesign.highpass` creates an object to specify highpass filters.
hilbert	`fdesign.hilbert` creates an object to specify an FIR Hilbert transformer.
interpolator	`fdesign.interpolator` creates an object to specify interpolators. Requires the DSP System Toolbox.

fdesign Response String	Description
isinchp	`fdesign.isinchp` creates an object to specify an inverse sinc highpass filter. Requires the DSP System Toolbox.
isinclp	`fdesign.isinclp` creates an object to specify an inverse sinc lowpass filters. Requires the DSP System Toolbox.
lowpass	`fdesign.lowpass` creates an object to specify lowpass filters.
notch	`fdesign.notch` creates an object to specify notch filters. Requires the DSP System Toolbox.
nyquist	`fdesign.nyquist` creates an object to specify nyquist filters. Requires the DSP System Toolbox.
octave	`fdesign.octave` creates an object to specify octave and fractional octave filters. Requires the DSP System Toolbox.
parameq	`fdesign.parameq` creates an object to specify parametric equalizer filters. Requires the DSP System Toolbox.
peak	`fdesign.peak` creates an object to specify peak filters. Requires the DSP System Toolbox.
polysrc	`fdesign.polysrc` creates an object to specify polynomial sample-rate converter filters. Requires the DSP System Toolbox.
pulseshaping	`fdesign.pulseshaping` creates an object to specify pulse-shaping filters.
rsrc	`fdesign.rsrc` creates an object to specify rational-factor sample-rate convertors. Requires the DSP System Toolbox.

Use the `doc fdesign.`*response* syntax at the MATLAB prompt to get help on a specific structure. Using `doc` in a syntax like

```
doc fdesign.lowpass
doc fdesign.bandstop
```

gets more information about the lowpass or bandstop structure objects.

Each `response` has a property `Specification` that defines the specifications to use to design your filter. You can use defaults or specify the `Specification` property when you construct the specifications object.

With the strings for the `Specification` property, you provide filter constraints such as the filter order or the passband attenuation to use when you construct your filter from the specification object.

Properties   `fdesign` returns a filter specification object. Every filter specification object has the following properties.

Property Name	Default Value	Description
Response	Depends on the chosen type	Defines the type of filter to design, such as an interpolator or bandpass filter. This is a read-only value.
Specification	Depends on the chosen type	Defines the filter characteristics used to define the desired filter performance, such as the cutoff frequency Fc or the filter order N.

Property Name	Default Value	Description
Description	Depends on the filter type you choose	Contains descriptions of the filter specifications used to define the object, and the filter specifications you use when you create a filter from the object. This is a read-only value.
NormalizedFrequency	Logical `true`	Determines whether the filter calculation uses normalized frequency from 0 to 1, or the frequency band from 0 to Fs/2, the sampling frequency. Accepts either `true` or `false` without single quotation marks. Audio weighting filters do not support normalized frequency.

In addition to these properties, filter specification objects may have other properties as well, depending on whether they design `dfilt` objects or `mfilt` objects.

Added Properties for mfilt Objects	Description
DecimationFactor	Specifies the amount to decrease the sampling rate. Always a positive integer.
InterpolationFactor	Specifies the amount to increase the sampling rate. Always a positive integer.
PolyphaseLength	Polyphase length is the length of each polyphase subfilter that composes the decimator or interpolator or rate-change

Added Properties for mfilt Objects	Description
	factor filters. Total filter length is the product of `pl` and the rate change factors. `pl` must be an even integer.

`d = fdesign.`*`response`*`(spec)`. In `spec`, you specify the variables to use that define your filter design, such as the passband frequency or the stopband attenuation. The specifications are applied to the filter design method you choose to design your filter.

For example, when you create a default lowpass filter specification object, `fdesign.lowpass` sets the passband frequency `Fp`, the stopband frequency `Fst`, the stopband attenuation `Ast`, and the passband ripple `Ap` :

```
H = fdesign.lowpass
% Without a terminating semicolon
% the filter specifications are displayed
```

The default specification `'Fp,Fst,Ap,Ast'` is only one of the possible specifications for `fdesign.lowpass`. To see all available specifications:

```
H = fdesign.lowpass;
set(H,'specification')
```

The DSP System Toolbox software supports all available specification strings. The Signal Processing Toolbox supports a subset of the specification strings. See the reference pages for the filter specification object to determine which specification strings your installation supports.

One important note is that the specification string you choose determines which design methods apply to the filter specifications object.

Specifications that do not contain the filter order result in minimum order designs when you invoke the `design` method:

```
d = fdesign.lowpass;
% Specification is Fp,Fst,Ap,Ast
Hd = design(d,'equiripple');
length(Hd.Numerator) % returns 43
% Filter order is 42
fvtool(Hd) %view magnitude
```

d = fdesign.*response*(...,Fs) specifies the sampling frequency in Hz to use in the filter specifications. The sampling frequency is a scalar trailing all other input arguments. If you specify a sampling frequency, all frequency specifications are in Hz.

d = fdesign.*response*(...,magunits) specifies the units for any magnitude specification you provide in the input arguments. magunits can be one of the following strings:

- 'linear' — specify the magnitude in linear units

- 'dB' — specify the magnitude in decibels

- 'squared' — specify the magnitude in power units

When you omit the magunits argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Using Filter Design Methods with Specification Objects

After you create a filter specification object, you use a filter design method to implement your filter with a selected algorithm. Use designmethods to determine valid design methods for your filter specification object.

```
d = fdesign.lowpass('N,Fc,Ap,Ast',10,0.2,0.5,40);
designmethods(d)
% Design FIR equiripple filter
hd = design(d,'equiripple');
```

When you use any of the design methods without providing an output argument, the resulting filter design appears in FVTool by default.

Along with filter design methods, `fdesign` works with supporting methods that help you create filter specification objects or determine which design methods work for a given specifications object.

Supporting Function	Description
setspecs	Set all of the specifications simultaneously.
designmethods	Return the design methods.
designopts	Return the input arguments and default values that apply to a specifications object and method

You can set filter specification values by passing them after the `Specification` argument, or by passing the values without the `Specification` string.

Filter object constructors take the input arguments in the same order as `setspecs` and the order in the strings for `Specification`. Enter `doc setspecs` at the prompt for more information about using `setspecs`.

When the first input to `fdesign` is not a valid `Specification` string like 'n,fc', `fdesign` assumes that the input argument is a filter specification and applies it using the default `Specification` string —`fp,fst,ap,ast` for a lowpass object, for example.

Examples   The following examples require only the Signal Processing Toolbox.

Example 1–Bandstop Filter

A bandstop filter specification object for data sampled at 8 kHz. The stopband between 2 and 2.4 kHz is attenuated at least 80 dB:

```
H = fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',...
         1600,2000,2400,2800,1,80,1,8000);
```

Example 2–Lowpass Filter

A lowpass filter specification object for data sampled at 10 kHz. The passband frequency is 500 Hz and the stopband frequency is 750 Hz.

The passband ripple is set to 1 dB and the required attenuation in the stopband is 80 dB.

```
H = fdesign.lowpass('Fp,Fst,Ap,Ast',500,750,1,80,10000);
```

Example 3–Highpass Filter

A default highpass filter specification object.

```
H = fdesign.highpass % Creates specifications object.
H.Description
```

Notice the correspondence between the property values in `Specification` and `Description` — in `Description` you see in words the definitions of the variables shown in `Specification`.

Example 4–Filter Specification and Design

Lowpass Butterworth filter specification object

Use a filter specification object to construct a lowpass Butterworth filter with default `Specification 'Fp,Fst,Ap,Ast'`. Set the passband edge frequency to 0.4π radians/sample, a stopband frequency of 0.5π radians/sample, a passband ripple of 1 dB, and 80 dB of stopband attenuation.

```
d = fdesign.lowpass(0.4,0.5,1,80);
```

Determine which design methods apply to d.

```
designmethods(d)
```

You can use d and the `butter` design method to design a Butterworth filter.

```
hd = design(d,'butter','matchexactly','passband');
fvtool(hd);
```

The resulting filter magnitude response shown by FVTool appears in the following figure.

If you have the DSP System Toolbox software installed, the preceding figure appears with the filter specification mask.

See Also   designmethods | designopts | fdatool | filterbuilder | fvtool

fdesign.arbmag

Purpose	Arbitrary response magnitude filter specification object

Syntax

```
D= fdesign.arbmag
D= fdesign.arbmag(SPEC)
D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)
D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)
D = fdesign.arbmag(...,Fs)
```

Description

D= fdesign.arbmag constructs an arbitrary magnitude filter specification object D.

D= fdesign.arbmag(SPEC) initializes the Specification property to SPEC. The input argument SPEC must be one of the strings shown in the following table. Specification strings are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP System Toolbox software.

- 'N,F,A' — Single band design (default)
- 'F,A,R' — Single band minimum order design *
- 'N,B,F,A' — Multiband design
- 'N,B,F,A,C' — Constrained multiband design *
- 'B,F,A,R' — Multiband minimum order design *
- 'Nb,Na,F,A' — Single band design *
- 'Nb,Na,B,F,A' — Multiband design *

The string entries are defined as follows:

- A — Amplitude vector. Values in A define the filter amplitude at frequency points you specify in f, the frequency vector. If you use A, you must use F as well. Amplitude values must be real. For complex values designs, use fdesign.arbmagnphase.

- B — Number of bands in the multiband filter

- C — Constrained band flag. This enables you to constrain the passband ripple in your multiband design. You cannot constrain the passband ripple in all bands simultaneously.

- F — Frequency vector. Frequency values in specified in F indicate locations where you provide specific filter response amplitudes. When you provide F, you must also provide A.

- N — Filter order for FIR filters and the numerator and denominator orders for IIR filters.

- Nb — Numerator order for IIR filters

- Na — Denominator order for IIR filter designs

- R — Ripple

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

Specifying Frequency and Amplitude Vectors

F and A are the input arguments you use to define the filter response desired. Each frequency value you specify in F must have a corresponding response value in A. The following table shows how F and A are related.

Define the frequency vector F as [0 0.25 0.3 0.4 0.5 0.6 0.7 0.75 1.0]

Define the response vector A as [1 1 0 0 0 0 0 1 1]

These specifications connect F and A as shown here:

F (Normalized Frequency)	A (Response Desired at F)
0	1
0.25	1
0.3	0
0.4	0

F (Normalized Frequency)	A (Response Desired at F)
0.5	0
0.6	0
0.7	0
0.75	1
1.0	1

Different specifications can have different design methods available. Use `designmethods` to get a list of design methods available for a given specification string and filter specification object.

Use `designopts` to get a list of design options available for a filter specification object and a given design method. Enter `help(D,METHOD)` to get detailed help on the available design options for a given design method.

`D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)` initializes the specifications with `specvalue1`, `specvalue2`. Use `get(D,'Description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ... `specvalueN`.

`D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)` uses the default specification string `'N,F,A'`, setting the filter order, filter frequency vector, and the amplitude vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

`D = fdesign.arbmag(...,Fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `Fs`.

Examples   **Design of a multiband arbitrary-magnitude filter**

Use fdesign.arbmag to design a 3–band filter.

Use the given frequency and amplitude vectors in "Specifying Frequency and Amplitude Vectors" on page 3-299.

```
N = 150;
B = 3;
F = [0 .25 .3 .4 .5 .6 .7 .75 1];
A = [1 1 0 0 0 0 0 1 1];
A1 = A(1:2);
A2 = A(3:7);
A3 = A(8:end);
F1 = F(1:2);
F2 = F(3:7);
F3 = F(8:end);
d = fdesign.arbmag('N,B,F,A',N,B,F1,A1,F2,A2,F3,A3);
Hd = design(d);
fvtool(Hd)
```

A response with two passbands — one roughly between 0 and 0.25 and
the second between 0.75 and 1 — results from the mapping between
F and A.

Design of a single band arbitrary-magnitude filter

Use fdesign.arbmag to design a single band equiripple filter.

```
n = 120;
f = linspace(0,1,100);  % 100 frequency points.
as = ones(1,100)-f*0.2;
absorb = [ones(1,30),(1-0.6*bohmanwin(10))',...
ones(1,5), (1-0.5*bohmanwin(8))',ones(1,47)];
a = as.*absorb;
d = fdesign.arbmag('N,F,A',n,f,a);
hd1 = design(d,'equiripple');
```

If you have the DSP System Toolbox, you can design a minimum-phase
equiripple filter.

```
hd2 = design(d,'equiripple','MinPhase',true);
hfvt = fvtool([hd1 hd2],'analysis','polezero');
legend(hfvt,'Equiripple Filter','Minimum-phase Equiripple Filter');
```

Design of a multiband minimum order arbitrary-magnitude filter

Use fdesign.arbmag to design a multiband minimum order filter.

This example requires the DSP System Toolbox.

Place the notches at 0.25π and 0.55π radians/sample

```
d = fdesign.arbmag('B,F,A,R');
d.NBands = 5;
d.B1Frequencies = [0 0.2];
d.B1Amplitudes = [1 1];
d.B1Ripple = 0.25;
d.B2Frequencies = 0.25;
d.B2Amplitudes = 0;
d.B3Frequencies = [0.3 0.5];
d.B3Amplitudes = [1 1];
d.B3Ripple = 0.25;
d.B4Frequencies = 0.55;
d.B4Amplitudes = 0;
d.B5Frequencies = [0.6 1];
d.B5Amplitudes = [1 1];
d.B5Ripple = 0.25;
Hd = design(d,'equiripple');
fvtool(Hd)
```

Design of a multiband constrained arbitrary-magnitude filter

Use fdesign.arbmag to design a multiband constrained FIR filter.

This example requires the DSP System Toolbox.

Force the frequency response at 0.15π radians/sample to 0 dB.

```
d = fdesign.arbmag('N,B,F,A,C',82,2);
d.B1Frequencies = [0 0.06 .1];
d.B1Amplitudes = [0 0 0];
d.B2Frequencies = [.15 1];
d.B2Amplitudes = [1 1];
```

```
% Design a filter with no constraints
Hd1 = design(d,'equiripple','B2ForcedFrequencyPoints',0.15);
% Add a constraint to the first band to increase attenuation
d.B1Constrained = true;
d.B1Ripple = .001;
Hd2 =  design(d,'equiripple','B2ForcedFrequencyPoints',0.15);
hfvt = fvtool(Hd1,Hd2);
legend(hfvt,'Original Design','Design with Constrained Stopband Ripple
```


See Also   design | designmethods | fdesign

fdesign.bandpass

Purpose   Bandpass filter specification object

Syntax
```
D = fdesign.bandpass
D = fdesign.bandpass(SPEC)
D = fdesign.bandpass(spec,specvalue1,specvalue2,...)
D = fdesign.bandpass(specvalue1,specvalue2,specvalue3,
specvalue4,...specvalue4,specvalue5,specvalue6)
D = fdesign.bandpass(...,Fs)
D = fdesign.bandpass(...,MAGUNITS)
```

Description   `D = fdesign.bandpass` constructs a bandpass filter specification object
D, applying default values for the properties `Fstop1`, `Fpass1`, `Fpass2`,
`Fstop2`, `Astop1`, `Apass`, and `Astop2` — one possible set of values you
use to specify a bandpass filter.

`D = fdesign.bandpass(SPEC)` constructs object D and sets its
`Specification` property to SPEC. Entries in the SPEC string represent
various filter response features, such as the filter order, that govern the
filter design. Valid entries for SPEC are shown below and used to define
the bandpass filter. The strings are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP
System Toolbox software.

- `'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'` (default spec)

- `'N,F3dB1,F3dB2'`

- `"N,F3dB1,F3dB2,Ap'` *

- `'N,F3dB1,F3dB2,Ast'` *

- `'N,F3dB1,F3dB2,Ast1,Ap,Ast2'` *

- `'N,F3dB1,F3dB2,BWp` *

- `'N,F3dB1,F3dB2,BWst'` *

- `'N,Fc1,Fc2'`

- `'N,Fc1,Fc2,Ast1,Ap,Ast2'`

- `'N,Fp1,Fp2,Ap'`

- `'N,Fp1,Fp2,Ast1,Ap,Ast2'`

- `'N,Fst1,Fp1,Fp2,Fst2'`

- `'N,Fst1,Fp1,Fp2,Fst2,C'` *

- `'N,Fst1,Fp1,Fp2,Fst2,Ap'` *

- `'N,Fst1,Fst2,Ast'`

- `'Nb,Na,Fst1,Fp1,Fp2,Fst2'` *

The string entries are defined as follows:

- `Ap` — amount of ripple allowed in the pass band. Also called Apass.

- `Ast1` — attenuation in the first stop band in decibels (the default units). Also called Astop1.

- `Ast2` — attenuation in the second stop band in decibels (the default units). Also called Astop2.

- `BWp` — bandwidth of the filter passband. Specified in normalized frequency units.

- `BWst` — bandwidth of the filter stopband. Specified in normalized frequency units.

- `C` — Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

  In the specification string `'N,Fst1,Fp1,Fp2,Fst2,C'`, you cannot specify constraints in both stopbands and the passband simultaneously. You can specify constraints in any one or two bands.

- `F3dB1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (IIR filters)

- `F3dB2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (IIR filters)

- `Fc1` — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (FIR filters)

- `Fc2` — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (FIR filters)

- `Fp1` — frequency at the edge of the start of the pass band. Specified in normalized frequency units. Also called Fpass1.

- `Fp2` — frequency at the edge of the end of the pass band. Specified in normalized frequency units. Also called Fpass2.

- `Fst1` — frequency at the edge of the start of the first stop band. Specified in normalized frequency units. Also called Fstop1.

- `Fst2` — frequency at the edge of the start of the second stop band. Specified in normalized frequency units. Also called Fstop2.

- `N` — filter order for FIR filters. Or both the numerator and denominator orders for IIR filters when na and nb are not provided.

- `Na` — denominator order for IIR filters

- `Nb` — numerator order for IIR filters

Graphically, the filter specifications look similar to those shown in the following figure.

Regions between specification values like `Fst1` and `Fp1` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandpass filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design methods apply to an object and the `Specification` property value.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.bandpass(spec,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.bandpass(specvalue1,specvalue2,specvalue3, specvalue4,...specvalue4,specvalue5,specvalue6)` constructs `D` with the default `Specification` property string, using the values you provide as input arguments for `specvalue1,specvalue2,specvalue3,specvalue4,specvalue4,specvalue5, specvalue6` and `specvalue7`.

`D = fdesign.bandpass(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

fdesign.bandpass

D = fdesign.bandpass(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Filter a discrete-time signal with a bandpass filter. The signal is a sum of three discrete-time sinusoids, π/8, π/2, and 3π/4 radians/sample.

```
n = 0:159;
x = cos(pi/8*n)+cos(pi/2*n)+sin(3*pi/4*n);
```

Design an FIR equiripple bandpass filter to remove the lowest and highest discrete-time sinusoids.

```
d = fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2',1/4,3/8,5/8,6/8,60,
Hd = design(d,'equiripple');
```

Apply the filter to the discrete-time signal.

```
y = filter(Hd,x);
freq = 0:(2*pi)/length(x):pi;
xdft = fft(x);
ydft = fft(y);
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(x)/2+1)),'r','linewidth',2);
legend('Original Signal','Bandpass Signal');
```

Design an IIR Butterworth filter of order 10 with 3–dB frequencies of 1 and 1.2 kHz. The sampling frequency is 10 kHz

```
d = fdesign.bandpass('N,F3dB1,F3dB2',10,1e3,1.2e3,1e4);
Hd = design(d,'butter');
fvtool(Hd)
```

This example requires the DSP System Toolbox software.

Design a constrained-band FIR equiripple filter of order 100 with a passband of [1, 1.4] kHz. Both stopband attenuation values are constrained to 60 dB. The sampling frequency is 10 kHz.

```
d = fdesign.bandpass('N,Fst1,Fp1,Fp2,Fst2,C',100,800,1e3,1.4e3,1.6e3,
d.Stopband1Constrained = true; d.Astop1 = 60;
d.Stopband2Constrained = true; d.Astop2 = 60;
Hd = design(d,'equiripple');
fvtool(Hd);
measure(Hd)
```

The passband ripple is slightly over 2 dB. Because the design constrains both stopbands, you cannot constrain the passband ripple.

See Also   fdesign, fdesign.bandstop, fdesign.highpass, fdesign.lowpass

fdesign.bandstop

Purpose	Bandstop filter specification object

Syntax

```
D = fdesign.bandstop
D = fdesign.bandstop(SPEC)
D = fdesign.bandstop(SPEC,specvalue1,specvalue2,...)
D = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,...
specvalue5,specvalue6,specvalue7)
D = fdesign.bandstop(...,Fs)
D = fdesign.bandstop(...,MAGUNITS)
```

Description   D = fdesign.bandstop constructs a bandstop filter specification object
D, applying default values for the properties Fpass1, Fstop1, Fstop2,
Fpass2, Apass1, Astop1 and Apass2.

D = fdesign.bandstop(SPEC) constructs object D and sets the
Specification property to SPEC. Entries in the SPEC string represent
various filter response features, such as the filter order, that govern
the filter design. Valid entries for SPEC are shown below. The strings
are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP
System Toolbox software.

- 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' (default spec)
- 'N,F3dB1,F3dB2'
- 'N,F3dB1,F3dB2,Ap' *
- 'N,F3dB1,F3dB2,Ap,Ast' *
- 'N,F3dB1,F3dB2,Ast' *
- 'N,F3dB1,F3dB2,BWp' *
- 'N,F3dB1,F3dB2,BWst' *
- 'N,Fc1,Fc2'

- `'N,Fc1,Fc2,Ap1,Ast,Ap2'`

- `'N,Fp1,Fp2,Ap'`

- `'N,Fp1,Fp2,Ap,Ast'`

- `'N,Fp1,Fst1,Fst2,Fp2'`

- `'N,Fp1,Fst1,Fst2,Fp2,C'` *

- `'N,Fp1,Fst1,Fst2,Fp2,Ap'` *

- `'N,Fst1,Fst2,Ast'`

- `'Nb,Na,Fp1,Fst1,Fst2,Fp2'` *

The string entries are defined as follows:

- Ap — amount of ripple allowed in the passband in decibels (the default units). Also called Apass.

- Ap1 — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass1.

- Ap2 — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass2.

- Ast — attenuation in the first stopband in decibels (the default units). Also called Astop1.

- BWp — bandwidth of the filter passband. Specified in normalized frequency units.

- BWst — bandwidth of the filter stopband. Specified in normalized frequency units.

- C — Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

  In the specification string `'N,Fp1,Fst1,Fst2,Fp2,C'`, you cannot specify constraints simultaneously in both passbands and the stopband. You can specify constraints in any one or two bands.

- F3dB1 — cutoff frequency for the point 3 dB point below the passband value for the first cutoff.

- F3dB2 — cutoff frequency for the point 3 dB point below the passband value for the second cutoff.

- Fc1 — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. (FIR filters)

- Fc2 — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. (FIR filters)

- Fp1 — frequency at the start of the pass band. Also called Fpass1.

- Fp2 — frequency at the end of the pass band. Also called Fpass2.

- Fst1 — frequency at the end of the first stop band. Also called Fstop1.

- Fst2 — frequency at the start of the second stop band. Also called Fstop2.

- N — filter order.

- Na — denominator order for IIR filters.

- Nb — numerator order for IIR filters.

Graphically, the filter specifications look similar to those shown in the following figure.

Regions between specification values like `Fp1` and `Fst1` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandstop filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design methods apply to an object and the `Specification` property value.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.bandstop(SPEC,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,... specvalue5,specvalue6,specvalue7)` constructs an object `D` with the default `Specification` property string , using the values you provide in `specvalue1,specvalue2,specvalue3,specvalue4,specvalue5, specvalue6` and `specvalue7`.

`D = fdesign.bandstop(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency. If you specify the sampling frequency as a trailing scalar, all frequencies in the specifications are in Hz as well.

`D = fdesign.bandstop(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units

- `'dB'` — specify the magnitude in dB (decibels)

- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

fdesign.bandstop

Examples Construct a bandstop filter to reject the discrete frequency band between 3π/8 and 5π/8 radians/sample. Apply the filter to a discrete-time signal consisting of the superposition of three discrete-time sinusoids.

Design an FIR equiripple filter and view the magnitude response.

```
d = fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',2/8,3/8,5/8,6/8,1,60
Hd = design(d,'equiripple');
fvtool(Hd)
```

Construct the discrete-time signal to filter.

```
n = 0:99;
x = cos(pi/5*n)+sin(pi/2*n)+cos(4*pi/5*n);
y = filter(Hd,x);
xdft = fft(x);
ydft = fft(y);
freq = 0:(2*pi)/length(x):pi;
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(y)/2+1)),'r','linewidth',2);
xlabel('Radians/Sample'); ylabel('Magnitude');
legend('Original Signal','Bandstop Signal');
```

Create a Butterworth bandstop filter for data sampled at 10 kHz. The stopband is [1,1.5] kHz. The order of the filter is 20.

```
d = fdesign.bandstop('N,F3dB1,F3dB2',20,1e3,1.5e3,1e4);
Hd = design(d,'butter');
fvtool(Hd);
```

Zoom in on the magnitude response plot to verify that the 3-dB down points are located at 1 and 1.5 kHz.

The following example requires the DSP System Toolbox license.

Design a constrained-band FIR equiripple filter of order 100 for data sampled at 10 kHz. You can specify constraints on at most two of the three bands: two passbands and one stopband. In this example, you choose to constrain the passband ripple to be 0.5 dB in each passband. Design the filter, visualize the magnitude response and measure the filter's design.

```
d = fdesign.bandstop('N,Fp1,Fst1,Fst2,Fp2,C',100,800,1e3,1.5e3,1.7e3,
d.Passband1Constrained = true; d.Apass1 = 0.5;
d.Passband2Constrained = true; d.Apass2 = 0.5;
Hd = design(d,'equiripple');
fvtool(Hd);
measure(Hd)
```

With this order filter and passband ripple constraints, you achieve approximately 50 dB of stopband attentuation.

See Also   fdesign, fdesign.bandpass, fdesign.highpass, fdesign.lowpass

fdesign.differentiator

Purpose	Differentiator filter specification object

Syntax

```
D = fdesign.differentiator
D = fdesign.differentiator(SPEC)
D = fdesign.differentiator(SPEC,specvalue1,specvalue2, ...)
D = fdesign.differentiator(specvalue1)
D = fdesign.differentiator(...,Fs)
D = fdesign.differentiator(...,MAGUNITS)
```

Description D = fdesign.differentiator constructs a default differentiator filter designer D with the filter order set to 31.

D = fdesign.differentiator(SPEC) initializes the filter designer Specification property to SPEC. You provide one of the following strings as input to replace SPEC. The string you provide is not case sensitive.

> **Note** Specifications strings marked with an asterisk require the DSP System Toolbox software.

- 'N' — Full band differentiator (default)
- 'N,Fp,Fst' — Partial band differentiator
- 'N,Fp,Fst,Ap' — Partial band differentiator *
- 'N,Fp,Fst,Ast' — Partial band differentiator *
- 'Ap' — Minimum order full band differentiator *
- 'Fp,Fst,Ap,Ast' — Minimum order partial band differentiator *

The string entries are defined as follows:

- Ap — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- Ast — attenuation in the stop band in decibels (the default units). Also called Astop.

- `Fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.

- `Fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.

- `N` — filter order.

By default, `fdesign.differentiator` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.differentiator(SPEC,specvalue1,specvalue2, ...)` initializes the filter designer specifications in `SPEC` with `specvalue1`, `specvalue2`, and so on. To get a description of the specifications `specvalue1`, `specvalue2`, and more, enter

`get(d,'`**description**`')`

at the Command prompt.

`D = fdesign.differentiator(specvalue1)` assumes the default specification string `N`, setting the filter order to the value you provide.

`D = fdesign.differentiator(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.differentiator(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units

- `'dB'` — specify the magnitude in dB (decibels)

- `'squared'` — specify the magnitude in power units

fdesign.differentiator

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Use an FIR equiripple differentiator to transform frequency modulation into amplitude modulation, which can be detected using an envelope detector.

Modulate a message signal consisting of a 20-Hz sine wave with a 1 kHz carrier frequency. The sampling frequency is 10 kHz .

```
t = linspace(0,1,1e4);
x = cos(2*pi*20*t);
Fc = 1e3;
Fs = 1e4;
y = modulate(x,Fc,Fs,'fm');
```

Design the equiripple FIR differentiator of order 31.

```
d = fdesign.differentiator(31,1e4);
Hd = design(d,'equiripple');
```

Filter the modulated signal and take the Hilbert transform to obtain the envelope.

```
y1 = filter(Hd,y);
y1 = hilbert(y1);
% Plot the envelope
plot(t.*1000,abs(y1));
xlabel('Milliseconds'); ylabel('Magnitude');
grid on;
title('Envelope of the Demodulated Signal');
```

From the preceding figure, you see that the envelope completes two cycles every 100 milliseconds. The envelope is oscillating at 20 Hz, which corresponds to the frequency of the message signal.

Design an FIR differentiator using least squares and plot the zero phase response.

```
d = fdesign.differentiator(33); % Filter order is 33.
hd = design(d,'firls');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')
```

Design a narrow band differentiator. Differentiate the first 25 percent of the frequencies in the Nyquist range and filter the higher frequencies.

```
Fs=20000; %sampling frequency
d = fdesign.differentiator('N,Fp,Fst',54,2500,3000,Fs);
Hd= design(d,'equiripple');
% Weight the stopband to increase attenuation
Hd1 = design(d,'equiripple','Wstop',4);
hfvt = fvtool(Hd,Hd1,'magnitudedisplay','zero-phase',...
'frequencyrange','[0, Fs/2)');
legend(hfvt,'Without stopband weighting',...
'With stopband weighting');
```

See Also   design | fdesign

fdesign.highpass

Purpose   Highpass filter specification object

Syntax

```
D = fdesign.highpass
D = fdesign.highpass(SPEC)
D = fdesign.highpass(SPEC,specvalue1,specvalue2,...)
D = fdesign.highpass(specvalue1,specvalue2,specvalue3,
specvalue4)
D = fdesign.highpass(...,Fs)
D = fdesign.highpass(...,MAGUNITS)
```

Description   `D = fdesign.highpass` constructs a highpass filter specification object D, applying default values for the specification string, `'Fst,Fp,Ast,Ap'`.

`D = fdesign.highpass(SPEC)` constructs object D and sets the `Specification` property to SPEC. Entries in the SPEC string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for SPEC are shown below. The strings are not case sensitive.

> **Note** Specifications strings marked with an asterisk require the DSP System Toolbox software.

- `'Fst,Fp,Ast,Ap'` (default spec)
- `'N,F3db'`
- `'N,F3db,Ap'` *
- `'N,F3db,Ast'` *
- `'N,F3db,Ast,Ap'` *
- `'N,F3db,Fp` *
- `'N,Fc'`
- `'N,Fc,Ast,Ap'`
- `'N,Fp,Ap'`

- `'N,Fp,Ast,Ap'`

- `'N,Fst,Ast'`

- `'N,Fst,Ast,Ap'`

- `'N,Fst,F3db'` *

- `'N,Fst,Fp'`

- `'N,Fst,Fp,Ap'` *

- `'N,Fst,Fp,Ast'` *

- `'Nb,Na,Fst,Fp'` *

The string entries are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- `Ast` — attenuation in the stop band in decibels (the default units). Also called Astop.

- `F3db` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.

- `Fc` — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.

- `Fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.

- `Fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.

- `N` — filter order.

- `Na` and `Nb` are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.

Regions between specification values like Fst and Fp are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a highpass filter specification object change depending on the Specification string. Use designmethods to determine which design method applies to an object and its specification string.

Use designopts to determine which design options are valid for a given design method. For detailed information on design options for a given design method, METHOD, enter help(D,METHOD) at the MATLAB command line.

D = fdesign.highpass(SPEC,specvalue1,specvalue2,...) constructs an object d and sets its specification values at construction time.

D = fdesign.highpass(specvalue1,specvalue2,specvalue3, specvalue4) constructs an object D with the default Specification property and the values you enter for specvalue1,specvalue2,....

D = fdesign.highpass(...,Fs) provides the sampling frequency for the filter specification object. Fs is in Hz and must be specified as a scalar trailing the other numerical values provided. If you specify a sampling frequency, all other frequency specifications are in Hz.

D = fdesign.highpass(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- `'linear'` — specify the magnitude in linear units

- `'dB'` — specify the magnitude in dB (decibels)

- `'squared'` — specify the magnitude in power units

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Higpass filter a discrete-time signal consisting of two sine waves.

Create a highpass filter specification object. Specify the passband frequency to be 0.25π radians/sample and the stopband frequency to be 0.15π radians/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d = fdesign.highpass('Fst,Fp,Ast,Ap',0.15,0.25,60,1);
```

Query the valid design methods for your filter specification object, d.

```
designmethods(d)
```

Create an FIR equiripple filter and view the filter magnitude response with fvtool.

```
Hd = design(d,'equiripple');
fvtool(Hd);
```

Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of π/8 and π/4 radians/sample and amplitudes of 1 and 0.25 respectively. Filter the discrete-time signal with the FIR equiripple filter object, Hd

```
n = 0:159;
x = cos((pi/8)*n)+0.25*sin((pi/4)*n);
y = filter(Hd,x);
Domega = (2*pi)/160;
freq = 0:(2*pi)/160:pi;
```

```
xdft = fft(x);
ydft = fft(y);
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(y)/2+1)),'r','linewidth',2);
legend('Original Signal','Lowpass Signal', ...
'Location','NorthEast');
ylabel('Magnitude'); xlabel('Radians/Sample');
```

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sampling frequency of 48 kHz.

```
d=fdesign.highpass('N,Fc',10,9600,48000);
designmethods(d)
% only valid design method is FIR window method
Hd = design(d);
% Display filter magnitude response
fvtool(Hd);
```

If you have the DSP System Toolbox software, you can specify the shape of the stopband and the rate at which the stopband decays.

Create two FIR equiripple filters with different linear stopband slopes. Specify the passband frequency to be 0.3π radians/sample and the stopband frequency to be 0.35π radians/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB. Design one filter with a 20 dB/rad/sample stopband slope and another filter with 40 dB/rad/sample.

```
D = fdesign.highpass('Fst,Fp,Ast,Ap',0.3,0.35,60,1);
Hd1 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',20);
Hd2 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',40);
hfvt = fvtool([Hd1 Hd2]);
legend(hfvt,'20 dB/rad/sample','40 dB/rad/sample');
```

See Also   design | designmethods | fdesign

fdesign.hilbert

Purpose   Hilbert filter specification object

Syntax
```
d = fdesign.hilbert
d = fdesign.hilbert(specvalue1,specvalue2)
d = fdesign.hilbert(spec)
d = fdesign.hilbert(spec,specvalue1,specvalue2)
d = fdesign.hilbert(...,Fs)
d = fdesign.hilbert(...,MAGUNITS)
```

Description   d = fdesign.hilbert constructs a default Hilbert filter designer d
with N, the filter order, set to 30 and TW, the transition width set to
0.1π radians/sample.

d = fdesign.hilbert(specvalue1,specvalue2) constructs a Hilbert
filter designer d assuming the default specification string 'N,TW'. You
input specvalue1 and specvalue2 for N and TW.

d = fdesign.hilbert(spec) initializes the filter designer
Specification property to spec. You provide one of the following
strings as input to replace spec. The specification strings are not case
sensitive.

Note Specifications strings marked with an asterisk require the DSP
System Toolbox software.

- 'N,TW' default spec string.

- 'TW,Ap' *

The string entries are defined as follows:

- Ap — amount of ripple allowed in the pass band in decibels (the
  default units). Also called Apass.

- N — filter order.

- TW — width of the transition region between the pass and stop bands.

By default, `fdesign.hilbert` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Different specification strings may have different design methods available. Use `designmethods(d)` to get a list of the design methods available for a given specification string.

`d = fdesign.hilbert(`*spec*`,specvalue1,specvalue2)` initializes the filter designer specifications in `spec` with `specvalue1`, `specvalue2`, and so on. To get a description of the specifications `specvalue1` and `specvalue2`, enter

`get(d,'`**description**`')`

at the Command prompt.

`d = fdesign.hilbert(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.hilbert(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Design a Hilbert transformer of order 30 with a transition width of 0.2π radians/sample. Plot the zero phase response from [-π,π) radians/sample and the impulse response.

```
d = fdesign.hilbert('N,TW',30,0.2);
```

```
% Show available design methods
designmethods(d)
% Use least square minimization to obtain linear-phase FIR filter
Hd = design(d,'equiripple');
% Display zero phase response from [-pi,pi)
fvtool(Hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')
```


The impulse response of this even order filter is antisymmetric (type III).

```
fvtool(Hd,'analysis','impulse')
```


Apply the filter to a discrete-time sinusoid with a frequency of π/2 radians/sample.

```
n = 0:99;
x = cos(pi/2*n);
y = filter(Hd,x);
% Correct for the filter delay
Delay = floor(length(Hd.Numerator)/2);
y = y(Delay+1:end);
```

fdesign.hilbert

Plot a the filter input and output and validate the approximate π/2 phase shift obtained with the Hilbert transformer.

```
stem(x(1:end-Delay),'markerfacecolor',[0 0 1]);
hold on;
stem(y,'Color',[1 0 0],'markerfacecolor',[1 0 0]);
axis([10 20 -1.5 1.5]); grid on;
xlabel('Samples'); ylabel('Amplitude');
legend('Input','Output','Location','NorthWest')
```

Because the frequency of the discrete-time sinusoid is π/2 radians/sample, a one sample shift corresponds to a phase shift of π/2.

Form the analytic signal and demonstrate that the frequency content of the analytic signal is zero for negative frequencies and approximately twice the spectrum of the input for positive frequencies.

```
x1 = x(1:end-Delay);
% Form the analytic signal
xa = x1+1j*y;
freq = -pi:(2*pi)/length(x1):pi-(2*pi)/length(x);
plot(freq,abs(fftshift(fft(x1))));
hold on;
plot(freq,abs(fftshift(fft(xa))),'r'); grid on;
xlabel('Radians/Sample'); ylabel('Magnitude');
legend('Input DFT','Analytic Signal DFT','Location','NorthWest');
```

Design a minimum-order Hilbert transformer that has a sampling frequency of 1 kHz. Specify the passband ripple to be 1 dB.

```
d = fdesign.hilbert('TW,Ap',1,0.1,1e3);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase', ...
'frequencyrange','[-Fs/2, Fs/2)');
```

See Also   design | fdesign | setspecs

fdesign.lowpass

Purpose	Lowpass filter specification

Syntax

```
D = fdesign.lowpass
D = fdesign.lowpass(SPEC)
D = fdesign.lowpass(SPEC,specvalue1,specvalue2,...)
D = fdesign.lowpass(specvalue1,specvalue2,specvalue3,
    specvalue4)
D = fdesign.lowpass(...,Fs)
D = fdesign.lowpass(...,MAGUNITS)
```

Description   D = fdesign.lowpass constructs a lowpass filter specification object D, applying default values for the default specification string 'Fp,Fst,Ap,Ast'.

D = fdesign.lowpass(SPEC) constructs object D and sets the Specification property to the string in SPEC. Entries in the SPEC string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for SPEC are shown below. The strings are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP System Toolbox software.

- 'Fp,Fst,Ap,Ast' (default spec)
- 'N,F3db'
- 'N,F3db,Ap' *
- 'N,F3db,Ap,Ast' *
- 'N,F3db,Ast' *
- 'N,F3db,Fst' *
- 'N,Fc'
- "N,Fc,Ap,Ast'

- `'N,Fp,Ap'`
- `'N,Fp,Ap,Ast'`
- `'N,Fp,Fst,Ap'` *
- `'N,Fp,F3db'` *
- `'N,Fp,Fst'`
- `'N,Fp,Fst,Ast'` *
- `'N,Fst,Ap,Ast'` *
- `'N,Fst,Ast'`
- `'Nb,Na,Fp,Fst'` *

The string entries are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- `Ast` — attenuation in the stop band in decibels (the default units). Also called Astop.

- `F3db` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.

- `Fc` — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.

- `Fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.

- `Fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.

- `N` — filter order.

- `Na` and `Nb` are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.

Regions between specification values like `Fp` and `Fst` are transition regions where the filter response is not explicitly defined.

`D = fdesign.lowpass(SPEC,specvalue1,specvalue2,...)` constructs an object `D` and sets the specification values at construction time using `specvalue1`, `specvalue2`, and so on for all of the specification variables in `SPEC`.

`D = fdesign.lowpass(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `D` with values for the default `Specification` property string `'Fp,Fst,Ap,Ast'` using the specifications you provide as input arguments `specvalue1,specvalue2,specvalue3,specvalue4`.

`D = fdesign.lowpass(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.lowpass(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units

- `'dB'` — specify the magnitude in dB (decibels)

- `'squared'` — specify the magnitude in power units

When you omit the MAGNUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Lowpass filter a discrete-time signal consisting of two sine waves.

Create a lowpass filter specification object. Specify the passband frequency to be 0.15π radians/sample and the stopband frequency to be 0.25π radians/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d=fdesign.lowpass('Fp,Fst,Ap,Ast',0.15,0.25,1,60);
```

Query the valid design methods for your filter specification object, d.

```
designmethods(d)
```

Create an FIR equiripple filter and view the filter magnitude response with fvtool.

```
Hd = design(d,'equiripple');
fvtool(Hd);
```

Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of π/8 and π/4 radians/sample and amplitudes of 1 and 0.25 respectively. Filter the discrete-time signal with the FIR equiripple filter object, Hd.

```
n = 0:159;
x = 0.25*cos((pi/8)*n)+sin((pi/4)*n);
y = filter(Hd,x);
Domega = (2*pi)/160;
freq = 0:(2*pi)/160:pi;
xdft = fft(x);
ydft = fft(y);
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(y)/2+1)),'r','linewidth',2);
```

fdesign.lowpass

```
legend('Original Signal','Highpass Signal', ...
'Location','NorthEast');
ylabel('Magnitude'); xlabel('Radians/Sample');
```

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sampling frequency of 48 kHz.

```
d=fdesign.lowpass('N,Fc',10,9600,48000);
designmethods(d)
% only valid design method is FIR window method
Hd = design(d);
% Display filter magnitude response
fvtool(Hd);
```

Zoom in on the magnitude response to verify that the -6 dB point is at 9.6 kHz.

If you have the DSP System Toolbox software, you can specify the shape of the stopband and the rate at which the stopband decays. The following example requires the DSP System Toolbox.

Create an FIR equiripple filter with a passband frequency of 0.2п radians/sample, a stopband frequency of 0.25п radians/sample, a passband ripple of 1 dB, and a stopband attenuation of 60 dB. Design the filter with a 20 dB/rad/sample linear stopband.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,1,60);
Hd = design(D,'equiripple','StopBandShape','linear','StopBandDecay',20);
fvtool(Hd);
```

See Also   design | designmethods | fdesign

| **Purpose** | Pulse-shaping filter specification object |

Syntax

```
D = fdesign.pulseshaping
D = fdesign.pulseshaping(sps)
D = fdesign.pulseshaping(sps,shape)
d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)
d = fdesign.pulseshaping(...,fs)
d = fdesign.pulseshaping(...,magunits)
```

Description   D = fdesign.pulseshaping constructs a specification object D, which can be used to design a minimum-order raised cosine filter object with a default stop band attenuation of 60dB and a rolloff factor of 0.25.

D = fdesign.pulseshaping(sps) constructs a minimum-order raised cosine filter specification object d with a positive integer-valued oversampling factor, SamplesPerSymbol .

D = fdesign.pulseshaping(sps,shape) constructs d where shape specifies the PulseShape property. Valid entries for shape are:

- 'Raised Cosine'

- 'Square Root Raised Cosine'

- 'Gaussian'

d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...) constructs d where spec defines the Specification properties. The string entries for spec specify various properties of the filter, including the order and frequency response. Valid entries for spec depend upon the shape property. For 'Raised Cosine' and 'Square Root Raised Cosine' filters, the valid entries for spec are:

- 'Ast,Beta' (minimum order; default)

- 'Nsym,Beta'

- 'N,Beta'

The string entries are defined as follows:

- `Ast` —stopband attenuation (in dB). The default stopband attenuation for a raised cosine filter is 60 dB. The default stopband attenuation for a square root raised cosine filter is 30 dB. If `Ast` is specified, the minimum-order filter is returned.

- `Beta` —rolloff factor expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.

- `Nsym` —filter order in symbols. The length of the impulse response is given by `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.

- `N` —filter order (must be even). The length of the impulse response is `N+1`.

If the `shape` property is specified as `'Gaussian'`, the valid entries for `spec` are:

- `'Nsym,BT'` (default)

The string entries are defined as follows:

- `Nsym`—filter order in symbols. `Nsym` defaults to 6. The length of the filter impulse response is `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.

- `BT` —the 3–dB bandwidth-symbol time product. `BT` is a positive real-valued scalar, which defaults to 0.3. Larger values of `BT` produce a narrower pulse width in time with poorer concentration of energy in the frequency domain.

`d = fdesign.pulseshaping(...,fs)` specifies the sampling frequency of the signal to be filtered. `fs` must be specified as a scalar trailing the other numerical values provided. For this case, `fs` is assumed to be in Hz and is used for analysis and visualization.

`d = fdesign.pulseshaping(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. Valid entries for `magunits` are:

- `linear` — specify the magnitude in linear units

- dB — specify the magnitude in dB (decibels)

- squared — specify the magnitude in power units

When you omit the magunits argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

After creating the specification object d, you can use the design function to create a filter object such as h in the following example:

```
d = fdesign.pulseshaping(8,'Raised Cosine','Nsym,Beta',6,0.25);
h = design(d);
```

Normally, the Specification property of the specification object also determines which design methods you can use when you create the filter object. Currently, regardless of the Specification property, the design function uses the window design method with all fdesign.pulseshaping specification objects. The window method creates an FIR filter with a windowed impulse response.

Examples

Pulse-shaping can be used to change the waveform of transmitted pulses so the signal bandwidth matches that of the communication channel. This helps to reduce distortion and intersymbol interference (ISI).

This example shows how to design a minimum-order raised cosine filter that provides a stop band attenuation of 60 dB, rolloff factor of 0.50, and 8 samples per symbol.

```
h  = fdesign.pulseshaping(8,'Raised Cosine','Ast,Beta',60,0.50);
   Hd = design(h);
   fvtool(Hd)
```

This code generates the following figure.

This example shows how to design a raised cosine filter that spans 8 symbol durations (i.e., of order 8 symbols), has a rolloff factor of 0.50, and oversampling factor of 10.

```
h  = fdesign.pulseshaping(10,'Raised Cosine','Nsym,Beta',8,0.50);
   Hd = design(h);
   fvtool(Hd, 'impulse')
```

This example shows how to design a square root raised cosine filter of order 42, rolloff factor of 0.25, and 10 samples per symbol.

```
h  = fdesign.pulseshaping(10,'Square Root Raised Cosine','N,Beta',42);
   Hd = design(h);
   fvtool(Hd, 'impulse')
```


The following example demonstrates how to create a Gaussian pulse-shaping filter with an oversampling factor (sps) of 10, a bandwidth-time symbol product of 0.2, and 8 symbol periods. The sampling frequency is specified as 10 kHz.

Purpose   FFT-based FIR filtering using overlap-add method

Syntax
```
y = fftfilt(b,x)
y = fftfilt(b,x,n)
y = fftfilt(gpuArrayb,gpuArrayX,n)
```

Description   `fftfilt` filters data using the efficient FFT-based method of
overlap-add, a frequency domain filtering technique that works only
for FIR filters.

`y = fftfilt(b,x)` filters the data in vector `x` with the filter described
by coefficient vector `b`. It returns the data vector `y`. The operation
performed by `fftfilt` is described in the *time domain* by the difference
equation:

$$y(n) = b(1)x(n) + b(2)x(n-1) + \cdots + b(nb+1)x(n-nb)$$

An equivalent representation is the *z*-transform or *frequency domain*
description:

$$Y(z) = \left(b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb}\right)X(z)$$

By default, `fftfilt` chooses an FFT length and data block length that
guarantee efficient execution time.

If `x` is a matrix, `fftfilt` filters its columns. If `b` is a matrix, `fftfilt`
applies the filter in each column of `b` to the signal vector `x`. If `b` and `x`
are both matrices with the same number of columns, the i-th column of
`b` is used to filter the i-th column of `x`.

`y = fftfilt(b,x,n)` uses `n` to determine the length of the FFT. See
"Algorithms" on page 3-351 for information.

`y = fftfilt(gpuArrayb,gpuArrayX,n)` filters the data in the
`gpuArray` object, `gpuArrayX`, with the FIR filter coefficients in the
`gpuArray`, `gpuArrayb`. See "Using gpuArray" for details on
gpuArray objects. Using `fftfilt` with `gpuArray` objects requires
Parallel Computing Toolbox software and a CUDA-enabled

fftfilt

NVIDIA GPU with compute capability 1.3 or above. See
http://www.mathworks.com/products/parallel-computing/requirements.html
for details. The filtered data, y, is a gpuArray object. See "Overlap-Add
Filtering on the GPU" on page 3-351 for example of overlap-add
filtering on the GPU.

fftfilt works for both real and complex inputs.

Comparison to filter function

When the input signal is relatively large, it is advantageous to use
fftfilt instead of filter, which performs N multiplications for each
sample in x, where N is the filter length. fftfilt performs 2 FFT
operations — the FFT of the signal block of length L plus the inverse
FT of the product of the FFTs — at the cost of

$$1/2*L*log2(L)$$

where L is the block length. It then performs L pointwise multiplications
for a total cost of

$$L+L*log2(L) = L*(1+log2(L))$$

multiplications. The cost ratio is therefore

$$L*(1+log2(L))/(N*L) = (1+log2(L))/N$$

which is approximately log2(L)/N.

Therefore, fftfilt becomes advantageous when log2(L) is less than N.

Examples Show that the results from fftfilt and filter are identical:

```
b = [1 2 3 4];
x = [1 zeros(1,99)]';
norm(fftfilt(b,x) - filter(b,1,x))
```

Overlap-Add Filtering on the GPU

The following example requires Parallel Computing
Toolbox software and a CUDA-enabled NVIDIA
GPU with compute capability 1.3 or above. See
http://www.mathworks.com/products/parallel-computing/requirements.html
for details.

Create a signal consisting of a sum of sine waves in white Gaussian
additive noise. The sine wave frequencies are 2.5, 5, 10, and 15 kHz.
The sampling frequency is 50 kHz.

```
Fs = 50e3;
t = 0:1/Fs:10-(1/Fs);
x = cos(2*pi*2500*t)+0.5*sin(2*pi*5000*t)+0.25*cos(2*pi*10000*t)+0.125
```

Design a lowpass FIR equiripple filter using `fdesign.lowpass`.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',5500,6000,0.5,50,50e3);
Hd = design(d,'equiripple');
b = Hd.Numerator;
```

Filter the data on the GPU using the overlap-add method. Put the
data on the GPU using `gpuArray`. Return the output to the MATLAB
workspace using `gather` and plot the power spectral density estimate
of the filtered data.

```
y = fftfilt(gpuArray(b),gpuArray(x));
plot(psd(spectrum.periodogram,gather(y),'Fs',50e3,'NFFT',length(y)));
```

Algorithms   `fftfilt` uses `fft` to implement the *overlap-add method* [1], a technique
that combines successive frequency domain filtered blocks of an input
sequence. `fftfilt` breaks an input sequence x into length L data
blocks, where L must be greater than the filter length N.

```
x  ├────────┼────────┼────────┼────────┤ . . . ├────────┤
           L       2L       3L              ceil(nx/L)*L
```

and convolves each block with the filter b by

```
y = ifft(fft(x(i:i+L-1),nfft).*fft(b,nfft));
```

where `nfft` is the FFT length. `fftfilt` overlaps successive output sections by `n-1` points, where `n` is the length of the filter, and sums them.

`fftfilt` chooses the key parameters `L` and `nfft` in different ways, depending on whether you supply an FFT length `n` and on the lengths of the filter and signal. If you do not specify a value for `n` (which determines FFT length), `fftfilt` chooses these key parameters automatically:

- If `length(x)`is greater than `length(b)`, `fftfilt` chooses values that minimize the number of blocks times the number of flops per FFT.

- If `length(b)` is greater than or equal to `length(x)`, `fftfilt` uses a single FFT of length

  ```
  2^nextpow2(length(b) + length(x) - 1)
  ```

  This essentially computes

  ```
  y = ifft(fft(B,nfft).*fft(X,nfft))
  ```

If you supply a value for `n`, `fftfilt` chooses an FFT length, `nfft`, of `2^nextpow2(n)`and a data block length of `nfft - length(b) + 1`. If `n` is less than `length(b)`, `fftfilt` sets `n` to `length(b)`.

References [1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing,* Prentice-Hall, 1989.

See Also conv | dfilt.fftfir | filter | filtfilt

filter

Purpose   Filter data with recursive (IIR) or nonrecursive (FIR) filter

Description   filter is a MATLAB function.

Signal-Specific **Filter Method of DFILT**
Information

Filter is also an overloaded method of the discrete-time filter object (dfilt). You can pass an object handle, data, and optionally, the dimension into the filter method.

The MATLAB filter function describes a zi input for initial conditions. Note that the recommended way of passing initial conditions into a dfilt is by using the states property. For more information, see the dfilt reference page.

Filter Normalization

Using the filter function on b and a coefficients normalizes the filter by forcing the a_0 coefficient to be equal to 1.

Using the filter method on a dfilt object does not normalize the a_0 coefficient.

FIR Filters

The denominator of FIR filters is, by definition, equal to 1. To use the filter function with the b coefficients from an FIR function, use y = filter(b,1,x) .

filterbuilder

Purpose   GUI-based filter design

Syntax   `filterbuilder(h)`
                 `filterbuilder('`*`response`*`')`

Description `filterbuilder` starts a GUI-based tool for building filters. It relies
                 on the `fdesign` object-object oriented filter design paradigm, and is
                 intended to reduce development time during the filter design process.
                 `filterbuilder` uses a specification-centered approach to find the best
                 algorithm for the desired response.

Note You must have the Signal Processing Toolbox installed to use
`fdesign` and `filterbuilder`. Some of the features described below
may be unavailable if your installation does not additionally include the
DSP System Toolbox. You can verify the presence of both toolboxes by
typing `ver` at the command prompt.

The `filterbuilder` GUI contains many features not available in
FDATool. For more information on how to use `filterbuilder`, see
"Filterbuilder Design Process".

To use `filterbuilder`, enter `filterbuilder` at the MATLAB command
line using one of three approaches:

- Simply enter `filterbuilder`. MATLAB opens a dialog for you to
  select a filter response type. After you select a filter response type,
  `filterbuilder` launches the appropriate filter design dialog box.

- Enter `filterbuilder(h)`, where h is an existing filter object. For
  example, if h is a bandpass filter, `filterbuilder(h)` opens the
  bandpass filter design dialog box. (The h object must have been
  created using `filterbuilder` or must be a `dfilt` ,`mfilt`, or filter
  System object created using fdesign.)

Note You must have the DSP System Toolbox software to create and import filter System objects.

- Enter filterbuilder('*response*'), replacing *response* with a response string from the following table. MATLAB opens a filter design dialog that corresponds to the response string.

Note You must have the DSP System Toolbox software to implement a number of the filter designs listed in the following table. If you only have the Signal Processing Toolbox software, you can design a limited set of the following filter-response types.

Response String	Description of Resulting Filter Design	Filter Object
arbgrpdelay	Arbitrary group delay filter design	fdesign.arbgrpdelay
arbmag	Arbitrary magnitude filter design	fdesign.arbmag
arbmagnphase	Arbitrary response filter (magnitude and phase)	fdesign.arbmagnphase
audioweighting	Audio weighting filter	fdesign.audioweighting
bandpass or bp	Bandpass filter	fdesign.bandpass
bandstop or bs	Bandstop filter	fdesign.bandstop
cic	CIC filter	fdesign.decimator(M,'cic',... or
ciccomp	CIC compensator	fdesign.ciccomp r(L,'cic',
comb	Comb filter	fdesign.comb

and
fdesign.interpolator

filterbuilder

Response String	Description of Resulting Filter Design	Filter Object
diff	Differentiator filter	`fdesign.differentiator`
fracdelay	Fractional delay filter	`fdesign.fracdelay`
halfband or `hb`	Halfband filter	`fdesign.halfband`
highpass or `hp`	Highpass filter	`fdesign.highpass`
hilb	Hilbert filter	`fdesign.hilbert`
isinc, isinclp, or isinchp	Inverse sinc lowpass or highpass filter	`fdesign.isinclp` and `fdesign.isinchp`
lowpass or `lp`	Lowpass filter (default)	`fdesign.lowpass`
notch	Notch filter	`fdesign.notch`
nyquist	Nyquist filter	`fdesign.nyquist`
octave	Octave filter	`fdesign.octave`
parameq	Parametric equalizer filter	`fdesign.parameq`
peak	Peak filter	`fdesign.peak`
pulseshaping	Pulse-shaping filter	`fdesign.pulseshaping`

Note Because they do not change the filter structure, the magnitude specifications and design method are tunable when using `filterbuilder`.

Filterbuilder Design Panes

Main Design Pane

The main pane of filterbuilder varies depending on the filter response type, but the basic structure is the same. The following figure shows the basic layout of the dialog box.

As you choose the response for the filter, the available options and design parameters displayed in the dialog box change. This display allows you to focus only on parameters that make sense in the context of your filter design.

Every filter design dialog box includes the options displayed at the top of the dialog box, shown in the following figure.

Save variable as: Hbp   View Filter Response

- **Save variable as** — When you click **Apply** to apply your changes or **OK** to close this dialog box, `filterbuilder` saves the current filter to your MATLAB workspace as a filter object with the name you enter.

- **View Filter Response** — Displays the magnitude response for the current filter specifications and design method by opening the Filter Visualization Tool (`fvtool`).

Note The `filterbuilder` dialog box includes an **Apply** option. Each time you click **Apply**, `filterbuilder` writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes without overwriting the variable in you workspace, change the variable name in **Save variable as** before you click **Apply**.

There are three tabs in the Filterbuilder dialog box, containing three panes: **Main**, **Data Types**, and **Code Generation**. The first pane changes according to the filter being designed. The last two panes are the same for all filters. These panes are discussed in the following sections.

Data Types Pane

The second tab in the Filterbuilder dialog box is shown in the following figure.

The **Arithmetic** drop down box allows the choice of Double precision, Single precision, or Fixed point. Some of these options may be unavailable depending on the filter parameters. The following table describes these options.

Arithmetic List Entry	Effect on the Filter
Double precision	All filtering operations and coefficients use double-precision, floating-point representations and math. When you use filterbuilder to

Arithmetic List Entry	Effect on the Filter
	create a filter, `double precision` is the default value for the Arithmetic property.
`Single precision`	All filtering operations and coefficients use single-precision floating-point representations and math.
`Fixed point`	This string applies selected default values, typically used on many digital processors, for the properties in the fixed-point filter. These properties include coefficient word lengths, fraction lengths, and various operating modes. This setting allows signed fixed data types only. Fixed-point filter design with `filterbuilder` is available only when you install Fixed-Point Toolbox™ software along with DSP System Toolbox software.

The following figure shows the **Data Types** pane after you select `Fixed point` for **Arithmetic** and set **Filter internals** to `Specify precision`. This figure shows the **Data Types** pane for the case where the **Use a System object to implement filter** check box is not selected in the **Main** pane.

Input signal

Specify the format the filter applies to data to be filtered. For all cases, `filterbuilder` implements filters that use binary point scaling and signed input. You set the word length and fraction length as needed.

Coefficients

Choose how you specify the word length and the fraction length of the filter numerator and denominator coefficients:

- `Specify word length` enables you to enter the word length of the coefficients in bits. In this mode, `filterbuilder` automatically sets the fraction length of the coefficients to the binary-point only scaling that provides the best possible precision for the value and word length of the coefficients.

- `Binary point scaling` enables you to enter the word length and the fraction length of the coefficients in bits. If applicable, enter separate fraction lengths for the numerator and denominator coefficients.

- The filter coefficients do not obey the **Rounding mode** and **Overflow mode** parameters that are available when you select `Specify precision` from the Filter internals list. Coefficients are always saturated and rounded to `Nearest`.

Section Input

Choose how you specify the word length and the fraction length of the fixed-point data type going into each section of an SOS filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- `Binary point scaling` enables you to enter the word and fraction lengths of the section input in bits.

- `Specify word length` enables you to enter the word lengths in bits.

Section Output

Choose how you specify the word length and the fraction length of the fixed-point data type coming out of each section of an SOS

filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- `Binary point scaling` enables you to enter the word and fraction lengths of the section output in bits.

- `Specify word length` enables you to enter the output word lengths in bits.

State

Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Use this parameter to specify how to designate the state word and fraction lengths. This parameter is not visible for direct form and direct form I filter structures because `filterbuilder` deduces the state directly from the input format. States always use signed representation:

- `Binary point scaling` enables you to enter the word length and the fraction length of the accumulator in bits.

- `Specify precision` enables you to enter the word length and fraction length in bits (if available).

Product

Determines how the filter handles the output of product operations. Choose from the following options:

- `Full precision` — Maintain full precision in the result.

- `Keep LSB` — Keep the least significant bit in the result when you need to shorten the data words.

- `Specify Precision` — Enables you to set the precision (the fraction length) used by the output from the multiplies.

Filter internals

Specify how the fixed-point filter performs arithmetic operations within the filter. The affected filter portions are filter products, sums, states, and output. Select one of these options:

- `Full precision` — Specifies that the filter maintains full precision in all calculations for products, output, and in the accumulator.

- `Specify precision` — Set the word and fraction lengths applied to the results of product operations, the filter output, and the accumulator. Selecting this option enables the word and fraction length controls.

Signed

Selecting this option directs the filter to use signed representations for the filter coefficients.

Word length

Sets the word length for the associated filter parameter in bits.

Fraction length

Sets the fraction length for the associate filter parameter in bits.

Accum

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

Determines how the accumulator outputs stored values. Choose from the following options:

- `Full precision` — Maintain full precision in the accumulator.

- `Keep MSB` — Keep the most significant bit in the accumulator.

- `Keep LSB` — Keep the least significant bit in the accumulator when you need to shorten the data words.

- `Specify Precision` — Enables you to set the precision (the fraction length) used by the accumulator.

Output

Sets the mode the filter uses to scale the output data after filtering. You have the following choices:

- `Avoid Overflow` — Set the output data fraction length to avoid causing the data to overflow. `Avoid overflow` is considered

the conservative setting because it is independent of the input data values and range.

- `Best Precision` — Set the output data fraction length to maximize the precision in the output data.

- `Specify Precision` — Set the fraction length used by the filtered data.

Fixed-point operational parameters

Parameters in this group control how the filter rounds fixed-point values and how it treats values that overflow.

Rounding mode

Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).

- `ceil` - Round toward positive infinity.

- `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.

- `zero/fix` - Round toward zero.

- `floor` - Round toward negative infinity.

- `nearest` - Round toward nearest. Ties round toward positive infinity.

- `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

Overflow mode

Sets the mode the filter uses to respond to overflow conditions in fixed-point arithmetic. Choose from the following options:

- `Saturate` — Limit the output to the largest positive or negative representable value.

- `Wrap` — Set overflowing values to the nearest representable value using modular arithmetic.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

Cast before sum

Specifies whether to cast numeric data to the appropriate accumulator format before performing sum operations. Selecting **Cast before sum** ensures that the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

If you clear **Cast before sum**, the filter prevents the addends from being cast to the sum format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use. The input format referenced by **Cast before sum** depends on the filter structure you are using.

The effect of clearing or selecting **Cast before sum** is as follows:

- Cleared — Configures filter summation operations to retain the addends in the format carried from the previous operation.

- Selected — Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually, selecting **Cast before sum** generates results from the summation that more closely match those found from digital signal processors.

Code Generation Pane

The code generation pane contains options for various implementations of the completed filter design. Depending on your installation, you can generate MATLAB, VHDL, and Verilog code from the designed filter. You can also choose to create or update a Simulink model from the designed filter. The following section explains these options.

HDL

For more information on this option, see "Opening the Filter Design HDL Coder™ GUI From the filterbuilder GUI".

MATLAB

Generate MATLAB code based on filter specifications

- **Generate function that returns your filter as an output**

  Selecting this option generates a function that designs either a DFILT/MFILT object or a system object (depending on whether you have selected the **Use a System object to implement the filter** check box) using fdesign. The function call returns a filter object.

- **Generate function that filters your data**

  Selecting this option generates a function that takes data as input, and outputs data filtered with the designed filter.

  Clicking on the **Generate MATLAB code** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable file.

Simulink Model

Generate Simulink blocks and subsystems from your designed filters

When the **Use a System object to implement filter** check box is selected in the **Main** pane, you are able to generate Simulink models as long as the **Arithmetic** is not set to Fixed point in the **Data Types** pane. If the **Arithmetic** is set to Fixed point, the **Generate Model** button in the **Simulink model** panel will be disabled.

Clicking on the **Generate Model** button brings up the **Export to Simulink** dialog box, as shown in the following figure.

You can set the following parameters in this dialog box:

- **Block Name** — The name for the new subsystem block, set to **Filter** by default.

- **Destination** — **Current** saves the generated model to the current Simulink model; **New** creates a new model to contain the generated block; **User Defined** creates a new model or subsystem to the user-specified location enumerated in the **User Defined** text box.

- **Overwrite generated 'Filter' block** — When this check box is selected, DSP System Toolbox software overwrites an existing block with the name specified in **Block Name**; when cleared, creates a new block with the same name.

- **Build model using basic elements** — When this check box is selected, DSP System Toolbox software builds the model using only basic blocks.

- **Optimize for zero gains** — When this check box is selected, DSP System Toolbox software removes all zero gain blocks from the model.

- **Optimize for unity gains** — When this check box is selected, DSP System Toolbox software replaces all unity gains with direct connections.

- **Optimize for negative gains** — When this check box is selected, DSP System Toolbox software removes all negative unity gain blocks, and changes sign at the nearest summation block.

- **Optimize delay chains** — When this check box is selected, DSP System Toolbox software replaces delay chains made up of *n* unit delays with a single delay by *n*.

- **Optimize for unity scale values** — When this check box is selected, DSP System Toolbox software removes all scale value multiplications by 1 from the filter structure.

- **Input processing** — Specify how the generated filter block or subsystem block processes the input. Depending on the type of filter you are designing, one or both of the following options may be available:

  - Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

  - Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

  For more information about sample- and frame-based processing, see "Sample- and Frame-Based Concepts".

- **Realize Model** — DSP System Toolbox software builds the model with the set parameters.

Filter Responses

Select your filter response from the filterbuilder **Response Selection** main menu.

If you have the DSP System Toolbox software, the following **Response Selection** menu appears.

Select your desired filter response from the menu and design your filter.

The following sections describe the options available for each response type.

Arbitrary Response Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

This dialog only applies if you have the DSP System Toolbox software. Select either FIR or IIR from the drop down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly. Arbitrary group delay designs are only available if **Impulse response** is IIR. Without the DSP System Toolbox, the only available arbitrary response filter design is FIR.

Order mode

This dialog only applies if you have the DSP System Toolbox software. Choose Minimum or Specify. Choosing Specify enables the **Order** dialog.

Order

This dialog only applies when **Order mode** is Specify. For an FIR design, specify the filter order. For an IIR design, you can specify an equal order for the numerator and denominator, or you can specify different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box. Because the Signal Processing Toolbox only supports FIR arbitrary-magnitude filters, you do not have the option to specify a denominator order.

Denominator order

Select the check box and enter the denominator order. This option is enabled only if IIR is selected for **Impulse response**.

Filter type

This dialog only applies if you have the DSP System Toolbox software and is only available for FIR filters. Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods

and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2 for `Decimator` and 3 for `Sample-rate converter`.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Response Specification

Number of Bands

Select the number of bands in the filter. Multiband design is available for both FIR and IIR filters.

Specify response as:

Specify the response as `Amplitudes`, `Magnitudes and phase`, `Frequency response`, or `Group delay`. `Amplitudes` is the only option if you do not have the DSP System Toolbox software. `Group delay` is only available for IIR designs.

Frequency units

Specify frequency units as either `Normalized`, `Hz`, `kHz`, `MHz`, or `GHz`.

Input Fs

Enter the input sampling frequency in the units specified in the **Frequency units** drop-down box. This option is enabled when **Frequency units** is set to an option in hertz.

Band Properties

These properties are modified automatically depending on the response chosen in the **Specify response as** drop-down box. Two or three columns are presented for input. The first column is always Frequencies. The other columns are either Amplitudes, Magnitudes, Phases, or Frequency Response. Enter the corresponding vectors of values for each column.

- **Frequencies** and **Amplitudes** — These columns are presented for input if you select Amplitudes in the **Specify response as** drop-down box.

- **Frequencies**, **Magnitudes**, and **Phases** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is Magnitudes and phases.

- **Frequencies** and **Frequency response** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is Frequency response.

Algorithm

The options for each design are specific for each design method. In the arbitrary response design, the available options also depend on the **Response specifications**. This section does not present all of the available options for all designs and design methods.

Design Method

Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

Design Options

- **Window** — Valid when the **Design method** is `Frequency Sampling`. Replace the square brackets with the name of a `window` function or function handle. For example, `'hamming'` or `@hamming`. If the window function takes parameters other than the length, use a cell array. For example, `{`kaiser',3.5}` or `{@chebwin,60}`.

- **Density factor** — Valid when the **Design method** is `equiripple`. Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

  Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

  The default changes to 20 for an IIR arbitrary group delay design.

- **Phase constraint** — Valid when the **Design method** is `equiripple`, you have the DSP System Toolbox installed, and **Specify response as** is set to `Amplitudes`. Choose one of `Linear`, `Minimum`, or `Maximum`.

- **Weights** — Uses the weights in **Weights** to weight the error for a single-band design. If you have multiple frequency bands, the **Weights** design option changes to **B1 Weights**, **B2 Weights** to designate the separate bands. Use **Bi Weights** to specify weights for the i-th band. The **Bi Weights** design option is only available when you specify the i-th band as an unconstrained.

- **Bi forced frequency point** — This option is only available in a multi-band constrained equiripple design when **Specify response as** is `Amplitudes`. **Bi forced frequency point** is

the frequency point in the i-th band at which the response is forced to be zero. The index **i** corresponds to the frequency bands in **Band properties**. For example, if you specify two bands in **Band properties**, you have **B1 forced frequency point** and **B2 forced frequency point**.

- **Norm** — Valid only for IIR arbitrary group delay designs. **Norm** is the norm used in the optimization. The default value is 128, which essentially equals the L-infinity norm. The norm must be even.

- **Max pole radius** — Valid only for IIR arbitrary group delay designs. Constrains the maximum pole radius. The default is 0.999999. Reducing the **Max pole radius** can produce a transfer function more resistant to quantization.

- **Init norm** — Valid only for IIR arbitrary group delay designs. The initial norm used in the optimization. The default initial norm is 2.

- **Init numerator** — Specifies an initial estimate of the filter numerator coefficients.

- **Init denominator** — Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems. In allpass filters, you only have to specify either the denominator or numerator coefficients. If you specify the denominator coefficients, you can obtain the numerator coefficients.

Filter implementation

Structure
Select the structure for the filter. The available filter structures depend on the parameters you select for your filter.

Use a System object to implement filter
Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned

off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Audio Weighting Filter Design Dialog Box — Main Pane

Filter specifications

- **Weighting type** — The weighting type defines the frequency response of the filter. The valid weighting types are: A, C , C-message, ITU-T 0.41, and ITU-R 468–4 weighting. See `fdesign.audioweighting` for definitions of the weighting types.

- **Class** — Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in `fvtool` for the analysis of the filter design.

- **Impulse response** — Impulse response type as one of `IIR` or `FIR`. For A, C , C-message, and ITU-R 468–4 filter, IIR is the only option. For a ITU-T 0.41 weighting filter, FIR is the only option.

- **Frequency units** — Choose `Hz`, `kHz`, `MHz`, or `GHz`. Normalized frequency designs are not supported for audio weighting filters.

- **Input Fs** — The sampling frequency in **Frequency units**. For example, if **Frequency units** is set to `kHz`, setting **Input Fs** to 40 is equivalent to a 40 kHz sampling frequency.

Algorithm

- **Design method** — Valid design methods depend on the weighting type. For type A and C weighting filters, the only valid design type is `ANSI S1.42`. This is an IIR design method that follows ANSI standard S1.42–2001. For a C message filter, the only valid design method is `Bell 41009`, which is an IIR design method following the Bell System Technical Reference PUB 41009. For a ITU-R 468–4 weighting filter, you can design an IIR or FIR filter. If you choose an IIR design, the design method is `IIR least p-norm`. If you choose an FIR design, the design method choices are: `Equirriple` or `Frequency Sampling`. For an ITU-T 0.41 weighting filter, the available FIR design methods are `equirriple` or `Frequency Sampling`

- **Scale SOS filter coefficients to reduce chance of overflow** — Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter implementation

- **Structure** — For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. For audio weighting IIR filter designs, you can choose direct form I or II biquad (SOS). You can also choose to implement these structures in transposed form.

  For FIR designs, you can choose direct form, direct-form transposed, direct-form symmetric, direct-form asymmetric structures, or an overlap and add structure.

- **Use a System object to implement filter** — Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Bandpass Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response
> Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode
> Select Minimum (the default) or Specify from the drop-down box. Selecting Specify enables the **Order** option so you can enter the filter order.

> If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type — This dialog only applies if you have the DSP System Toolbox software.
> Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

> - Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

> - Selecting Sample-rate converter activates both factors.

Order

> Enter the filter order. This option is enabled only if you select Specify for **Order mode**.

Decimation Factor

> Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

Interpolation Factor

> Enter the decimation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

In the figure, regions between specification values such as Fstop1 and Fpass1 represent transition regions where the filter response is not explicitly defined.

filterbuilder

Frequency constraints

Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.

- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.

- `3dB points` — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3-dB point is the frequency for the point 3 dB below the passband value.

- `3dB points and passband width` — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the passband. (IIR filters)

- `3dB points and stopband widths` — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the stopband. (IIR filters)

- `6dB points` — Define the filter response by specifying the locations of the 6-dB points. The 6-dB point is the frequency for the point 6dB below the passband value. (FIR filters)

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in hertz, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop1

Enter the frequency at the edge of the end of the first stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass1

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fpass2

Enter the frequency at the edge of the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop2

Enter the frequency at the edge of the start of the second stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

Specify as Unconstrained or Constrained bands. You must have the DSP System Toolbox software to select Constrained bands. Selecting Constrained bands enables dialogs for both stopbands and the passband: **Astop1**, **Astop2**, and **Apass**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to `Constrained bands` enables the **Wstop** and **Wpass** options under **Design options**.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in dB (decibels). This is the default setting.

- `Squared` — Specify the magnitude in squared units.

Astop1

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

> Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).
>
> Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

> Valid when the **Design method** is `equiripple` and you have the DSP System Toolbox installed. Choose one of `Linear`, `Minimum`, or `Maximum`.

Minimum order

> This option only applies when you have the DSP System Toolbox software and **Order mode** is `Minimum`.
>
> Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Wstop1
>   Weight for the first stopband.

Wpass
>   Passband weight.

Wstop2
>   Weight for the second stopband.

Max pole radius
>   Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

Init norm
>   Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

Init numerator
>   Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

Init denominator
>   Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

Filter implementation

Structure
>   For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter
>   Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Bandstop Filter Design Dialog Box — Main Pane

filterbuilder

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

> Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

> ---
> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.
> ---

Order mode

> Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option so you can enter the filter order.
>
> If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

> Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.
>
> - Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
>
> - Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.

- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.

- `3dB points` — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3 dB point is the frequency for the point 3 dB point below the passband value.

- `3dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband (IIR filters).

- `3dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband (IIR filters).

- `6dB points` — Define the filter response by specifying the locations of the 6-dB points (FIR filters). The 6-dB point is the frequency for the point 6 dB point below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

> When you design an interpolator, Fs represents the sampling frequency at the filter output rather than the filter input. This option is available only when you set **Filter type** is `interpolator`.

Fpass1

> Enter the frequency at the edge of the end of the first passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop1

> Enter the frequency at the edge of the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop2

> Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fpass2

> Enter the frequency at the edge of the start of the second passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

Specify as `Unconstrained` or `Constrained bands`. You must have the DSP System Toolbox software to select `Constrained bands`. Selecting `Constrained bands` enables dialogs for both passbands and the stopband: **Apass1**, **Apass2**, and **Astop**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to `Constrained bands` enables the **Wstop** and **Wpass** options under **Design options**.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in decibels (default).

- `Squared` — Specify the magnitude in squared units.

Apass1

Enter the filter ripple allowed in the first passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels

Apass2

Enter the filter ripple allowed in the second passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the

specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow
Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor
Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint
Valid when the **Design method** is `equiripple` and you have the DSP System Toolbox installed. Choose one of `Linear`, `Minimum`, or `Maximum`.

Minimum order

This option only applies when you have the DSP System Toolbox software and **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Wpass1

Weight for the first passband.

Wstop

Stopband weight.

Wpass2

Weight for the second passband.

Match exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` .

Max pole radius

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

Init norm

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

Init numerator

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

Init denominator

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

Filter implementation

Structure
>   For the filter specifications and design method you select, this
>   parameter lists the filter structures available to implement your
>   filter. By default, FIR filters use direct-form structure, and IIR
>   filters use direct-form II filters with SOS.

Use a System object to implement filter
>   Selecting this check box gives you the choice of using a system
>   object to implement the filter. By default the check box is turned
>   off. When the current design method or structure is not supported
>   by a system object filter, then this check box is disabled.

CIC Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your CIC filter format, such as the filter type and the differential delay.

Filter type

> Select whether your filter will be a `decimator` or an `interpolator`. Your choice determines the type of filter and the design methods and structures that are available to implement your filter. Selecting `decimator` or `interpolator` activates the **Factor** option. When you design an interpolator, you enable the **Output Fs** parameter.
>
> When you design either a decimator or interpolator, the resulting filter is a CIC filter that decimates or interpolates your input signal.

Differential Delay

> Specify the differential delay of your CIC filter as an integer value greater than or equal to 1. The default value is 1. The differential delay changes the shape, number, and location of nulls in the filter response. Increasing the differential delay increases the sharpness of the nulls and the response between the nulls. In practice, differential delay values of 1 or 2 are the most common.

Factor

> Specify the decimation or interpolation factor for your filter as an integer value greater than or equal to 1. The default value is 2.

Frequency specifications

Frequency units

> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Filter implementation

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

CIC Compensator Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the filter order mode and the filter type.

Order mode

> Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

> Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.
>
> - Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
>
> - Selecting Sample-rate converter activates both factors.
>
> When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

> Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

Decimation Factor

> Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

Interpolation Factor

> Enter the decimation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default factor value is 2.

Number of CIC sections

> Specify the number of sections in the CIC filter for which you are designing this compensator. Select the number of sections from the drop-down list or enter the number.

Differential Delay

> Specify the differential delay of your target CIC filter. The default value is 1. Most CIC filters use 1 or 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve.

Frequency specifications

Frequency units

> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select Normalized (0 1) to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The

default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select Any, Even, or Odd from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

> **Note** Generally, **Minimum order** designs are not available for IIR filters.

Match exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select passband or stopband or both from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- 1/f — Specifies that the stopband attenuation changes exponentially as the frequency increases, where f is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. filterbuilder applies that slope to the stopband.

- When you set **Stopband shape** to 1/f, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay.

filterbuilder applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Comb Filter Design Dialog Box—Main Pane

filterbuilder

Filter specifications

Parameters in this group enable you to specify the type of comb filter and the number of peaks or notches.

Comb Type

Select `Notch` or `Peak` from the drop-down list. `Notch` creates a comb filter that attenuates a set of harmonically related frequencies. `Peak` creates a comb filter that amplifies a set of harmonically related frequencies.

Order mode

Select `Order` or `Number of Peaks/Notches` from the drop-down menu.

Select `Order` to enter the desired filter order in the

dialog box. The comb filter has notches or peaks at increments of 2/`Order` in normalized frequency units.

Select `Number of Peaks` or `Number of Notches` to specify the number of peaks or notches and the `Shelving filter order`

Shelving filter order

> The Shelving filter order is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

Frequency specifications

Parameters in this group enable you to specify the frequency constraints and frequency units.

Frequency specifications

> Select Quality factor or Bandwidth.
>
> Quality factor is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the –3 dB point.
>
> Bandwidth specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the –3 dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

Frequency Units

> Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input Fs** dialog box.

Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Bandwidth gain — Specify the gain at which the bandwidth is measured. The default is –3 dB.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method
> The IIR Butterworth design is the only option for peaking or notching comb filters.

Filter implementation

Structure
> For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use a System object to implement filter
> Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off.

Differentiator Filter Design Dialog Box — Main Pane

filterbuilder

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order. Graphically, the filter specifications look similar to those shown in the following figure.

In the figure, regions between specification values such as **Fpass** (f_1) and **Fstop** (f_3) represent transition regions where the filter response is not explicitly defined.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order
Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve.

Frequency constraints
This option is only available when you specify the order of the filter design. Supported options are `Unconstrained` and `Passband edge` and `stopband edge`.

Frequency units
Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the

frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend on the value of the **Frequency constraints**. If the value of **Frequency constraints** is Unconstrained, **Magnitude constraints** must be Unconstrained. If the value of **Frequency constraints** is Passband edge and stopband edge, **Magnitude constraints** can be Unconstrained, Passband ripple, or Stopband attenuation.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Wpass

Passband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Wstop

Stopband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned

off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Fractional Delay Filter Design Dialog Box — Main Pane

Frequency specifications

Parameters in this group enable you to specify your filter format, such as the fractional delay and the filter order.

Order

>   If you choose Specify for **Order mode**, enter your filter
>   order in this field, or select the order from the drop-down
>   list. filterbuilder designs a filter with the order you specify.

Fractional delay

>   Specify a value between 0 and 1 samples for the filter fractional
>   delay. The default value is 0.5 samples.

Frequency units

>   Use this parameter to specify whether your frequency settings are
>   normalized or in absolute frequency. Select Normalized (0 1)
>   to enter frequencies in normalized form. This behavior is the
>   default. To enter frequencies in absolute values, select one of the
>   frequency units from the drop-down list—Hz, kHz, MHz, or GHz.
>   Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

>   Fs, specified in the units you selected for **Frequency units**,
>   defines the sampling frequency at the filter input. When you
>   provide an input sampling frequency, all frequencies in the
>   specifications are in the selected units as well. This parameter is
>   available when you select one of the frequency options from the
>   **Frequency units** list.

Halfband Filter Design Dialog Box — Main Pane

3-421

Filter specifications

Parameters in this group enable you to specify your filter type and order.

Impulse response

> Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

> ---
> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.
> ---

Order mode

> Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

> Select Single-rate, Decimator, or Interpolator. By default, filterbuilder specifies single-rate filters.

> When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that decimates or interpolates your input by a factor of two.

Order

> Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.

In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. For FIR halfband filters, the available design options are Equiripple and Kaiser window. For IIR halfband filters, the available design options are Butterworth, Elliptic, and IIR quasi-linear phase.

Design Options

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- 1/f — Specifies that the stopband attenuation changes exponentially as the frequency increases, where f is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. filterbuilder applies that slope to the stopband.

- When you set **Stopband shape** to 1/f, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. filterbuilder applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

filterbuilder

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Highpass Filter Design Dialog Box — Main Pane

filterbuilder

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option so you can enter the filter order.

If your **Impulse response** is IIR, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

This option is only available if you have the DSP System Toolbox software. Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a highpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

In the figure, the region between specification values Fstop and Fpass represents the transition region where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Stopband edge and passband edge` — Define the filter by specifying the frequencies for the edges for the stopband and passband.

- `Passband edge` — Define the filter by specifying the frequency for the edge of the passband.

- `Stopband edge` — Define the filter by specifying the frequency for the edges of the stopband.

- `Stopband edge and 3dB point` — Define the filter by specifying the stopband edge frequency and the 3-dB down point (IIR designs).

- `3dB point and passband edge` — Define the filter by specifying the 3-dB down point and passband edge frequency (IIR designs).

- `3dB point` — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).

- `6dB point` — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the

specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

> Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

> Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).
>
> Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a

reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

This option only applies when you have the DSP System Toolbox software and when the **Design method** is equiripple. Select one of Linear, Minimum, or Maximum.

Minimum order — This option only applies when you have the DSP System Toolbox software and the **Order mode** is Minimum.

Select Any (default), Even, or Odd. Selecting Even or Odd forces the minimum-order design to be an even or odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband when you select Passband or Stopband.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- 1/f — Specifies that the stopband attenuation changes exponentially as the frequency increases, where f is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Wpass
   Passband weight. This option only applies when **Impulse response** is `FIR` and **Order mode** is `Specify`.

Wstop
   Stopband weight. This option only applies when **Impulse response** is `FIR` and **Order mode** is `Specify`.

Filter implementation

Structure
   For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter
   Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Hilbert Filter Design Dialog Box — Main Pane

Impulse response

> Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

> ---
> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.
> ---

Order mode

> This option is only available if you have the DSP System Toolbox software. Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

> This option is only available if you have the DSP System Toolbox software. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

> - Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
> - Selecting `Sample-rate converter` activates both factors.

> When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

> Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

In the figure, the regions between 0 and f_1 and between f_2 and 1 represent the transition regions where the filter response is explicitly defined by the transition width.

Frequency units

> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

> Specify the width of the transitions at the ends of the passband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

> Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.
>
> - `Linear` — Specify the magnitude in linear units.
>
> - `dB` — Specify the magnitude in decibels (default)
>
> - `Squared` — Specify the magnitude in squared units.

Apass

> Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

> Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

> Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).
>
> Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a

reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

FIR Type

This option is only available in a minimum-order design. Specify whether to design a type 3 or a type 4 FIR filter. The filter type is defined as follows:

- Type 3 — FIR filter with even order antisymmetric coefficients

- Type 4 — FIR filter with odd order antisymmetric coefficients
Select 3 or 4 from the drop-down list.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Inverse Sinc Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Order mode

    Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Response type

    Select Lowpass or Highpass to design an inverse sinc lowpass or highpass filter.

Filter type

    Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

    When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

    Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

Decimation Factor

    Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

Interpolation Factor

    Enter the decimation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

Regions between specification values such as Fpass and Fstop represent transition regions where the filter response is not explicitly defined.

Frequency constraints

This option is only available when you specify the filter order. The following options are available:

- Passband and stopband edges — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- Passband edge — Define the filter by specifying frequencies for the edges of the passband.

- Stopband edge — Define the filter by specifying frequencies for the edges of the stopbands.

- 6dB point — The 6-dB point is the frequency for the point 6 dB point below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select Normalized (0 1) to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in decibels (default)

- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Available options are `Linear`, `Minimum`, and `Maximum`.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options;

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Sinc frequency factor

A frequency dilation factor. The sinc frequency factor, C, parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)^{\wedge}(-P)$ and for a highpass design through $H(\omega) = \text{sinc}(C(1-\omega))^{\wedge}(-P)$.

Sinc power

Negative power of passband magnitude response. The sinc power, P, parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)^{\wedge}(-P)$ and for a highpass design through $H(\omega) = \text{sinc}(C(1-\omega))^{\wedge}(-P)$.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

filterbuilder

Lowpass Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

If your **Impulse response** is IIR, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

This option is only available if you have the DSP System Toolbox. Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to the one shown in the following figure.

In the figure, regions between specification values such as F_{pass} and F_{stop} represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edge` — Define the filter by specifying the frequencies for the edge of the stopband and passband.

- `Passband edge` — Define the filter by specifying the frequency for the edge of the passband.

- `Stopband edge` — Define the filter by specifying the frequency for the edges of the stopband.

- `Passband edge and 3dB point` — Define the filter by specifying the passband edge frequency and the 3-dB down point (IIR designs).

- `3dB point and stopband edge` — Define the filter by specifying the 3-dB down point and stopband edge frequency (IIR designs).

- `3dB point` — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).

- `6dB point` — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is

available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default)

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method

Lists the design methods available for the frequency and
magnitude specifications you entered. When you change the
specifications for a filter, such as changing the impulse response,
the methods available to design filters changes as well. The
default IIR design method is usually Butterworth, and the default
FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter
coefficients to reduce the chances that the inputs or calculations
in the filter overflow and exceed the representable range of the
filter. Clearing this option removes the scaling. This parameter
applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This
section does not present all of the available options for all designs and
design methods. There are many more that you encounter as you select
different design methods and filter specifications. The following options
represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over
which the design method optimization evaluates your filter
response function. The number of equally spaced points in the grid
is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely
approximates an ideal equiripple filter but increases the time
required to design the filter. The default value of 16 represents a
reasonable trade between the accurate approximation to the ideal
filter and the time to design the filter.

Phase constraint
> This option only applies when you have the DSP System Toolbox software and when the **Design method** is `equiripple`. Select one of `Linear`, `Minimum`, or `Maximum`.

Minimum order — This option only applies when you have the DSP System Toolbox software and the **Order mode** is `Minimum`.
> Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Match Exactly
> Specifies that the resulting filter design matches either the passband or stopband when you select `Passband` or `Stopband`.

Stopband Shape
> Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

> - `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

> - `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

> - `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay
> When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

> - When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

> - When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the
  exponent n in the relation $(1/f)^n$ to define the stopband decay.
  `filterbuilder` applies the $(1/f)^n$ relation to the stopband to
  result in an exponentially decreasing stopband attenuation.

Wpass

> Passband weight. This option only applies when **Impulse
> response** is `FIR` and **Order mode** is `Specify`.

Wstop

> Stopband weight. This option only applies when **Impulse
> response** is `FIR` and **Order mode** is `Specify`.

Filter implementation

Structure

> For the filter specifications and design method you select, this
> parameter lists the filter structures available to implement your
> filter. By default, FIR filters use direct-form structure, and IIR
> filters use direct-form II filters with SOS.

Use a System object to implement filter

> Selecting this check box gives you the choice of using a system
> object to implement the filter. By default the check box is turned
> off. When the current design method or structure is not supported
> by a system object filter, then this check box is disabled.

Notch

See "Peak/Notch Filter Design Dialog Box — Main Pane" on page 3-473.

Nyquist Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Band

> Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region, bw, is calculated using the value for Band:
>
> bw = Fs/(2*Band).

Impulse response

> Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

> Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

> Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.
>
> - Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
>
> - Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

In the figure, BW is the width of the transition region and **Band** determines the location of the center of the region.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stopbands and passbands.

- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.

- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.

- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.

- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.

- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

> Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

> Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.
>
> - Linear — Specify the magnitude in linear units.
>
> - dB — Specify the magnitude in decibels (default)
>
> - Squared — Specify the magnitude in squared units.

Astop

> Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and designs the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Octave Filter Design Dialog Box — Main Pane

Filter specifications

Order

Specify filter order. Possible values are: `4, 6, 8, 10.`

Bands per octave

Specify the number of bands per octave. Possible values are: `1, 3, 6, 12, 24.`

Frequency units

Specify frequency units as `Hz` or `kHz`.

Input Fs

Specify the input sampling frequency in the frequency units specified previously.

Center Frequency

Select from the drop-down list of available center frequency values.

Algorithm

Design Method

Butterworth is the design method used for this type of filter.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Filter implementation

Structure

Specify filter structure. Choose from:

- Direct-form I SOS

- Direct-form II SOS

- Direct-form I transposed SOS

- Direct-form II transposed SOS

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default, the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Parametric Equalizer Filter Design Dialog Box — Main Pane

filterbuilder

Filter specifications

Order mode

Select Minimum to design a minimum order filter that meets the design specifications, or Specify to enter a specific filter order. The order mode also affects the possible frequency constraints, which in turn limit the gain specifications. For example, if you specify a Minimum order filter, the available frequency constraints are:

- Center frequency, bandwidth, passband width

- Center frequency, bandwidth, stopband width

If you select Specify, the available frequency constraints are:

- Center frequency, bandwidth

- Center frequency, quality factor

- Shelf type, cutoff frequency, quality factor

- Shelf type, cutoff frequency, shelf slope parameter

- Low frequency, high frequency

Order

This parameter is enabled only if the **Order mode** is set to Specify. Enter the filter order in this text box.

Frequency specifications

Depending on the filter order, the possible frequency constraints change. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

Frequency constraints

Select the specification to represent the frequency constraints. The following options are available:

- Center frequency, bandwidth, passband width (available for minimum order only)

- `Center frequency, bandwidth, stopband width` (available for minimum order only)

- `Center frequency, bandwidth` (available for a specified order only)

- `Center frequency, quality factor` (available for a specified order only)

- `Shelf type, cutoff frequency, quality factor` (available for a specified order only)

- `Shelf type, cutoff frequency, shelf slope parameter` (available for a specified order only)

- `Low frequency, high frequency` (available for a specified order only)

Frequency units

Select the frequency units from the available drop down list (`Normalized`, `Hz`, `kHz`, `MHz`, `GHz`). If `Normalized` is selected, then the **Input Fs** box is disabled for input.

Input Fs

Enter the input sampling frequency. This input box is disabled for input if `Normalized` is selected in the **Frequency units** input box.

Center frequency

Enter the center frequency in the units specified by the value in **Frequency units**.

Bandwidth

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to `db(sqrt(.5))`, or −3 dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: `Center frequency, bandwidth, passband width`, `Center frequency, bandwidth, stopband width`, or `Center frequency, bandwidth`.

Passband width

> The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, passband width`.

Stopband width

> The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, stopband width`.

Low frequency

> Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

High frequency

> Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

Gain specifications

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

Gain constraints

> Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- Reference, center frequency, bandwidth, passband

- Reference, center frequency, bandwidth, stopband

- Reference, center frequency, bandwidth,passband, stopband

- Reference, center frequency, bandwidth

Gain units

Specify the gain units either `dB` or `squared`. These units are used for all gain specifications in the dialog box.

Reference gain

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

Bandwidth gain

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a `bandwidth` parameter, or is `Low frequency, high frequency`.

Center frequency gain

Specify the center frequency in **Gain units**

Passband gain

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

Stopband gain

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constaints** are `Center frequency, bandwidth, stopband width`.

Boost/cut gain

The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the `Shelf type` parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

Algorithm

Design method

Select the design method from the drop-down list. Different IIR design methods are available depending on the filter constraints you specify.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Filter implementation

Structure

Select filter structure. The possible choices are:

- Direct-form I SOS

- Direct-form II SOS

- Direct-form I transposed SOS

- Direct-form II transposed SOS

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Peak/Notch Filter Design Dialog Box — Main Pane

Filter specifications

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

Response
: Select Peak or Notch from the drop-down box.

Order
: Enter the filter order. The order must be even.

Frequency specifications

This group of parameters allows you to specify frequency constraints and units.

Frequency Constraints
: Select the frequency constraints for filter specification. There are two choices as follows:

    - Center frequency and quality factor
    - Center frequency and bandwidth

Frequency units
: The frequency units are normalized by default. If you specify units other than normalized, filterbuilder assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz.

Input Fs
: This input box is enabled if **Frequency units** other than Normalized (0 to 1) are specified. Enter the input sampling frequency.

Center frequency
: Enter the center frequency in the units you specified in **Frequency units**.

Quality Factor

>This input box is enabled only when Center frequency and quality factor is chosen for the **Frequency Constraints**. Enter the quality factor.

Bandwidth

>This input box is enabled only when Center frequency and bandwidth is chosen for the **Frequency Constraints**. Enter the bandwidth.

Magnitude specifications

This group of parameters allows you to specify the magnitude constraints, as well as their values and units.

Magnitude Constraints

>Depending on the choice of constraints, the other input boxes are enabled or disabled. Select from four magnitude constraints available:
>
>- Unconstrained
>
>- Passband ripple
>
>- Stopband attenuation
>
>- Passband ripple and stopband attenuation

Magnitude units

>Select the magnitude units: either dB or squared.

Apass

>This input box is enabled if the magnitude constraints selected are Passband ripple or Passband ripple and stopband attenuation. Enter the passband ripple.

Astop

>This input box is enabled if the magnitude constraints selected are Stopband attenuation or Passband ripple and stopband attenuation. Enter the stopband attenuation.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method
> Lists all design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter the methods available to design filters changes as well.

Scale SOS filter coefficients to reduce chance of overflow
> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter implementation

Structure
> Lists all available filter structures for the filter specifications and design method you select. The typical options are:
>
> - Direct-form I SOS
>
> - Direct-form II SOS
>
> - Direct-form I transposed SOS
>
> - Direct-form II transposed SOS

Use a System object to implement filter
> Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Pulse-shaping Filter Design Dialog Box—Main Pane

Filter specifications

Parameters in this group enable you to specify the shape and length of the filter.

Pulse shape

Select the shape of the impulse response from the following options:

- `Raised Cosine`

- `Square Root Raised Cosine`

- `Gaussian`

Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- `Minimum`— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.

- `Specify order`—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be `Order+1` .

- `Specify symbols`—This option enables the user to specify the length of the impulse response in an alternative manner. If `Specify symbols` is chosen, the **Order** input box changes to the **Number of symbols** input box.

Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be **Number of symbols*Samples per symbol**+1. The product **Number of symbols*Samples per symbol** must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product **Number of symbols*Samples per symbol** must be an even number. The filter length will be **Number of symbols*Samples per symbol**+1.

Filter Type

This option is only available if you have the DSP System Toolbox software. Choose `Single rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. If you select `Decimator` or `Interpolator`, the decimation and interpolation factors default to the value of the **Samples per symbol**. If you select `Sample-rate converter`, the interpolation factor defaults to **Samples per symbol** and the decimation factor defaults to 3.

Frequency specifications

Parameters in this group enable you to specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

Rolloff factor

The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1)`, `Hz`, `kHz`, `MHz`, `GHz`

For a Gaussian pulse shape, the available frequency specifications are:

Bandwidth-time product

This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (BT) must be a positive real number.

Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1)`, `Hz`, `kHz`, `MHz`, `GHz`

Magnitude specifications

If the **Order mode** is specified as `Minimum`, the **Magnitude units** may be selected from:

- `dB`—Specify the magnitude in decibels (default).

- `Linear`—Specify the magnitude in linear units.

Algorithm

The only **Design method** available for FIR pulse-shaping filters is the `Window` method.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Purpose   2-norm or infinity-norm of digital filter

Syntax

```
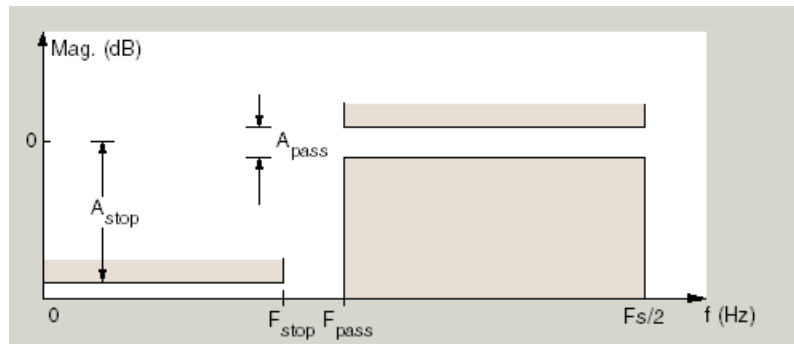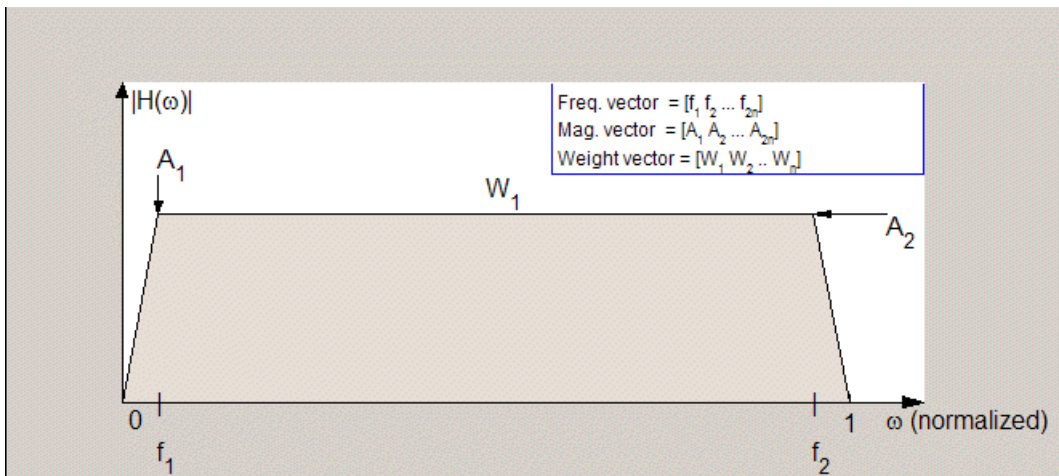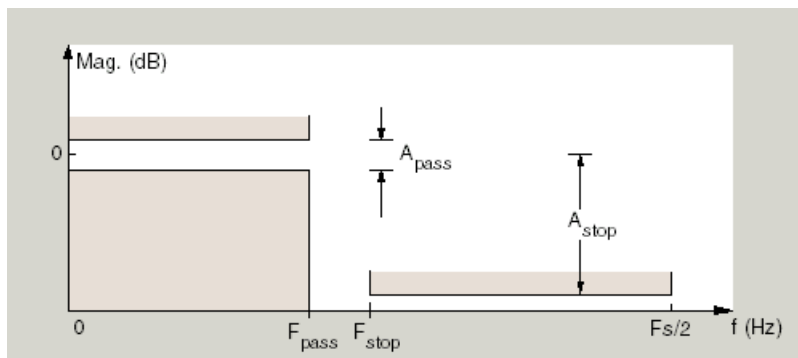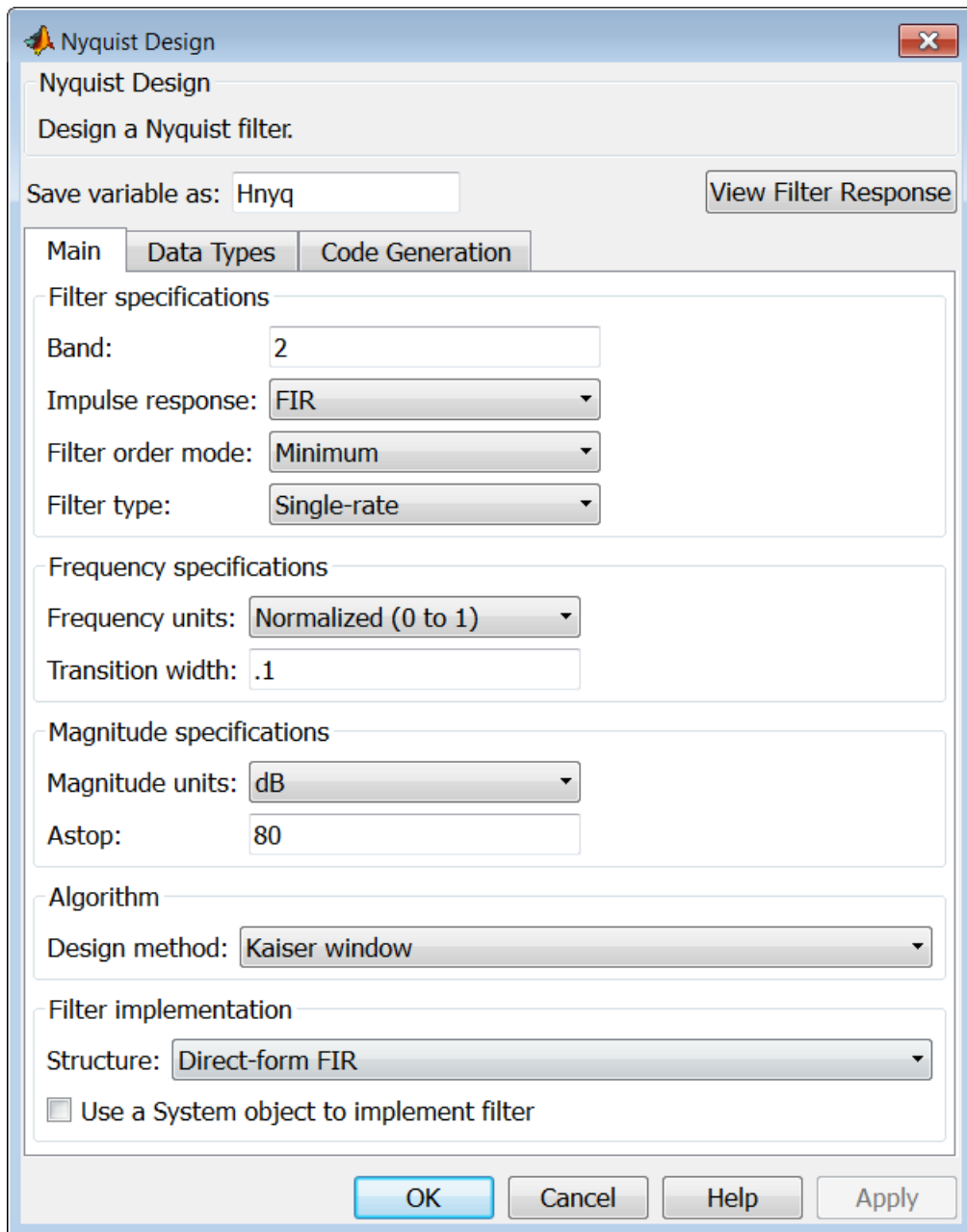filternorm(b,a)
filternorm(b,a,pnorm)
filternorm(b,a,2,tol)
```

Description   A typical use for filter norms is in digital filter scaling to reduce quantization effects. Scaling often improves the signal-to-noise ratio of the filter without resulting in data overflow. You, also, can use the 2-norm to compute the energy of the impulse response of a filter.

filternorm(b,a) computes the 2-norm of the digital filter defined by the numerator coefficients in b and denominator coefficients in a.

filternorm(b,a,pnorm) computes the 2- or infinity-norm (inf-norm) of the digital filter, where pnorm is either 2 or inf.

filternorm(b,a,2,tol) computes the 2-norm of an IIR filter with the specified tolerance, tol. The tolerance can be specified only for IIR 2-norm computations. pnorm in this case must be 2. If tol is not specified, it defaults to 1e-8.

Examples   Compute the 2-norm with a tolerance of 1e-10 of an IIR filter:

```
[b,a]=butter(5,.5);
L2=filternorm(b,a,2,1e-10)

L2 =

    0.7071
```

Compute the inf-norm of an FIR filter:

```
b=firpm(30,[.1 .9],[1 1],'Hilbert');
Linf=filternorm(b,1,inf)

Linf =
```

1.0028

Algorithms   Given a filter with frequency reponse H(e^jω), the L_p-norm for 1≤p<∞ is given by

$$||H(e^{j\omega}||_p = (\frac{1}{2\pi}\int_{-\pi}^{\pi}|H(e^{j\omega})|^p \ d\omega)^{1/p}$$

For the case p=∞, the L_∞ norm is

$$||H(e^{j\omega})||_\infty = \max_{-\pi \le \omega \le \pi}|H(e^{j\omega})|$$

For the case p=2, Parseval's theorem states that

$$||H(e^{j\omega})||_2 = (\frac{1}{2\pi}\int_{-\pi}^{\pi}|H(e^{j\omega})|^2 \ d\omega)^{1/2} = (\sum_{n}|h(n)|^2)^{1/2}$$

where h(n) is the impulse response of the filter. The energy of the impulse response is the squared L_2 norm.

References   [1] Jackson, L.B., *Digital Filters and Signal Processing, Third Edition,* Kluwer Academic Publishers, 1996, Chapter 11.

See Also   zp2sos | norm

Purpose   Zero-phase digital filtering

Syntax   y = filtfilt(b,a,x)
                   y = filtfilt(SOS,G,x)

Description   y = filtfilt(b,a,x) performs zero-phase digital filtering by
                   processing the input data, x, in both the forward and reverse directions
                   [1]. filtfilt operates along the first nonsingleton dimension of x.
                   The vector b provides the numerator coefficients of the filter and the
                   vector a provides the denominator coefficients. If you use an all-pole
                   filter, enter 1 for b. If you use an all-zero filter (FIR), enter 1 for a.
                   After filtering the data in the forward direction, filtfilt reverses the
                   filtered sequence and runs it back through the filter. The result has the
                   following characteristics:

                   • Zero-phase distortion

                   • A filter transfer function, which equals the squared magnitude of the
                     original filter transfer function

                   • A filter order that is double the order of the filter specified by b and a

                   filtfilt minimizes start-up and ending transients by matching initial
                   conditions, and you can use it for both real and complex inputs. Do not
                   use filtfilt with differentiator and Hilbert FIR filters, because the
                   operation of these filters depends heavily on their phase response.

                   Note The length of the input x must be more than three times the
                   filter order defined as max(length(b)-1,length(a)-1).

                   ---

                   y = filtfilt(SOS,G,x) zero-phase filters the data x using the
                   second-order section (biquad) filter represented by the matrix SOS and
                   scale values G. The matrix SOS is an L-by-6 matrix containing the L
                   second-order sections. The matrix SOS must be of the form:

$$\begin{pmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ b_{0L} & b_{1L} & b_{2L} & a_{0L} & a_{1L} & a_{2L} \end{pmatrix}$$

where each row are the coefficients of a biquad filter. The vector of filter scale values, G, must have a length between 1 and L+1.

Note When implementing zero-phase filtering using a second-order section filter, the length of the input x must be more than 6 samples.

Examples   Zero-phase filtering helps preserve features in the filtered time waveform exactly where those features occur in the unfiltered waveform. To illustrate the use of filtfilt for zero-phase filtering, consider an electrocardiogram waveform as an example.

```
plot(ecg(500)); %plot ECG signal
axis([0 500 -1.25 1.25]);
```

The QRS complex is an important feature in the ECG waveform beginning around time point 160 in this example.

The following example corrupts the ECG waveform with additive noise, constructs a lowpass FIR equiripple filter, and filters the noisy waveform using both zero-phase and conventional filtering. Because the filter is an all-zero (FIR) filter, the denominator equals 1. Seed the random number generator for reproducible results.

```
rng default;
x=ecg(500)'+0.25*randn(500,1); %noisy waveform
h=fdesign.lowpass('Fp,Fst,Ap,Ast',0.15,0.2,1,60);
d=design(h,'equiripple'); %Lowpass FIR filter
y=filtfilt(d.Numerator,1,x); %zero-phase filtering
```

```
y1=filter(d.Numerator,1,x); %conventional filtering
subplot(211);
plot([y y1]);
title('Filtered Waveforms');
legend('Zero-phase Filtering','Conventional Filtering');
subplot(212);
plot(ecg(500));
title('Original Waveform');
```

Zero-phase filtering reduces noise in the signal and preserves the QRS complex at the same time it occurs in the original signal. Conventional filtering reduces noise in the signal, but delays the QRS complex.

Repeat the above using a Butterworth second-order section filter:

```
h=fdesign.lowpass('N,F3dB',12,0.15);
d1 = design(h,'butter');
y = filtfilt(d1.sosMatrix,d1.ScaleValues,x);
plot(x,'b-.'); hold on;
plot(y,'r','linewidth',3);
legend('Noisy ECG','Zero-phase Filtering','location','NorthEast');
```

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing,* Prentice-Hall, 1989, pp. 284–285.

[2] Mitra, S.K., *Digital Signal Processing, 2nd ed.,* McGraw-Hill, 2001, Sections 4.4.2 and 8.2.5.

[3] Gustafsson, F., Determining the initial states in forward-backward filtering, *IEEE Transactions on Signal Processing*, April 1996, Volume 44, Issue 4, pp. 988–992.

See Also   fftfilt | filter | filter2

filtic

Purpose   Initial conditions for transposed direct-form II filter implementation

Syntax   `z = filtic(b,a,y,x)`
                    `z = filtic(b,a,y)`

Description   `z = filtic(b,a,y,x)` finds the initial conditions, `z`, for the delays in the *transposed direct-form II* filter implementation given past outputs `y` and inputs `x`. The vectors `b` and `a` represent the numerator and denominator coefficients, respectively, of the filter's transfer function.

The vectors `x` and `y` contain the most recent input or output first, and oldest input or output last.

$$x = [x(-1), x(-2), x(-3), \ldots, x(-n)]$$
$$y = [y(-1), y(-2), y(-3), \ldots, y(-m)]$$

where `n` is `length(b)-1` (the numerator order) and `m` is `length(a)-1` (the denominator order). If `length(x)` is less than `n`, `filtic` pads it with zeros to length `n`; if `length(y)` is less than `m`, `filtic` pads it with zeros to length `m`. Elements of `x` beyond `x(n-1)` and elements of `y` beyond `y(m-1)` are unnecessary so `filtic` ignores them.

Output `z` is a column vector of length equal to the larger of *n* and *m*. `z` describes the state of the delays given past inputs `x` and past outputs `y`.

`z = filtic(b,a,y)` assumes that the input `x` is 0 in the past.

The transposed direct-form II structure is shown in the following illustration.

n-1 is the filter order.

filtic works for both real and complex inputs.

Algorithms   filtic performs a reverse difference equation to obtain the delay states z.

Diagnostics   If any of the input arguments y, x, b, or a is not a vector (that is, if any argument is a scalar or array), filtic gives the following error message:

Requires vector inputs.

References   [1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 296, 301-302.

See Also   filter | filtfilt

filtstates

Purpose   Filter states

Syntax   Hs = filtstates.*structure*(input1,...)

Description Hs = filtstates.*structure*(input1,...) returns a filter states
object Hs, which contains the filter states.

You can extract a filtstates object from the states property of an
object with

```
Hd = dfilt.df1
Hs = Hd.states
```

or, for an mfilt object in the DSP System Toolbox product, with

```
Hm = mfilt.cicdecim
Hs = Hm.states
```

Structures

Structures for filtstates specify the type of filter structure. Available
types of structures for filtstates are shown below.

filtstates.structure	Description
filtstates.dfiir	filtstates for IIR direct-form I filters (dfilt.df1, dfilt.df1t, dfilt.df1sos, and dfilt.df1tsos)
filtstates.cic	filtstates for cascaded integrator comb filters. (Available only with DSP System Toolbox and Fixed-Point Toolbox products.)

Refer to the particular filtstates.*structure* reference page or use
the syntax help filtstates.*structure* at the MATLAB prompt for
more information.

See Also   filtstates.dfiir | dfilt | dfilt.df1 | dfilt.df1t | dfilt.df1sos
                    | dfilt.df1tsos

filtstates.dfiir

Purpose	IIR direct-form filter states
Syntax	`Hs = filtstates.dfiir(numstates,denstates)`

Description `Hs = filtstates.dfiir(numstates,denstates)` returns an IIR direct-form filter states object `Hs` with two properties — `Numerator` and `Denominator`, which contain the filter states. These two properties are column vectors with each column representing a separate channel of filter states. The number of states is always one less than the number of filter numerator or denominator coefficients.

You can extract a filtstates object from the states property of an IIR direct-form I object with

```
Hd = dfilt.df1
Hs = Hd.states
```

Methods

You can use the following methods on a `filtstates.dfiir` object.

Method	Description
double	Converts a `filtstates` object to a double-precision vector containing the values of the numerator and denominator states. The numerator states are listed first in this vector, followed by the denominator states.
single	Converts a `filtstates` object to a single-precision vector containing the values of the numerator and denominator states. (This method is used with the DSP System Toolbox product.)

Examples This example demonstrates the interaction of `filtstates` with a `dfilt.df1` object.

```
[b,a] = butter(4,0.5);    % Design butterworth filter
```

```
Hd = dfilt.df1(b,a);      % Create dfilt object
Hs = Hd.states            % Extract filter states object
                          %  from dfilt states property
Hs.Numerator = [1,1,1,1]  % Modify numerator states
Hd.states = Hs            % Set modified states back to
                          %  original object

Dbl = double(Hs)          % Create double vector from
                          %  states
```

See Also   filtstates | dfilt | dfilt.df1 | dfilt.df1t | dfilt.df1sos |
                 dfilt.df1tsos

findpeaks

Purpose	Find local maxima
Syntax	*pks* = findpeaks(*data*) [*pks*,*locs*] = findpeaks(*data*) [...] = findpeaks(*data*,'Name',*value*)

Description   *pks* = findpeaks(*data*) returns local maxima or peaks, *pks*, in the input *data*. *data* requires a row or column vector with real-valued elements with a minimum length of three. findpeaks compares each element of *data* to its neighboring values. If an element of *data* is larger than both of its neighbors or equals Inf, the element is a local peak. If there are no local maxima, *pks* is an empty vector.

[*pks*,*locs*] = findpeaks(*data*) returns the indices of the local peaks.

[...] = findpeaks(*data*,'Name',*value*) accepts one or more comma-separated name/value pairs. Specify 'Name' inside single quotes. 'Name' is not case sensitive.

Input Arguments

Name-Value Pair Arguments

MINPEAKHEIGHT

Minimum peak height

Specify the minimum peak height as a real-valued scalar. findpeaks only returns peaks that exceed the MINPEAKHEIGHT. Sometimes, specifying a minimum peak height reduces processing time.

> **Default:** -Inf

MINPEAKDISTANCE

Minimum peak separation

Specify the minimum peak distance, or minimum separation between peaks as a positive integer. You can use the 'MINPEAKDISTANCE' option to specify that the algorithm ignore small peaks that occur in the neighborhood of a larger peak. When you specify a value for

'MINPEAKDISTANCE, the algorithm initially identifies all the peaks in the input data and sorts those peaks in descending order. Beginning with the largest peak, the algorithm ignores all identified peaks not separated by more than the value of 'MINPEAKDISTANCE'.

Default: 1

THRESHOLD

Minimum height difference

Specify the threshold height difference between a peak and its neighboring values as a positive real number. findpeaks only returns peaks that exceed their neighbors by at least the value of 'THRESHOLD'.

Default: 0

NPEAKS

Number of peaks

Specify the maximum number of peaks to return as a positive integer. findpeaks operates from the first element of the input data and terminates when the number of peaks reaches the value of 'NPEAKS'.

Default: Returns all peaks that meet the specified criteria

SORTSTR

Peak sorting

Specify whether to return the peaks in order. Possible values for 'SORTSTR' are 'ascend', 'descend', and 'none'. 'ascend' returns peaks in ascending, or increasing order from the smallest to largest value. The option 'descend' specifies peaks in descending order, from the largest to smallest value. Using 'none' returns peaks in the order they occur in the input data. Specify the value string in lowercase only.

Default: 'none'

findpeaks

Examples   Find peaks in a vector:

```
data = [2 12 4 6 9 4 3 1 19 7];
pks=findpeaks(data);
% returns the 1x3 vector [12 9 19];
```

Find peaks separated by more than three elements and return their locations:

```
data = [2 12 4 6 9 4 3 1 19 7];
[pks,locs]=findpeaks(data,'minpeakdistance',3);
% returns pks=[12 19]
% locs=[2 9]
```

Create a signal with 11 peaks. Find each peak and mark the peaks in a plot:

```
x = linspace(0,1,1024);
Pos = [0.1 0.13 0.15 0.23 0.25 0.40 ...
 0.44 0.65  0.76 0.78 0.81];
Hgt = [ 4   5    3    4   5  4.2 2.1 4.3  3.1 5.1 4.2];
Wdt = [.005 .005 .006 .01 .01 .03 .01 .01  .005 .008 .005];
PeakSig = zeros(size(x));
  for n =1:length(Pos)
    PeakSig = ...
PeakSig + Hgt(n)./( 1 + abs((x - Pos(n))./Wdt(n))).^4;
  end
% find peaks with defaults
[pks,locs] = findpeaks(PeakSig);
plot(x,PeakSig); hold on;
% offset values of peak heights for plotting
plot(x(locs),pks+0.05,'k^','markerfacecolor',[1 0 0]);
```

fir1

Purpose   Window-based finite impulse response filter design

Syntax   b = fir1(n,Wn)
                 b = fir1(n,Wn,'*ftype*')
                 b = fir1(n,Wn,window)
                 b = fir1(n,Wn,'*ftype*',window)
                 b = fir1(...,'*normalization*')

Description fir1 implements the classical method of windowed linear-phase
                 FIR digital filter design [1]. It designs filters in standard lowpass,
                 highpass, bandpass, and bandstop configurations. By default the filter
                 is normalized so that the magnitude response of the filter at the center
                 frequency of the passband is 0 dB.

Note Use fir2 for windowed filters with arbitrary frequency response.

---

b = fir1(n,Wn) returns row vector b containing the n+1 coefficients
of an order n lowpass FIR filter. This is a Hamming-window based,
linear-phase filter with normalized cutoff frequency Wn. The output
filter coefficients, b, are ordered in descending powers of *z*.

$$B(z) = b(1) + b(2)z^{-1} + \ldots + b(n+1)z^{-N}$$

Wn is a number between 0 and 1, where 1 corresponds to the Nyquist
frequency.

If Wn is a two-element vector, Wn = [w1 w2], fir1 returns a bandpass
filter with passband w1 < ω < w2.

If Wn is a multi-element vector, Wn = [w1 w2 w3 w4 w5 ... wn], fir1
returns an order n multiband filter with bands 0 < ω < w1, w1 < ω < w2,
..., wn < ω < 1.

By default, the filter is scaled so that the center of the first passband
has a magnitude of exactly 1 after windowing.

b = fir1(n,Wn,'*ftype*') specifies a filter type, where '*ftype*' is:

- 'high' for a highpass filter with cutoff frequency Wn.

- 'stop' for a bandstop filter, if Wn = [w1 w2]. The stopband frequency range is specified by this interval.

- 'DC-1' to make the first band of a multiband filter a passband.

- 'DC-0' to make the first band of a multiband filter a stopband.

fir1 always uses an even filter order for the highpass and bandstop configurations. This is because for odd orders, the frequency response at the Nyquist frequency is 0, which is inappropriate for highpass and bandstop filters. If you specify an odd-valued n, fir1 increments it by 1.

b = fir1(n,Wn,window) uses the window specified in column vector window for the design. The vector window must be n+1 elements long. If no window is specified, fir1 uses a Hamming window (see hamming) of length n+1.

b = fir1(n,Wn,'*ftype*',window) accepts both '*ftype*' and window parameters.

b = fir1(...,'*normalization*') specifies whether or not the filter magnitude is normalized. The string '*normalization*' can be the following:

- 'scale' (default): Normalize the filter so that the magnitude response of the filter at the center frequency of the passband is 0 dB.

- 'noscale': Do not normalize the filter.

The group delay of the FIR filter designed by fir1 is n/2.

Algorithms   fir1 uses the window method of FIR filter design [1]. If $w(n)$ denotes a window, where $1 \le n \le N$, and the impulse response of the ideal filter is $h(n)$, then the windowed digital filter coefficients are given by

$$b(n) = w(n)h(n) \quad 1 \le n \le N$$

Examples   **Example 1**

Design a 48th-order FIR bandpass filter with passband $0.35 \leq \omega \leq 0.65$:

```
b = fir1(48,[0.35 0.65]);
freqz(b,1,512)
```


Example 2

The chirp.mat file contains a signal, y, that has most of its power above fs/4, or half the Nyquist frequency. Design a 34th-order FIR highpass filter to attenuate the components of the signal below fs/4. Use a cutoff frequency of 0.48 and a Chebyshev window with 30 dB of ripple:

```
load chirp          % Load y and fs.
b = fir1(34,0.48,'high',chebwin(35,30));
freqz(b,1,512)
```

References   [1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979. Algorithm 5.2.

See Also   cfirpm | filter | fir2 | fircls | fircls1 | firls | freqz | kaiserord | firpm | window

fir2

Purpose Frequency sampling-based finite impulse response filter design

Syntax
```
b = fir2(n,f,m)
b = fir2(n,f,m,window)
b = fir2(n,f,m,npt)
b = fir2(n,f,m,npt,window)
b = fir2(n,f,m,npt,lap)
b = fir2(n,f,m,npt,lap,window)
```

Description fir2 designs frequency sampling-based digital FIR filters with arbitrarily shaped frequency response.

Note Use fir1 for windows-based standard lowpass, bandpass, highpass, and bandstop configurations.

---

b = fir2(n,f,m) returns row vector b containing the n+1 coefficients of an order n FIR filter. The frequency-magnitude characteristics of this filter match those given by vectors f and m:

- f is a vector of frequency points in the range from 0 to 1, where 1 corresponds to the Nyquist frequency. The first point of f must be 0 and the last point 1. The frequency points must be in increasing order.

- m is a vector containing the desired magnitude response at the points specified in f.

- f and m must be the same length.

- Duplicate frequency points are allowed, corresponding to steps in the frequency response.

Use plot(f,m) to view the filter shape.

The output filter coefficients, b, are ordered in descending powers of z.

$$B(z) = b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}$$

fir2 always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued n, fir2 increments it by 1.

b = fir2(n,f,m,window) uses the window specified in the column vector window. The vector window must be n+1 elements long. If no window is specified, fir2 uses a Hamming window (see hamming) of length n+1.

b = fir2(n,f,m,npt) or

b = fir2(n,f,m,npt,window) specifies the number of points, npt, for the grid onto which fir2 linearly interpolates the frequency response with or without the window specification. npt must be greater than 1/2 the filter order (npt>n/2). If desired, you can interpolate f and m before passing them to fir2.

b = fir2(n,f,m,npt,lap) and

b = fir2(n,f,m,npt,lap,window) specify the size of the region, lap, that fir2 inserts around duplicate frequency points, with or without a window specification.

See "Algorithms" on page 3-506 for more on npt and lap.

Examples

Design a 30th-order lowpass filter and overplot the desired frequency response with the actual frequency response:

```
f = [0 0.6 0.6 1]; m = [1 1 0 0];
b = fir2(30,f,m);
[h,w] = freqz(b,1,128);
plot(f,m,w/pi,abs(h))
legend('Ideal','fir2 Designed')
title('Comparison of Frequency Response Magnitudes')
```

fir2

Comparison of Frequency Response Magnitudes

Algorithms   The desired frequency response is linearly interpolated onto a dense, evenly spaced grid of length npt. npt is 512 by default. If two successive values of f are the same, a region of lap points is set up around this frequency to provide a smooth but steep transition in the requested frequency response. By default, lap is 25. The filter coefficients are obtained by applying an inverse fast Fourier transform to the grid and multiplying by a window; by default, this is a Hamming window.

References   [1] Mitra, S.K., *Digital Signal Processing A Computer Based Approach, First Edition*, McGraw-Hill, New York, 1998, pp. 462-468.

[2] Jackson, L.B., *Digital Filters and Signal Processing, Third Edition*, Kluwer Academic Publishers, Boston, 1996, pp. 301-307.

See Also   butter | cheby1 | cheby2 | ellip | fir1 | maxflat | firpm | yulewalk

Purpose   Constrained least square, FIR multiband filter design

Syntax   b = fircls(n,f,amp,up,lo)
                 fircls(n,f,amp,up,lo,'*design_flag*')

Description   b = fircls(n,f,amp,up,lo) generates a length n+1 linear phase FIR
                 filter b. The frequency-magnitude characteristics of this filter match
                 those given by vectors f and amp:

- f is a vector of transition frequencies in the range from 0 to 1, where
  1 corresponds to the Nyquist frequency. The first point of f must be 0
  and the last point 1. The frequency points must be in increasing
  order.

- amp is a vector describing the piecewise constant desired amplitude of
  the frequency response. The length of amp is equal to the number of
  bands in the response and should be equal to length(f)-1.

- up and lo are vectors with the same length as amp. They define the
  upper and lower bounds for the frequency response in each band.

fircls always uses an even filter order for configurations with a
passband at the Nyquist frequency (that is, highpass and bandstop
filters). This is because for odd orders, the frequency response at the
Nyquist frequency is necessarily 0. If you specify an odd-valued n,
fircls increments it by 1.

fircls(n,f,amp,up,lo,'*design_flag*') enables you to monitor the
filter design, where '*design_flag*' can be

- 'trace', for a textual display of the design error at each iteration
  step.

- 'plots', for a collection of plots showing the filter's full-band
  magnitude response and a zoomed view of the magnitude response
  in each sub-band. All plots are updated at each iteration step. The
  O's on the plot are the estimated extremals of the new iteration and
  the X's are the estimated extremals of the previous iteration, where
  the extremals are the peaks (maximum and minimum) of the filter

ripples. Only ripples that have a corresponding O and X are made equal.

- `'both'`, for both the textual display and plots.

Examples Design an order 150 lowpass filter:

```
n=150;
f=[0 0.4 1];
a=[1 0];
up=[1.02 0.01];
lo =[0.98 -0.01];
b = fircls(n,f,a,up,lo,'both');   % Display plots of bands
%  The Bound Violations indicate iterations as
%  the design converges.
fvtool(b)                         % Display magnitude plot
```

fircls

Note Normally, the lower value in the stopband will be specified as negative. By setting `lo` equal to `0` in the stopbands, a nonnegative frequency response amplitude can be obtained. Such filters can be spectrally factored to obtain minimum phase filters.

Algorithms   `fircls` uses an iterative least-squares algorithm to obtain an equiripple response. The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

References   [1] Selesnick, I.W., M. Lang, and C.S. Burrus, "Constrained Least Square Design of FIR Filters without Specified Transition

Bands," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing, Vol. 2* (May 1995), pp. 1260-1263.

[2] Selesnick, I.W., M. Lang, and C.S. Burrus. "Constrained Least Square Design of FIR Filters without Specified Transition Bands." *IEEE Transactions on Signal Processing, Vol. 44*, No. 8 (August 1996).

See Also   fircls1 | firls | firpm

fircls1

Purpose   Constrained least square, lowpass and highpass, linear phase, FIR
                   filter design

Syntax   b = fircls1(n,wo,dp,ds)
                   b = fircls1(n,wo,dp,ds,'high')
                   b = fircls1(n,wo,dp,ds,wt)
                   b = fircls1(n,wo,dp,ds,wt,'high')
                   b = fircls1(n,wo,dp,ds,wp,ws,k)
                   b = fircls1(n,wo,dp,ds,wp,ws,k,'high')
                   b = fircls1(n,wo,dp,ds,...,'*design_flag*')

Description   b = fircls1(n,wo,dp,ds) generates a lowpass FIR filter b, where n+1
                   is the filter length, wo is the normalized cutoff frequency in the range
                   between 0 and 1 (where 1 corresponds to the Nyquist frequency), dp is
                   the maximum passband deviation from 1 (passband ripple), and ds is
                   the maximum stopband deviation from 0 (stopband ripple).

                   b = fircls1(n,wo,dp,ds,'high') generates a highpass FIR filter b.
                   fircls1 always uses an even filter order for the highpass configuration.
                   This is because for odd orders, the frequency response at the Nyquist
                   frequency is necessarily 0. If you specify an odd-valued n, fircls1
                   increments it by 1.

                   b = fircls1(n,wo,dp,ds,wt) and

                   b = fircls1(n,wo,dp,ds,wt,'high') specifies a frequency wt above
                   which (for wt > wo) or below which (for wt < wo) the filter is guaranteed
                   to meet the given band criterion. This will help you design a filter that
                   meets a passband or stopband edge requirement. There are four cases:

                   • Lowpass:

                       ▪ 0 < wt < wo < 1: the amplitude of the filter is within dp of 1 over
                         the frequency range $0 < \omega < wt$.

                       ▪ 0 < wo < wt < 1: the amplitude of the filter is within ds of 0 over
                         the frequency range $wt < \omega < 1$.

                   • Highpass:

- $0 <$ wt $<$ wo < 1: the amplitude of the filter is within ds of 0 over the frequency range $0 < \omega <$ wt.

- $0 <$ wo $<$ wt < 1: the amplitude of the filter is within dp of 1 over the frequency range wt $< \omega < 1$.

b = fircls1(n,wo,dp,ds,wp,ws,k) generates a lowpass FIR filter b with a weighted function, where n+1 is the filter length, wo is the normalized cutoff frequency, dp is the maximum passband deviation from 1 (passband ripple), and ds is the maximum stopband deviation from 0 (stopband ripple). wp is the passband edge of the L2 weight function and ws is the stopband edge of the L2 weight function, where wp $<$ wo $<$ ws. k is the ratio (passband L2 error)/(stopband L2 error)

$$k = \frac{\int_0^{w_p} |A(\omega) - D(\omega)|^2 \, d\omega}{\int_{w_z}^{\pi} |A(\omega) - D(\omega)|^2 \, d\omega}$$

b = fircls1(n,wo,dp,ds,wp,ws,k,'high') generates a highpass FIR filter b with a weighted function, where ws $<$ wo $<$ wp.

b = fircls1(n,wo,dp,ds,...,'*design_flag*') enables you to monitor the filter design, where '*design_flag*' can be

- 'trace', for a textual display of the design table used in the design

- 'plots', for plots of the filter's magnitude, group delay, and zeros and poles. All plots are updated at each iteration step. The O's on the plot are the estimated extremals of the new iteration and the X's are the estimated extremals of the previous iteration, where the extremals are the peaks (maximum and minimum) of the filter ripples. Only ripples that have a corresponding O and X are made equal.

- 'both', for both the textual display and plots

fircls1

Examples   Design an order 55 lowpass filter with a cutoff frequency at 0.3:

```
n = 55;     wo = 0.3;
dp = 0.02;  ds = 0.008;
b = fircls1(n,wo,dp,ds,'both');    % Display plots of bands
    Bound Violation = 0.0870385343920
    Bound Violation = 0.0149343456540
    Bound Violation = 0.0056513587932
    Bound Violation = 0.0001056264205
    Bound Violation = 0.0000967624352
    Bound Violation = 0.0000000226538
    Bound Violation = 0.0000000000038
%  The above Bound Violations indicate iterations as
%  the design converges.
fvtool(b)            % Display magnitude plot
```

fircls1

Algorithms   fircls1 uses an iterative least-squares algorithm to obtain an equiripple response. The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

References   [1] Selesnick, I.W., M. Lang, and C.S. Burrus, "Constrained Least Square Design of FIR Filters without Specified Transition Bands," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing, Vol. 2* (May 1995), pp.1260-1263.

[2] Selesnick, I.W., M. Lang, and C.S. Burrus, "Constrained Least Square Design of FIR Filters without Specified Transition Bands," *IEEE Transactions on Signal Processing, Vol. 44*, No. 8 (August 1996).

See Also   fircls | firls | firpm

Purpose   Least square linear-phase FIR filter design

Syntax   b = firls(n,f,a)
                  b = firls(n,f,a,w)
                  b = firls(n,f,a,'*ftype*')
                  b = firls(n,f,a,w,'*ftype*')

Description   firls designs a linear-phase FIR filter that minimizes the weighted,
                  integrated squared error between an ideal piecewise linear function and
                  the magnitude response of the filter over a set of desired frequency
                  bands.

                  b = firls(n,f,a) returns row vector b containing the n+1 coefficients
                  of the order n FIR filter whose frequency-amplitude characteristics
                  approximately match those given by vectors f and a. The output filter
                  coefficients, or "taps," in b obey the symmetry relation.

                  $$b(k) = b(n + 2 - k), \qquad k = 1, ..., n + 1$$

                  These are type I (n odd) and type II (n even) linear-phase filters. Vectors
                  f and a specify the frequency-amplitude characteristics of the filter:

                  • f is a vector of pairs of frequency points, specified in the range
                    between 0 and 1, where 1 corresponds to the Nyquist frequency.
                    The frequencies must be in increasing order. Duplicate frequency
                    points are allowed and, in fact, can be used to design a filter exactly
                    the same as those returned by the fir1 and fir2 functions with a
                    rectangular (rectwin) window.

                  • a is a vector containing the desired amplitude at the points specified
                    in f.

                    The desired amplitude function at frequencies between pairs of
                    points ($f(k)$, $f(k+1)$) for k odd is the line segment connecting the points
                    ($f(k)$, $a(k)$) and ($f(k+1)$, $a(k+1)$).

                    The desired amplitude function at frequencies between pairs of
                    points ($f(k)$, $f(k+1)$) for k even is unspecified. These are transition or
                    "don't care" regions.

- `f` and `a` are the same length. This length must be an even number.

`firls` always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `firls` increments it by 1.

The figure below illustrates the relationship between the `f` and `a` vectors in defining a desired amplitude response.


```
f = [0 .3 .4 .6 .7 .9]
a = [0  1  0  0 .5 .5]
```

`b = firls(n,f,a,w)` uses the weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

`b = firls(n,f,a,'`*ftype*`')` and

`b = firls(n,f,a,w,'`*ftype*`')` specify a filter type, where `'`*ftype*`'` is:

- `'hilbert'` for linear-phase filters with odd symmetry (type III and type IV). The output coefficients in `b` obey the relation

    b(k) = –b(n + 2 – k), k = 1, ... , n + 1.

    This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

- `'differentiator'` for type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of $(1/f)^2$ so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, the filters minimize the relative integrated squared error (the integral of the square of the ratio of the error to the desired amplitude).

Examples   **Example 1**

Design an order 255 lowpass filter with transition band:

```
b = firls(255,[0 0.25 0.3 1],[1 1 0 0]);
```

Example 2

Design a 31 coefficient differentiator:

```
b = firls(30,[0 0.9],[0 0.9*pi],'differentiator');
```

An ideal differentiator has the response

$$D(w) = jw$$

The amplitudes include a pi multiplier because the frequencies are normalized by pi.

Example 3

Design a 24th-order anti-symmetric filter with piecewise linear passbands and plot the desired and actual frequency response:

```
F = [0 0.3  0.4 0.6  0.7 0.9];
A = [0  1    0   0   0.5 0.5];
b = firls(24,F,A,'hilbert');
for i=1:2:6,
   plot([F(i) F(i+1)],[A(i) A(i+1)],'--'), hold on
end
[H,f] = freqz(b,1,512,2);
plot(f,abs(H)), grid on, hold off
```

```
legend('Ideal','firls Design')
```


Algorithms   Reference [1] describes the theoretical approach behind `firls`. The function solves a system of linear equations involving an inner product matrix of size roughly `n/2` using the MATLAB \ operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for n even and odd respectively, while the `'hilbert'` and `'differentiator'` flags produce type III (n even) and IV (n odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type I	Even	$b(k) = b(n + 2 - k), k = 1, ..., n + 1$	No restriction	No restriction
Type II	Even	$b(k) = b(n + 2 - k), k = 1, ..., n + 1$	No restriction	$H(1) = 0$

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type III	Odd	$b(k) = -b(n+2-k), k = 1,...,n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd	$b(k) = -b(n+2-k), k = 1,...,n+1$	$H(0) = 0$	No restriction

Diagnostics One of the following diagnostic messages is displayed when an incorrect argument is used:

```
F must be even length.
F and A must be equal lengths.
Requires symmetry to be 'hilbert' or 'differentiator'.
Requires one weight per band.
Frequencies in F must be nondecreasing.
Frequencies in F must be in range [0,1].
```

A more serious warning message is

```
Warning: Matrix is close to singular or badly scaled.
```

This tends to happen when the product of the filter length and transition width grows large. In this case, the filter coefficients b might not represent the desired filter. You can check the filter by looking at its frequency response.

References [1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 54-83.

[2] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 256-266.

See Also fir1 | fir2 | firrcos | firpm

firpm

Purpose	Parks-McClellan optimal FIR filter design

Syntax
```
b = firpm(n,f,a)
b = firpm(n,f,a,w)
b = firpm(n,f,a, 'ftype')
b = firpm(n,f,a,w, 'ftype')
b = firpm(...,{lgrid})
[b,err] = firpm(...)
[b,err,res] = firpm(...)
b = firpm(n,f,@fresp,w)
b = firpm(n,f,@fresp,w,'ftype')
```

Description firpm designs a linear-phase FIR filter using the Parks-McClellan algorithm [1]. The Parks-McClellan algorithm uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with an optimal fit between the desired and actual frequency responses. The filters are optimal in the sense that the maximum error between the desired frequency response and the actual frequency response is minimized. Filters designed this way exhibit an equiripple behavior in their frequency responses and are sometimes called *equiripple* filters. firpm exhibits discontinuities at the head and tail of its impulse response due to this equiripple nature.

b = firpm(n,f,a) returns row vector b containing the n+1 coefficients of the order n FIR filter whose frequency-amplitude characteristics match those given by vectors f and a.

The output filter coefficients (taps) in b obey the symmetry relation:

$$b(k) = b(n+2-k), \quad k = 1,...,n+1$$

Vectors f and a specify the frequency-magnitude characteristics of the filter:

• f is a vector of pairs of normalized frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order.

- a is a vector containing the desired amplitudes at the points specified in f.

  The desired amplitude at frequencies between pairs of points (*f(k)*, *f(k+1)*) for *k* odd is the line segment connecting the points (*f(k)*, *a(k)*) and (*f(k+1)*, *a(k+1)*).

  The desired amplitude at frequencies between pairs of points (*f(k)*, *f(k+1)*) for *k* even is unspecified. The areas between such points are transition or "don't care" regions.

- f and a must be the same length. The length must be an even number.

The relationship between the f and a vectors in defining a desired frequency response is shown in the illustration below.

firpm always uses an even filter order for configurations with even symmetry and a nonzero passband at the Nyquist frequency. This is because for impulse responses exhibiting even symmetry and odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued n, firpm increments it by 1.

firpm

`b = firpm(n,f,a,w)` uses the weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

Note `b = firpm(n,f,a,w)` is a synonym for `b = firpm(n,f,{@firpmfrf,a},w)`, where, `@firpmfrf` is the predefined frequency response function handle for `firpm`. If desired, you can write your own response function. Use `help private/firpmfrf` for information.

`b = firpm(n,f,a, 'ftype')` and

`b = firpm(n,f,a,w, 'ftype')` specify a filter type, where '*ftype*' is

- `'hilbert'`, for linear-phase filters with odd symmetry (type III and type IV)

  The output coefficients in `b` obey the relation $b(k) = -b(n+2-k)$, $k = 1, ..., n+1$. This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

  For example,

  ```
  h = firpm(30,[0.1 0.9],[1 1],'hilbert');
  ```

  designs an approximate FIR Hilbert transformer of length 31.

- `'differentiator'`, for type III and type IV filters, using a special weighting technique

  For nonzero amplitude bands, it weights the error by a factor of $1/f$ so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, these filters minimize the maximum relative error (the maximum of the ratio of the error to the desired amplitude).

`b = firpm(...,{lgrid})` uses the integer `lgrid` to control the density of the frequency grid, which has roughly `(lgrid*n)/(2*bw)` frequency

points, where bw is the fraction of the total frequency band interval [0,1] covered by f. Increasing lgrid often results in filters that more exactly match an equiripple filter, but that take longer to compute. The default value of 16 is the minimum value that should be specified for lgrid. Note that the {lgrid} argument must be a 1-by-1 cell array.

[b,err] = firpm(...) returns the maximum ripple height in err.

[b,err,res] = firpm(...) returns a structure res with the following fields.

res.fgrid	Frequency grid vector used for the filter design optimization
res.des	Desired frequency response for each point in res.fgrid
res.wt	Weighting for each point in opt.fgrid
res.H	Actual frequency response for each point in res.fgrid
res.error	Error at each point in res.fgrid (res.des-res.H)
res.iextr	Vector of indices into res.fgrid for extremal frequencies
res.fextr	Vector of extremal frequencies

You can also use firpm to write a function that defines the desired frequency response. The predefined frequency response function handle for firpm is @firpmfrf, which designs a linear-phase FIR filter.

b = firpm(n,f,@*fresp*,w) returns row vector b containing the n+1 coefficients of the order n FIR filter whose frequency-amplitude characteristics best approximate the response returned by function handle @*fresp*. The function is called from within firpm with the following syntax.

[dh,dw] = fresp(n,f,gf,w)

The arguments are similar to those for firpm:

firpm

- n is the filter order.

- f is the vector of normalized frequency band edges that appear monotonically between 0 and 1, where 1 is the Nyquist frequency.

- gf is a vector of grid points that have been linearly interpolated over each specified frequency band by firpm. gf determines the frequency grid at which the response function must be evaluated, and contains the same data returned by cfirpm in the fgrid field of the opt structure.

- w is a vector of real, positive weights, one per band, used during optimization. w is optional in the call to firpm; if not specified, it is set to unity weighting before being passed to fresp.

- dh and dw are the desired complex frequency response and band weight vectors, respectively, evaluated at each frequency in grid gf.

b = firpm(n,f,@*fresp*,w,'*ftype*') designs antisymmetric (odd) filters, where '*ftype*' is either 'd' for a differentiator or 'h' for a Hilbert transformer. If you do not specify an *ftype*, a call is made to *fresp* to determine the default symmetry property sym. This call is made using the syntax.

```
sym = fresp('defaults',{n,f,[],w,p1,p2,...})
```

The arguments n, f, w, etc., may be used as necessary in determining an appropriate value for sym, which firpm expects to be either 'even' or 'odd'. If *fresp* does not support this calling syntax, firpm defaults to even symmetry.

Examples   Graph the desired and actual frequency responses of a 17th-order Parks-McClellan bandpass filter:

```
f = [0 0.3 0.4 0.6 0.7 1]; a = [0 0 1 1 0 0];
b = firpm(17,f,a);
[h,w] = freqz(b,1,512);
plot(f,a,w/pi,abs(h))
legend('Ideal','firpm Design')
```

Algorithms   firpm is a MEX-file version of the original Fortran code from [1], altered to design arbitrarily long filters with arbitrarily many linear bands.

firpm designs type I, II, III, and IV linear-phase filters. Type I and type II are the defaults for n even and n odd, respectively, while type III (n even) and type IV (n odd) are obtained with the 'hilbert' and 'differentiator' flags. The different types of filters have different symmetries and certain constraints on their frequency responses (see [5] for more details).

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type I	Even	even: $$b(k) = b(n+2-k), k = 1,...,n+1$$	No restriction	No restriction
Type II	Odd	even: $$b(k) = b(n+2-k), k = 1,...,n+1$$	No restriction	*H(1)=0*firpm increments the filter order by 1 if you attempt

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
				to construct a type II filter with a nonzero passband at the Nyquist frequency.
Type III	Even	odd: $$b(k) = -b(n + 2 - k), k = 1,...,n + 1$$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd	odd: $$b(k) = -b(n + 2 - k), k = 1,...,n + 1$$	$H(0) = 0$	No restriction

Diagnostics   If you get the following warning message,

```
-- Failure to Converge --
Probable cause is machine rounding error.
```

it is possible that the filter design may still be correct. Verify the design by checking its frequency response.

References   [1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, Algorithm 5.1.

[2] *Selected Papers in Digital Signal Processing*, II, IEEE Press, New York, 1979.

[3] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, New York:, 1987, p. 83.

[4] Rabiner, L.R., J.H. McClellan, and T.W. Parks, "FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximations," Proc. IEEE 63 (1975).

[5] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 256-266.

See Also   butter | cheby1 | cheby2 | cfirpm | ellip | fir1 | fir2 | fircls | fircls1 | firls | firrcos | firpmord | function_handle | yulewalk

firpmord

Purpose   Parks-McClellan optimal FIR filter order estimation

Syntax   `[n,fo,ao,w] = firpmord(f,a,dev)`
`[n,fo,ao,w] = firpmord(f,a,dev,fs)`
`c = firpmord(f,a,dev,fs,'cell')`

Description   `[n,fo,ao,w] = firpmord(f,a,dev)` finds the approximate order, normalized frequency band edges, frequency band amplitudes, and weights that meet input specifications `f`, `a`, and `dev`.

- `f` is a vector of frequency band edges (between 0 and $F_s/2$, where F_s is the sampling frequency), and `a` is a vector specifying the desired amplitude on the bands defined by `f`. The length of `f` is two less than twice the length of `a`. The desired function is piecewise constant.

- `dev` is a vector the same size as `a` that specifies the maximum allowable deviation or ripples between the frequency response and the desired amplitude of the output filter for each band.

Use `firpm` with the resulting order `n`, frequency vector `fo`, amplitude response vector `ao`, and weights `w` to design the filter `b` which approximately meets the specifications given by `firpmord` input parameters `f`, `a`, and `dev`.

`b = firpm(n,fo,ao,w)`

`[n,fo,ao,w] = firpmord(f,a,dev,fs)` specifies a sampling frequency `fs`. `fs` defaults to 2 Hz, implying a Nyquist frequency of 1 Hz. You can therefore specify band edges scaled to a particular application's sampling frequency.

In some cases `firpmord` underestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1` or `n+2`.

`c = firpmord(f,a,dev,fs,'cell')` generates a cell-array whose elements are the parameters to `firpm`.

Examples   **Example 1**

Design a minimum-order lowpass filter with a 500 Hz passband cutoff
frequency and 600 Hz stopband cutoff frequency, with a sampling
frequency of 2000 Hz, at least 40 dB attenuation in the stopband, and
less than 3 dB of ripple in the passband:

```
rp = 3;            % Passband ripple
rs = 40;           % Stopband ripple
fs = 2000;         % Sampling frequency
f = [500 600];     % Cutoff frequencies
a = [1 0];         % Desired amplitudes
% Compute deviations
dev = [(10^(rp/20)-1)/(10^(rp/20)+1)  10^(-rs/20)];
[n,fo,ao,w] = firpmord(f,a,dev,fs);
b = firpm(n,fo,ao,w);
freqz(b,1,1024,fs);
title('Lowpass Filter Designed to Specifications');
```

Note that the filter falls slightly short of meeting the stopband attenuation and passband ripple specifications. Using n+1 in the call to firpm instead of n achieves the desired amplitude characteristics.

Example 2

Design a lowpass filter with a 1500 Hz passband cutoff frequency and 2000 Hz stopband cutoff frequency, with a sampling frequency of 8000 Hz, a maximum stopband amplitude of 0.1, and a maximum passband error (ripple) of 0.01:

```
[n,fo,ao,w] = firpmord([1500 2000],[1 0],[0.01 0.1],8000 );
b = firpm(n,fo,ao,w);
```

This is equivalent to

```
c = firpmord( [1500 2000],[1 0],[0.01 0.1],8000,'cell');
b = firpm(c{:});
```

Note In some cases, firpmord underestimates or overestimates the order n. If the filter does not meet the specifications, try a higher order such as n+1 or n+2.

Results are inaccurate if the cutoff frequencies are near 0 or the Nyquist frequency.

Algorithms   firpmord uses the algorithm suggested in [1]. This method is inaccurate for band edges close to either 0 or the Nyquist frequency (fs/2).

References   [1] Rabiner, L.R., and O. Herrmann, "The Predictability of Certain Optimum Finite Impulse Response Digital Filters," *IEEE Trans. on Circuit Theory, Vol. CT-20*, No. 4 (July 1973), pp. 401-408.

[2] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing. Englewood Cliffs*, NJ: Prentice-Hall, 1975, pp. 156-157.

See Also   buttord | cheb1ord | cheb2ord | ellipord | kaiserord | firpm

firrcos

Purpose   Raised cosine FIR filter design

Syntax
```
b = firrcos(n,Fc,df)
b = firrcos(n,Fc,df,Fs)
b = firrcos(n,Fc,df,Fs,'bandwidth')
b = firrcos(n,Fc,df,Fs,'type')
b = firrcos(...,'type',delay)
b = firrcos(...,'type',delay,window)
b = firrcos(n,Fc,r,Fs,'rolloff')
b = firrcos(...,'rolloff','type')
[b,a] = firrcos(...)
```

Description   `b = firrcos(n,Fc,df)` uses a default sampling frequency of `Fs = 2`.

`b = firrcos(n,Fc,df,Fs)` or, equivalently,

`b = firrcos(n,Fc,df,Fs,'bandwidth')` returns an order `n` lowpass linear-phase FIR filter with a raised cosine transition band. The order `n` must be even. The filter has cutoff frequency `Fc`, transition bandwidth `df`, and sampling frequency `Fs`, all in hertz. `df` must be small enough so that `Fc ± df/2` is between `0` and `Fs/2`. The coefficients in `b` are normalized so that the nominal passband gain is always equal to 1.

`b = firrcos(n,Fc,df,Fs,'type')` designs either a normal raised cosine filter or a square root raised cosine filter according to how you specify the string `'type'`. Specify `'type'` as:

- `'normal'`, for a regular raised cosine filter. This is the default, and is also in effect when the `'type'` argument is left empty, `[]`, or unspecified.

- `'sqrt'`, for a square root raised cosine filter.

`b = firrcos(...,'type',delay)` specifies an integer delay in the range `[0,n+1]`. The default is `n/2` for all `n`.

`b = firrcos(...,'type',delay,window)` applies a length `n+1` window to the designed filter to reduce the ripple in the frequency response. `window` must be a length `n+1` column vector. If no window is specified, a

rectangular (`rectwin`) window is used. Care must be exercised when using a window with a delay other than the default.

`b = firrcos(n,Fc,r,Fs,'rolloff')` interprets the third argument, `r`, as the rolloff factor instead of the transition bandwidth, `df`. `r` must be in the range `[0,1]`.

`b = firrcos(...,'rolloff','`*type*`')` specifies the type of raised cosine filter.

`[b,a] = firrcos(...)` always returns `a = 1`.

Examples   Design an order 20 raised cosine FIR filter with cutoff frequency 0.25 of the Nyquist frequency and a transition bandwidth of 0.25:

```
h = firrcos(20,0.25,0.25);
freqz(h,1)
```


See Also   `fir1` | `fir2` | `firls` | `firpm`

flattopwin

Purpose   Flat Top weighted window

Syntax   w = flattopwin(L)
                 w = flattopwin(L,sflag)

Description Flat Top windows have very low passband ripple (< 0.01 dB) and
                 are used primarily for calibration purposes. Their bandwidth is
                 approximately 2.5 times wider than a Hann window.

                 w = flattopwin(L) returns the L-point symmetric flat top window
                 in column vector w.

                 w = flattopwin(L,sflag) returns the L-point symmetric flat
                 top window using sflag window sampling, where sflag is either
                 'symmetric' or 'periodic'. The 'periodic' flag is useful for
                 DFT/FFT purposes, such as in spectral analysis. The DFT/FFT contains
                 an implicit periodic extension and the periodic flag enables a signal
                 windowed with a periodic window to have perfect periodic extension.
                 When 'periodic' is specified, flattopwin computes a length L+1
                 window and returns the first L points. When using windows for filter
                 design, the 'symmetric' flag should be used.

Algorithms Flat top windows are summations of cosines. The coefficients of a flat
                 top window are computed from the following equation

$$w(n) = a_0 - a_1\cos\left(\frac{2\pi n}{N}\right) + a_2\cos\left(\frac{4\pi n}{N}\right) - a_3\cos\left(\frac{6\pi n}{N}\right) + a_4\cos\left(\frac{8\pi n}{N}\right)$$

where $0 \le n \le N$ and $w(n) = 0$ elsewhere and the window length is L = N +1. The coefficient values are

Coefficient	Value
a_0	0.21557895
a_1	0.41663158

Coefficient	Value
a_2	0.277263158
a_3	0.083578947
a_4	0.006947368

Examples   Create a 64-point, symmetric Flat Top window and view the window using WVTool:

```
w = flattopwin(64);
wvtool(w);
```


References   [1] D'Antona, Gabriele. and A. Ferrero, *Digital Signal Processing for Measurement Systems*, New York: Springer Media, Inc., 2006, pp. 70–72.

[2] Gade, Svend and H. Herlufsen, "Use of Weighting Functions in DFT/FFT Analysis (Part I)," Brüel & Kjær, *Windows to FFT Analysis (Part I) Technical Review, No. 3*, 1987, pp. 19-21.

See Also   `blackman | hamming | hann`

Purpose   Frequency response of analog filters

Syntax   h = freqs(b,a,w)
                [h,w] = freqs(b,a,n)
                freqs

Description freqs returns the complex frequency response $H(j\omega)$ (Laplace transform) of an analog filter

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{a(1)s^m + a(2)s^{m-1} + \cdots + a(m+1)}$$

given the numerator and denominator coefficients in vectors b and a.

h = freqs(b,a,w) returns the complex frequency response of the analog filter specified by coefficient vectors b and a. freqs evaluates the frequency response along the imaginary axis in the complex plane at the angular frequencies in rad/sec specified in real vector w, where w is a vector containing more than one frequency.

[h,w] = freqs(b,a,n) uses n frequency points to compute the frequency response h, where n is a real, scalar value. The frequency vector w is auto-generated and has length n. If you omit n as an input, 200 frequency points are used. If you do not need the generated frequency vector returned, you can use the form h = freqs(b,a,n) to return only the frequency response h.

freqs with no output arguments plots the magnitude and phase response versus frequency in the current figure window.

freqs works only for real input systems and positive frequencies.

Examples   Find and graph the frequency response of the transfer function given by:

$$H(s) = \frac{0.2s^2 + 0.3s + 1}{s^2 + 0.4s + 1}$$

a = [1 0.4 1];

```
b = [0.2 0.3 1];
w = logspace(-1,1);
freqs(b,a,w)
```


You can also create the plot with

```
h = freqs(b,a,w);
mag = abs(h);
phase = angle(h);
subplot(2,1,1), loglog(w,mag)
subplot(2,1,2), semilogx(w,phase)
```

To convert to hertz, degrees, and decibels, use

```
f = w/(2*pi);
mag = 20*log10(mag);
phase = phase*180/pi;
```

Algorithms   freqs evaluates the polynomials at each frequency point, then divides the numerator response by the denominator response:

```
s = i*w;
h = polyval(b,s)./polyval(a,s);
```

See Also   abs | angle | freqz | invfreqs | logspace | polyval

freqsamp

Purpose   Real or complex frequency-sampled FIR filter from specification object

Syntax   hd = design(d,'freqsamp')
                     hd = design(...,'filterstructure',structure)
                     hd = design(...,'window',window)

Description   hd = design(d,'freqsamp') designs a frequency-sampled filter
                     specified by the filter specifications object d.

                     hd = design(...,'filterstructure',structure) returns a filter with
                     the filter structure you specify by the structure input argument.
                     structure is dffir by default and can be any one of the following filter
                     structures.

Structure String	Description of Resulting Filter Structure
dffir	Direct-form FIR filter
dffirt	Transposed direct-form FIR filter
dfsymfir	Symmetrical direct-form FIR filter
dfasymfir	Asymmetrical direct-form FIR filter
fftfir	Fast Fourier transform FIR filter

hd = design(...,'window',window) designs filters using the window
specified by the string in window. Provide the input argument window as

- A string for the window type. For example, use 'bartlett' , or
  'hamming'. See window for the full list of windows available in the
  Signal Processing Toolbox User's Guide.

- A function handle that references the window function. When the
  window function requires more than one input, use a cell array to
  hold the required arguments. The first example shows a cell array
  input argument.

- The window vector itself.

Examples   These examples design FIR filters that have arbitrary magnitude responses. In the first filter, the response has three distinct sections and the resulting filter is real.

The second example creates a complex filter.

```
b1 = 0:0.01:0.18;
b2 = [.2 .38 .4 .55 .562 .585 .6 .78];
b3 = [0.79:0.01:1];
a1 = .5+sin(2*pi*7.5*b1)/4;    % Sinusoidal response section.
a2 = [.5 2.3 1 1 -.2 -.2 1 1]; % Piecewise linear response section.
a3 = .2+18*(1-b3).^2;          % Quadratic response section.
f = [b1 b2 b3];
a = [a1 a2 a3];
n = 300;
d = fdesign.arbmag('n,f,a',n,f,a); % First specifications object.
hd = design(d,'freqsamp','window',{@kaiser,.5}); % Filter.
fvtool(hd)
```

The plot from FVTool shows the response for hd.

Now design the arbitrary-magnitude complex FIR filter. Recall that vector f contains frequency locations and vector a contains the desired filter response values at the locations specified in f.

```
f = [-1 -.93443 -.86885 -.80328 -.7377 -.67213 -.60656 -.54098 ...
-.47541,-.40984 -.34426 -.27869 -.21311 -.14754 -.081967 ...
-.016393 .04918 .11475,.18033 .2459 .31148 .37705 .44262 ...
 .5082 .57377 .63934 .70492 .77049,.83607 .90164 1];
a = [.0095848 .021972 .047249 .099869 .23119 .57569 .94032 ...
.98084 .99707,.99565 .9958 .99899 .99402 .99978 .99995 .99733 ...
.99731 .96979 .94936,.8196 .28502 .065469 .0044517 .018164 ...
.023305 .02397 .023141 .021341,.019364 .017379 .016061];
```

```
n = 48;
d = fdesign.arbmag('n,f,a',n,f,a); % Second spec. object.
hdc = design(d,'freqsamp','window','rectwin'); % Filter.
fvtool(hdc)
```

FVTool shows you the response for hdc from -1 to 1 in normalized frequency because the filter's transfer function is not symmetric around 0. Since the Fourier transform of the filter does not exhibit conjugate symmetry, design(d,...) returns a complex—valued filter for hdc .

See Also   design | designmethods | fdesign.arbmag

freqz

Purpose	Frequency response of digital filter

Syntax
```
[h,w] = freqz(b,a,n)
h = freqz(b,a,w)
[h,w] = freqz(b,a,n,'whole')
[h,f] = freqz(b,a,n,fs)
h = freqz(b,a,f,fs)
[h,f] = freqz(b,a,n,'whole',fs)
freqz(b,a,...)
freqz(Hd)
```

Description   `[h,w] = freqz(b,a,n)` returns the frequency response vector h and the corresponding angular frequency vector w for the digital filter whose transfer function is determined by the (real or complex) numerator and denominator polynomials represented in the vectors b and a, respectively. The vectors h and w are both of length n. n must be a positive integer greater than or equal to two. The angular frequency vector w has values ranging from 0 to π radians per sample. If you do not specify the integer n, or you specify it as the empty vector [ ], the frequency response is calculated using the default value of 512 samples.

`h = freqz(b,a,w)` returns the frequency response vector h calculated at the frequencies (in radians per sample) supplied by the vector w. w must be a vector and have a minimum length of two.

`[h,w] = freqz(b,a,n,'whole')` uses n sample points around the entire unit circle to calculate the frequency response. The frequency vector w has length n and has values ranging from 0 to 2π radians per sample.

`[h,f] = freqz(b,a,n,fs)` returns the frequency response vector h and the corresponding frequency vector f for the digital filter whose transfer function is determined by the (real or complex) numerator and denominator polynomials represented in the vectors b and a, respectively. The vectors h and f are both of length n. For this syntax, the frequency response is calculated using the sampling frequency specified by the scalar fs (in hertz). The frequency vector f is calculated in units of hertz (Hz). The frequency vector f has values ranging from 0 to fs/2 Hz.

`h = freqz(b,a,f,fs)` returns the frequency response vector `h` calculated at the frequencies (in Hz) supplied in the vector `f`. The vector `f` can be any length.

`[h,f] = freqz(b,a,n,'whole',fs)` uses `n` points around the entire unit circle to calculate the frequency response. The frequency vector `f` has length `n` and has values ranging from `0` to `fs` Hz.

`freqz(b,a,...)` plots the magnitude and unwrapped phase of the frequency response of the filter. The plot is displayed in the current figure window.

`freqz(Hd)` plots the magnitude and unwrapped phase of the frequency response of the filter. The plot is displayed in `fvtool`. The input `Hd` is a `dfilt` filter object or an array of `dfilt` filter objects.

Tips
It is best to choose a power of 2 for the third input argument `n`, because `freqz` uses an FFT algorithm to calculate the frequency response. See the reference description of `fft` for more information.

Examples
Plot the magnitude and phase response of an FIR filter:

```
b = fir1(80,0.5,kaiser(81,8));
freqz(b,1);
```

The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

```
d=fdesign.lowpass('N,Fc',80,0.5);
Hd=design(d);
freqz(Hd);
```

Algorithms   The frequency response [1] of a digital filter can be interpreted as the transfer function evaluated at $z = e^{j\omega}$. You can always write a rational transfer function in the following form.

$$H(e^{j\omega}) = \frac{\displaystyle\sum_{k=0}^{M-1} b(k)e^{-j\omega k}}{\displaystyle\sum_{l=0}^{N-1} a(l)e^{-j\omega l}}.$$

freqz determines the transfer function from the (real or complex) numerator and denominator polynomials you specify, and returns the complex frequency response $H(e^{j\omega})$ of a digital filter. The frequency

response is evaluated at sample points determined by the syntax that you use.

`freqz` generally uses an FFT algorithm to compute the frequency response whenever you don't supply a vector of frequencies as an input argument. It computes the frequency response as the ratio of the transformed numerator and denominator coefficients, padded with zeros to the desired length.

When you do supply a vector of frequencies as an input argument, then `freqz` evaluates the polynomials at each frequency point using Horner's method of nested polynomial evaluation [1], dividing the numerator response by the denominator response.

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing, Prentice-Hall*, 1989, pp. 203-205.

See Also   `abs` | `angle` | `fft` | `filter` | `freqs` | `impz` | `invfreqs` | `logspace`

Purpose   Open Filter Visualization Tool

Syntax   fvtool(b,a)
                   fvtool(b_1,a_1,b_2,a_2,...b_n,a_n)
                   fvtool(Hd)
                   fvtool(Hd_1,Hd_2,...)
                   h = fvtool(...)

Description   fvtool(b,a) opens FVTool and computes the magnitude response of
                   the digital filter defined with numerator, b and denominator, a. Using
                   FVTool you can display the phase response, group delay, impulse
                   response, step response, pole-zero plot, and coefficients of the filter. You
                   can export the displayed response to a file with **File > Export**.

                   fvtool(b_1,a_1,b_2,a_2,...b_n,a_n) opens FVTool and computes the
                   magnitude responses of multiple filters defined with numerators, b_1...b_n
                   and denominators, a_1...a_n.

                   fvtool(Hd) opens FVTool and computes the magnitude responses of
                   the filters in the dfilt objects, Hd.

                   fvtool(Hd_1,Hd_2,...) opens FVTool and computes the magnitude
                   responses of the filters in the dfilt objects Hd1, Hd2, etc.

                   If you have the DSP System Toolbox product installed, you can also use
                   fvtool(H) and fvtool(H_1,H_2,...) to analyze:

                   • Quantized filter objects (dfilt with arithmetic set to 'single' or
                     'fixed')

                   • Multirate filter (mfilt) objects

                   • Any of the following filter System objects: dsp.FIRFilter,
                     dsp.IIRFilter, dsp.AllpoleFilter, dsp.BiquadFilter,
                     dsp.FIRInterpolator, dsp.CICInterpolator, dsp.FIRDecimator,
                     dsp.CICDecimator, or dsp.FIRRateConverter

                   • Adaptive filter (adaptfilt) objects

                   When the input filter is a dfilt or mfilt object, FVTool
                   performs fixed-point analysis if the arithmetic property of the

filter objects is set to 'fixed'. However, for filter System objects, `fvtool(H,'Arithmetic',ARITH,...)` analyzes H, based on the arithmetic specified in the `ARITH` input.

`ARITH` can be one of `'double'`, `'single'`, or `'fixed'`. The `'Arithmetic'` input is only relevant for the analysis of filter System objects. The arithmetic setting `ARITH`, applies to all the filter System objects that you input to fvtool. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

System Object State	Coefficient Data Type	Rule
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

If you do not specify the arithmetic for non-CIC structures, and the System object is in an unlocked state, the function uses double-precision arithmetic. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Analysis methods `noisepsd` and `freqrespest` have behavior restrictions in `fvtool`. To see the rules, click the links to these methods.

`h = fvtool(...)` returns a figure handle `h`. You can use this handle to interact with FVTool from the command line. See "Controlling FVTool from the MATLAB Command Line" on page 3-560.

FVTool has two toolbars.

- An extended version of the MATLAB plot editing toolbar. The following table shows the toolbar icons specific to FVTool.

Icon	Description
⤢	Restore default view. This view displays buffer regions around the data and shows only significant data. To see the response using standard MATLAB plotting, which shows all data values, use **View > Full View**.
🔲	Toggle legend

Icon	Description
▦	Toggle grid
✎	Link to FDATool (appears only if FVTool was started from FDATool)
⊞	Toggle Add mode/Replace mode (appears only if FVTool was launched from FDATool)

- Analysis toolbar with the following icons

📈	Magnitude response of the current filter. See `freqz` and `zerophase` for more information.
	To see the zero-phase response, right-click the *y*-axis label of the Magnitude plot and select **Zero-phase** from the context menu.
📉	Phase response of the current filter. See `phasez` for more information.
📊	Superimposes the magnitude response and the phase response of the current filter. See `freqz` for more information.
$\frac{d\phi}{d\omega}$	Shows the group delay of the current filter. Group delay is the average delay of the filter as a function of frequency. See `grpdelay` for more information.
$\frac{-\phi}{\omega}$	Shows the phase delay of the current filter. Phase delay is the time delay the filter imposes on each component of the input signal. See `phasedelay` for more information.
↑	Impulse response of the current filter. The impulse response is the response of the filter to a impulse input. See `impz` for more information.

⌐	Step response of the current filter. The step response is the response of the filter to a step input. See stepz for more information.
⊕	Pole-zero plot, which shows the pole and zero locations of the current filter on the *z*-plane. See zplane for more information.
[b,a]	Filter coefficients of the current filter, which depend on the filter structure (e.g., direct-form, lattice, etc.) in a text box. For SOS filters, each section is displayed as a separate filter.
❶	Detailed filter information.

Linking to FDATool

In fdatool, selecting **View > Filter Visualization Tool** or the **Full View Analysis** toolbar button ▢ when an analysis is displayed starts FVTool for the current filter. You can synchronize FDATool and FVTool with the **FDAToolLink** toolbar button ⌖. Any changes made to the filter in FDATool are immediately reflected in FVTool.

Two FDATool link modes are provided via the **Set Link Mode** toolbar button:

- Replace ▦ — removes the filter currently displayed in FVTool and inserts the new filter.

- Add ▦ — retains the filter currently displayed in FVTool and adds the new filter to the display.

Modifying the Axes

You can change the *x*- or *y*-axis units by right-clicking the mouse on the axis label or by right-clicking on the plot and selecting **Analysis Parameters**. Available options for the axes units are as follows.

Plot	X-Axis Units	Y-Axis Units
Magnitude	Normalized Frequency Linear Frequency	Magnitude Magnitude(dB) Magnitude squared Zero-Phase
Phase	Normalized Frequency Linear Frequency	Phase Continuous Phase Degrees Radians
Magnitude and Phase	Normalized Frequency Linear Frequency	(*y*-axis on left side) Magnitude Magnitude(dB) Magnitude squared Zero-Phase (*y*-axis on right side) Phase Continuous Phase Degrees Radians
Group Delay	Normalized Frequency Linear Frequency	Samples Time
Phase Delay	Normalized Frequency Linear Frequency	Degrees Radians
Impulse Response	Samples Time	Amplitude

Plot	X-Axis Units	Y-Axis Units
Step Response	Samples Time	Amplitude
Pole-Zero	Real Part	Imaginary Part

Modifying the Plot

You can use any of the plot editing toolbar buttons to change the properties of your plot.

Analysis Parameters are parameters that apply to the displayed analyses. To display them, right-click in the plot area and select **Analysis Parameters** from the menu. (Note that you can access the menu only if the **Edit Plot** button is inactive.) The following analysis parameters are displayed. (If more than one response is displayed, parameters applicable to each plot are displayed.) Not all of these analysis fields are displayed for all types of plots:

- **Normalized Frequency** — if checked, frequency is normalized between 0 and 1, or if not checked, frequency is in Hz

- **Frequency Scale** — y-axis scale (Linear or Log)

- **Frequency Range** — range of the frequency axis or Specify freq. vector

- **Number of Points** — number of samples used to compute the response

- **Frequency Vector** — vector to use for plotting, if Specify freq. vector is selected in **Frequency Range**.

- **Magnitude Display** — y-axis units (Magnitude, Magnitude (dB), Magnitude squared, or Zero-Phase)

- **Phase Units** — y-axis units (Degrees or Radians)

- **Phase Display** — type of phase plot (Phase or Continuous Phase)

- **Group Delay Units** — y-axis units (Samples or Time)

- **Specify Length** — length type of impulse or step response (`Default` or `Specified`)

- **Length** — number of points to use for the impulse or step response

In addition to the above analysis parameters, you can change the plot type for Impulse and Step Response plots by right-clicking and selecting **Line with Marker**, **Stem** or **Line** from the context menu. You can change the *x*-axis units by right-clicking the *x*-axis label and selecting `Samples` or `Time.`

To save the displayed parameters as the default values to use when FDATool or FVTool is opened, click **Save as default**.

To restore the default values, click **Restore original defaults**.

Data tips display information about a particular point in the plot. See "Data Cursor — Displaying Data Values Interactively" in the MATLAB documentation for information on data tips.

If you have the DSP System Toolbox software, FVTool displays a specification mask along with your designed filter on a magnitude plot.

Note To use **View > Passband zoom**, your filter must have been designed using `fdesign` or FDATool. Passband zoom is not provided for cascaded integrator-comb (CIC) filters because CICs do not have conventional passbands.

Overlaying a Response

You can overlay a second response on the plot by selecting **Analysis > Overlay Analysis** and selecting an available response. A second *y*-axis is added to the right side of the response plot. The Analysis Parameters dialog box shows parameters for the *x*-axis and both *y*-axes. See "Example 2" on page 3-564 for a sample Analysis Parameters dialog box.

Controlling FVTool from the MATLAB Command Line

After you obtain the handle for FVTool, you can control some aspects of FVTool from the command line. In addition to the standard Handle Graphics® properties (see Handle Graphics in the MATLAB documentation), FVTool has the following properties:

- `'Filters'` — returns a cell array of the filters in FVTool.

- `'Analysis'` — displays the specified type of analysis plot. The following table lists the analyses and corresponding analysis strings. Note that the only analyses that use filter internals are magnitude response estimate and round-off noise power, which are available only with the DSP System Toolbox product.

Analysis Type	Analysis String
Magnitude plot	`'magnitude'`
Phase plot	`'phase'`
Magnitude and phase plot	`` `freq' ``
Group delay plot	`'grpdelay'`
Phase delay plot	`` `phasedelay' ``
Impulse response plot	`'impulse'`
Step response plot	`'step'`
Pole-zero plot	`'polezero'`
Filter coefficients	`'coefficients'`
Filter information	`'info'`

Analysis Type	Analysis String
Magnitude response estimate (available only with the DSP System Toolbox product, see freqrespest for more information)	`'magestimate'`
Round-off noise power (available only with the DSP System Toolbox product, see noisepsd for more information)	`'noisepower'`

- `'Grid'` — controls whether the grid is `'on'` or `'off'`

- `'Legend'` — controls whether the legend is `'on'` or `'off'`

- `'Fs'` — controls the sampling frequency of filters in FVTool. The sampling frequency vector must be of the same length as the number of filters or a scalar value. If it is a vector, each value is applied to its corresponding filter. If it is a scalar, the same value is applied to all filters.

- SosViewSettings — (This option is available only if you have the DSP System Toolbox product.) For second-order sections filters, this controls how the filter is displayed. The SOSViewSettings property contains an object so you must use this syntax to set it: set(h.SOSViewSettings,'View',*viewtype*), where *viewtype* is one of the following:

  - `'Complete'` — Displays the complete response of the overall filter

  - `'Individual'` — Displays the response of each section separately

  - 'Cumulative' — Displays the response for each section accumulated with each prior section. If your filter has three sections, the first plot shows section one, the second plot shows the accumulation of sections one and two, and the third plot show the accumulation of all three sections.

You can also define whether to use `SecondaryScaling`, which determines where the sections should be split. The secondary scaling points are the scaling locations between the recursive and the nonrecursive parts of the section. The default value is `false`, which does not use secondary scaling. To turn on secondary scaling, use this syntax: `set(h.SOSViewSettings,'View','Cumulative',true)`

- `'UserDefined'` — Allows you to define which sections to display and the order in which to display them. Enter a cell array where each section is represented by its index. If you enter one index, only that section is plotted. If you enter a range of indices, the combined response of that range of sections is plotted. For example, if your filter has four sections, entering `{1:4}` plots the combined response for all four sections, and entering `{1,2,3,4}` plots the response for each section individually.

Note You can change other properties of FVTool from the command line using the `set` function. Use `get(h)` to view property tags and current property settings.

You can use the following methods with the FVTool handle.

`addfilter(h,filtobj)` adds a new filter to FVTool. The new filter, `filtobj`, must be a `dfilt` filter object. You can specify the sampling frequency of the new filter with `addfilter(h,filtobj,'Fs',10)`.

`setfilter(h,filtobj)` replaces the filter in FVTool with the filter specified in `filtobj`. You can set the sampling frequency as described above.

`deletefilter(h, index)` deletes the filter at the FVTool cell array `index` location.

`legend(h,str1,str2,...)` creates a legend in FVTool by associating `str1` with filter 1, `str2` with filter 2, etc. See `legend` in the MATLAB documentation for information.

For more information on using FVTool from the command line, see the example fvtooldemo.

Examples   **Example 1**

Display the magnitude response of an elliptic filter, starting FVTool from the command line:

```
[b,a]=ellip(6,3,50,300/500);
fvtool(b,a);
```

Example 2

Display and analyze multiple FIR filters, starting FVTool from the command line. Then, display the associated analysis parameters for the magnitude:

```
b1 = firpm(20,[0 0.4 0.5 1],[1 1 0 0]);
b2 = firpm(40,[0 0.4 0.5 1],[1 1 0 0]);
fvtool(b1,1,b2,1);
```

Example 3

Create a lowpass, equiripple filter of order 20 in FDATool and display it in FVTool.

```
fdatool          % Start FDATool
```

Set these parameters in `fdatool`:

Parameter	Setting
Response Type	Lowpass
Design Method	FIR Equiripple
Filter Order	Specify order: 20
Density factor	16
Frequency specifications -- units	Normalized (0 to 1)
Wpass	0.4

Parameter	Setting
Wstop	0.5
Magnitude specifications -- Wpass and Wstop	1

and then click the **Design Filter** button.

Click the **Full View Analysis** button to start FVTool.

Example 4

Create an elliptic filter and use some of FVTool's figure handle commands:

```
[b,a]=ellip(6,3,50,300/500);
h = fvtool(b,a);     % Create handle, h and start FVTool
                     % with magnitude plot
```

```
set(h,'Analysis','phase')   % Change display to phase plot
```

```
set(h,'Legend','on')        % Turn legend on
legend(h,'Phase plot')      % Add legend text
```

```
get(h)                    % View all properties
                          % FVTool-specific properties are
                          %  at the end of this list.
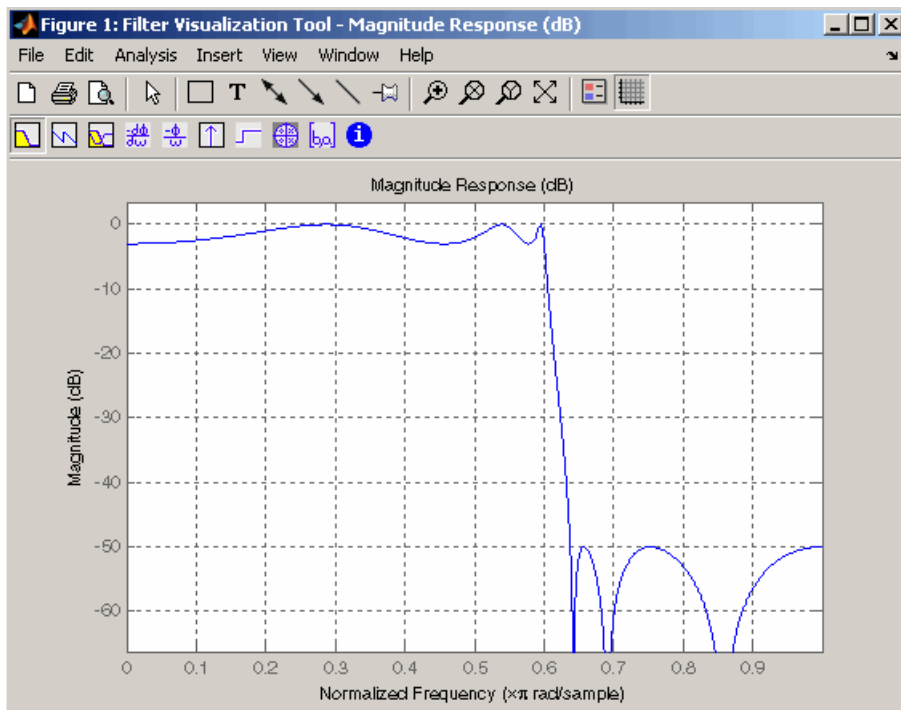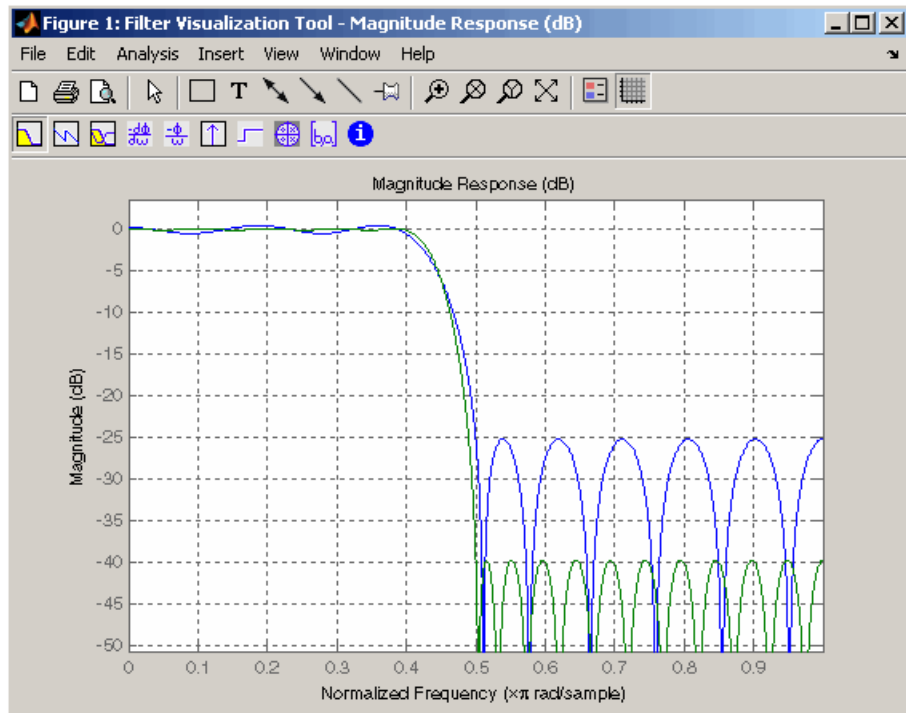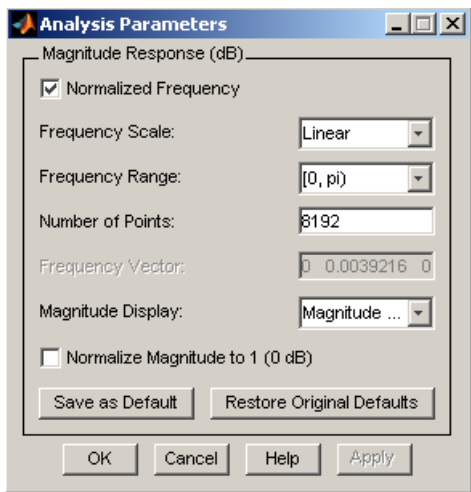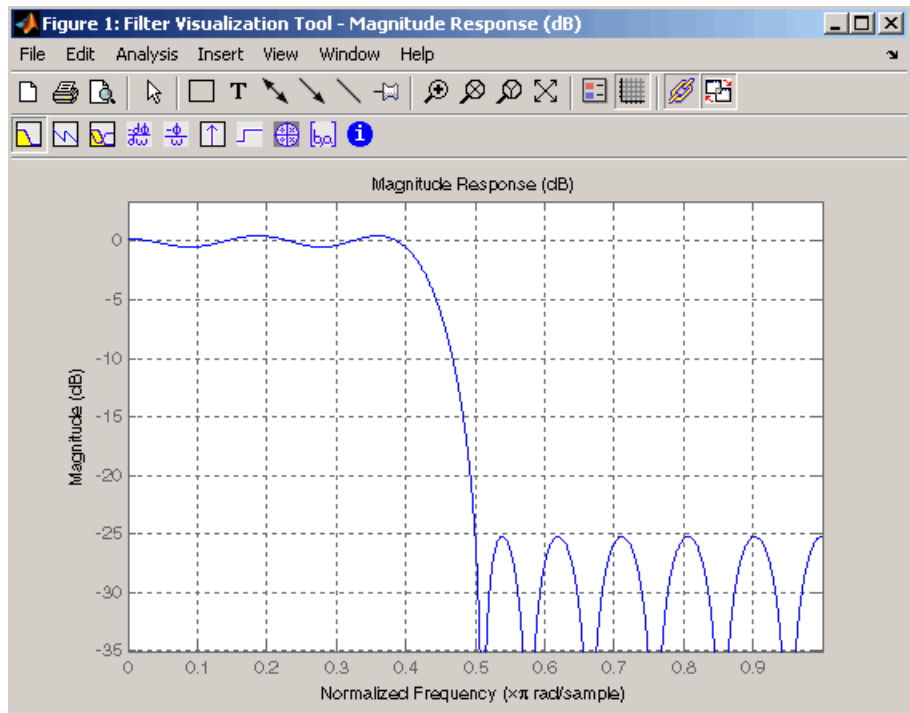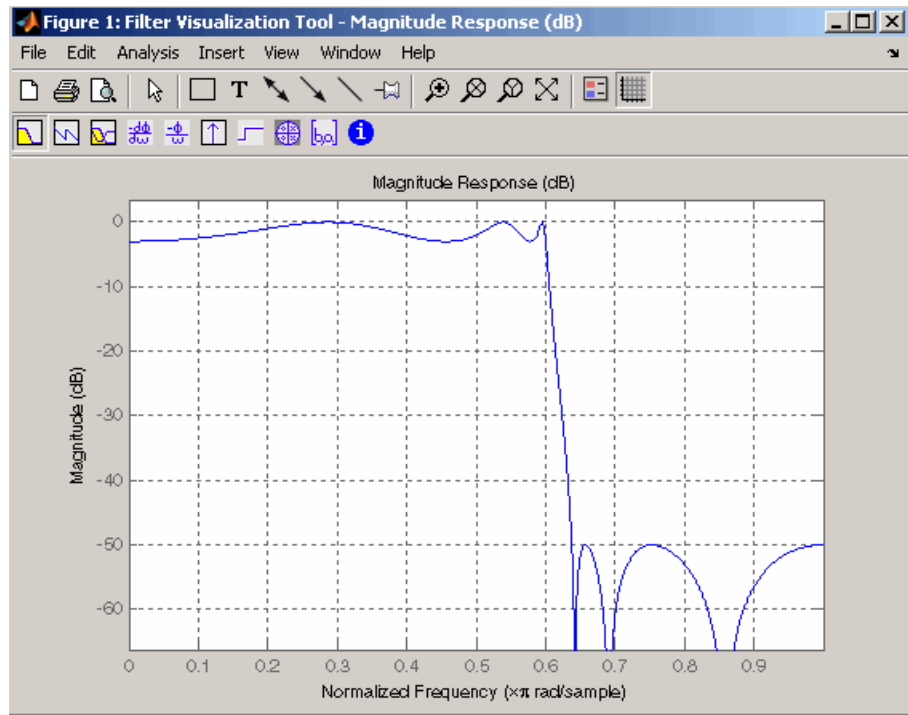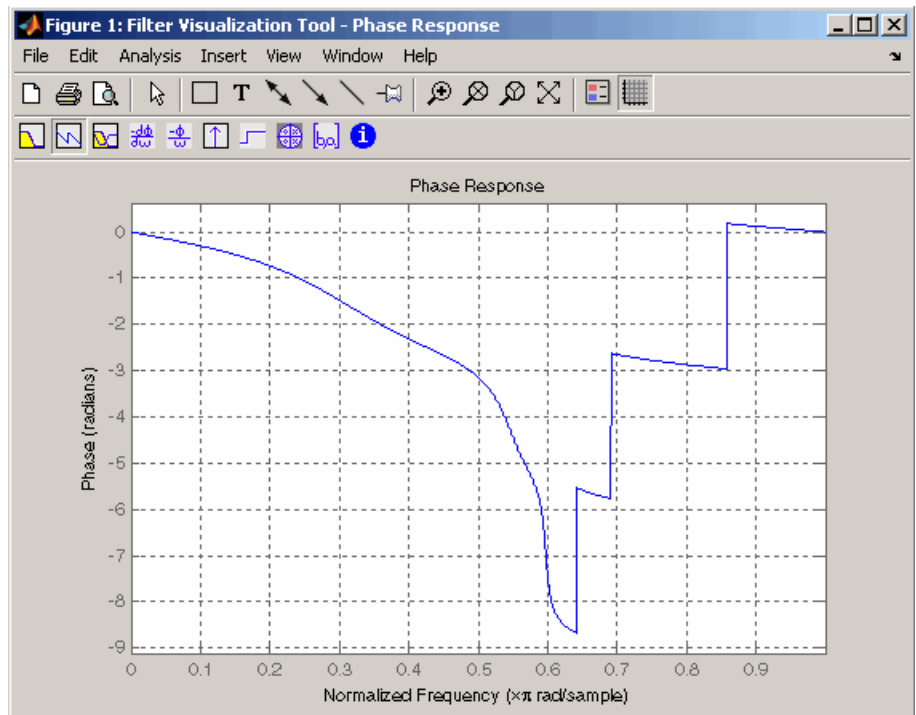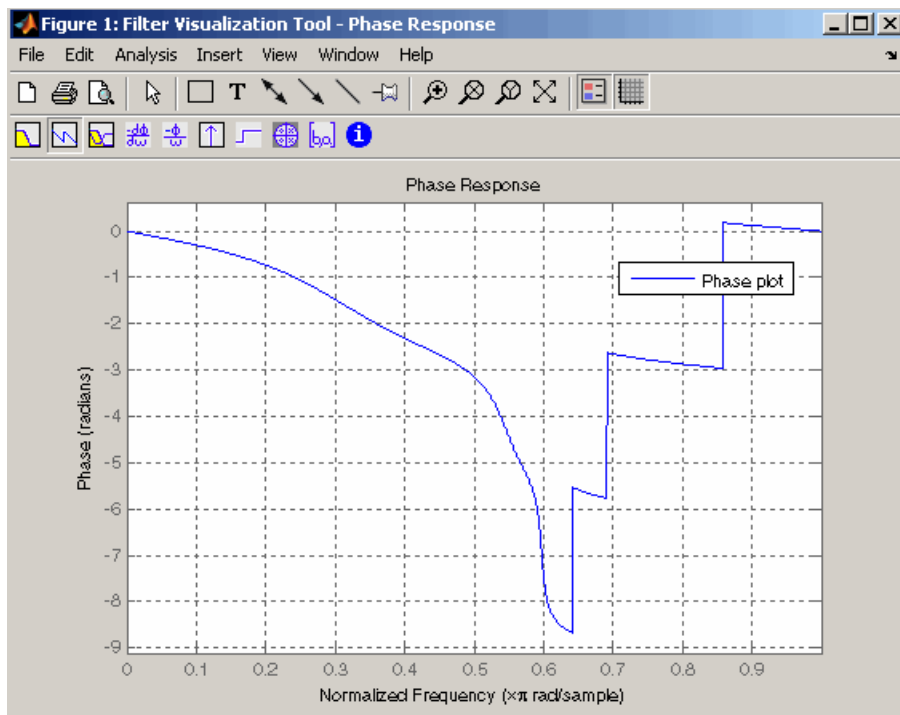

                AlphaMap: [1x64 double]
         CloseRequestFcn: 'closereq'
                   Color: [0.8314 0.8157 0.7843]
                ColorMap: [64x3 double]
             CurrentAxes: 208.0084
        CurrentCharacter: ''
           CurrentObject: []
            CurrentPoint: [0 0]
            DockControls: 'on'
            DoubleBuffer: 'on'
```

```
             FileName: ''
          FixedColors: [11x3 double]
        IntegerHandle: 'on'
       InvertHardcopy: 'on'
          KeyPressFcn: ''
              MenuBar: 'none'
           MinColormap: 64
                 Name: 'Filter Visualization Tool - Phase Response'
             NextPlot: 'new'
          NumberTitle: 'on'
           PaperUnits: 'inches'
      PaperOrientation: 'portrait'
         PaperPosition: [0.2500 2.5000 8 6]
     PaperPositionMode: 'manual'
             PaperSize: [8.5000 11]
             PaperType: 'usletter'
               Pointer: 'arrow'
       PointerShapeCData: [16x16 double]
     PointerShapeHotSpot: [1 1]
               Position: [360 292 560 345]
               Renderer: 'painters'
           RendererMode: 'auto'
                 Resize: 'on'
              ResizeFcn: ''
          SelectionType: 'normal'
                Toolbar: 'auto'
                  Units: 'pixels'
     WindowButtonDownFcn: ''
   WindowButtonMotionFcn: ''
       WindowButtonUpFcn: ''
             WindowStyle: 'normal'
             BeingDeleted: 'off'
           ButtonDownFcn: ''
                 Children: [15x1 double]
                 Clipping: 'on'
                CreateFcn: ''
                DeleteFcn: ''
```

```
              BusyAction: 'queue'
        HandleVisibility: 'on'
                 HitTest: 'on'
           Interruptible: 'on'
                  Parent: 0
                Selected: 'off'
      SelectionHighlight: 'on'
                     Tag: 'filtervisualizationtool'
           UIContextMenu: []
                UserData: []
                 Visible: 'on'
        AnalysisToolbar: 'on'
          FigureToolbar: 'on'
                 Filters: {[1x1 dfilt.df2t]}
                    Grid: 'on'
                  Legend: 'on'
              DesignMask: 'off'
                      Fs: 1
         SOSViewSettings: [1x1 dspopts.sosview]
                Analysis: 'phase'
        OverlayedAnalysis: ''
           ShowReference: 'on'
           PolyphaseView: 'off'
     NormalizedFrequency: 'on'
          FrequencyScale: 'Linear'
          FrequencyRange: '[O, pi)'
          NumberofPoints: 8192
          FrequencyVector: [1x256 double]
              PhaseUnits: 'Radians'
            PhaseDisplay: 'Phase'
```

See Also fdatool | sptool

Purpose   Fast Walsh–Hadamard transform

Syntax   y = fwht(x)
                 y = fwht(x,n)
                 y = fwht(x,n,ordering)

Description y = fwht(x) returns the coefficients of the discrete Walsh–Hadamard transform of the input x. If x is a matrix, the FWHT is calculated on each column of x. The FWHT operates only on signals with length equal to a power of 2. If the length of x is less than a power of 2, its length is padded with zeros to the next greater power of two before processing.

y = fwht(x,n) returns the n-point discrete Walsh–Hadamard transform, where n must be a power of 2. x and n must be the same length. If x is longer than n, x is truncated; if x is shorter than n, x is padded with zeros.

y = fwht(x,n,ordering) specifies the ordering to use for the returned Walsh–Hadamard transform coefficients. To specify ordering, you must enter a value for the length n or, to use the default behavior, specify an empty vector [ ] for n. Valid values for ordering are the following strings:

Ordering	Description
'sequency'	Coefficients in order of increasing sequency value, where each row has an additional zero crossing. This is the default ordering.
'hadamard'	Coefficients in normal Hadamard order.
'dyadic'	Coefficients in Gray code order, where a single bit change occurs from one coefficient to the next.

For more information on the Walsh functions and ordering, see "Walsh–Hadamard Transform".

Examples   This example shows a simple input signal and the resulting transformed signal.

x = [19 -1 11 -9 -7 13 -15 5];

```
y = fwht(x);
```

y contains nonzero values at these locations: 0, 1, 3, and 6. By forming the Walsh functions with the sequency values of 0, 1, 3, and 6, we can recreate x, as follows.

```
w0 = [1 1 1 1 1 1 1 1];
w1 = [1 1 1 1 -1 -1 -1 -1];
w3 = [1 1 -1 -1 1 1 -1 -1];
w6 = [1 -1 1 -1 -1 1 -1 1];
w = 2*w0 + 3*w1 + 4*w3 + 10*w6;
y1=fwht(w);
x1 = ifwht(y);
```

Algorithms   The fast Walsh-Hadamard tranform algorithm is similar to the Cooley-Tukey algorithm used for the FFT. Both use a butterfly structure to determine the transform coefficients. See the references for details.

References   [1] Beauchamp, K.G., *Applications of Walsh and Related Functions*, Academic Press, 1984.

[2] Beer, T., *Walsh Transforms*, American Journal of Physics, Volume 49, Issue 5, May 1981.

See Also   ifwht | dct | idct | fft | ifft

Purpose   Gaussian-modulated sinusoidal pulse

Syntax
```
yi = gauspuls(t,fc,bw)
yi = gauspuls(t,fc,bw,bwr)
[yi,yq] = gauspuls(...)
[yi,yq,ye] = gauspuls(...)
tc = gauspuls('cutoff',fc,bw,bwr,tpe)
```

Description   `gauspuls` generates Gaussian-modulated sinusoidal pulses.

`yi = gauspuls(t,fc,bw)` returns a unity-amplitude Gaussian RF pulse at the times indicated in array `t`, with a center frequency `fc` in hertz and a fractional bandwidth `bw`, which must be greater than 0. The default value for `fc` is 1000 Hz and for `bw` is 0.5.

`yi = gauspuls(t,fc,bw,bwr)` returns a unity-amplitude Gaussian RF pulse with a fractional bandwidth of `bw` as measured at a level of `bwr` dB with respect to the normalized signal peak. The fractional bandwidth reference level `bwr` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `bwr` is -6 dB. Note that the fractional bandwidth is specified in terms of power ratios. This corresponds to the -3 dB point expressed in magnitude ratios.

`[yi,yq] = gauspuls(...)` returns both the in-phase and quadrature pulses.

`[yi,yq,ye] = gauspuls(...)` returns the RF signal envelope.

`tc = gauspuls('cutoff',fc,bw,bwr,tpe)` returns the cutoff time `tc` (greater than or equal to 0) at which the trailing pulse envelope falls below `tpe` dB with respect to the peak envelope amplitude. The trailing pulse envelope level `tpe` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `tpe` is -60 dB.

Tips   Default values are substituted for empty or omitted trailing input arguments.

gauspuls

Examples   Plot a 50 kHz Gaussian RF pulse with 60% bandwidth, sampled at
a rate of 1 MHz. Truncate the pulse where the envelope falls 40 dB
below the peak:

```
tc = gauspuls('cutoff',50e3,0.6,[],-40);
t = -tc : 1e-6 : tc;
yi = gauspuls(t,50e3,0.6);
plot(t,yi)
```


See Also   chirp | cos | diric | pulstran | rectpuls | sawtooth | sin | sinc
| square | tripuls

Purpose   Gaussian FIR pulse-shaping filter

Syntax   h = gaussfir(bt)
               h = gaussfir(bt,n)
               h = gaussfir(bt,n,o)

Description   This filter is used primarily in Gaussian minimum shift keying (GMSK) communications applications.

h = gaussfir(bt) designs a low pass FIR Gaussian pulse-shaping filter and returns the filter coefficients in the h vector. bt is the 3-dB bandwidth-symbol time product where b is the one-sided bandwidth in hertz and t is in seconds. Smaller bt products produce larger pulse widths. The number of symbol periods (n) defaults to 3 and the oversampling factor (o) defaults to 2.

The length of the impulse response of the filter is given by 2*o*n+1. The coefficients h are normalized so that the nominal passband gain is always equal to 1.

h = gaussfir(bt,n) uses n number of symbol periods between the start of the filter impulse response and its peak.

h = gaussfir(bt,n,o) uses an oversampling factor of o, which is the number of samples per symbol.

Examples   Design a Gaussian filter to be used in a Global System for Mobile (GSM) communications GMSK scheme.

```
bt = .3;          % 3-dB bandwidth-symbol time
o = 8;            % Oversampling factor
n = 2;            % 2 symbol periods to the filters peak
h = gaussfir(bt,n,o);
hfvt = fvtool(h,'impulse');
```

gaussfir

References   [1] Rappaport T.S., Wireless Communications Principles and Practice, 2nd Edition, Prentice Hall, 2001.

[2] Krishnapura N., Pavan S., Mathiazhagan C., Ramamurthi B., "A Baseband Pulse Shaping Filter for Gaussian Minimum Shift Keying," *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, 1998.

See Also   firrcos | rcosfir

Purpose   Gaussian window

Syntax   *w* = gausswin(*N*)
                   w = gausswin(*N*,*Alpha*)

Description   *w* = gausswin(*N*) returns an N-point Gaussian window in the column
                   vector w. L is a positive integer. The coefficients of a Gaussian window
                   are computed from the following equation.

$$w(n) = e^{-\frac{1}{2}\left(\alpha\frac{n}{N/2}\right)^2}$$

where $-\dfrac{N}{2} \le n \le \dfrac{N}{2}$, and α is inversely proportional to the standard
deviation of a Gaussian random variable.

w = gausswin(*N*,*Alpha*) returns an N-point Gaussian window where
Alpha is proportional to reciprocal of the standard deviation. The width
of the window is inversely related to the value of α; a larger value of α
produces a more narrow window. If α is omitted, it defaults to 2.5.

Note If the window appears to be clipped, increase the number of
points (N).

Examples   Create a 64-point Gaussian window and display the result in WVTool:

```
L=64;
wvtool(gausswin(L))
```

Note The shape of this window is similar in the frequency domain because the Fourier transform of a Gaussian is also a Gaussian.

References   [1] Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE. Vol. 66*, No. 1 (January 1978).

[2] Roberts, Richard A., and C.T. Mullis. *Digital Signal Processing.* Reading, MA: Addison-Wesley, 1987, pp. 135-136.

See Also   chebwin | kaiser | tukeywin | window | wintool | wvtool

Purpose   Gaussian monopulse

Syntax   y = gmonopuls(t,fc)
                   tc = gmonopuls('cutoff',fc)

Description   y = gmonopuls(t,fc) returns samples of the unity-amplitude
                   Gaussian monopulse with center frequency fc (in hertz) at the times
                   indicated in array t. By default, fc = 1000 Hz.

                   tc = gmonopuls('cutoff',fc) returns the time duration between the
                   maximum and minimum amplitudes of the pulse.

Tips   Default values are substituted for empty or omitted trailing input
                   arguments.

Examples   **Example 1**

                   Plot a 2 GHz Gaussian monopulse sampled at a rate of 100 GHz:

                   ```
                   fc = 2E9; fs=100E9;
                   tc = gmonopuls('cutoff',fc);
                   t  = -2*tc : 1/fs : 2*tc;
                   y = gmonopuls(t,fc); plot(t,y)
                   ```

gmonopuls

Example 2

Construct a pulse train from the monopulse of Example 1 using a
spacing of 7.5 ns:

```
fc = 2E9;  fs=100E9;            % Center freq, sample freq
D = [2.5 10 17.5]' * 1e-9;      % Pulse delay times
tc = gmonopuls('cutoff',fc);    % Width of each pulse
t  = 0 : 1/fs : 150*tc;         % Signal evaluation time
yp = pulstran(t,D,@gmonopuls,fc);
plot(t,yp)
```

See Also   chirp | gauspuls | pulstran | rectpuls | tripuls

goertzel

Purpose	Discrete Fourier transform with second-order Goertzel algorithm

Syntax	`dft_data = goertzel(data)`
	`dft_data = goertzel(data,freq_indices)`
	`dft_data = goertzel(data,freq_indices,dim)`

Description	`dft_data = goertzel(data)` returns the discrete Fourier transform (DFT) of the input data `data` using a second-order Goertzel algorithm. If `data` is a matrix, `goertzel` computes the DFT of each column separately. `dft_data = goertzel(data,freq_indices)` returns the DFT for the frequency indices `freq_indices`. `dft_data = goertzel(data,freq_indices,dim)` computes the DFT of the matrix `data` along the dimension `dim`.
Examples	Estimate the frequency of the two tones generated by pressing the 1 button on a telephone keypad:

```
f=[697 770 852 941 1209 1336 1477];
% frequencies for numbers 0:9 on keypad
Fs = 8000; %sampling frequency
N = 205; %Number of points
% Tones generated by a "1": 697 and 1209 Hz
data = sum(sin(2*pi*[697;1209]*(0:N-1)/Fs));
% Indices of the DFT for the frequencies f
freq_indices = round(f/Fs*N)+1;
%Compute DFT using Goertzel algorithm
dft_data = goertzel(data,freq_indices);
%Plot the DFT magnitudes
stem(f,abs(dft_data));
set(gca,'xtick',f); xlabel('Hz');
ylabel('Magnitude');
```

Algorithms   The Goertzel algorithm implements the DFT as a recursive difference equation. To establish this difference equation, express the DFT as the convolution of an N-point input $x(n)$ with the impulse response $h(n) = W_N^{-kn}u(n)$, where $W_N^{-kn} = e^{-i2\pi k/N}$ and $u(n)$ is the unit step sequence.

The z-transform of the impulse response is:

$$H(z) = \frac{1 - W_N^k z^{-1}}{1 - 2\cos(2\pi k/N)z^{-1} + z^{-2}}$$

The direct form II implementation is:

References   Proakis, J.G. and D.G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*, Upper Saddle River, NJ: Prentice Hall, 1996, pp. 480–481.

Alternatives   You can also compute the DFT with:

- fft less efficient than the Goertzel algorithm when you only need the DFT at a few frequencies.

- czt the chirp z-transform. czt calculates the z-transform of an input on a circular or spiral contour and includes the DFT as a special case.

Purpose   Average filter delay (group delay)

Syntax   grpdelay(b,a)
                 [gd,w] = grpdelay(b,a,l)
                 [gd,f] = grpdelay(b,a,n,fs)
                 [gd,w] = grpdelay(b,a,n,'whole')
                 [gd,f] = grpdelay(b,a,n,'whole', fs)
                 gd = grpdelay(b,a,w)
                 gd = grpdelay(b,a,f,fs)
                 grpdelay(Hd)

Description   The *group delay* of a filter is a measure of the average delay of the
                 filter as a function of frequency. It is the negative first derivative of
                 the phase response of the filter. If the frequency response of a filter
                 is $H(e^{j\omega})$, then the group delay is

$$\tau_g(\omega) = -\frac{d\theta(\omega)}{d\omega}$$

where $\theta(\omega)$ is the phase, or argument, of phase $H(e^{j\omega})$.

grpdelay(b,a) with no output arguments plots the group delay versus
frequency in the current figure window.

[gd,w] = grpdelay(b,a,l) returns the l-point group delay of the
digital filter

$$H(z) = \frac{B(z)}{A(z)} = \frac{\displaystyle\sum_{n=0}^{N-1} b(n+1)z^{-n}}{\displaystyle\sum_{l=0}^{M-1} a(l+1)z^{-l}},$$

given the numerator and denominator coefficients in vectors b and a.
grpdelay returns both gd, the group delay, which has units of samples,
and w, a vector containing the n frequency points in radians. grpdelay

evaluates the group delay at n points equally spaced around the upper half of the unit circle, so w contains n points between 0 and π.

[gd,f] = grpdelay(b,a,n,fs) specifies a positive sampling frequency fs in hertz. It returns a length n vector f containing the actual frequency points at which the group delay is calculated, also in hertz. f contains n points between 0 and fs/2.

[gd,w] = grpdelay(b,a,n,'whole') and

[gd,f] = grpdelay(b,a,n,'whole', fs) use n points around the whole unit circle (from 0 to 2π, or from 0 to fs).

gd = grpdelay(b,a,w) and

gd = grpdelay(b,a,f,fs) return the group delay evaluated at the points in w (in radians) or f (in hertz), respectively, where fs is the sampling frequency in hertz.

grpdelay(Hd) plots the group delay and displays the plot in fvtool. The input Hd is a dfilt filter object or an array of dfilt filter objects.

grpdelay works for both real and complex filters.

Examples   Plot the group delay of Butterworth filter b(z)/a(z):

```
[b,a] = butter(6,0.2);
grpdelay(b,a,128)
```

The same example using a dfilt object and displaying the result in the
Filter Visualization Tool (fvtool) is

```
[b,a] = butter(6,0.2);
Hd=dfilt.df1(b,a);
grpdelay(Hd,128)
```

Plot both the group and phase delays of a system on the same graph:

```
[b,a] = butter(6,0.2);
gd = grpdelay(b,a,512);
gd(1) = [];           % Avoid NaNs
[h,w] = freqz(b,a,512); h(1) = []; w(1) = [];
pd = -unwrap(angle(h))./w;
plot(w,gd,w,pd,':')
axis([0 pi min(gd) max(gd)]);
xlabel('Frequency (rad/sec)'); grid;
legend('Group Delay','Phase Delay');
```

Algorithms   grpdelay multiplies the filter coefficients by a unit ramp. After Fourier transformation, this process corresponds to differentiation.

See Also   cceps | fft | freqz | fvtool | hilbert | icceps | rceps

hamming

Purpose   Hamming window

Syntax   w = hamming(L)
w = hamming(L,'*sflag*')

Description   w = hamming(L) returns an L-point symmetric Hamming window in the
column vector w. L should be a positive integer. The coefficients of a
Hamming window are computed from the following equation.

$$w(n) = 0.54 - 0.46\cos\left(2\pi\frac{n}{N}\right), \quad 0 \le n \le N$$

The window length is $L = N + 1$.

w = hamming(L,'*sflag*') returns an L-point Hamming window using
the window sampling specified by '*sflag*', which can be either
'periodic' or 'symmetric' (the default). The 'periodic' flag
is useful for DFT/FFT purposes, such as in spectral analysis. The
DFT/FFT contains an implicit periodic extension and the periodic flag
enables a signal windowed with a periodic window to have perfect
periodic extension. When 'periodic' is specified, hamming computes a
length L+1 window and returns the first L points. When using windows
for filter design, the 'symmetric' flag should be used.

Note If you specify a one-point window (L=1), the value 1 is returned.

Examples   Create a 64-point Hamming window and display the result in WVTool:

```
L=64;
wvtool(hamming(L))
```

References [1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 447-448.

See Also blackman | flattopwin | hann | window | wintool | wvtool

hann

Purpose   Hann (Hanning) window

Syntax   w = hann(L)
                   w = hann(L,'*sflag*')

Description   w = hann(L) returns an L-point symmetric Hann window in the column
                   vector w. L must be a positive integer. The coefficients of a Hann
                   window are computed from the following equation.

$$w(n) = 0.5\left(1 - \cos\left(2\pi\frac{n}{N}\right)\right), \quad 0 \le n \le N$$

The window length is $L = N + 1$.

w = hann(L,'*sflag*') returns an L-point Hann window using the
window sampling specified by '*sflag*', which can be either 'periodic'
or 'symmetric' (the default). The 'periodic' flag is useful for
DFT/FFT purposes, such as in spectral analysis. The DFT/FFT contains
an implicit periodic extension and the periodic flag enables a signal
windowed with a periodic window to have perfect periodic extension.
When 'periodic' is specified, hann computes a length L+1 window and
returns the first L points. When using windows for filter design, the
'symmetric' flag should be used.

Note If you specify a one-point window (L=1), the value 1 is returned.

Examples   Create a 64-point Hann window and display the result in WVTool:

```
L=64;
wvtool(hann(L))
```

References   [1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 447-448.

See Also   blackman | flattopwin | hamming | window | wintool | wvtool

hilbert

Purpose   Discrete-time analytic signal using Hilbert transform

Syntax   x = hilbert(xr)
                 x = hilbert(xr,n)

Description x = hilbert(xr) returns a complex helical sequence, sometimes
                 called the *analytic signal*, from a real data sequence. The analytic
                 signal x = xr + i*xi has a real part, xr, which is the original data,
                 and an imaginary part, xi, which contains the Hilbert transform. The
                 imaginary part is a version of the original real sequence with a 90°
                 phase shift. Sines are therefore transformed to cosines and vice versa.
                 The Hilbert transformed series has the same amplitude and frequency
                 content as the original real data and includes phase information that
                 depends on the phase of the original data.

                 If xr is a matrix, x = hilbert(xr) operates columnwise on the matrix,
                 finding the Hilbert transform of each column.

                 x = hilbert(xr,n) uses an n point FFT to compute the Hilbert
                 transform. The input data xr is zero-padded or truncated to length n,
                 as appropriate.

                 The Hilbert transform is useful in calculating instantaneous attributes
                 of a time series, especially the amplitude and frequency. The
                 instantaneous amplitude is the amplitude of the complex Hilbert
                 transform; the instantaneous frequency is the time rate of change of the
                 instantaneous phase angle. For a pure sinusoid, the instantaneous
                 amplitude and frequency are constant. The instantaneous phase,
                 however, is a sawtooth, reflecting the way in which the local phase
                 angle varies linearly over a single cycle. For mixtures of sinusoids, the
                 attributes are short term, or local, averages spanning no more than
                 two or three points.

                 Reference [1] describes the Kolmogorov method for minimum phase
                 reconstruction, which involves taking the Hilbert transform of the
                 logarithm of the spectral density of a time series. The toolbox function
                 rceps performs this reconstruction.

For a discrete-time analytic signal x, the last half of fft(x) is zero, and the first (DC) and center (Nyquist) elements of fft(x) are purely real.

Examples

```
xr = [1 2 3 4];
x = hilbert(xr)
x
```

You can see that the imaginary part, imag(x) = [1 -1 -1 1], is the Hilbert transform of xr, and the real part, real(x) = [1 2 3 4], is simply xr itself. Note that the last half of fft(x) = [10 -4+4i -2 0] is zero (in this example, the last half is just the last element), and that the DC and Nyquist elements of fft(x), 10 and -2 respectively, are purely real.

Algorithms

The analytic signal for a sequence x has a *one-sided Fourier transform*, that is, negative frequencies are 0. To approximate the analytic signal, hilbert calculates the FFT of the input sequence, replaces those FFT coefficients that correspond to negative frequencies with zeros, and calculates the inverse FFT of the result.

In detail, hilbert uses a four-step algorithm:

1 It calculates the FFT of the input sequence, storing the result in a vector x.

2 It creates a vector h whose elements h(i) have the values:

- 1 for i = 1, (n/2)+1
- 2 for i = 2, 3, ... , (n/2)
- 0 for i = (n/2)+2, ... , n

3 It calculates the element-wise product of x and h.

4 It calculates the inverse FFT of the sequence obtained in step 3 and returns the first n elements of the result.

If the input data `xr` is a matrix, `hilbert` operates in a similar manner, extending each step above to handle the matrix case.

References

[1] Claerbout, J.F., *Fundamentals of Geophysical Data Processing*, McGraw-Hill, 1976, pp.59-62.

[2] Marple, S.L., "Computing the discrete-time analytic signal via FFT," IEEE Transactions on Signal Processing, Vol. 47, No. 9 (September 1999), pp. 2600-2603.

[3] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, 2nd ed., Prentice-Hall, 1998.

See Also   fft | ifft | rceps

Purpose   Inverse complex cepstrum

Syntax   x = icceps(xhat,nd)

Description
_____

Note icceps only works on real data.
_____

x = icceps(xhat,nd) returns the inverse complex cepstrum of the
real data sequence xhat, removing nd samples of delay. If xhat was
obtained with cceps(x), then the amount of delay that was added to x
was the element of round(unwrap(angle(fft(x)))/pi) corresponding
to π radians.

References   [1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal
                   Processing*, Prentice-Hall, 1989.

See Also   cceps | hilbert | rceps | unwrap

idct

Purpose	Inverse discrete cosine transform

Syntax

```
x = idct(y)
x = idct(y,n)
```

Description The inverse discrete cosine transform reconstructs a sequence from its discrete cosine transform (DCT) coefficients. The idct function is the inverse of the dct function.

$x = idct(y)$ returns the inverse discrete cosine transform of y

$$x(n) = \sum_{k=1}^{N} w(k)y(k)\cos(\frac{\pi(2n-1)(k-1)}{2N}) \quad n = 1, 2, \dots N$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{N}} & k = 1 \\ \sqrt{\dfrac{2}{N}} & 2 \le k \le N \end{cases}$$

and N = length(x), which is the same as length(y). The series is indexed from n = 1 and k = 1 instead of the usual n = 0 and k = 0 because MATLAB vectors run from 1 to N instead of from 0 to N-1.

$x = idct(y,n)$ appends zeros or truncates the vector y to length n before transforming.

If y is a matrix, idct transforms its columns.

References [1] Jain, A.K., *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989.

[2] Pennebaker, W.B., and J.L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993, Chapter 4.

See Also dct | dct2 | idct2 | ifft

Purpose   Inverse Fast Walsh–Hadamard transform

Syntax
```
y = iwht(x)
y = ifwht(x,n)
y = ifwht(x,n,ordering)
```

Description   `y = iwht(x)` returns the coefficients of the inverse discrete fast Walsh–Hadamard transform of the input `x`. If `x` is a matrix, the inverse fast Walsh-Hadamard tranform is calculated on each column of `x`. The inverse fast Walsh-Hadamard tranform operates only on signals with length equal to a power of 2. If the length of `x` is less than a power of 2, its length is padded with zeros to the next greater power of two before processing.

`y = ifwht(x,n)` returns the n-point inverse discrete Walsh–Hadamard transform, where `n` must be a power of 2.

`y = ifwht(x,n,ordering)` specifies the ordering to use for the returned inverse Walsh–Hadamard transform coefficients. To specify ordering, you must enter a value for the length `n` or, to use the default behavior, specify an empty vector `[]` for `n`. Valid values for ordering are the following strings:

Ordering	Description
`'sequency'`	Coefficients in order of ascending sequency value, where each row has an additional zero crossing. This is the default ordering.
`'hadamard'`	Coefficients in normal Hadamard order.
`'dyadic'`	Coefficients in Gray code order, where a single bit change occurs from one coefficient to the next.

Algorithms   The inverse fast Walsh-Hadamard tranform algorithm is similar to the Cooley-Tukey algorithm used for the inverse FFT. Both use a butterfly structure to determine the transform coefficients. See the references below for details.

ifwht

References   [1] Beauchamp, K.G., *Applications of Walsh and Related Functions*, Academic Press, 1984.

[2] Beer, T., *Walsh Transforms*, American Journal of Physics, Volume 49, Issue 5, May 1981.

See Also   fwht | dct | idct | fft | ifft

Purpose   Impulse invariance method for analog-to-digital filter conversion

Syntax   [bz,az] = impinvar(b,a,fs)
                 [bz,az] = impinvar(b,a,fs,tol)

Description [bz,az] = impinvar(b,a,fs) creates a digital filter with numerator
                and denominator coefficients bz and az, respectively, whose impulse
                response is equal to the impulse response of the analog filter with
                coefficients b and a, scaled by 1/fs. If you leave out the argument fs, or
                specify fs as the empty vector [], it takes the default value of 1 Hz.

                [bz,az] = impinvar(b,a,fs,tol) uses the tolerance specified by
                tol to determine whether poles are repeated. A larger tolerance
                increases the likelihood that impinvar interprets closely located poles
                as multiplicities (repeated ones). The default is 0.001, or 0.1% of a pole's
                magnitude. Note that the accuracy of the pole values is still limited to
                the accuracy obtainable by the roots function.

Examples   ### Example 1

                Convert an analog lowpass filter to a digital filter using impinvar with
                a sampling frequency of 10 Hz:

                ```
                [b,a] = butter(4,0.3,'s');
                [bz,az] = impinvar(b,a,10);
                ```

                ### Example 2

                Illustrate the relationship between analog and digital impulse
                responses [2].

                Note This example requires the impulse function from Control System
                Toolbox™ software.

                The steps used in this example are:

1 Create an analog Butterworth filter

2 Use impinvar with a sampling frequency Fs of 10 Hz to scale the coefficients by 1/Fs. This compensates for the gain that will be introduced in Step 4 below.

3 Use Control System Toolbox impulse function to plot the continuous-time unit impulse response of an LTI system.

4 Plot the digital impulse response, multiplying the numerator by a constant (Fs) to compensate for the 1/Fs gain introduced in the impulse response of the derived digital filter.

```
[b,a] = butter(4,0.3,'s');
[bz,az] = impinvar(b,a,10);
sys = tf(b,a);
impulse(sys);
hold on;
impz(10*bz,az,[],10);
```

Zooming the resulting plot shows that the analog and digital impulse responses are the same.

Algorithms   impinvar performs the impulse-invariant method of analog-to-digital transfer function conversion discussed in reference [1]:

1 It finds the partial fraction expansion of the system represented by b and a.

2 It replaces the poles p by the poles exp(p/fs).

3 It finds the transfer function coefficients of the system from the residues from step 1 and the poles from step 2.

References   [1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp.206-209.

[2] Antoniou, Andreas, *Digital Filters*, McGraw Hill, Inc, 1993, pp.221-224.

See Also   bilinear | lp2bp | lp2bs | lp2hp | lp2lp

impz

Purpose	Impulse response of digital filter

Syntax

```
[h,t] = impz(b,a)
[h,t] = impz(b,a,n)
[h,t] = impz(b,a,n,fs)
impz(b,a)
impz(Hd)
```

Description

[h,t] = impz(b,a) computes the impulse response of the filter with numerator coefficients b and denominator coefficients a. impz chooses the number of samples and returns the response in the column vector h and sample times in the column vector t (where t = [0:n-1]', and n = length(t) is computed automatically).

[h,t] = impz(b,a,n) computes n samples of the impulse response when n is an integer (t = [0:n-1]'). If n is a vector of integers, impz computes the impulse response at those integer locations, starting the response computation from 0 (and t = n or t = [0 n]). If, instead of n, you include the empty vector [ ] for the second argument, the number of samples is computed automatically by default.

[h,t] = impz(b,a,n,fs) computes n samples and produces a vector t of length n so that the samples are spaced 1/fs units apart.

impz(b,a) with no output arguments plots the impulse response and displays the response in the current figure window.

impz(Hd) plots the impulse response of the filter and displays the plot in fvtool. The input Hd is a dfilt filter object or an array of dfilt filter objects.

Note impz works for both real and complex input systems.

Examples

Plot the first 50 samples of the impulse response of a fourth-order lowpass elliptic filter with cutoff frequency of 0.4 times the Nyquist frequency:

```
[b,a] = ellip(4,0.5,20,0.4);
impz(b,a,50)
```


The same example using a dfilt object and displaying the result in the Filter Visualization Tool (fvtool) is

```
[b,a] = ellip(4,0.5,20,0.4);
Hd = dfilt.df1(b,a)
impz(Hd,50)
```

Algorithms   impz filters a length n impulse sequence using

```
filter(b,a,[1 zeros(1,n-1)])
```

and plots the results using stem.

To compute n in the auto-length case, impz either uses n = length(b) for the FIR case or first finds the poles using p = roots(a), if length(a) is greater than 1.

If the filter is unstable, n is chosen to be the point at which the term from the largest pole reaches 10^6 times its original value.

If the filter is stable, n is chosen to be the point at which the term due to the largest amplitude pole is 5*10^-5 of its original amplitude.

If the filter is oscillatory (poles on the unit circle only), impz computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms, n is chosen to equal five periods of the slowest oscillation or the point at which the term due to the largest (nonunity) amplitude pole is 5*10^-5 of its original amplitude, whichever is greater.

impz also allows for delays in the numerator polynomial. The number of delays is incorporated into the computation for the number of samples.

See Also   impulse | stem

interp

Purpose   Interpolation — increase sampling rate by integer factor

Syntax   y = interp(x,r)
                y = interp(x,r,l,alpha)
                [y,b] = interp(x,r,l,alpha)

Description Interpolation increases the original sampling rate for a sequence to a
                higher rate. interp performs lowpass interpolation by inserting zeros
                into the original sequence and then applying a special lowpass filter.
                The filter returned by intfilt is identical to the filter used by interp.

                y = interp(x,r) increases the sampling rate of x by a factor of r. The
                interpolated vector y is r times longer than the original input x.

                y = interp(x,r,l,alpha) specifies l (filter length) and alpha (cut-off
                frequency). The default value for l is 4 and the default value for alpha
                is 0.5.

                [y,b] = interp(x,r,l,alpha) returns vector b containing the filter
                coefficients used for the interpolation.

Examples   Interpolate a signal by a factor of four:

```
t = 0:0.001:1;      % Time vector
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = interp(x,4);
subplot(121);
stem(x(1:30));
axis([0 30 -2 2]);
title('Original Signal');
subplot(122);
stem(y(1:120));
title('Interpolated Signal');
axis([0 120 -2 2]);
```

Algorithms   interp uses the lowpass interpolation Algorithm 8.1 described in [1]:

1 It expands the input vector to the correct length by inserting zeros between the original data values.

2 It designs a special symmetric FIR filter that allows the original data to pass through unchanged and interpolates between so that the mean-square errors between the interpolated points and their ideal values are minimized.

3 It applies the filter to the input vector to produce the interpolated output vector.

The length of the FIR lowpass interpolating filter is 2*l*r+1. The number of original sample values used for interpolation is 2*l. Ordinarily, l should be less than or equal to 10. The original signal is assumed to be band limited with normalized cutoff frequency $0 \leq$ alpha ≤ 1, where 1 is half the original sampling frequency (the Nyquist

frequency). The default value for `l` is 4 and the default value for `alpha` is 0.5.

Diagnostics   If `r` is not an integer, `interp` gives the following error message:

```
Resampling rate R must be an integer.
```

References   [1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, Algorithm 8.1.

See Also   decimate | downsample | interp1 | intfilt | resample | spline | upfirdn | upsample

Purpose   Interpolation FIR filter design

Syntax   b = intfilt(l,p,alpha)
                 b = intfilt(l,n,'Lagrange')

Description b = intfilt(l,p,alpha) designs a linear phase FIR filter that
                 performs ideal bandlimited interpolation using the nearest 2*p nonzero
                 samples, when used on a sequence interleaved with l-1 consecutive
                 zeros every l samples. It assumes an original bandlimitedness of alpha
                 times the Nyquist frequency. The returned filter is identical to that
                 used by interp. b is length 2*l*p-1

                 alpha is inversely proportional to the transition bandwidth of the
                 filter and it also affects the bandwith of the don't-care regions in
                 the stopband. Specifying alpha allows you to specify how much of
                 the Nyquist interval your input signal occupies. This is beneficial,
                 particularly for signals to be interpolated, because it allows you to
                 increase the transition bandwidth without affecting the interpolation
                 and results in better stopband attenuation for a given l and p. If you
                 set alpha to 1, your signal is assumed to occupy the entire Nyquist
                 interval. Setting alpha to less than one allows for don't-care regions in
                 the stopband. For example, if your input occupies half the Nyquist
                 interval, you could set alpha to 0.5.

                 b = intfilt(l,n,'Lagrange') designs an FIR filter that performs
                 nth-order Lagrange polynomial interpolation on a sequence interleaved
                 with l-1 consecutive zeros every r samples. b has length (n + 1)*l for n
                 even, and length (n + 1)*l-1 for n odd. If both n and l are even, the
                 filter designed is not linear phase.

                 Both types of filters are basically lowpass and have a gain of l in the
                 passband..

Examples   Design a digital interpolation filter to upsample a signal by four, using
                 the bandlimited method:

```
alpha = 0.5;                  % "Bandlimitedness" factor
h1 = intfilt(4,2,alpha);      % Bandlimited interpolation
```

The filter `h1` works best when the original signal is bandlimited to alpha times the Nyquist frequency. Create a bandlimited noise signal:

```
x = filter(fir1(40,0.5),1,randn(200,1));   % Bandlimit
```

Now zero pad the signal with three zeros between every sample. The resulting sequence is four times the length of `x`:

```
xr = reshape([x zeros(length(x),3)]',4*length(x),1);
```

Interpolate using the `filter` command:

```
y = filter(h1,1,xr);
```

`y` is an interpolated version of `x`, delayed by seven samples (the group-delay of the filter). Zoom in on a section of one hundred samples to see this:

```
plot(100:200,y(100:200),7+(101:4:196),x(26:49),'o')
```

`intfilt` also performs Lagrange polynomial interpolation of the original signal. For example, first-order polynomial interpolation is just linear interpolation, which is accomplished with a triangular filter:

```
h2 = intfilt(4,1,'l');  % Lagrange interpolation
```

Algorithms   The bandlimited method uses `firls` to design an interpolation FIR filter. The polynomial method uses Lagrange's polynomial interpolation formula on equally spaced samples to construct the appropriate filter.

See Also   decimate | downsample | interp | resample | upsample

Purpose   Identify continuous-time filter parameters from frequency response data

Syntax
```
[b,a] = invfreqs(h,w,n,m)
[b,a] = invfreqs(h,w,n,m,wt)
[b,a] = invfreqs(h,w,n,m,wt,iter)
[b,a] = invfreqs(h,w,n,m,wt,iter,tol)
[b,a] = invfreqs(h,w,n,m,wt,iter,tol,'trace')
[b,a] = invfreqs(h,w,'complex',n,m,...)
```

Description   invfreqs is the inverse operation of freqs. It finds a continuous-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, invfreqs is useful in converting magnitude and phase data into transfer functions.

[b,a] = invfreqs(h,w,n,m) returns the real numerator and denominator coefficient vectors b and a of the transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{a(1)s^m + a(2)s^{m-1} + \cdots + a(m+1)}$$

whose complex frequency response is given in vector h at the frequency points specified in vector w. Scalars n and m specify the desired orders of the numerator and denominator polynomials.

The length of h must be the same as the length of w. invfreqs uses conj(h) at -w to ensure the proper frequency domain symmetry for a real filter.

[b,a] = invfreqs(h,w,n,m,wt) weights the fit-errors versus frequency, where wt is a vector of weighting factors the same length as w.

[b,a] = invfreqs(h,w,n,m,wt,iter) and

[b,a] = invfreqs(h,w,n,m,wt,iter,tol) provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The iter parameter tells invfreqs to end the iteration when the solution has converged, or after iter iterations, whichever comes first. invfreqs defines convergence as occurring when the norm of the (modified) gradient

invfreqs

vector is less than `tol`, where `tol` is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use

```
invfreqs(h,w,n,m,[],iter,tol)
```

`[b,a] = invfreqs(h,w,n,m,wt,iter,tol,'trace')` displays a textual progress report of the iteration.

`[b,a] = invfreqs(h,w,'complex',n,m,...)` creates a complex filter. In this case no symmetry is enforced, and the frequency is specified in radians between -π and π.

Tips   When building higher order models using high frequencies, it is important to scale the frequencies, dividing by a factor such as half the highest frequency present in `w`, so as to obtain well conditioned values of `a` and `b`. This corresponds to a rescaling of time.

Examples   **Example 1**

Convert a simple transfer function to frequency response data and then back to the original filter coefficients:

```
a = [1 2 3 2 1 4]; b = [1 2 3 2 3];
[h,w] = freqs(b,a,64);
[bb,aa] = invfreqs(h,w,4,5)
bb =
    1.0000    2.0000    3.0000    2.0000    3.0000
aa =
    1.0000    2.0000    3.0000    2.0000    1.0000    4.0000
```

Notice that `bb` and `aa` are equivalent to `b` and `a`, respectively. However, `aa` has poles in the right half-plane and thus the system is unstable. Use `invfreqs`'s iterative algorithm to find a stable approximation to the system:

```
[bbb,aaa] = invfreqs(h,w,4,5,[],30)
bbb =
    0.6816    2.1015    2.6694    0.9113    -0.1218
```

```
aaa =
    1.0000   3.4676   7.4060   6.2102   2.5413   0.0001
```

Example 2

Suppose you have two vectors, `mag` and `phase`, that contain magnitude and phase data gathered in a laboratory, and a third vector `w` of frequencies. You can convert the data into a continuous-time transfer function using `invfreqs`:

```
[b,a] = invfreqs(mag.*exp(j*phase),w,2,3);
```

Algorithms   By default, `invfreqs` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b,a} \sum_{k=1}^{n} wt(k)|h(k)A(w(k)) - B(w(k))|^2$$

by creating a system of linear equations and solving them with the MATLAB \ operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the polynomials `a` and `b`, respectively, at the frequency $w(k)$, and n is the number of frequency points (the length of `h` and `w`). This algorithm is based on Levi [1]. Several variants have been suggested in the literature, where the weighting function `wt` gives less attention to high frequencies.

The superior ("output-error") algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points.

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k) - \frac{B(w(k))}{A(w(k))}\right|^2$$

invfreqs

References   [1] Levi, E.C., "Complex-Curve Fitting," *IRE Trans. on Automatic Control*, Vol.AC-4 (1959), pp.37-44.

[2] Dennis, J.E., Jr., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

See Also   freqs | freqz | invfreqz | prony

Purpose   Identify discrete-time filter parameters from frequency response data

Syntax
```
[b,a] = invfreqz(h,w,n,m)
[b,a] = invfreqz(h,w,n,m,wt)
[b,a] = invfreqz(h,w,n,m,wt,iter)
[b,a] = invfreqz(h,w,n,m,wt,iter,tol)
[b,a] = invfreqz(h,w,n,m,wt,iter,tol,'trace')
[b,a] = invfreqz(h,w,'complex',n,m,...)
```

Description   invfreqz is the inverse operation of freqz; it finds a discrete-time
transfer function that corresponds to a given complex frequency
response. From a laboratory analysis standpoint, invfreqz can be used
to convert magnitude and phase data into transfer functions.

[b,a] = invfreqz(h,w,n,m) returns the real numerator and
denominator coefficients in vectors b and a of the transfer function

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1)+b(2)z^{-1}+\cdots+b(n+1)z^{-n}}{a(1)+a(2)z^{-1}+\cdots+a(m+1)z^{-m}}$$

whose complex frequency response is given in vector h at the frequency
points specified in vector w. Scalars n and m specify the desired orders
of the numerator and denominator polynomials.

Frequency is specified in radians between 0 and π, and the length of h
must be the same as the length of w. invfreqz uses conj(h) at -w to
ensure the proper frequency domain symmetry for a real filter.

[b,a] = invfreqz(h,w,n,m,wt) weights the fit-errors versus
frequency, where wt is a vector of weighting factors the same length as w.

[b,a] = invfreqz(h,w,n,m,wt,iter) and

[b,a] = invfreqz(h,w,n,m,wt,iter,tol) provide a superior
algorithm that guarantees stability of the resulting linear system and
searches for the best fit using a numerical, iterative scheme. The iter
parameter tells invfreqz to end the iteration when the solution has
converged, or after iter iterations, whichever comes first. invfreqz
defines convergence as occurring when the norm of the (modified)

gradient vector is less than `tol`, where `tol` is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use

```
invfreqz(h,w,n,m,[],iter,tol)
```

`[b,a] = invfreqz(h,w,n,m,wt,iter,tol,'trace')` displays a textual progress report of the iteration.

`[b,a] = invfreqz(h,w,'complex',n,m,...)` creates a complex filter. In this case no symmetry is enforced, and the frequency is specified in radians between -π and π.

Examples   Convert a simple transfer function to frequency response data and then back to the original filter coefficients:

```
a = [1 2 3 2 1 4]; b = [1 2 3 2 3];
[h,w] = freqz(b,a,64);
[bb,aa] = invfreqz(h,w,4,5)
bb =
    1.0000    2.0000    3.0000    2.0000    3.0000
aa =
    1.0000    2.0000    3.0000    2.0000    1.0000    4.0000
```

Notice that `bb` and `aa` are equivalent to `b` and `a`, respectively. However, `aa` has poles outside the unit circle and thus the system is unstable. Use `invfreqz`'s iterative algorithm to find a stable approximation to the system:

```
[bbb,aaa] = invfreqz(h,w,4,5,[],30)
bbb =
    0.2427    0.2788    0.0069    0.0971    0.1980
aaa =
    1.0000   -0.8944    0.6954    0.9997   -0.8933    0.6949
```

Algorithms   By default, `invfreqz` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b,a} \sum_{k=1}^{n} wt(k) |h(k)A(w(k)) - B(w(k))|^2$$

by creating a system of linear equations and solving them with the MATLAB \ operator. Here $A(\omega(k))$ and $B(\omega(k))$ are the Fourier transforms of the polynomials a and b, respectively, at the frequency $\omega(k)$, and n is the number of frequency points (the length of h and w). This algorithm is a based on Levi [1].

The superior ("output-error") algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points.

$$\min_{b,a} \sum_{k=1}^{n} wt(k) \left| h(k) - \frac{B(w(k))}{A(w(k))} \right|^2$$

References

[1] Levi, E.C., "Complex-Curve Fitting," IRE Trans. on Automatic Control, Vol. AC-4 (1959), pp. 37-44.

[2] Dennis, J.E., Jr., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, 1983.

See Also   freqs | freqz | invfreqz | prony

is2rc

Purpose	Convert inverse sine parameters to reflection coefficients
Syntax	k = is2rc(isin)
Description	k = is2rc(isin) returns a vector of reflection coefficients k from a vector of inverse sine parameters isin.
Examples	isin = [0.2000 0.8727 0.0020 0.0052 -0.0052]; k = is2rc(isin);
References	[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.
See Also	ac2rc | lar2rc | poly2rc | rc2is

Purpose   Kaiser window

Syntax   `w = kaiser(L,beta)`

Description   `w = kaiser(L,beta)` returns an `L`-point Kaiser window in the column vector `w`. `beta` is the Kaiser window parameter that affects the sidelobe attenuation of the Fourier transform of the window. The default value for `beta` is 0.5.

To obtain a Kaiser window that designs an FIR filter with sidelobe attenuation of α dB, use the following ß.

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

Increasing `beta` widens the main lobe and decreases the amplitude of the sidelobes (i.e., increases the attenuation).

Examples   Create a 200-point Kaiser window with a beta of 2.5 and display the result using WVTool:

```
w = kaiser(200,2.5);
wvtool(w)
```

kaiser

References   [1] Kaiser, J.F., "Nonrecursive Digital Filter Design Using the I0- sinh Window Function," Proc. 1974 *IEEE Symp. Circuits and Systems*, (April 1974), pp. 20-23.

[2] *Selected Papers in Digital Signal Processing II*, IEEE Press, New York, 1975.

[3] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 453.

See Also   chebwin | gausswin | kaiserord | tukeywin | window | wintool | wvtool

Purpose   Kaiser window FIR filter design estimation parameters

Syntax   [n,Wn,beta,ftype] = kaiserord(f,a,dev)
[n,Wn,beta,ftype] = kaiserord(f,a,dev,fs)
c = kaiserord(f,a,dev,fs,'cell')

Description   kaiserord returns a filter order n and beta parameter to specify
a Kaiser window for use with the fir1 function. Given a set
of specifications in the frequency domain, kaiserord estimates
the minimum FIR filter order that will approximately meet the
specifications. kaiserord converts the given filter specifications into
passband and stopband ripples and converts cutoff frequencies into the
form needed for windowed FIR filter design.

[n,Wn,beta,ftype] = kaiserord(f,a,dev) finds the approximate
order n, normalized frequency band edges Wn, and weights that meet
input specifications f, a, and dev. f is a vector of band edges and a is a
vector specifying the desired amplitude on the bands defined by f. The
length of f is twice the length of a, minus 2. Together, f and a define
a desired piecewise constant response function. dev is a vector the
same size as a that specifies the maximum allowable error or deviation
between the frequency response of the output filter and its desired
amplitude, for each band. The entries in dev specify the passband
ripple and the stopband attenuation. You specify each entry in dev as a
positive number, representing absolute filter gain (not in decibels).

Note If, in the vector dev, you specify unequal deviations across bands,
the minimum specified deviation is used, since the Kaiser window
method is constrained to produce filters with minimum deviation in
all of the bands.

fir1 can use the resulting order n, frequency vector Wn, multiband
magnitude type ftype, and the Kaiser window parameter beta. The
ftype string is intended for use with fir1; it is equal to 'high' for a
highpass filter and 'stop' for a bandstop filter. For multiband filters,

kaiserord

it can be equal to `'dc-0'` when the first band is a stopband (starting at $f = 0$) or `'dc-1'` when the first band is a passband.

To design an FIR filter `b` that approximately meets the specifications given by `kaiser` parameters `f`, `a`, and `dev`, use the following command.

```
b = fir1(n,Wn,kaiser(n+1,beta),ftype,'noscale')
```

`[n,Wn,beta,ftype] = kaiserord(f,a,dev,fs)` uses a sampling frequency `fs` in Hz. If you don't specify the argument `fs`, or if you specify it as the empty vector `[]`, it defaults to 2 Hz, and the Nyquist frequency is 1 Hz. You can use this syntax to specify band edges scaled to a particular application's sampling frequency. The frequency band edges in `f` must be from 0 to `fs`/2.

`c = kaiserord(f,a,dev,fs,'cell')` is a cell-array whose elements are the parameters to `fir1`.

Note In some cases, `kaiserord` underestimates or overestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1`, `n+2`, and so on, or a try lower order.

Results are inaccurate if the cutoff frequencies are near 0 or the Nyquist frequency, or if `dev` is large (greater than 10%).

Tips   Be careful to distinguish between the meanings of filter length and filter order. The filter *length* is the number of impulse response samples in the FIR filter. Generally, the impulse response is indexed from $n = 0$ to $n = L–1$ where L is the filter length. The filter *order* is the highest power in a z-transform representation of the filter. For an FIR transfer function, this representation is a polynomial in z, where the highest power is $z^{L–1}$ and the lowest power is z^0. The filter order is one less than the length ($L–1$) and is also equal to the number of zeros of the z polynomial.

Examples　**Example 1**

Design a lowpass filter with passband defined from 0 to 1 kHz and stopband defined from 1500 Hz to 4 kHz. Specify a passband ripple of 5% and a stopband attenuation of 40 dB:

```
fsamp = 8000;
fcuts = [1000 1500];
mags = [1 0];
devs = [0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
freqz(hh)
```


Example 2

Design an odd-length bandpass filter (note that odd length means even order, so the input to fir1 must be an even integer):

```
fsamp = 8000;
fcuts = [1000 1300 2210 2410];
mags = [0 1 0];
devs = [0.01 0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
```

kaiserord

```
n = n + rem(n,2);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
[H,f] = freqz(hh,1,1024,fsamp);
plot(f,abs(H)), grid on
```


Example 3

Design a lowpass filter with a passband cutoff of 1500 Hz, a stopband cutoff of 2000 Hz, passband ripple of 0.01, stopband ripple of 0.1, and a sampling frequency of 8000 Hz:

```
[n,Wn,beta,ftype] = kaiserord([1500 2000],[1 0],...
                              [0.01 0.1],8000);
b = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
```

This is equivalent to

```
c = kaiserord([1500 2000],[1 0],[0.01 0.1],8000,'cell');
b = fir1(c{:});
```

Algorithms   kaiserord uses empirically derived formulas for estimating the orders of lowpass filters, as well as differentiators and Hilbert transformers. Estimates for multiband filters (such as bandpass filters) are derived from the lowpass design formulas.

The design formulas that underlie the Kaiser window and its application to FIR filter design are

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

where $\alpha = -20\log_{10}\delta$ is the stopband attenuation expressed in decibels (recall that $\delta_p = \delta_s$ is required).

The design formula is

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where n is the filter order and $\Delta\omega$ is the width of the smallest transition region.

References

[1] Kaiser, J.F., "Nonrecursive Digital Filter Design Using the - sinh Window Function," Proc. 1974 IEEE Symp. *Circuits and Systems*, (April 1974), pp. 20-23.

[2] *Selected Papers in Digital Signal Processing II*, IEEE Press, New York, 1975, pp. 123-126.

[3] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 458-562.

See Also   `fir1` | `kaiser` | `firpmord`

kaiserwin

Purpose   Kaiser window filter from specification object

Syntax   h = design(d,'kaiserwin')
                 h = design(d,'kaiserwin',designoption,value,designoption,...
                 value,...)

Description h = design(d,'kaiserwin') designs a digital filter hd, or a multirate
                 filter hm that uses a Kaiser window. For kaiserwin to work properly,
                 the filter order in the specifications object must be even. In addition,
                 higher order filters (filter order greater than 120) tend to be more
                 accurate for smaller transition widths. kaiserwin returns a warning
                 when your filter order may be too low to design your filter accurately.

                 h =
                 design(d,'kaiserwin',designoption,value,designoption,...
                 value,...) returns a filter where you specify design options as input
                 arguments and the design process uses the Kaiser window technique.

                 To determine the available design options, use designmethods with
                 the specification object and the design method as input arguments as
                 shown.

                 designopts(d,'method')

                 For complete help about using kaiserwin, refer to the command line
                 help system. For example, to get specific information about using
                 kaiserwin with d, the specification object, enter the following at the
                 MATLAB prompt.

                 help(d,'kaiserwin')

Examples   This example designs a direct form FIR filter from a lowpass filter
                 specification object.

                 d=fdesign.lowpass;
                 Hd=design(d,'kaiserwin');
                 fvtool(Hd)

See Also   design | fdesign

lar2rc

Purpose	Convert log area ratio parameters to reflection coefficients
Syntax	k = lar2rc(g)
Description	k = lar2rc(g) returns a vector of reflection coefficients k from a vector of log area ratio parameters g.
Examples	g = [0.6389 4.5989 0.0063 0.0163 -0.0163]; k = lar2rc(g);
References	[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.
See Also	ac2rc	is2rc	poly2rc	rc2lar

Purpose	Convert lattice filter parameters to transfer function form

Syntax

```
[num,den] = latc2tf(k,v)
[num,den] = latc2tf(k,'iiroption')
num = latc2tf(k,'firoption')
```

Description

[num,den] = latc2tf(k,v) finds the transfer function numerator num and denominator den from the IIR lattice coefficients k and ladder coefficients v.

[num,den] = latc2tf(k,'*iiroption*') produces an IIR filter transfer function according to the value of the string '*iiroption*':

- 'allpole': Produces an all-pole filter transfer function from the associated all-pole IIR lattice filter coefficients k.

- 'allpass': Produces an allpass filter transfer function from the associated allpass IIR lattice filter coefficients k.

num = latc2tf(k,'*firoption*') produces an FIR filter according to the value of the string '*firoption*':

- 'min': Produces a minimum-phase FIR filter numerator from the associated minimum-phase FIR lattice filter coefficients k.

- 'max': Produces a maximum-phase FIR filter numerator from the associated maximum-phase FIR lattice filter coefficients k.

- 'FIR': Produces a general FIR filter numerator from the lattice filter coefficients k (default, if you leave off the string altogether).

See Also   latcfilt | tf2latc

latcfilt

Purpose

Lattice and lattice-ladder filter implementation

Syntax

```
[f,g] = latcfilt(k,x)
[f,g] = latcfilt(k,v,x)
[f,g] = latcfilt(k,1,x)
[f,g,zf] = latcfilt(...,'ic',zi)
[f,g,zf] = latcfilt(...,dim)
```

Description

When filtering data, lattice coefficients can be used to represent

- FIR filters

- All-pole IIR filters

- Allpass IIR filters

- General IIR filters

`[f,g] = latcfilt(k,x)` filters x with the FIR lattice coefficients in the vector k. The forward lattice filter result is f and g is the backward filter result. If $|k| \leq 1$, f corresponds to the minimum-phase output, and g corresponds to the maximum-phase output.

If k and x are vectors, the result is a (signal) vector. Matrix arguments are permitted under the following rules:

- If x is a matrix and k is a vector, each column of x is processed through the lattice filter specified by k.

- If x is a vector and k is a matrix, each column of k is used to filter x, and a signal matrix is returned.

- If x and k are both matrices with the same number of columns, then the *i*th column of k is used to filter the *i*th column of x. A signal matrix is returned.

`[f,g] = latcfilt(k,v,x)` filters x with the IIR lattice coefficients k and ladder coefficients v. Both k and v must be vectors, while x can be a signal matrix.

[f,g] = latcfilt(k,1,x) filters x with the IIR lattice specified by k, where k and x can be vectors or matrices. f is the all-pole lattice filter result and g is the allpass filter result.

[f,g,zf] = latcfilt(...,'ic',zi) accepts a length-k vector zi specifying the initial condition of the lattice states. Output zf is a length-k vector specifying the final condition of the lattice states.

[f,g,zf] = latcfilt(...,dim) filters x along the dimension dim. To specify a dim value, the FIR lattice coefficients k must be a vector and you must specify all previous input parameters in order. Use the empty vector [ ] for any parameters you do not want to specify. zf returns the final conditions in columns, regardless of the shape of x.

Examples   Filter data with an FIR lattice filter:

```
%create data
x=randn(512,1);
%reflection coefficients for 3-point MA filter
[f,g]=latcfilt([1/2 1],x);
%compare f vector to dfilt.latticemamin output
Hd=dfilt.latticemamin([1/2 1]);
y=filter(Hd,x);
isequal(y,f) %returns 1
%compare g vector to dfilt.latticemamax output
Hd1=dfilt.latticemamax([1/2 1]);
y1=filter(Hd1,x);
isequal(g,y1) %returns 1
```

See Also   dfilt.latticemamax | dfilt.latticemamin | filter

levinson

Purpose	Levinson-Durbin recursion

Syntax	`a = levinson(r)`
	`a = levinson(r,n)`
	`[a,e] = levinson(r,n)`
	`[a,e,k] = levinson(r,n)`

Description The Levinson-Durbin recursion is an algorithm for finding an all-pole IIR filter with a prescribed deterministic autocorrelation sequence. It has applications in filter design, coding, and spectral estimation. The filter that `levinson` produces is minimum phase.

`a = levinson(r)` finds the coefficients of a `length(r)-1` order autoregressive linear process which has `r` as its autocorrelation sequence. `r` is a real or complex deterministic autocorrelation sequence. If `r` is a matrix, `levinson` finds the coefficients for each column of `r` and returns them in the rows of `a`. `n=length(r)-1` is the default order of the denominator polynomial $A(z)$; that is, `a = [1 a(2) ... a(n+1)]`. The filter coefficients are ordered in descending powers of z^{-1}.

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

`a = levinson(r,n)` returns the coefficients for an autoregressive model of order n.

`[a,e] = levinson(r,n)` returns the prediction error, e, of order n.

`[a,e,k] = levinson(r,n)` returns the reflection coefficients k as a column vector of length n.

Note k is computed internally while computing the a coefficients, so returning k simultaneously is more efficient than converting a to k with `tf2latc`.

Algorithms   `levinson` solves the symmetric Toeplitz system of linear equations

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(n)^* \\ r(2) & r(1) & \cdots & r(n-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(n) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

where `r = [r(1) ... r(n+1)]` is the input autocorrelation vector, and $r(i)^*$ denotes the complex conjugate of $r(i)$. The input `r` is typically a vector of autocorrelation coefficients where lag 0 is the first element $r(1)$. The algorithm requires $O(n^2)$ flops and is thus much more efficient than the MATLAB \ command for large `n`. However, the `levinson` function uses \ for low orders to provide the fastest possible execution.

References   [1] Ljung, L., *System Identification: Theory for the User*, Prentice-Hall, 1987, pp. 278-280.

See Also   `lpc` | `prony` | `rlevinson` | `schurrc` | `stmcb`

lp2bp

Purpose	Transform lowpass analog filters to bandpass

Syntax

```
[bt,at] = lp2bp(b,a,Wo,Bw)
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)
```

Description   lp2bp transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandpass filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the butter, cheby1, cheby2, and ellip functions.

lp2bp can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

[bt,at] = lp2bp(b,a,Wo,Bw) transforms an analog lowpass filter prototype given by polynomial coefficients into a bandpass filter with center frequency Wo and bandwidth Bw. Row vectors b and a specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*.

$$\frac{B(s)}{A(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalars Wo and Bw specify the center frequency and bandwidth in units of rad/s. For a filter with lower band edge w1 and upper band edge w2, use Wo = sqrt(w1*w2) and Bw = w2-w1.

lp2bp returns the frequency transformed filter in row vectors bt and at.

State-Space Form

[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw) converts the continuous-time state-space lowpass filter prototype in matrices A, B, C, D shown below

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

into a bandpass filter with center frequency `Wo` and bandwidth `Bw`. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

The bandpass filter is returned in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithms   `lp2bp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. Consider the state-space system

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where u is the input, x is the state vector, and y is the output. The Laplace transform of the first equation (assuming zero initial conditions) is

$$sX(s) = AX(s) + BU(s)$$

Now if a bandpass filter is to have center frequency ω_0 and bandwidth B_w, the standard s-domain transformation is

$$s = Q(p^2 + 1)/p$$

where $Q = \omega_0/B_w$ and $p = s/\omega_0$. Substituting this for s in the Laplace transformed state-space equation, and considering the operator p as d/dt results in

$$Q\ddot{x} + Qx = \dot{A}x + B\dot{u}$$

or

$$Q\ddot{x} - \dot{A}x - B\dot{u} = -Qx$$

Now define

$$Q\dot{\omega} = -Qx$$

which, when substituted, leads to

$$Q\dot{x} = Ax + Q\omega + Bu$$

The last two equations give equations of state. Write them in standard form and multiply the differential equations by ω_0 to recover the time/frequency scaling represented by p and find state matrices for the bandpass filter:

```
Q = Wo/Bw; [ma,m] = size(A);
At = Wo*[A/Q eye(ma,m);-eye(ma,m) zeros(ma,m)];
Bt = Wo*[B/Q; zeros(ma,n)];
Ct = [C zeros(mc,ma)];
Dt = d;
```

If the input to lp2bp is in transfer function form, the function transforms it into state-space form before applying this algorithm.

See Also   bilinear | impinvar | lp2bs | lp2hp | lp2lp

Purpose   Transform lowpass analog filters to bandstop

Syntax   ```
[bt,at] = lp2bs(b,a,Wo,Bw)
[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw)
```

Description lp2bs transforms analog lowpass filter prototypes with a cutoff angular
frequency of 1 rad/s into bandstop filters with desired bandwidth and
center frequency. The transformation is one step in the digital filter
design process for the butter, cheby1, cheby2, and ellip functions.

lp2bs can perform the transformation on two different linear system
representations: transfer function form and state-space form. In both
cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

[bt,at] = lp2bs(b,a,Wo,Bw) transforms an analog lowpass filter
prototype given by polynomial coefficients into a bandstop filter with
center frequency Wo and bandwidth Bw. Row vectors b and a specify
the coefficients of the numerator and denominator of the prototype in
descending powers of *s*.

$$\frac{B(s)}{A(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalars Wo and Bw specify the center frequency and bandwidth in units
of radians/second. For a filter with lower band edge w1 and upper band
edge w2, use Wo = sqrt(w1*w2) and Bw = w2-w1.

lp2bs returns the frequency transformed filter in row vectors bt and at.

State-Space Form

[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw) converts the
continuous-time state-space lowpass filter prototype in matrices A, B, C,
D shown below

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

lp2bs

into a bandstop filter with center frequency Wo and bandwidth Bw. For a filter with lower band edge w1 and upper band edge w2, use Wo = sqrt(w1*w2) and Bw = w2-w1.

The bandstop filter is returned in matrices At, Bt, Ct, Dt.

Algorithms lp2bs is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a bandstop filter is to have center frequency ω_0 and bandwidth B_w, the standard *s*-domain transformation is

$$s = \frac{p}{Q(p^2 + 1)}$$

where $Q = \omega_0/B_w$ and $p = s/\omega_0$. The state-space version of this transformation is

```
Q = Wo/Bw;
At = [Wo/Q*inv(A) Wo*eye(ma);-Wo*eye(ma) zeros(ma)];
Bt = -[Wo/Q*(A\B); zeros(ma,n)];
Ct = [C/A zeros(mc,ma)];
Dt = D - C/A*B;
```

See lp2bp for a derivation of the bandpass version of this transformation.

See Also bilinear | impinvar | lp2bp | lp2hp | lp2lp

Purpose   Transform lowpass analog filters to highpass

Syntax   [bt,at] = lp2hp(b,a,Wo)
                  [At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo)

Description   lp2hp transforms analog lowpass filter prototypes with a cutoff angular
                  frequency of 1 rad/s into highpass filters with desired cutoff angular
                  frequency. The transformation is one step in the digital filter design
                  process for the butter, cheby1, cheby2, and ellip functions.

                  The lp2hp function can perform the transformation on two different
                  linear system representations: transfer function form and state-space
                  form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

[bt,at] = lp2hp(b,a,Wo) transforms an analog lowpass filter
prototype given by polynomial coefficients into a highpass filter with
cutoff angular frequency Wo. Row vectors b and a specify the coefficients
of the numerator and denominator of the prototype in descending
powers of s.

$$\frac{B(s)}{A(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalar Wo specifies the cutoff angular frequency in units of
radians/second. The frequency transformed filter is returned in row
vectors bt and at.

State-Space Form

[At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo) converts the continuous-time
state-space lowpass filter prototype in matrices A, B, C, D below

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

into a highpass filter with cutoff angular frequency Wo. The highpass
filter is returned in matrices At, Bt, Ct, Dt.

lp2hp

Algorithms   lp2hp is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a highpass filter is to have cutoff angular frequency ω_0, the standard *s*-domain transformation is

$$s = \frac{\omega_0}{p}$$

The state-space version of this transformation is

```
At = Wo*inv(A);
Bt = -Wo*(A\B);
Ct = C/A;
Dt = D - C/A*B;
```

See lp2bp for a derivation of the bandpass version of this transformation.

See Also   bilinear | impinvar | lp2bp | lp2bs | lp2lp

Purpose

Change cutoff frequency for lowpass analog filter

Syntax

```
[bt,at] = lp2lp(b,a,Wo)
[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)
```

Description

`lp2lp` transforms an analog lowpass filter prototype with a cutoff angular frequency of 1 rad/s into a lowpass filter with any specified cutoff angular frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

The `lp2lp` function can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2lp(b,a,Wo)` transforms an analog lowpass filter prototype given by polynomial coefficients into a lowpass filter with cutoff angular frequency Wo. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*.

$$\frac{B(s)}{A(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalar Wo specifies the cutoff angular frequency in units of radians/second. `lp2lp` returns the frequency transformed filter in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)` converts the continuous-time state-space lowpass filter prototype in matrices A, B, C, D below

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

into a lowpass filter with cutoff angular frequency `Wo`. `lp2lp` returns the lowpass filter in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithms   `lp2lp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a lowpass filter is to have cutoff angular frequency ω_0, the standard *s*-domain transformation is

$$s = p / \omega_0$$

The state-space version of this transformation is

```
At = Wo*A;
Bt = Wo*B;
Ct = C;
Dt = D;
```

See `lp2bp` for a derivation of the bandpass version of this transformation.

See Also   `bilinear | impinvar | lp2bp | lp2bs | lp2hp`

Purpose   Linear prediction filter coefficients

Syntax   `[a,g] = lpc(x,p)`

Description   `lpc` determines the coefficients of a forward linear predictor by minimizing the prediction error in the least squares sense. It has applications in filter design and speech coding.

`[a,g] = lpc(x,p)` finds the coefficients of a `pth`-order linear predictor (FIR filter) that predicts the current value of the real-valued time series `x` based on past samples.

$$\hat{x}(n) = -a(2)x(n-1) - a(3)x(n-2) - \cdots - a(p+1)x(n-p)$$

`p` is the order of the prediction filter polynomial, `a = [1 a(2) ... a(p+1)]`. If `p` is unspecified, `lpc` uses as a default `p = length(x)-1`. If `x` is a matrix containing a separate signal in each column, `lpc` returns a model estimate for each column in the rows of matrix `a` and a column vector of prediction error variances `g`. The length of `p` must be less than or equal to the length of `x`.

Examples   Estimate a data series using a third-order forward predictor, and compare to the original signal.

First, create the signal data as the output of an autoregressive process driven by white noise. Use the last 4096 samples of the AR process output to avoid start-up transients:

```
noise = randn(50000,1);  % Normalized white Gaussian noise
x = filter(1,[1 1/2 1/3 1/4],noise);
x = x(45904:50000);
```

Compute the predictor coefficients, estimated signal, prediction error, and autocorrelation sequence of the prediction error:

```
a = lpc(x,3);
est_x = filter([0 -a(2:end)],1,x);    % Estimated signal
e = x - est_x;                         % Prediction error
```

```
[acs,lags] = xcorr(e,'coeff');   % ACS of prediction error
```

The prediction error, $e(n)$, can be viewed as the output of the prediction error filter $A(z)$ shown below, where $H(z)$ is the optimal linear predictor, $x(n)$ is the input signal, and $\hat{x}(n)$ is the predicted signal.
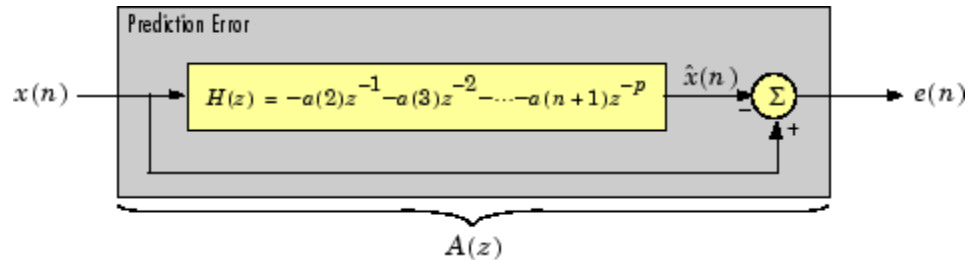
Compare the predicted signal to the original signal.

```
plot(1:97,x(4001:4097),1:97,est_x(4001:4097),'--');
title('Original Signal vs. LPC Estimate');
xlabel('Sample Number'); ylabel('Amplitude'); grid;
legend('Original Signal','LPC Estimate')
```

Look at the autocorrelation of the prediction error:

```
plot(lags,acs);
title('Autocorrelation of the Prediction Error');
xlabel('Lags'); ylabel('Normalized Value'); grid;
```

The prediction error is approximately white Gaussian noise, as expected for a third-order AR input process.

Algorithms   lpc uses the autocorrelation method of autoregressive (AR) modeling to find the filter coefficients. The generated filter might not model the process exactly even if the data sequence is truly an AR process of the correct order. This is because the autocorrelation method implicitly windows the data, that is, it assumes that signal samples beyond the length of x are 0.

lpc computes the least squares solution to

$$Xa = b$$

where

$$X = \begin{bmatrix} x(1) & 0 & \cdots & 0 \\ x(2) & x(1) & \ddots & \vdots \\ \vdots & x(2) & \ddots & 0 \\ x(m) & \vdots & \ddots & x(1) \\ 0 & x(m) & \ddots & x(2) \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & x(m) \end{bmatrix}, \quad a = \begin{bmatrix} 1 \\ a(2) \\ \vdots \\ a(p+1) \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

and m is the length of x. Solving the least squares problem via the normal equations

$$X^H X a = X^H b$$

leads to the Yule-Walker equations

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \ddots & \vdots \\ \vdots & \ddots & \ddots & r(2)^* \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

where $r = [r(1)\ r(2)\ ...\ r(p+1)]$ is an autocorrelation estimate for x computed using xcorr. The Yule-Walker equations are solved in $O(p^2)$ flops by the Levinson-Durbin algorithm (see levinson).

References   [1] Jackson, L.B., *Digital Filters and Signal Processing, Second Edition,* Kluwer Academic Publishers, 1989. pp. 255-257.

See Also   aryule | levinson | prony | pyulear | stmcb

lsf2poly

Purpose　　　　Convert line spectral frequencies to prediction filter coefficients

Syntax　　　　`a = lsf2poly(lsf)`

Description　　`a = lsf2poly(lsf)` returns a vector `a` containing the prediction filter coefficients from the vector `lsf` of line spectral frequencies. If `lsf` is a matrix of size MxN with separate channels of line spectral frequencies in each column, the returned `a` matrix has the resulting prediction filter coefficients in rows and is of size Nx(M+1).

Examples
```
lsf = [0.7842    1.5605    1.8776    1.8984    2.3593];
a = lsf2poly(lsf)
a =
    1.0000   0.6148   0.9899   0.0001   0.0031  -0.0081
```

References　　[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.

[2] Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

See Also　　　ac2poly | poly2lsf | rc2poly

Purpose   Convert magnitude to decibels (dB)

Syntax   ydb = mag2db(y)

Description   ydb = mag2db(y) returns the corresponding decibel (dB) value ydb for
                  a given magnitude y. The relationship between magnitude and decibels
                  is ydb = $20*\log_{10}(y)$.

See Also   db2mag

marcumq

Purpose   Generalized Marcum Q function

Syntax   Q = marcumq(a,b)
                     Q = marcumq(a,b,m)

Description   Q = marcumq(a,b) computes the Marcum Q function of a and b, defined
                     by

$$Q(a,b) = \int_b^\infty x \exp\left(-\frac{(x^2 + a^2)}{2}\right) I_0(ax)dx$$

where a and b are nonnegative real numbers. In this expression, I_0 is
the modified Bessel function of the first kind of zero order.

Q = marcumq(a,b,m) computes the generalized Marcum Q, defined by

$$Q\ (a,b) = \frac{1}{a^{m-1}} \int_b^\infty x^m \exp\left(-\frac{(x^2 + a^2)}{2}\right) I_{m-1}(ax)dx$$

where a and b are nonnegative real numbers, and m is a positive integer.
In this expression, I_{m-1} is the modified Bessel function of the first kind
of order m-1.

If any of the inputs is a scalar, it is expanded to the size of the other
inputs.

Algorithms   marcumq uses the algorithm developed in [3]. The paper describes two
                    error criteria: a relative error criterion and an absolute error criterion.
                    marcumq utilizes the absolute error criterion.

References   [1] Cantrell, P. E., and A. K. Ojha, "Comparison of Generalized
                    Q-Function Algorithms," *IEEE Transactions on Information Theory*,
                    Vol. IT-33, July, 1987, pp. 591–596.

[2] Marcum, J. I., "A Statistical Theory of Target Detection by Pulsed Radar: Mathematical Appendix," RAND Corporation, Santa Monica, CA, Research Memorandum RM-753, July 1, 1948. Reprinted in *IRE Transactions on Information Theory*, Vol. IT-6, April, 1960, pp. 59–267.

[3] Shnidman, D. A., "The Calculation of the Probability of Detection and the Generalized Marcum Q-Function," *IEEE Transactions on Information Theory*, Vol. IT-35, March, 1989, pp. 389–400.

See Also   besseli

maxflat

Purpose	Generalized digital Butterworth filter design

Syntax	`[b,a] = maxflat(n,m,Wn)`
	`b = maxflat(n,'sym',Wn)`
	`[b,a,b1,b2] = maxflat(n,m,Wn)`
	`[b,a,b1,b2,sos,g] = maxflat(n,m,Wn)`
	`[...] = maxflat(n,m,Wn,'design_flag')`

Description `[b,a] = maxflat(n,m,Wn)` is a lowpass Butterworth filter with numerator and denominator coefficients b and a of orders n and m respectively. Wn is the normalized cutoff frequency at which the magnitude response of the filter is equal to $1/\sqrt{2}$ (approx. -3 dB). Wn must be between 0 and 1, where 1 corresponds to the Nyquist frequency.

`b = maxflat(n,'sym',Wn)` is a symmetric FIR Butterworth filter. n must be even, and Wn is restricted to a subinterval of [0,1]. The function raises an error if Wn is specified outside of this subinterval.

`[b,a,b1,b2] = maxflat(n,m,Wn)` returns two polynomials b1 and b2 whose product is equal to the numerator polynomial b (that is, `b = conv(b1,b2)`). b1 contains all the zeros at `z = -1`, and b2 contains all the other zeros.

`[b,a,b1,b2,sos,g] = maxflat(n,m,Wn)` returns the second-order sections representation of the filter as the filter matrix sos and the gain g.

`[...] = maxflat(n,m,Wn,'design_flag')` enables you to monitor the filter design, where '*design_flag*' is

- `'trace'` for a textual display of the design table used in the design

- `'plots'` for plots of the filter's magnitude, group delay, and zeros and poles

- `'both'` for both the textual display and plots

Examples
```
n = 10; m = 2; Wn = 0.2;
[b,a] = maxflat(n,m,Wn)
```

```
fvtool(b,a)                    % Display the magnitude plot
```
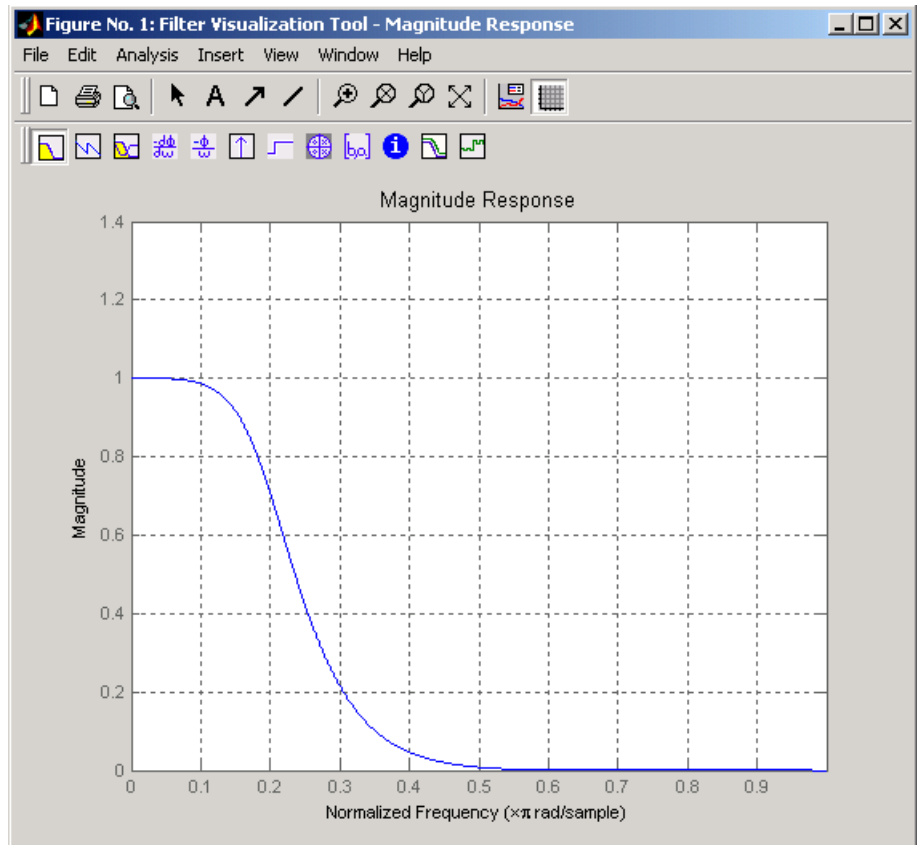

Algorithms   The method consists of the use of formulae, polynomial root finding, and a transformation of polynomial roots.

References   [1] Selesnick, I.W., and C.S. Burrus, "Generalized Digital Butterworth Filter Design," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing, Vol. 3* (May 1996).

maxflat

butter | filter | freqz

medfilt1

Purpose   1-D median filtering

Syntax   ```
                   y = medfilt1(x,n)
                   y = medfilt1(x,n,blksz)
                   y = medfilt1(x,n,blksz,dim)
                   ```

**Description**    `y = medfilt1(x,n)` applies an order `n` one-dimensional median filter to
                   vector `x`; the function considers the signal to be 0 beyond the end points.
                   Output `y` has the same length as `x`.

                   For `n` odd, `y(k)` is the median of `x(k-(n-1)/2:k+(n-1)/2)`.

                   For `n` even, `y(k)` is the median of `x(k-n/2)`, `x(k-(n/2)+1)`, ...,
                   `x(k+(n/2)-1)`. In this case, `medfilt1` sorts the numbers, then takes
                   the average of the `n/2` and `(n/2)+1` elements.

                   The default for `n` is 3.

                   `y = medfilt1(x,n,blksz)` uses a `for`-loop to compute `blksz` (block
                   size) output samples at a time. Use `blksz << length(x)` if you are low
                   on memory, since `medfilt1` uses a working matrix of size `n`-by-`blksz`.
                   By default, `blksz = length(x)`; this provides the fastest execution if
                   you have sufficient memory.

                   If `x` is a matrix, `medfilt1` median filters its columns using

                    `y(:,i) = medfilt1(x(:,i),n,blksz)`

                   in a loop over the columns of `x`.

                   `y = medfilt1(x,n,blksz,dim)` specifies the dimension, `dim`, along
                   which the filter operates.

**References**     [1] Pratt, W.K., *Digital Image Processing*, John Wiley & Sons, 1978,
                   pp. 330-333.

**See Also**       filter | medfilt2 | median

# midcross

**Purpose**    Mid-reference level crossing for bilevel waveform

**Syntax**
```
C = midcross(X)
C = midcross(X,FS)
C = midcross(X,T)
[C,MIDLEV] = midcross(...)
C = midcross(X,Name,Value)
midcross(...)
```

**Description**    `C = midcross(X)` returns a vector, `C`, of time instants where each transition of the input signal, `X`, crosses the 50% reference level. The sample instants correspond to the indices of the input vector. Because `midcross` uses interpolation to determine the crossing instant, `C` may contain values that do not correspond to sampling instants. To determine the transitions, `midcross` estimates the state levels of `X` by a histogram method. `midcross` identifies all intervals which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See "State-Level Tolerances" on page 3-660.

`C = midcross(X,FS)` specifies the sample rate, `FS`, in hertz as a positive scalar. The first sample instant corresponds to t=0. Because `midcross` uses interpolation to determine the crossing instant, `C` may contain values that do not correspond to sampling instants.

`C = midcross(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`. Because `midcross` uses interpolation to determine the crossing instant, `C` may contain values that do not correspond to sampling instants.

`[C,MIDLEV] = midcross(...)` returns the waveform value corresponding to the mid-reference level.

`C = midcross(X,Name,Value)` returns the time instants corresponding to mid-reference level crossings with additional options specified by one or more `Name,Value` pair arguments.

midcross(...) plots the signal and marks the location of the mid-crossings (mid-reference level instants) and the associated reference levels. midcross also plots the state levels with upper and lower state boundaries.

**Input Arguments**

**X**

Bilevel waveform. X is a real-valued row or column vector.

**FS**

Sample rate in hertz.

**T**

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

**Name-Value Pair Arguments**

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

**MidPct**

Mid-reference level as a percentage of the waveform amplitude.

> **Default:** 50

**StateLevels**

Low and high state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low state level. The second element is the high state level. If you do not specify low- and high-state levels, midcross estimates the state levels from the input waveform using the histogram method.

**Tolerance**

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-660.

> **Default:** 2

**Output Arguments**

**C**

Time instants of the mid-reference level crossings.

**MIDLEV**

Mid-reference level.

**Definitions**

### Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level, *S_1*, and high–state level, *S_2*, is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

### Mid Reference Level Instant

Let $y_{50\%}$ denote the mid–reference level.

Let $t_{50\%-}$ and $t_{50\%+}$ denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.

Let $y_{50\%-}$ and $y_{50\%+}$ denote the waveform values at $t_{50\%-}$ and $t_{50\%+}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

### State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a

scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \tfrac{\alpha}{100}(S_2 - S_1)$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

State Levels ± 2% Tolerance

**Examples**     **Mid-Reference Level Instant of Bilevel Waveform**

Assuming a sampling interval of 1, compute the mid-reference level instant of a bilevel waveform and plot the result.

```
load('transitionex.mat', 'x');
C = midcross(x);
plot(x); hold on;
plot([C C],[-0.5 2.5],'r','linewidth',2);
```


The instant at which the waveform crosses the 50% reference level is 21.5. Note that this is not a sampling instant present in the input vector because midcross uses interpolation to identify the mid-reference level crossing.

Mid-Reference Level Instant with Sampling Frequency

Compute the mid-reference level instant using the sampling rate for a bilevel waveform sampled at 4 MHz.

```
load('transitionex.mat','x','t');
Fs = 1/(t(2)-t(1));
C = midcross(x,Fs);
```

Mid Reference Level Instant Using Sample Instants

Compute the mid-reference level instants using a vector of sample times equal in length to the bilevel waveform. The sampling rate is 4 MHz.

```
load('transitionex.mat','x','t');
C = midcross(x,t);
```

Mid-Reference Level Value of Bilevel Waveform

Compute the level corresponding to the mid-reference level instant. Plot the result.

```
load('transitionex.mat','x','t');
[C,MIDLEV] = midcross(x,t);
plot(t,x); hold on;
plot([C C],[-0.5 2.5],'r','linewidth',2);
plot([0 t(end)],[MIDLEV MIDLEV],'r','linewidth',2);
axis tight;
```

60% Reference Level Instant and Waveform Value

Obtain the 60% reference level instant and value for a bilevel waveform.

```
load('transitionex.mat','x','t');
[C,Lev60] = midcross(x,t,'MidPct',60);
```

References [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003. p. 20.

midcross

See Also   falltime | pulsewidth | risetime | settlingtime | statelevels

modulate

Purpose   Modulation for communications simulation

Syntax   y = modulate(x,fc,fs,'*method*')
                 y = modulate(x,fc,fs,'*method*',opt)
                 [y,t] = modulate(x,fc,fs)

Description y = modulate(x,fc,fs,'*method*') and

y = modulate(x,fc,fs,'*method*',opt) modulate the real message signal x with a carrier frequency fc and sampling frequency fs, using one of the options listed below for '*method*'. Note that some methods accept an option, opt.

Note Use modulate and demod in the Signal Processing Toolbox with real-valued signals to obtain real-valued outputs. modulate and demod are not intended to accept complex-valued inputs or produce complex-valued outputs.

Method	Description
amdsb-sc or am	Amplitude modulation, double sideband, suppressed carrier. Multiplies x by a sinusoid of frequency fc. y = x.*cos(2*pi*fc*t)
amdsb-tc	Amplitude modulation, double sideband, transmitted carrier. Subtracts scalar opt from x and multiplies the result by a sinusoid of frequency fc. y = (x-opt).*cos(2*pi*fc*t) If the opt parameter is not present, modulate uses a default of min(min(x)) so that the message signal (x-opt) is entirely nonnegative and has a minimum value of 0.

modulate

Method	Description
amssb	Amplitude modulation, single sideband. Multiplies x by a sinusoid of frequency fc and adds the result to the Hilbert transform of x multiplied by a phase shifted sinusoid of frequency fc. `y = x.*cos(2*pi*fc*t)+imag(hilbert(x)).*sin(2*pi*fc*t)` This effectively removes the upper sideband.

fm	Frequency modulation. Creates a sinusoid with instantaneous frequency that varies with the message signal x.
	
	`y = cos(2*pi*fc*t + opt*cumsum(x))`
	
	cumsum is a rectangular approximation to the integral of x. modulate uses opt as the constant of frequency modulation. If opt is not present, modulate uses a default of
	
	`opt = (fc/fs)*2*pi/(max(max(x)))`
	
	so the maximum frequency excursion from fc is fc Hz.

pm	Phase modulation. Creates a sinusoid of frequency fc whose phase varies with the message signal x.
	
	`y = cos(2*pi*fc*t + opt*x)`
	
	modulate uses opt as the constant of phase modulation. If opt is not present, modulate uses a default of
	
	`opt = pi/(max(max(x)))`
	
	so the maximum phase excursion is π radians.

Method	Description
pwm	Pulse-width modulation. Creates a pulse-width modulated signal from the pulse widths in x. The elements of x must be between 0 and 1, specifying the width of each pulse in fractions of a period. The pulses start at the beginning of each period, that is, they are left justified. `modulate(x,fc,fs,'pwm','centered')` yields pulses centered at the beginning of each period. y is length `length(x)*fs/fc`.
ppm	Pulse-position modulation. Creates a pulse-position modulated signal from the pulse positions in x. The elements of x must be between 0 and 1, specifying the left edge of each pulse in fractions of a period. opt is a scalar between 0 and 1 that specifies the length of each pulse in fractions of a period. The default for opt is 0.1. y is length `length(x)*fs/fc`.
qam	Quadrature amplitude modulation. Creates a quadrature amplitude modulated signal from signals x and opt. `y = x.*cos(2*pi*fc*t) + opt.*sin(2*pi*fc*t)` opt must be the same size as x.

If you do not specify '*method*', then modulate assumes am. Except for the pwm and ptm cases, y is the same size as x.

If x is an array, modulate modulates its columns.

`[y,t] = modulate(x,fc,fs)` returns the internal time vector t that modulate uses in its computations.

See Also   demod | vco

mscohere

Purpose   Magnitude squared coherence

Syntax
```
Cxy = mscohere(x,y)
Cxy = mscohere(x,y,window)
Cxy = mscohere(x,y,window,noverlap)
[Pxy,W] = mscohere(x,y,window,noverlap,nfft)
[Cxy,F] = mscohere(x,y,window,noverlap,nfft,fs)
[...] = mscohere(x,y,...,'twosided')
mscohere(...)
```

Description   Cxy = mscohere(x,y) finds the magnitude squared coherence estimate
Cxy of the input signals x and y using Welch's averaged, modified
periodogram method. The magnitude squared coherence estimate is a
function of frequency with values between 0 and 1 that indicates how
well x corresponds to y at each frequency. The magnitude squared
coherence is a function of the power spectral densities (P_{xx}(f) and P_{yy}(f))
of x and y and the cross power spectral density (P_{xy}(f)) of x and y.

$$C_{xy}(f) = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)}$$

x and y must be the same length. For real x and y, mscohere returns
a one-sided coherence estimate and for complex x or y, it returns a
two-sided estimate.

mscohere uses the following default values:

Parameter	Description	Default Value
nfft	FFT length which determines the frequencies at which the coherence is estimated For real x and y, the length of Cxy is (nfft/2+1) if nfft is even or (nfft+1)/2 if nfft is odd. For complex x or y, the length of Cxy is nfft. If nfft is greater than the signal length, the data is zero-padded. If nfft is less than the signal length, the segment is wrapped using datawrap so that the length is equal to nfft.	Maximum of 256 or the next power of 2 greater than the length of each section of x or y
fs	Sampling frequency	1
window	Windowing function and number of samples to use for each section	Periodic Hamming window of length to obtain eight equal sections of x and y
noverlap	Number of samples by which the sections overlap	Value to obtain 50% overlap

Note You can use the empty matrix [] to specify the default value for any input argument except x or y. For example, Pxy = mschoere(x,y,[],[],128) uses a Hamming window, default noverlap to obtain 50% overlap, and the specified 128 nfft.

`Cxy = mscohere(x,y,window)` specifies a windowing function, divides x and y into equal overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for `window`, `Cxy` uses a Hamming window of that length. `mscohere` zero pads the sections if the window length exceeds `nfft`.

`Cxy = mscohere(x,y,window,noverlap)` overlaps the sections of x by `noverlap` samples. `noverlap` must be an integer smaller than the length of `window`.

`[Pxy,W] = mscohere(x,y,window,noverlap,nfft)` uses the specified FFT length `nfft` to calculate the coherence estimate. It also returns `W`, which is the vector of normalized frequencies (in rad/sample) at which the coherence is estimated. For real x and y, `Cxy` length is (`nfft`/2 +1) if `nfft` is even and if `nfft` is odd, the length is (`nfft`+1)/2. For complex x or y, the length of `Cxy` is `nfft`. For real signals, the range of `W` is [0, pi] when `nfft` is even and [0, pi) when `nfft` is odd. For complex signals, the range of `W` is [0, 2*pi).

`[Cxy,F] = mscohere(x,y,window,noverlap,nfft,fs)` returns `Cxy` as a function of frequency and a vector `F` of frequencies at which the coherence is estimated. `fs` is the sampling frequency in Hz. For real signals, the range of `F` is [0, `fs`/2] when `nfft` is even and [0, `fs`/2) when `nfft` is odd. For complex signals, the range of `F` is [0, `fs`).

`[...] = mscohere(x,y,...,'twosided')` returns a coherence estimate with frequencies that range over the whole Nyquist interval. Specifying `'onesided'` uses half the Nyquist interval.

`mscohere(...)` plots the magnitude squared coherence versus frequency in the current figure window.

Note If you estimate the magnitude squared coherence with a single window, or section, the value is identically 1 for all frequencies [1] . You must use at least two sections.

Examples   Compute and plot the coherence estimate between two colored noise
sequences x and y:

```
rng default;
h = fir1(30,0.2,rectwin(31));
h1 = ones(1,10)/sqrt(10);
r = randn(16384,1);
x = filter(h1,1,r);
y = filter(h,1,x);
mscohere(x,y,hanning(1024),512,1024)
```

mscohere

Algorithms mscohere estimates the magnitude squared coherence function [2] using Welch's overlapped averaged periodogram method (see references [3] and [4]).

References [1] Stoica, P., and R. Moses. *Introduction to Spectral Analysis.* Upper Saddle River, NJ: Prentice-Hall, 2005. pp. 67-68.

[2] Kay, S.M. *Modern Spectral Estimation.* Englewood Cliffs, NJ: Prentice-Hall, 1988. pp. 453-455.

[3] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1975.

[4] Welch, P.D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." *IEEE Trans. Audio Electroacoust. Vol. AU-15 (June 1967)*, pp. 70-73.

See Also cpsd | periodogram | pwelch | spectrum | tfestimate

Purpose   Nuttall-defined minimum 4-term Blackman-Harris window

Syntax   w = nuttallwin(N)
                w = nuttalwin(N,SFLAG)

Description w = nuttallwin(N) returns a Nuttall defined N-point, 4-term
symmetric Blackman-Harris window in the column vector w. The
window is minimum in the sense that its maximum sidelobes
are minimized. The coefficients for this window differ from the
Blackman-Harris window coefficients computed with blackmanharris
and produce slightly lower sidelobes.

w = nuttalwin(N,SFLAG) uses SFLAG window sampling. SFLAG can
be 'symmetric' or 'periodic'. The default is 'symmetric'. You can
find the equations defining the symmetric and periodic windows in
"Definitions" on page 3-676.

Examples   Compare 64-point Blackman-Harris and Nuttall's Blackman-Harris
windows and plot them using WVTool:

```
L = 64;
w = blackmanharris(L);
y = nuttallwin(L);
wvtool(w,y)
```

The maximum difference between the two windows is

```
max(abs(y-w))
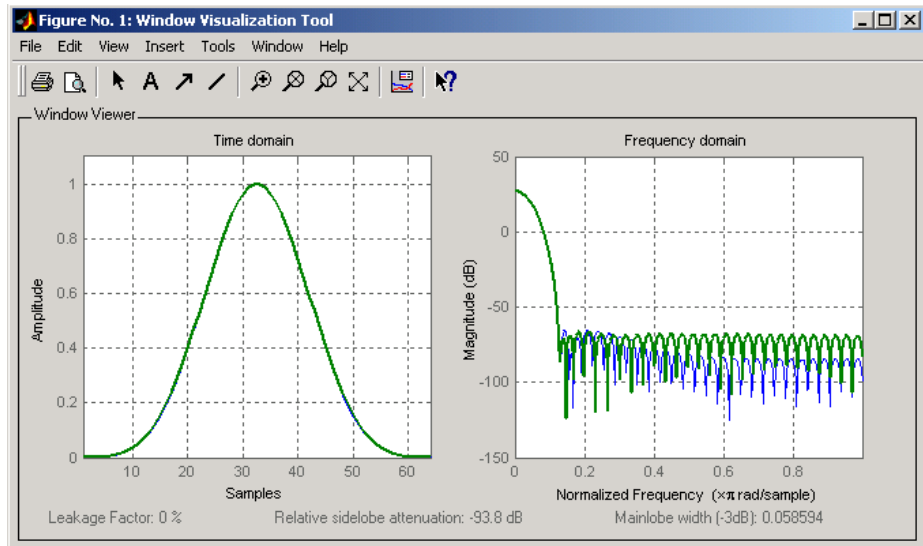
ans =

    0.0099
```

Definitions   The equation for the **symmetric** Nuttall defined 4-term Blackman-Harris window is

$$w(n) = a_0 - a_1 \cos\left(2\pi\frac{n}{N-1}\right) + a_2 \cos\left(4\pi\frac{n}{N-1}\right) - a_3 \cos\left(6\pi\frac{n}{N-1}\right)$$

where *n*= 0,1,2, ... *N*-1.

The equation for the **periodic** Nuttall defined 4-term Blackman-Harris window is

$$w(n) = a_0 - a_1 \cos\left(2\pi \frac{n}{N}\right) + a_2 \cos\left(4\pi \frac{n}{N}\right) - a_3 \cos\left(6\pi \frac{n}{N}\right)$$

where $n = 0, 1, 2, \ldots N\text{-}1$. The periodic window is N-periodic.

The coefficients for this window are

$a_0 = 0.3635819$

$a_1 = 0.4891775$

$a_2 = 0.1365995$

$a_3 = .0106411$

References   [1] Nuttall, Albert H. "Some Windows with Very Good Sidelobe Behavior." *IEEE Transactions on Acoustics, Speech, and Signal Processing.* Vol. ASSP-29 (February 1981). pp. 84-91.

See Also   barthannwin | bartlett | blackmanharris | bohmanwin | parzenwin | rectwin | triang | window | wintool | wvtool

overshoot

Purpose   Overshoot metrics of bilevel waveform transitions

Syntax
```
OS = overshoot(X)
OS = overshoot(X,FS)
OS = overshoot(X,T)
[OS,OSLEV,OSINST] = overshoot(...)
[...] = overshoot(...,Name,Value)
overshoot(...)
```

Description   `OS = overshoot(X)` returns the greatest absolute deviations larger than the final state levels of each transition in the bilevel waveform, `X`. The overshoots, `OS`, are expressed as a percentage of the difference between the state levels. The length of `OS` corresponds to the number of transitions detected in the input signal. The sample instants in `X` correspond to the vector indices. To determine the transitions, `overshoot` estimates the state levels of the input waveform by a histogram method. `overshoot` identifies all intervals which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See "State-Level Tolerances" on page 3-683.

`OS = overshoot(X,FS)` specifies the sampling frequency in hertz. The sampling frequency determines the sample instants corresponding to the elements in `X`. The first sample instant in `X` corresponds to t=0.

`OS = overshoot(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

`[OS,OSLEV,OSINST] = overshoot(...)` returns the levels, `OSLEV`, and sample instants,`OSINST`, of the overshoots for each transition.

`[...]  = overshoot(...,Name,Value)` returns the greatest deviations larger than the final state level with additional options specified by one or more `Name,Value` pair arguments.

`overshoot(...)` plots the bilevel waveform and marks the location of the overshoot of each transition as well as the lower and upper

reference-level instants and the associated reference levels. The state levels and associated lower and upper-state boundaries are also plotted.

Input Arguments

X

Bilevel waveform. X is a real-valued row or column vector.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

PctRefLevels

Reference levels as a percentage of the waveform amplitude. The lower-state level is defined to be 0 percent. The upper-state level is defined to be 100 percent. The value of 'PCTREFLEVELS' is a two-element real row vector whose elements correspond to the lower and upper percent reference levels.

> **Default:** [10 90]

Region

Specifies the region over which to compute the overshoot. Valid values for 'Region' are 'Preshoot' or 'Postshoot'. If you specify 'Preshoot', the end of the pretransition aberration region is defined as the last instant where the signal exits the first state. If you specify 'Postshoot', the start of the posttransition aberration region is defined as the instant when the signal enters the second state.

> **Default:** 'Postshoot'

SeekFactor

Aberration region duration. Specifies the duration of the region over which to compute the overshoot for each transition as a multiple of the corresponding transition duration. If the edge of the waveform is reached, or a complete intervening transition is detected before the duration aberration region duration elapses, the duration is truncated to the edge of the waveform or the start of the intervening transition.

Default: 3

StateLevels

Lower and upper state levels. Specifies the levels to use for the lower and upper state levels as a two-element real row vector whose first and second elements correspond to the lower and upper state levels of the input waveform.

Tolerance

Specifies the tolerance that the initial and final levels of each transition must be within the respective state levels. The 'Tolerance' value is a scalar expressed as the percentage of the difference between the upper and lower state levels.

Default: 2

Output Arguments

OS

Overshoots expressed as a percentage of the state levels. The overshoot percentages are computed based on the greatest deviation from the final state level in each transition. By default overshoots are computed for posttransition aberration regions. See "Overshoot" on page 3-681.

OSLEV

Level of the pretransition or posttransition overshoot.

OSINST

Sample instants of pretransition or posttransition overshoots. If you specify the sampling frequency or sampling instants, the overshoot instants are in seconds. If you do not specify the sampling frequency or sampling instants, the overshoot instants are the indices of the input vector.

Definitions

Overshoot

For a positive-going (positive-polarity) pulse, overshoot expressed as a percentage is

$$100\frac{(O - S_2)}{(S_2 - S_1)}$$

where O is the maximum deviation greater the high-state level, S_2 is the high state, and S_1 is the low state.

For a negative-going (negative-polarity) pulse, overshoot expressed as a percentage is

$$100\frac{(O - S_1)}{(S_2 - S_1)}$$

The following figure illustrates the calculation of overshoot for a positive-going transition.

overshoot

The red dashed lines indicate the estimated state levels. The double-sided black arrow depicts the difference between the high and low-state levels. The solid black line indicates the difference between the overshoot value and the high-state level.

State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.

State Levels ± 2% Tolerance

Examples **Overshoot Percentage in Posttransition Aberration Region**

Determine the maximum percent overshoot relative to the high-state level in a 2.3 V clock waveform.

Load the 2.3 V clock data. Plot the waveform. In this example, you see that the maximum overshoot in the posttransition region occurs near index 22.

```
load('transitionex.mat', 'x');
plot(x);
set(gca,'xtick',[1 6 12 18 22 28 34 40]);
```

Determine the maximum percent overshoot.

```
os = overshoot(x);
```

Overshoot Percentage, Levels, and Sample Instant in Posttransition Aberration Region

Determine the maximum percent overshoot relative to the high-state level, the level of the overshoot, and the sample instant in a 2.3 V clock waveform.

Load the 2.3 V clock data with sampling instants. Plot the waveform. The clock data is sampled at 4 MHz.

```
load('transitionex.mat', 'x','t');
plot(t,x);
```

Determine the maximum percent overshoot, the level of the overshoot in volts, and the sampling instant where the maximum overshoot occurs. Plot the result.

```
[os,oslev,osinst] = overshoot(x,t);
plot(t.*1e6,x); xlabel('Microseconds');
hold on; grid on;
plot(osinst*1e6,oslev,'ro','markerfacecolor',[1 0 0]);
```

Overshoot Percentage, Levels, and Sample Instant in Pretransition Aberration Region

Determine the maximum percent overshoot relative to the low-state level, the level of the overshoot, and the sample instant in a 2.3 V clock

waveform. Specify the `'Region'` as `'Preshoot'` to output pretransition metrics.

Load the 2.3 V clock data with sampling instants. Plot the waveform. The clock data is sampled at 4 MHz.

```
load('transitionex.mat', 'x','t');
plot(t,x);
```

Determine the maximum percent overshoot, the level of the overshoot in volts, and the sampling instant where the maximum overshoot occurs. Plot the result.

```
load('transitionex.mat', 'x','t');
[os,oslev,osinst] = overshoot(x,t,'Region','Preshoot');
plot(t.*1e6,x); xlabel('Microseconds');
hold on; grid on;
plot(osinst*1e6,oslev,'ro','markerfacecolor',[1 0 0]);
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Standard 181, 2003, pp. 15–17.

See Also   settlingtime | statelevels | overshoot

Purpose   Parzen (de la Valle-Poussin) window

Syntax   w = parzenwin(L)

Description   w = parzenwin(L) returns the L-point Parzen (de la Valle-Poussin) window in column vector w. Parzen windows are piecewise cubic approximations of Gaussian windows. Parzen window sidelobes fall off as $1/\omega^4$.

Examples   Compare 64-point Parzen and Gaussian windows and display the result using sigwin window objects and wintool:

wintool(sigwin.parzenwin(64),sigwin.gausswin(64))

parzenwin

Algorithms   The following equation defines the *N*–point Parzen window over the interval $-\frac{(N-1)}{2} \le n \le \frac{(N-1)}{2}$:

$$w(n) = \begin{cases} 1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3 & 0 \le |n| \le (N-1)/4 \\ 2\left(1 - \frac{|n|}{N/2}\right)^3 & (N-1)/4 < |n| \le (N-1)/2 \end{cases}$$

References   [1] Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, No. 1 (January 1978).

See Also   barthannwin | bartlett | blackmanharris | bohmanwin | nuttallwin | rectwin | triang | window | wintool | wvtool

Purpose   PSD using Burg method

Syntax   Pxx = pburg(x,p)
                Pxx = pburg(x,p,nfft)
                [Pxx,w] = pburg(...)
                [Pxx,w] = pburg(x,p,w)
                Pxx = pburg(x,p,nfft,fs)
                Pxx = pburg(x,p,f,fs)
                [Pxx,f] = pburg(x,p,nfft,fs)
                [Pxx,f] = pburg(x,p,f,fs)
                [Pxx,f] = pburg(x,p,nfft,fs,'*range*')
                [Pxx,w] = pburg(x,p,nfft,'*range*')
                pburg(...)

Description Pxx = pburg(x,p) implements the Burg algorithm, a parametric
                spectral estimation method, and returns Pxx, an estimate of the power
                spectral density (PSD) of the vector x. The entries of x represent
                samples of a discrete-time signal, and p is the integer specifying the
                order of an autoregressive (AR) prediction model for the signal, used in
                estimating the PSD.

                The power spectral density is calculated in units of power per radians
                per sample. Real-valued inputs produce full power one-sided (in
                frequency) PSDs (by default), while complex-valued inputs produce
                two-sided PSDs.

                In general, the length of the FFT and the values of the input x
                determine the length of Pxx and the range of the corresponding
                normalized frequencies. For this syntax, the (default) FFT length is
                256. The following table indicates the length of Pxx and the range of the
                corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

Pxx = pburg(x,p,nfft) uses the integer FFT length nfft to calculate the PSD vector Pxx.

[Pxx,w] = pburg(...) also returns w, a vector of normalized angular frequencies at which the two-sided PSD is estimated. Pxx and w have the same length. The units for w are rad/sample.

The length of Pxx and the frequency range for w depend on nfft and the values of the input x. The following table indicates the length of Pxx and the frequency range for w in this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	[0, π]
Real-valued	Odd	(nfft + 1)/2	[0, π)
Complex-valued	Even or odd	nfft	[0, 2π)

[Pxx,w] = pburg(x,p,w) uses a vector of normalized frequencies w with two or more elements to compute the PSD at those frequencies and returns a two-sided PSD.

Pxx = pburg(x,p,nfft,fs)

or

`Pxx = pburg(x,p,f,fs)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx` or uses the vector of frequencies `f` in Hz and the sampling frequency `fs` to compute the two-sided PSD vector `Pxx` at those frequencies. If you specify `nfft` as the empty vector `[]`, it uses the default value of 256. If you specify `fs` as the empty vector `[]`, the sampling frequency `fs` defaults to 1 Hz. The spectral density produced is calculated in units of power per Hz.

`[Pxx,f] = pburg(x,p,nfft,fs)`

or

`[Pxx,f] = pburg(x,p,f,fs)` returns the frequency vector `f`. In this case, the units for the frequency vector are in Hz. The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pburg(x,p,nfft,fs,'range')` or

`[Pxx,w] = pburg(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.

  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

pburg

- If you don't specify fs, the frequency range is [0, 2π).

- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real x. This is the default for determining the frequency range for real-valued x. Note that `'onesided'` is not valid if you pass in a vector of frequencies (f or w).

Note You can put the string argument `'range'` anywhere in the input argument list after p.

---

pburg(...) with no outputs plots the PSD in the current figure window. The frequency range on the plot is the same as the range of output w (or f) for a given set of parameters.

Tips
The power spectral density is computed as the distribution of power per unit frequency. This algorithm depends on your selecting an appropriate model order for your signal.

Examples
The Burg method estimates the spectral density by fitting an AR prediction model of a given order to the signal, so first generate a signal from an AR (all-pole) model of a given order. Use freqz to check the magnitude of the frequency response of your AR filter. Then, generate the input signal x by filtering white noise through the AR filter. Estimate the PSD of x based on a fourth-order AR prediction model because in this case we know that the original AR system model a has order 4:

```
% Define AR filter coefficients
a = [1 -2.2137 2.9403 -2.1697 0.9606];
[H,w] = freqz(1,a,256);     % AR filter freq response
% Scale to make one-sided PSD
Hp = plot(w/pi,20*log10(2*abs(H)/(2*pi)),'r');
hold on;
rng default;
x = filter(1,a,randn(256,1));           % AR system output
pburg(x,4,511);
```

```
xlabel('Normalized frequency (\times \pi rad/sample)')
ylabel('One-sided PSD (dB/rad/sample)')
legend('PSD of model output','PSD estimate of x')
```


Algorithms   You can use linear prediction filters to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

The Burg method fits an AR linear prediction filter model of the specified order to the input signal by minimizing (using least squares) the arithmetic mean of the forward and backward prediction errors. The spectral density then is computed from the frequency response of

the prediction filter. The AR filter parameters are constrained to satisfy the Levinson-Durbin recursion.

References   [1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also   arburg | lpc | pcov | peig | periodogram | pmcov | pmtm | pmusic | pwelch | pyulear | spectrum.burg

Purpose   PSD using covariance method

Syntax   Pxx = pcov(x,p)
                   Pxx = pcov(x,p,nfft)
                   [Pxx,w] = pcov(...)
                   [Pxx,w] = pcov(x,p,w)
                   Pxx = pcov(x,p,nfft,fs)
                   Pxx = pcov(x,p,f,fs)
                   [Pxx,f] = pcov(x,p,nfft,fs)
                   [Pxx,f] = pcov(x,p,f,fs)
                   [Pxx,f] = pcov(x,p,nfft,fs,'*range*')
                   [Pxx,w] = pcov(x,p,nfft,'*range*')
                   pcov(...)

Description   Pxx = pcov(x,p) implements the covariance algorithm, a parametric
                   spectral estimation method, and returns Pxx, an estimate of the power
                   spectral density (PSD) of the vector x. The entries of x represent
                   samples of a discrete-time signal, and where p is the integer specifying
                   the order of an autoregressive (AR) prediction model for the signal,
                   used in estimating the PSD.

                   The power spectral density is calculated in units of power per radians
                   per sample. Real-valued inputs produce full power one-sided (in
                   frequency) PSDs (by default), while complex-valued inputs produce
                   two-sided PSDs.

                   In general, the length of the FFT and the values of the input x
                   determine the length of Pxx and the range of the corresponding
                   normalized frequencies. For this syntax, the (default) FFT length is
                   256. The following table indicates the length of Pxx and the range of the
                   corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

Pxx = pcov(x,p,nfft) uses the integer FFT length nfft to calculate the PSD vector Pxx.

[Pxx,w] = pcov(...) also returns w, a vector of normalized angular frequencies at which the two-sided PSD is estimated. Pxx and w have the same length. The units for w are rad/sample.

The length of Pxx and the frequency range for w depend on nfft and the values of the input x. The following table indicates the length of Pxx and the frequency range for w in this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	[0, π]
Real-valued	Odd	(nfft + 1)/2	[0, π)
Complex-valued	Even or odd	nfft	[0, 2π)

[Pxx,w] = pcov(x,p,w) uses a vector of normalized frequencies w with two or more elements to compute the PSD at those frequencies and returns a two-sided PSD.

Pxx = pcov(x,p,nfft,fs)

or

`Pxx = pcov(x,p,f,fs)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx` or uses the vector of frequencies `f` in Hz and the sampling frequency `fs` to compute the two-sided PSD vector `Pxx` at those frequencies. If you specify `nfft` as the empty vector `[]`, it uses the default value of 256. If you specify `fs` as the empty vector `[]`, the sampling frequency `fs` defaults to 1 Hz. The spectral density produced is calculated in units of power per Hz.

`[Pxx,f] = pcov(x,p,nfft,fs)`

or

`[Pxx,f] = pcov(x,p,f,fs)` returns the frequency vector f. In this case, the units for the frequency vector are in Hz. The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pcov(x,p,nfft,fs,'`*range*`')` or

`[Pxx,w] = pcov(x,p,nfft,'`*range*`')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'`*range*`'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.

  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

- If you don't specify fs, the frequency range is $[0, 2\pi)$.

• 'onesided': Compute the one-sided PSD over the frequency ranges specified for real x. This is the default for determining the frequency range for real-valued x. Note that 'onesided' is not valid if you pass in a vector of frequencies (f or w).

Note You can put the string argument '*range*' anywhere in the input argument list after p.

---

pcov(...) with no outputs plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output w (or f) for a given set of parameters.

Tips The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples Because the covariance method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use freqz to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using pcov:

```
a = [1 -2.2137 2.9403 -2.1697 0.9606];    % AR filter coefficients
freqz(1,a)                                 % AR filter frequency response
title('AR System Frequency Response')
```

Now generate the input signal x by filtering white noise through the AR filter. Estimate the PSD of x based on a fourth-order AR prediction model since in this case we know that the original AR system model a has order 4:

```
% Signal generated from AR filter
x = filter(1,a,randn(256,1));
% Fourth-order estimate
pcov(x,4)
```

Algorithms   Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

The covariance method estimates the PSD of a signal using the covariance method. The covariance (or nonwindowed) method fits an AR linear prediction filter model to the signal by minimizing the forward prediction error (based on causal observations of your input signal) in the least squares sense. The spectral estimate returned by pcov is the squared magnitude of the frequency response of this AR model.

pcov

References   [1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also   arcov | lpc | pburg | peig | periodogram | pmcov | pmtm | pmusic | pwelch | pyulear | spectrum.cov

Purpose   Maximum-to-minimum difference

Syntax   Y = peak2peak(X)
                     Y = peak2peak(X,DIM)

Description   Y = peak2peak(X) returns the difference between the maximum and
                     minimum values in X. peak2peak operates along the first nonsingleton
                     dimension of X by default. For example, if X is a row or column vector,
                     Y is a real-valued scalar. If Y is an N-by-M matrix with N>1, Y is a
                     1-by-M row vector containing the maximum-to-minimum differences
                     of the columns of X.

                     Y = peak2peak(X,DIM) computes the maximum-to-minimum
                     differences of X along the dimension, DIM.

Input   **X**
Arguments
                     Real- or complex-valued input vector or matrix. By default, peak2peak
                     acts along the first nonsingleton dimension of X. For complex-valued
                     inputs, peak2peak identifies the maximum and minimum in absolute
                     value. peak2peak subtracts the complex number with the minimum
                     modulus from the complex number with the maximum modulus.

                     DIM

                     Dimension for maximum-to-minimum difference. The optional DIM
                     input argument specifies the dimension along which to compute the
                     maximum-to-minimum differences.

                     Default: First nonsingleton dimension

Output   **Y**
Arguments
                     Maximum-to-minimum difference. For vectors, Y is a real-valued
                     scalar. For matrices, Y contains the maximum-to-minimum differences
                     computed along the specified dimension, DIM. By default, DIM is the
                     first nonsingleton dimension.

peak2peak

Examples

Peak-to-Peak Difference of Sinusoid

Compute the maximum-to-minimum difference of a 100-Hz sinusoid sampled at 1 kHz.

```
t = 0:0.001:1-0.001;
X = cos(2*pi*100*t);
Y = peak2peak(X);
```

Peak-to-Peak Difference of Complex Exponential

Compute the maximum-to-minimum difference of a complex exponential with a frequency of π/4 radians/sample.

Create a complex exponential with a frequency of π/4 radians/sample. Find the peak-to-peak difference.

```
n = 0:99;
x = exp(1j*pi/4*n);
maxmin = peak2peak(x);
```

Peak-to-Peak Differences of 2-D Matrix

Create a matrix where each column is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the column index.

Compute the maximum-to-minimum differences of the columns.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)';
X = repmat(x,1,4);
amp = 1:4;
amp = repmat(amp,1e3,1);
X = X.*amp;
Y = peak2peak(X);
```

Peak-to-Peak Differences of 2-D Matrix Along Specified Dimension

Create a matrix where each row is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the row index.

Compute the maximum-to-minimum differences of the rows specifying the dimension equal to 2 with the DIM argument.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t);
X = repmat(x,4,1);
amp = (1:4)';
amp = repmat(amp,1,1e3);
X = X.*amp;
Y = peak2peak(X,2);
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

See Also   min | max | peak2rms

peak2rms

Purpose	Peak-magnitude-to-RMS ratio
Syntax	`Y = peak2rms(X)` `Y = peak2rms(X,DIM)`

Description   `Y = peak2rms(X)` returns the ratio of the largest absolute value in X to the root-mean-square (RMS) value of X. `peak2rms` operates along the first nonsingleton dimension of X. For example, if X is a row or column vector, Y is a real-valued scalar. If Y is an N-by-M matrix with N>1, Y is a 1-by-M row vector containing the peak-magnitude-to-RMS levels of the columns of Y.

`Y = peak2rms(X,DIM)` computes the peak-magnitude-to-RMS level of X along the dimension, DIM.

Input Arguments

X

Real– or complex-valued input vector or matrix. By default, `peak2rms` acts along the first nonsingleton dimension of X.

DIM

Dimension for peak-magnitude-to-RMS ratio. The optional DIM input argument specifies the dimension along which to compute the peak-magnitude-to-RMS level.

Default: First nonsingleton dimension

Output Arguments

Y

Peak-magnitude-to-RMS ratio. For vectors, Y is a real-valued scalar. For matrices, Y contains the peak-magnitude-to-RMS levels computed along the specified dimension, DIM. By default, DIM is the first nonsingleton dimension.

Definitions   **Peak-magnitude-to-RMS Level**

The peak-magnitude-to-RMS ratio is

$$\frac{||X||_\infty}{\sqrt{\frac{1}{N}\sum_{n=1}^{N}|X_n|^2}}$$

where the *l*-infinity norm and RMS values are computed along the specified dimension.

Examples

Peak-magnitude-to-RMS Ratio of Sinusoid

Compute the peak-magnitude-to-RMS ratio of a 100-Hz sinusoid sampled at 1 kHz.

```
t = 0:0.001:1-0.001;
X = cos(2*pi*100*t);
Y = peak2rms(X);
```

Peak-magnitude-to-RMS Ratio of Complex Exponential

Compute the peak-magnitude-to-RMS ratio of a complex exponential with a frequency of π/4 radians/sample.

Create a complex exponential with a frequency of π/4 radians/sample. Find the peak-magnitude-to-RMS ratio.

```
n = 0:99;
X = exp(1j*pi/4*n);
Y = peak2rms(X);
```

Peak-magnitude-to-RMS ratio of 2-D Matrix

Create a matrix where each column is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the column index.

Compute the peak-magnitude-to-RMS ratio of the columns.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)';
X = repmat(x,1,4);
```

```
amp = 1:4;
amp = repmat(amp,1e3,1);
X = X.*amp;
Y = peak2rms(X);
```

Peak-magnitude-to-RMS ratio of 2-D Matrix Along Specified Dimension

Create a matrix where each row is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the row index.

Compute the peak-magnitude-to-RMS ratio of the rows specifying the dimension equal to 2 with the DIM argument.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t);
X = repmat(x,4,1);
amp = (1:4)';
amp = repmat(amp,1,1e3);
X = X.*amp;
Y = peak2rms(X,2);
```

References [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Standard 181, 2003.

Purpose   Pseudospectrum using eigenvector method

Syntax
```
[S,w] = peig(x,p)
[S,w] = peig(x,p,w)
[S,w] = peig(...,nfft)
[S,f] = peig(x,p,nfft,fs)
[S,f] = peig(x,p,f,fs)
[S,f] = peig(...,'corr')
[S,f] = peig(x,p,nfft,fs,nwin,noverlap)
[...] = peig(...,'range')
[...,v,e] = peig(...)
peig(...)
```

Description   `[S,w] = peig(x,p)` implements the eigenvector spectral estimation method and returns S, the pseudospectrum estimate of the input signal x, and w, a vector of normalized frequencies (in rad/sample) at which the pseudospectrum is evaluated. The pseudospectrum is calculated using estimates of the eigenvectors of a correlation matrix associated with the input data x, where x is specified as either:

- A row or column vector representing one observation of the signal

- A rectangular array for which each row of x represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that x'*x is an estimate of the correlation matrix

---

Note You can use the output of `corrmtx` to generate such an array x.

You can specify the second input argument p as either:

- A scalar integer. In this case, the signal subspace dimension is p.

- A two-element vector. In this case, p(2), the second element of p, represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}*p(2) are assigned to the noise subspace.

In this case, `p(1)` specifies the maximum dimension of the signal subspace.

Note If the inputs to `peig` are real sinusoids, set the value of p to double the number of input signals. If the inputs are complex sinusoids, set p equal to the number of inputs.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

`S` and `w` have the same length. In general, the length of the FFT and the values of the input `x` determine the length of the computed `S` and the range of the corresponding normalized frequencies. The following table indicates the length of `S` (and `w`) and the range of the corresponding normalized frequencies for this syntax.

S Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of S and w	Range of the Corresponding Normalized Frequencies
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

`[S,w] = peig(x,p,w)` returns the pseudospectrum in the vector `S` computed at the normalized frequencies specified in vector `w`, which has two or more elements

`[S,w] = peig(...,nfft)` specifies the integer length of the FFT `nfft` used to estimate the pseudospectrum. The default value for `nfft` (entered as an empty vector `[ ]`) is 256.

The following table indicates the length of `S` and `w`, and the frequency range for `w` for this syntax.

S and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of S and w	Range of w
Real-valued	Even	`(nfft/2 + 1)`	$[0, \pi]$
Real-valued	Odd	`(nfft + 1)/2`	$[0, \pi)$
Complex-valued	Even or odd	`nfft`	$[0, 2\pi)$

`[S,f] = peig(x,p,nfft,fs)` returns the pseudospectrum in the vector S evaluated at the corresponding vector of frequencies f (in Hz). You supply the sampling frequency fs in Hz. If you specify fs with the empty vector [], the sampling frequency defaults to 1 Hz.

The frequency range for f depends on nfft, fs, and the values of the input x. The length of S (and f) is the same as in the S and Frequency Vector Characteristics on page 3-709 above. The following table indicates the frequency range for f for this syntax.

S and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[S,f] = peig(x,p,f,fs)` returns the pseudospectrum in the vector S computed at the frequencies specified in vector f, which has two or more elements

`[S,f] = peig(...,'corr')` forces the input argument x to be interpreted as a correlation matrix rather than matrix of signal data. For this syntax x must be a square matrix, and all of its eigenvalues must be nonnegative.

[S,f] = peig(x,p,nfft,fs,nwin,noverlap) allows you to specify nwin, a scalar integer indicating a rectangular window length, or a real-valued vector specifying window coefficients. Use the scalar integer noverlap in conjunction with nwin to specify the number of input sample points by which successive windows overlap. noverlap is not used if x is a matrix. The default value for nwin is 2*p(1) and noverlap is nwin-1.

With this syntax, the input data x is segmented and windowed before the matrix used to estimate the correlation matrix eigenvalues is formulated. The segmentation of the data depends on nwin, noverlap, and the form of x. Comments on the resulting windowed segments are described in the following table.

Windowed Data Depending on x and nwin

Input data x	Form of nwin	Windowed Data
Data vector	Scalar	Length is nwin
Data vector	Vector of coefficients	Length is length(nwin)
Data matrix	Scalar	Data is not windowed.
Data matrix	Vector of coefficients	length(nwin) must be the same as the column length of x, and noverlap is not used.

See the table, Eigenvector Length Depending on Input Data and Syntax on page 3-712, for related information on this syntax.

Note The arguments nwin and noverlap are ignored when you include the string 'corr' in the syntax.

[...] = peig(...,'*range*') specifies the range of frequency values to include in f or w. This syntax is useful when x is real. '*range*' can be either:

- 'whole': Compute the pseudospectrum over the frequency range [0,fs). This is the default for determining the frequency range for complex-valued x.

  - If you specify fs as the empty vector, [], the frequency range is [0,1).

  - If you don't specify fs, the frequency range is $[0, 2\pi)$.

- 'half': Compute the pseudospectrum over the frequency ranges specified for real x. This is the default for determining the frequency range for real-valued x.

Note You can put the string arguments '*range*' or 'corr' anywhere in the input argument list after p.

[...,v,e] = peig(...) returns the matrix v of noise eigenvectors, along with the associated eigenvalues in the vector e. The columns of v span the noise subspace of dimension size(v,2). The dimension of the signal subspace is size(v,1)-size(v,2). For this syntax, e is a vector of estimated eigenvalues of the correlation matrix.

peig(...) with no output arguments plots the pseudospectrum in the current figure window.

Tips   In the process of estimating the pseudospectrum, peig computes the noise and signal subspaces from the estimated eigenvectors v_j and eigenvalues λ_j of the signal's correlation matrix. The smallest of these eigenvalues is used in conjunction with the threshold parameter p(2) to affect the dimension of the noise subspace in some cases.

The length *n* of the eigenvectors computed by peig is the sum of the dimensions of the signal and noise subspaces. This eigenvector length

depends on your input (signal data or correlation matrix) and the syntax you use.

The following table summarizes the dependency of the eigenvector length on the input argument.

Eigenvector Length Depending on Input Data and Syntax

Form of Input Data x	Comments on the Syntax	Length n of Eigenvectors
Row or column vector	nwin is specified as a scalar integer.	nwin
Row or column vector	nwin is specified as a vector.	length(nwin)
Row or column vector	nwin is not specified.	2*p(1)
l-by-*m* matrix	If nwin is specified as a scalar, it is not used. If nwin is specified as a vector, length(nwin) must equal *m*.	*m*
m-by-*m* nonnegative definite matrix	The string 'corr' is specified and nwin is not used.	*m*

You should specify nwin > p(1) or length(nwin) > p(1) if you want p(2) > 1 to have any effect.

Examples   Implement the eigenvector method to find the pseudospectrum of the sum of three sinusoids in noise, using the default FFT length of 256. The inputs are complex sinusoids so you set p equal to the number of inputs. Use the modified covariance method for the correlation matrix estimate:

```
n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
X=corrmtx(s,12,'mod');
```

```
peig(X,3,'whole')              % Uses default NFFT of 256
```

Algorithms   The eigenvector method estimates the pseudospectrum from a signal or a correlation matrix using a weighted version of the MUSIC algorithm derived from Schmidt's eigenspace analysis method [1] [2]. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content. The eigenvalues and eigenvectors of the signal's correlation matrix are estimated using svd if you don't supply the correlation matrix. This algorithm is particularly suitable for signals that are the sum of sinusoids with additive white Gaussian noise.

The eigenvector method produces a pseudospectrum estimate given by

$$P_{ev}(f) = \frac{1}{\left(\displaystyle\sum_{k=p+1}^{N} |\mathbf{v}_k^H \mathbf{e}(f)|^2 \right) / \lambda_k}$$

where N is the dimension of the eigenvectors and $\mathbf{v}_k$ is the kth eigenvector of the correlation matrix of the input signal. The integer p is the dimension of the signal subspace, so the eigenvectors $\mathbf{v}_k$ used in the sum correspond to the smallest eigenvalues λ_k of the correlation matrix. The eigenvectors used span the noise subspace. The vector $\mathbf{e}(f)$ consists of complex exponentials, so the inner product

$$\mathbf{v}_k^H \mathbf{e}(f)$$

amounts to a Fourier transform. This is used for computation of the pseudospectrum. The FFT is computed for each $\mathbf{v}_k$ and then the squared magnitudes are summed and scaled.

References   [1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 373-378.

peig

[2] Schmidt, R.O, "Multiple Emitter Location and Signal Parameter Estimation," *IEEE Trans. Antennas Propagation*, Vol. AP-34 (March 1986), pp.276-280.

[3] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also   corrmtx | dspdata | pburg | periodogram | pmtm | pmusic | prony | pwelch | rooteig | rootmusic | spectrum.eigenvector

Purpose   PSD using periodogram

Syntax
```
[Pxx,w] = periodogram(x)
[Pxx,w] = periodogram(x,window)
[Pxx,w] = periodogram(x,window,nfft)
[Pxx,w] = periodogram(x,window,w)
[Pxx,f] = periodogram(x,window,nfft,fs)
[Pxx,f] = periodogram(x,window,f,fs)
[Pxx,f] = periodogram(x,window,nfft,fs,'range')
[Pxx,w] = periodogram(x,window,nfft,'range')
periodogram(...)
```

Description   `[Pxx,w] = periodogram(x)` returns the power spectral density (PSD) estimate `Pxx` of the sequence `x` using a periodogram. The power spectral density is calculated in units of power per radians per sample. The corresponding vector of frequencies `w` is computed in radians per sample, and has the same length as `Pxx`.

A real-valued input vector `x` produces a full power one-sided (in frequency) PSD (by default), while a complex-valued `x` produces a two-sided PSD.

In general, the length N of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) length N of the FFT is the larger of 256 and the next power of 2 greater than the length of `x`. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	$(N/2) + 1$	$[0, \pi]$
Complex-valued	N	$[0, 2\pi)$

[Pxx,w] = periodogram(x,window) returns the PSD estimate Pxx computed using the modified periodogram method. The vector window specifies the coefficients of the window used in computing a modified periodogram of the input signal. Both input arguments must be vectors of the same length. When you don't supply the second argument window, or set it to the empty vector [ ], a rectangular window (rectwin) is used by default. In this case the standard periodogram is calculated.

[Pxx,w] = periodogram(x,window,nfft) uses the modified periodogram to estimate the PSD while specifying the length of the FFT with the integer nfft. If you set nfft to the empty vector [ ], it takes the default value for *N* listed in the previous syntax.

The length of Pxx and the frequency range for w depend on nfft and the values of the input x. The following table indicates the length of Pxx and the frequency range for w for this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	$[0, \pi]$
Real-valued	Odd	(nfft + 1)/2	$[0, \pi)$
Complex-valued	Even or odd	nfft	$[0, 2\pi)$

Note periodogram uses an nfft-point FFT of the windowed data (x.*window) to compute the periodogram. If the value you specify for nfft is less than the length of x, then x.*window is wrapped modulo nfft. If the value you specify for nfft is greater than the length of x, then x.*window is zero-padded to compute the FFT.

[Pxx,w] = periodogram(x,window,w) estimates the two-sided PSD at the normalized frequencies specified in the vector w using the Goertzel algorithm. The frequencies of w are rounded to the nearest DFT bin

commensurate with the resolution of the signal. The units of w are rad/sample.

`[Pxx,f] = periodogram(x,window,nfft,fs)` uses the sampling frequency `fs` specified as an integer in hertz (Hz) to compute the PSD vector (`Pxx`) and the corresponding vector of frequencies (`f`). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify `fs` as the empty vector `[]`, the sampling frequency defaults to 1 Hz.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = periodogram(x,window,f,fs)` uses the vector of frequencies `f` at which the PSD is estimated. The frequencies of `f` are rounded to the nearest DFT bin commensurate with the resolution of the signal.

`[Pxx,f] = periodogram(x,window,nfft,fs,'`*range*`')` or

`[Pxx,w] = periodogram(x,window,nfft,'`*range*`')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'`*range*`'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.

  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

periodogram

- If you don't specify fs, the frequency range is [0, 2π).

- 'onesided': Compute the one-sided PSD over the frequency ranges specified for real x. This is the default for determining the frequency range for real-valued x.

Note You can put the string argument '*range*' anywhere in the input argument list after window.

---

periodogram(...) with no outputs plots the power spectral density in dB per unit frequency in the current figure window. The frequency range on the plot is the same as the range of output w (or f) for the syntax you use.

Examples   Compute the periodogram of a 200 Hz signal embedded in additive noise using the default window:

```
Fs = 1000;
t = 0:1/Fs:.3;
x = cos(2*pi*t*200)+0.1*randn(size(t));
periodogram(x,[],'onesided',512,Fs)
```

Algorithms   The periodogram for a sequence $[x_1, \ldots, x_N]$ is given by the following formula:

$$S(e^{j\omega}) = \frac{1}{2\pi N} \Big| \sum_{n=1}^{N} x_n e^{-j\omega n} \Big|^2$$

where ω is in units of radians/sample.

If you define the frequency variable in Hz, the periodogram is defined as:

periodogram

$$S(f) = \frac{1}{FsN} \Big| \sum_{n=1}^{N} x_n e^{-j(2\pi f/Fs)n} \Big|^2$$

where *Fs* is the sampling frequency.

The periodogram is an estimate of the PSD of the signal defined by the sequence $[x_1, \ldots, x_N]$.

If you weight your signal sequence by a window $[w_1, \ldots, w_N]$, then the weighted or modified periodogram is defined as:

$$S(e^{j\omega}) = \frac{1}{2\pi N} \Big| \sum_{n=1}^{N} x_n w_n e^{-j\omega n} \Big|^2 \Big/ \frac{1}{N} \sum_{n=1}^{N} |w_n|^2$$

Canceling the common factors and denoting the squared ℓ^2 norm of the window sequence by $||w||^2$, the modified periodogram can be simplified as:

$$S(e^{j\omega}) = \frac{1}{2\pi} \frac{\Big| \sum_{n=1}^{N} x_n w_n e^{-j\omega n} \Big|^2}{||w||^2}$$

In either case, `periodogram` uses an `nfft`-point FFT to compute the power spectral density.

References   [1] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997, pp. 24-26.

[2] Welch, P.D, "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms," *IEEE Trans. Audio Electroacoustics, Vol. AU-15 (June 1967)*, pp. 70-73.

[3] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp.730-742.

See Also   dspdata.msspectrum | pburg | pcov | peig | pmcov | pmtm | pmusic | pwelch | pyulear | spectrum.periodogram

phasedelay

Purpose   Phase delay of digital filter

Syntax
```
[phi,w] = phasedelay(b,a,n)
[phi,w] = phasedelay(b,a,n,'whole')
phi = phasedelay(b,a,w)
[phi,f] = phasedelay(b,a,n,fs)
[phi,f] = phasedelay(b,a,n,'whole',fs)
phi = phasedelay(b,a,f,fs)
[phi,w,s] = phasedelay(...)
[phi,f,s] = phasedelay(...)
phasedelay(b,a,...)
```

Description   `[phi,w] = phasedelay(b,a,n)` returns the n-point phase delay
response vector `phi` and the n-point frequency reponse vector `w` (in
radians/sample) of the filter defined by numerator coefficients `b` and
denominator coefficients `a`. The phase delay response is evaluated at `n`
equally spaced points around the upper half of the unit circle. If `n` is
omitted, it defaults to 512.

`[phi,w] = phasedelay(b,a,n,'whole')` uses n equally spaced points
around the whole unit circle.

`phi = phasedelay(b,a,w)` returns the phase delay response at
frequencies specified in vector `w` (in radians/sample). The frequencies
are normally between 0 and π.

`[phi,f] = phasedelay(b,a,n,fs)` and `[phi,f] =
phasedelay(b,a,n,'whole',fs)` return the phase delay
vector `f` (in Hz), using the sampling frequency `fs` (in Hz).

`phi = phasedelay(b,a,f,fs)` returns the phase delay response at the
frequencies specified in vector `f` (in Hz), using the sampling frequency
`fs` (in Hz)..

`[phi,w,s] = phasedelay(...)` and `[phi,f,s] = phasedelay(...)`
return plotting information, where `s` is a structure with fields you can
change to display different frequency response plots.

`phasedelay(b,a,...)` with no output arguments, plots the phase
delay response of the filter.

Examples   **Example 1**

Plot the phase delay response of a constrained least squares FIR filter:

```
b=fircls1(54,.3,.02,.008);
phasedelay(b)
```


Example 2

Plot the phase delay response of an elliptic filter:

```
[b,a] = ellip(10,.5,20,.4);
phasedelay(b,a,512,'whole')
```

phasedelay

See Also   freqz | fvtool | phasez | grpdelay

Purpose   Phase response of digital filter

Syntax
```
[phi,w] = phasez(b,a,n)
[phi,w]=phasez(Hd,n)
[phi,w] = phasez(b,a,n,'whole')
[phi,w]=phasez(Hd,n,'whole')
phi = phasez(b,a,w)
phi=phasez(Hd,w)
[phi,f] = phasez(b,a,n,fs)
[phi,f]=phasez(Hd,n,fs)
phi = phasez(b,a,f,fs)
phi=phasez(Hd,f,fs)
[phi,w,s] = phasez(...)
[phi,f,s] = phasez(...)
phasez(b,a,...)
phasez(Hd)
```

Description   [phi,w] = phasez(b,a,n) returns the n-point unwrapped phase
response vector, phi, in radians and frequency vector, w, in
radians/sample for the filter coefficients specified in b and a. The values
of the frequency vector, w, range from 0 to pi. If n is omitted, the length
of the phase response vector defaults to 512.

[phi,w]=phasez(Hd,n) returns the n-point unwrapped phase response
vector, phi, in radians and frequency vector, w, in radians/sample for
the discrete-time filter object Hd. If n is unspecified for discrete-time
filter objects, the length of the phase response vector defaults to 8192.

[phi,w] = phasez(b,a,n,'whole') or
[phi,w]=phasez(Hd,n,'whole') returns frequency and unwrapped
phase response vectors evaluated at n equally-spaced points around the
unit circle from 0 to 2*pi radians/sample.

phi = phasez(b,a,w) or phi=phasez(Hd,w) returns the unwrapped
phase response in radians at frequencies specified in w (radians/sample).
The frequencies are normally between 0 and pi.

[phi,f] = phasez(b,a,n,fs) and [phi,f]=phasez(Hd,n,fs) return
the unwrapped phase vector phi in radians and the frequency vector

phasez

in Hz. The frequency vector ranges from 0 to the Nyquist frequency, `fs/2`. If the `'whole'` option is used, the frequency vector ranges from 0 to the sampling frequency.

`phi = phasez(b,a,f,fs)` and `phi=phasez(Hd,f,fs)` return the phase response in radians at the frequencies specified in the vector `f` (in Hz) using the sampling frequency `fs` (in Hz).

`[phi,w,s] = phasez(...)` and `[phi,f,s] = phasez(...)` return plotting information, where `s` is a structure array with fields you can change to display different frequency response plots.

`phasez(b,a,...)` or `phasez(Hd)` with no output arguments, plots the phase response of the filter. If the filter coefficients are inputted, the current figure window is used. If the user inputs a discrete-time filter object, `fvtool` is used to plot the phase response.

Examples

Example 1

Plot the phase response of a constrained least squares FIR filter:

```
b=fircls1(54,.3,.02,.008);
phasez(b)
```

Example 2

In the next example, we design an equiripple lowpass default filter
object and display the result:

```
d=fdesign.lowpass;
Hd=design(d,'equiripple');
phasez(Hd)
```


Example 3

Plot the phase response of an elliptic filter:

```
d=fdesign.lowpass('Fp,Fst,Ap,Ast',0.4,0.5,1,60);
Hd=design(d,'ellip');
phasez(Hd)
```

phasez

See Also   freqz | fvtool | phasedelay | grpdelay

Purpose   PSD using modified covariance method

Syntax   Pxx = pmcov(x,p)
                Pxx = pmcov(x,p,nfft)
                [Pxx,w] = pmcov(...)
                [Pxx,w] = pmcov(x,p,w)
                Pxx = pmcov(x,p,nfft,fs)
                Pxx = pmcov(x,p,f,fs)
                [Pxx,f] = pmcov(x,p,nfft,fs)
                [Pxx,f] = pmcov(x,p,f,fs)
                [Pxx,f] = pmcov(x,p,nfft,fs,'*range*')
                [Pxx,w] = pmcov(x,p,nfft,'*range*')
                pmcov(...)

Description Pxx = pmcov(x,p) implements the modified covariance algorithm, a
                parametric spectral estimation method, and returns Pxx, an estimate
                of the power spectral density (PSD) of the vector x. The entries of
                x represent samples of a discrete-time signal, and p is the integer
                specifying the order of an autoregressive (AR) prediction model for the
                signal, used in estimating the PSD.

                The power spectral density is calculated in units of power per radians
                per sample. Real-valued inputs produce full power one-sided (in
                frequency) PSDs (by default), while complex-valued inputs produce
                two-sided PSDs.

                In general, the length of the FFT and the values of the input x
                determine the length of Pxx and the range of the corresponding
                normalized frequencies. For this syntax, the (default) FFT length is
                256. The following table indicates the length of Pxx and the range of the
                corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

Pxx = pmcov(x,p,nfft) uses the integer FFT length nfft to calculate the PSD vector Pxx.

[Pxx,w] = pmcov(...) also returns w, a vector of normalized angular frequencies at which the two-sided PSD is estimated. Pxx and w have the same length. The units for w are rad/sample.

The length of Pxx and the frequency range for w depend on nfft and the values of the input x. The following table indicates the length of Pxx and the frequency range for w in this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	[0, π]
Real-valued	Odd	(nfft + 1)/2	[0, π]
Complex-valued	Even or odd	nfft	[0, 2π)

[Pxx,w] = pmcov(x,p,w) uses a vector of normalized frequencies w with two or more elements to compute the PSD at those frequencies and returns a two-sided PSD.

Pxx = pmcov(x,p,nfft,fs)

or

Pxx = pmcov(x,p,f,fs) uses the integer FFT length nfft to calculate the PSD vector Pxx or uses the vector of frequencies f in Hz and the

sampling frequency `fs` to compute the two-sided PSD vector `Pxx` at those frequencies. If you specify `nfft` as the empty vector `[]`, it uses the default value of 256. If you specify `fs` as the empty vector `[]`, the sampling frequency `fs` defaults to 1 Hz. The spectral density produced is calculated in units of power per Hz.

`[Pxx,f] = pmcov(x,p,nfft,fs)`

or

`[Pxx,f] = pmcov(x,p,f,fs)` returns the frequency vector f. In this case, the units for the frequency vector are in Hz. The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pmcov(x,p,nfft,fs,'`*range*`')` or

`[Pxx,w] = pmcov(x,p,nfft,'`*range*`')` specifies the range of frequency values to include in f or w. This syntax is useful when x is real. `'`*range*`'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued x.

  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

  - If you don't specify `fs`, the frequency range is [0, 2π).

- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`. Note that `'onesided'` is not valid if you pass in a vector of frequencies (`f` or `w`).

Note You can put the string argument `'range'` anywhere in the input argument list after `p`.

`pmcov(...)` with no outputs plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

Tips The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples Because the modified covariance method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using `pmcov`:

```
a = [1 -2.2137 2.9403 -2.1697 0.9606];    % AR filter coefficients
freqz(1,a)                                % AR filter frequency response
title('AR System Frequency Response')
```

Now generate the input signal x by filtering white noise through the AR filter. Estimate the PSD of x based on a fourth-order AR prediction model since in this case we know that the original AR system model a has order 4:

```
x = filter(1,a,randn(256,1));    % AR filter output
pmcov(x,4)                        % Fourth-order estimate
```

Algorithms

Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

pmcov estimates the PSD of the signal vector using the modified covariance method. This method fits an autoregressive (AR) linear prediction filter model to the signal by simultaneously minimizing the forward and backward prediction errors (based on causal observations of your input signal) in the least squares sense. The spectral estimate returned by pmcov is the magnitude squared frequency response of this AR model.

References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.

pmcov

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also   armcov | lpc | pburg | pcov | peig | periodogram | pmtm | pmusic | pwelch | prony | pyulear | spectrum.mcov

Purpose   PSD using multitaper method (MTM)

Syntax
```
[Pxx,w] = pmtm(x,nw)
[Pxx,w] = pmtm(x,nw,nfft)
[Pxx,w] = pmtm(x,nw,w)
[Pxx,f] = pmtm(x,nw,nfft,fs)
[Pxx,w] = pmtm(x,nw,f,fs)
[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs)
[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs,p)
[Pxx,Pxxc,f] = pmtm(x,e,v,nfft,fs,p)
[Pxx,Pxxc,f] = pmtm(x,dpss_params,nfft,fs,p)
[...] = pmtm(...,'DropLastTaper',dropflag)
[...] = pmtm(...,'method')
[...] = pmtm(...,'range')
pmtm(...)
```

Description   pmtm estimates the power spectral density (PSD) of the time series *x* using the multitaper method (MTM) described in [1]. This method uses linear or nonlinear combinations of modified periodograms to estimate the PSD. These periodograms are computed using a sequence of orthogonal tapers (windows in the frequency domain) specified from the discrete prolate spheroidal sequences (see dpss).

[Pxx,w] = pmtm(x,nw) estimates the PSD Pxx for the input signal x, using 2*nw-1 discrete prolate spheroidal sequences as data tapers for the multitaper estimation method. nw is the time-bandwidth product for the discrete prolate spheroidal sequences. If you specify nw as the empty vector [], a default value of 4 is used. Other typical choices are 2, 5/2, 3, or 7/2. pmtm also returns w, a vector of frequencies at which the PSD is estimated. Pxx and w have the same length. The units for frequency are rad/sample.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce (by default) full power one-sided (in frequency) PSDs, while complex-valued inputs produce two-sided PSDs.

In general, the length *N* of the FFT and the values of the input x determine the length of Pxx and the range of the corresponding normalized frequencies. For this syntax, the (default) length *N* of the FFT is the larger of 256 and the next power of 2 greater than the length of the segment. The following table indicates the length of Pxx and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	(*N*/2) +1	[0, π]
Complex-valued	*N*	[0, 2π)

[Pxx,w] = pmtm(x,nw,nfft) uses the multitaper method to estimate the PSD while specifying the length of the FFT with the integer nfft. If you specify nfft as the empty vector [], it adopts the default value for *N* described in the previous syntax.

The length of Pxx and the frequency range for w depend on nfft and the values of the input x. The following table indicates the length of Pxx and the frequency range for w for this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	[0, π]
Real-valued	Odd	(nfft + 1)/2	[0, π)
Complex-valued	Even or odd	nfft	[0, 2π)

[Pxx,w] = pmtm(x,nw,w) estimates the two-sided PSD at the normalized frequencies specified in the vector w using the Goertzel algorithm. The frequencies of w are rounded to the nearest DFT bin

commensurate with the resolution of the signal. The units of w are rad/sample.

[Pxx,f] = pmtm(x,nw,nfft,fs) uses the sampling frequency fs specified as an integer in hertz (Hz) to compute the PSD vector (Pxx) and the corresponding vector of frequencies (f). In this case, the units for the frequency vector f are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify fs as the empty vector [], the sampling frequency defaults to 1 Hz.

The frequency range for f depends on nfft, fs, and the values of the input x. The length of Pxx is the same as in the table, PSD and Frequency Vector Characteristics on page 3-736 above. The following table indicates the frequency range for f for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	[0, fs/2]
Real-valued	Odd	[0, fs/2)
Complex-valued	Even or odd	[0, fs)

[Pxx,w] = pmtm(x,nw,f,fs) estimates the two-sided PSD at the frequencies specified in the vector f using the Goertzel algorithm. The frequencies of f are rounded to the nearest DFT bin commensurate with the resolution of the signal. The units of w are rad/sample.

[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs) returns Pxxc, the 95% confidence interval for Pxx. Confidence intervals are computed using a chi-squared approach. Pxxc is a two-column matrix with the same number of rows as Pxx. Pxxc(:,1) is the lower bound of the confidence interval and Pxxc(:,2) is the upper bound of the confidence interval.

[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs,p) returns Pxxc, the p*100% confidence interval for Pxx, where p is a scalar between 0 and 1. If you don't specify p, or if you specify p as the empty vector [], the default 95% confidence interval is used.

[Pxx,Pxxc,f] = pmtm(x,e,v,nfft,fs,p) returns the PSD estimate Pxx, the confidence interval Pxxc, and the frequency vector f from the data tapers contained in the columns of the matrix e, and their concentrations in the vector v. The length of v is the same as the number of columns in e. You can obtain the data to supply as these arguments from the outputs of dpss.

[Pxx,Pxxc,f] = pmtm(x,dpss_params,nfft,fs,p) uses the cell array dpss_params containing the input arguments to dpss (listed in order, but excluding the first argument) to compute the data tapers. For example, pmtm(x,{3.5,'trace'},512,1000) calculates the prolate spheroidal sequences for nw = 3.5, using nfft = 512, and fs = 1000, and displays the method that dpss uses for this calculation. See dpss for other options.

[...] = pmtm(...,'DropLastTaper',dropflag) specifies whether the last taper or eigenvector should be dropped during the MTM calculation of the PSD. The last taper is associated with an eigenvalue that is typically much smaller than one and this taper is usually not included in the calculation. The default value of dropflag is true. Setting dropflag to false forces the last taper to be included in the calculation.

[...] = pmtm(...,'*method*') specifies the algorithm used for combining the individual spectral estimates. The string '*method*' can be one of the following:

- 'adapt': Thomson's adaptive nonlinear combination (default)

- 'unity': A linear combination of the weighted periodograms with unity weights

- 'eigen': A linear combination of the weighted periodograms with eigenvalue weights

[...] = pmtm(...,'*range*') specifies the range of frequency values to include in f or w. This syntax is useful when x is real. '*range*' can be either:

- 'twosided': Compute the two-sided PSD over the frequency range [0,fs). This is the default for determining the frequency range for complex-valued x.

- If you specify fs as the empty vector, [], the frequency range is [0,1).

- If you don't specify fs, the frequency range is [0, 2π).

- 'onesided': Compute the one-sided PSD over the frequency ranges specified for real x. This is the default for determining the frequency range for real-valued x.

Note You can put the string arguments '*range*' or '*method*' anywhere after the input argument nw or v.

---

pmtm(...) with no output arguments plots the PSD estimate and the confidence intervals in the current figure window. If you don't specify fs, the 95% confidence interval is plotted. If you do specify fs, the confidence intervals plotted depend on the value of p.

Examples   This example analyzes a sinusoid in white noise:

```
fs = 1000;
t = 0:1/fs:0.3;
x = cos(2*pi*t*200) + 0.1*randn(size(t));
[Pxx,Pxxc,f] = pmtm(x,3.5,512,fs,0.99);
hpsd = dspdata.psd([Pxx Pxxc],'Fs',fs);
plot(hpsd)
```

pmtm

References   [1] Percival, D.B., and A.T. Walden, *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*, Cambridge University Press, 1993.

[2] Thomson, D.J., "Spectrum estimation and harmonic analysis," *Proceedings of the IEEE, Vol. 70* (1982), pp. 1055-1096.

See Also   dpss | pburg | pcov | peig | periodogram | pmcov | pmusic | pwelch | pyulear | spectrum.mtm

Purpose   Pseudospectrum using MUSIC algorithm

Syntax
```
[S,w] = pmusic(x,p)
[S,w] = pmusic(x,p,w)
[S,w] = pmusic(...,nfft)
[S,f] = pmusic(x,p,nfft,fs)
[S,f] = pmusic(x,p,f,fs)
[S,f] = pmusic(...,'corr')
[S,f] = pmusic(x,p,nfft,fs,nwin,noverlap)
[...] = pmusic(...,'range')
[...,v,e] = pmusic(...)
pmusic(...)
```

Description   `[S,w] = pmusic(x,p)` implements the MUSIC (Multiple Signal
Classification) algorithm and returns `S`, the pseudospectrum estimate
of the input signal `x`, and a vector `w` of normalized frequencies
(in rad/sample) at which the pseudospectrum is evaluated. The
pseudospectrum is calculated using estimates of the eigenvectors of a
correlation matrix associated with the input data `x`, where `x` is specified
as either:

- A row or column vector representing one observation of the signal

- A rectangular array for which each row of `x` represents a separate
  observation of the signal (for example, each row is one output of
  an array of sensors, as in array processing), such that `x'*x` is an
  estimate of the correlation matrix

Note You can use the output of `corrmtx` to generate such an array `x`.

You can specify the second input argument `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.

- A two-element vector. In this case, `p(2)`, the second element of
  `p`, represents a threshold that is multiplied by λ_{min}, the smallest
  estimated eigenvalue of the signal's correlation matrix. Eigenvalues

below the threshold λ_{min}*p(2) are assigned to the noise subspace. In this case, p(1) specifies the maximum dimension of the signal subspace.

Note If the inputs to pmusic are real sinusoids, set the value of p to double the number of input signals. If the inputs are complex sinusoids, set p equal to the number of inputs.

The extra threshold parameter in the second entry in p provides you more flexibility and control in assigning the noise and signal subspaces.

S and w have the same length. In general, the length of the FFT and the values of the input x determine the length of the computed S and the range of the corresponding normalized frequencies. The following table indicates the length of S (and w) and the range of the corresponding normalized frequencies for this syntax.

S Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of S and w	Range of the Corresponding Normalized Frequencies
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

[S,w] = pmusic(x,p,w) returns the pseudospectrum in the vector S computed at the normalized frequencies specified in vector w, which has two or more elements

[S,w] = pmusic(...,nfft) specifies the integer length of the FFT nfft used to estimate the pseudospectrum. The default value for nfft (entered as an empty vector []) is 256.

The following table indicates the length of S and w, and the frequency range for w in this syntax.

S and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of S and w	Range of w
Real-valued	Even	`(nfft/2 + 1)`	[0, π]
Real-valued	Odd	`(nfft + 1)/2`	[0, π)
Complex-valued	Even or odd	`nfft`	[0, 2π)

`[S,f] = pmusic(x,p,nfft,fs)` returns the pseudospectrum in the vector S evaluated at the corresponding vector of frequencies f (in Hz). You supply the sampling frequency fs in Hz. If you specify fs with the empty vector `[]`, the sampling frequency defaults to 1 Hz.

The frequency range for f depends on nfft, fs, and the values of the input x. The length of S (and f) is the same as in the S and Frequency Vector Characteristics on page 3-743 above. The following table indicates the frequency range for f for this syntax.

S and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[S,f] = pmusic(x,p,f,fs)` returns the pseudospectrum in the vector S computed at the frequencies specified in vector f, which has two or more elements

`[S,f] = pmusic(...,'corr')` forces the input argument x to be interpreted as a correlation matrix rather than matrix of signal data. For this syntax x must be a square matrix, and all of its eigenvalues must be nonnegative.

[S,f] = pmusic(x,p,nfft,fs,nwin,noverlap) allows you to specify nwin, a scalar integer indicating a rectangular window length, or a real-valued vector specifying window coefficients. Use the scalar integer noverlap in conjunction with nwin to specify the number of input sample points by which successive windows overlap. noverlap is not used if x is a matrix. The default value for nwin is 2*p(1) and noverlap is nwin-1.

With this syntax, the input data x is segmented and windowed before the matrix used to estimate the correlation matrix eigenvalues is formulated. The segmentation of the data depends on nwin, noverlap, and the form of x. Comments on the resulting windowed segments are described in the following table.

Windowed Data Depending on x and nwin

Input data x	Form of nwin	Windowed Data
Data vector	Scalar	Length is nwin
Data vector	Vector of coefficients	Length is length(nwin)
Data matrix	Scalar	Data is not windowed.
Data matrix	Vector of coefficients	length(nwin) must be the same as the column length of x, and noverlap is not used.

See the Eigenvector Length Depending on Input Data and Syntax on page 3-746 below for related information on this syntax.

Note The arguments nwin and noverlap are ignored when you include the string 'corr' in the syntax.

[...] = pmusic(...,'*range*') specifies the range of frequency values to include in f or w. This syntax is useful when x is real. '*range*' can be either:

- `'whole'`: Compute the pseudospectrum over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.

  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

  - If you don't specify `fs`, the frequency range is $[0, 2\pi)$.

- `'half'`: Compute the pseudospectrum over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

Note You can put the string arguments `'range'` or `'corr'` anywhere in the input argument list after `p`.

`[...,v,e] = pmusic(...)` returns the matrix `v` of noise eigenvectors, along with the associated eigenvalues in the vector `e`. The columns of `v` span the noise subspace of dimension `size(v,2)`. The dimension of the signal subspace is `size(v,1)-size(v,2)`. For this syntax, `e` is a vector of estimated eigenvalues of the correlation matrix.

`pmusic(...)` with no output arguments plots the pseudospectrum in the current figure window.

Tips   In the process of estimating the pseudospectrum, `pmusic` computes the noise and signal subspaces from the estimated eigenvectors v_j and eigenvalues λ_j of the signal's correlation matrix. The smallest of these eigenvalues is used in conjunction with the threshold parameter `p(2)` to affect the dimension of the noise subspace in some cases.

The length *n* of the eigenvectors computed by `pmusic` is the sum of the dimensions of the signal and noise subspaces. This eigenvector length depends on your input (signal data or correlation matrix) and the syntax you use.

The following table summarizes the dependency of the eigenvector length on the input argument.

Eigenvector Length Depending on Input Data and Syntax

Form of Input Data x	Comments on the Syntax	Length n of Eigenvectors
Row or column vector	nwin is specified as a scalar integer.	nwin
Row or column vector	nwin is specified as a vector.	length(nwin)
Row or column vector	nwin is not specified.	2*p(1)
l-by-*m* matrix	If nwin is specified as a scalar, it is not used. If nwin is specified as a vector, length(nwin) must equal *m*.	*m*
m-by-*m* nonnegative definite matrix	The string 'corr' is specified and nwin is not used.	*m*

You should specify nwin > p(1) or length(nwin) > p(1) if you want p(2) > 1 to have any effect.

Examples

Example 1: pmusic with no Sampling Specified

This example analyzes a signal vector x, assuming that two real sinusoidal components are present in the signal subspace. In this case, the dimension of the signal subspace is 4 because each real sinusoid is the sum of two complex exponentials:

```
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
pmusic(x,4)      % Set p to 4 because two real inputs
```

Example 2: Specifying Sampling Frequency and Subspace Dimensions

This example analyzes the same signal vector x with an eigenvalue cutoff of 10% above the minimum. Setting p(1) = Inf forces the signal/noise subspace decision to be based on the threshold parameter p(2). Specify the eigenvectors of length 7 using the nwin argument, and set the sampling frequency fs to 8 kHz:

```
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
[P,f] = pmusic(x,[Inf,1.1],[],8000,7); % Window length = 7
```

Example 3: Entering a Correlation Matrix

Supply a positive definite correlation matrix R for estimating the spectral density. Use the default 256 samples:

```
R = toeplitz(cos(0.1*pi*[0:6])) + 0.1*eye(7);
[P,f] = pmusic(R,4,'corr');
```

Example 4: Entering a Signal Data Matrix Generated from corrmtx

Enter a signal data matrix Xm generated from data using corrmtx:

```
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
Xm = corrmtx(x,7,'mod');
[P,w] = pmusic(Xm,2);
```

Example 5: Using Windowing to Create the Effect of a Signal Data Matrix

Use the same signal, but let pmusic form the 100-by-7 data matrix using its windowing input arguments. In addition, specify an FFT of length 512:

```
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
[PP,ff] = pmusic(x,2,512,[],7,0);
```

Algorithms   The name MUSIC is an acronym for MUltiple SIgnal Classification. The MUSIC algorithm estimates the pseudospectrum from a signal or a correlation matrix using Schmidt's eigenspace analysis method [1]. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content. This algorithm is particularly suitable for signals that are the sum of sinusoids with additive white Gaussian noise. The eigenvalues and eigenvectors of the signal's correlation matrix are estimated if you don't supply the correlation matrix.

The MUSIC pseudospectrum estimate is given by

$$P_{music}(f) = \frac{1}{e^H(f)\left(\sum\limits_{k=p+1}^{N} v_k v_k^H\right)e(f)} = \frac{1}{\sum\limits_{k=p+1}^{N} |v_k^H e(f)|^2}$$

where N is the dimension of the eigenvectors and v_k is the k-th eigenvector of the correlation matrix. The integer p is the dimension of the signal subspace, so the eigenvectors v_k used in the sum correspond to the smallest eigenvalues and also span the noise subspace. The vector e(f) consists of complex exponentials, so the inner product

$$v_k^H e(f)$$

amounts to a Fourier transform. This is used for computation of the pseudospectrum estimate. The FFT is computed for each v_k and then the squared magnitudes are summed.

References   [1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 373-378.

[2] Schmidt, R.O, "Multiple Emitter Location and Signal Parameter Estimation," *IEEE Trans. Antennas Propagation, Vol. AP-34* (March 1986), pp. 276-280.

[3] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1997.

See Also   corrmtx | dspdata | pburg | peig | periodogram | pmtm | prony | pwelch | rooteig | rootmusic | spectrum.music

poly2ac

Purpose	Convert prediction filter polynomial to autocorrelation sequence
Syntax	`r = poly2ac(a,efinal)`
Description	`r = poly2ac(a,efinal)` finds the autocorrelation vector `r` corresponding to the prediction filter polynomial `a`. The autocorrelation sequence produced is approximately the same as that of the output of the autoregressive prediction filter whose coefficients are determined by `a`. `poly2ac` also produces the final `length(r)` step prediction error `efinal`. If `a(1)` is not equal to 1, `poly2ac` normalizes the prediction filter polynomial by `a(1)`. `a(1)` cannot be 0.
Tips	You can apply this function to both real and complex polynomials.

Examples	`a = [1.0000 0.6147 0.9898 0.0004 0.0034 -0.0077];`
	`efinal = 0.2;`
	`r = poly2ac(a,efinal)`

References	[1] Kay, S.M. *Modern Spectral Estimation*, Englewood Cliffs, NJ, Prentice-Hall, 1988
See Also	`ac2poly` | `poly2rc` | `rc2ac`

Purpose   Convert prediction filter coefficients to line spectral frequencies

Syntax   lsf = poly2lsf(a)

Description   lsf = poly2lsf(a) returns a vector lsf of line spectral frequencies
                   from a vector a of prediction filter coefficients.

Examples   a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082];
                   lsf = poly2lsf(a)

References   [1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time
                   Processing of Speech Signals*, Prentice-Hall, 1993.

                   [2] Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech
                   Signals*, Prentice-Hall, 1978.

See Also   lsf2poly

poly2rc

Purpose   Convert prediction filter polynomial to reflection coefficients

Syntax   ```
k = poly2rc(a)
[k,r0] = poly2rc(a,efinal)
```

Description   `k = poly2rc(a)` converts the prediction filter polynomial `a` to the reflection coefficients of the corresponding lattice structure. `a` can be real or complex, and `a(1)` cannot be 0. If `a(1)` is not equal to 1, poly2rc normalizes the prediction filter polynomial by `a(1)`. `k` is a row vector of size `length(a)-1`.

[k,r0] = poly2rc(a,efinal)` returns the zero-lag autocorrelation, r0, based on the final prediction error, `efinal`.

A simple, fast way to check if `a` has all of its roots inside the unit circle is to check if each of the elements of `k` has magnitude less than 1.

```
stable = all(abs(poly2rc(a))<1)
```

Examples   ```
a = [1.0000   0.6149   0.9899   0.0000   0.0031 -0.0082];
efinal = 0.2;
[k,r0] = poly2rc(a,efinal)
```

**Limitations**    If `abs(k(i)) == 1` for any `i`, finding the reflection coefficients is an ill-conditioned problem. `poly2rc` returns some `NaN`s and provide a warning message in this case.

**Algorithms**     `poly2rc` implements this recursive relationship:

$$k(n) = a_n(n)$$
$$a_{n-1}(m) = \frac{a_n(m) - k(n)a_n(n-m)}{1 - k(n)^2}, \quad m = 1, 2, \cdots, n-1$$

This relationship is based on Levinson's recursion [1]. To implement it, poly2rc loops through `a` in reverse order after discarding its first element. For each loop iteration `i`, the function:

**1** Sets `k(i)` equal to `a(i)`

**2** Applies the second relationship above to elements 1 through `i` of the vector `a`.

```
a = (a-k(i)*fliplr(a))/(1-k(i)^2);
```

References [1] Kay, S.M. *Modern Spectral Estimation*, Englewood Cliffs, NJ, Prentice-Hall, 1988.

See Also ac2rc | latc2tf | latcfilt | poly2ac | rc2poly | tf2latc

polyscale

Purpose	Scale roots of polynomial
Syntax	`b = polyscale(a,alpha)`

Description   `b = polyscale(a,alpha)` scales the roots of a polynomial in the *z*-plane, where `a` is a vector containing the polynomial coefficients and `alpha` is the scaling factor.

If `alpha` is a real value in the range `[0 1]`, then the roots of `a` are radially scaled toward the origin in the *z*-plane. Complex values for `alpha` allow arbitrary changes to the root locations.

Tips   By reducing the radius of the roots in an autoregressive polynomial, the bandwidth of the spectral peaks in the frequency response is expanded (flattened). This operation is often referred to as *bandwidth expansion*.

See Also   `polystab | roots`

polystab

Purpose	Stabilize polynomial
Syntax	b = polystab(a)

Description   polystab stabilizes a polynomial with respect to the unit circle; it reflects roots with magnitudes greater than 1 inside the unit circle.

b = polystab(a) returns a row vector b containing the stabilized polynomial, where a is a vector of polynomial coefficients, normally in the *z*-domain.

$$A(z) = a(1) + a(2)z^{-1} + \ldots + a(m+1)z^{-m}$$

Examples   polystab can convert a linear-phase filter into a minimum-phase filter with the same magnitude response:

```
h = fir1(25,0.4);
hmin = polystab(h) * norm(h)/norm(polystab(h));
```

Algorithms   polystab finds the roots of the polynomial and maps those roots found outside the unit circle to the inside of the unit circle:

```
v = roots(a);
vs = 0.5*(sign(abs(v)-1)+1);
v = (1-vs).*v + vs./conj(v);
b = a(1)*poly(v);
```

See Also   roots

pow2db

Purpose   Convert power to decibels (dB)

Syntax   ydb = pow2db(y)

Description   ydb = pow2db(y) returns the corresponding decibel (dB) value ydb for
                     a given power value y. The relationship between power and decibels
                     is ydb = $10*\log_{10}$(y).

See Also   db2pow

Purpose   Prony method for filter design

Syntax   [Num,Den] = prony(impulse_resp,num_ord,denom_ord)

Description   [Num,Den] = prony(impulse_resp,num_ord,denom_ord) returns
the numerator Num and denominator Den coefficients for a causal
rational system function with impulse response impulse_resp. The
system function has numerator order num_ord and denominator
order denom_ord. The lengths of Num and Den are num_ord+1 and
denom_ord+1. If the length of impulse_resp is less than the largest
order (num_ord or denom_ord), impulse_resp is padded with zeros.
Enter 0 in num_ord for an all-pole system function. For an all-zero
system function, enter a 0 for denom_ord.

Definitions   The *system function* is the z-transform of the impulse response *h[n]*:

$$H(z) = \sum_{n=-\infty}^{\infty} h[n]z^{-n}$$

A *rational system function* is a ratio of polynomials in z^{-1}. By convention
the numerator polynomial is *B(z)* and the denominator is *A(z)*. The
following equation describes a causal rational system function of
numerator order num_ord *q* and denominator order denom_ord *p* :

$$H(z) = \frac{\sum_{k=0}^{q} b[k]z^{-k}}{1 + \sum_{l=1}^{p} a[l]z^{-l}}$$

where *a[0]=1*.

Examples   Fit IIR model to an impulse response of lowpass filter:

```
d=fdesign.lowpass('Nb,Na,F3dB',4,4,0.2);
% Butterworth filter design
Hd=design(d,'butter');
```

```
% Obtain impulse response
impulse_resp=filter(Hd,[1 zeros(1,31)]);
% Find system function of order 4
denom_order=4; num_order=4;
[Num,Den]=prony(impulse_resp,num_order,denom_order);
% Compare impulse response with input
subplot(211);
stem(impz(Num,Den,length(impulse_resp)));
title('Impulse Response with Prony Design');
subplot(212);
stem(impulse_resp); title('Input Impulse Response');
```


Fit FIR model to an impulse response of highpass filter:

```
d=fdesign.highpass('N,F3dB',10,0.8);
Hd=design(d,'maxflat');
% Impulse response
impulse_resp=filter(Hd,[1 zeros(1,31)]);
% Find all-zero system function of order 10
```

```
num_order=10; denom_order=0;
[Num,Den]=prony(impulse_resp,num_order,denom_order);
% Compare Num to Hd.Numerator.
% Num and Hd.Numerator will not be identical but the
% coefficients will be close in value.
```

References   Parks, T.W., and C.S. Burrus *Digital Filter Design*, John Wiley & Sons, 1987, pp., 226–228.

See Also   design | fdesign | impz | levinson | lpc

How To   • "Parametric Modeling"

pulseperiod

Purpose Period of bilevel pulse

Syntax
```
P = pulseperiod(X)
P = pulseperiod(X,FS)
P = pulseperiod(X,T)
[P,INITCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] =
pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] =
pulseperiod(...,
    Name,Value)
```

Description P = pulseperiod(X) returns a vector, P, containing the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition in the bilevel waveform,X. If pulseperiod does not find two positive-polarity transitions, P is empty. To determine the transitions for each pulse, pulseperiod estimates the state levels of the input waveform by a histogram method and identifies all regions which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See "State-Level Tolerances" on page 3-765. Because pulseperiod uses interpolation to determine the mid-reference level instants, P may contain values that do not correspond to sampling instants of the bilevel waveform, X.

P = pulseperiod(X,FS) specifies the sampling rate in hertz as a positive scalar. The first sample instant in X corresponds to t=0. Because pulseperiod uses interpolation to determine the mid-reference level instants, P may contain values that do not correspond to sampling instants of the bilevel waveform, X.

P = pulseperiod(X,T) specifies the sampling instants in a vector equal in length to X. Because pulseperiod uses interpolation to

determine the mid-reference level instants, P may contain values that do not correspond to sampling instants of the bilevel waveform, X.

`[P,INITCROSS] = pulseperiod(...)` returns the mid-reference level instants of the first transition of each pulse.

`[P,INITCROSS,FINALCROSS] = pulseperiod(...)` returns the mid-reference level instants of the final transition of each pulse.

`[P,INITCROSS,FINALCROSS,NEXTCROSS] = pulseperiod(...)` returns the mid-reference level instants of next detected transition after each pulse.

`[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulseperiod(...)` returns the mid-reference level,`MIDLEV`.

`[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulseperiod(...,Name,Value)` returns the pulse periods with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

X

Bilevel waveform. If the waveform, X, does not contain at least two transitions, `pulseperiod` outputs an empty matrix.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

MidPct

Mid-reference level as a percentage of the waveform amplitude.

> **Default:** 50

pulseperiod

Polarity

Pulse polarity. Specify the polarity as `'positive'` or `'negative'`. If you specify `'positive'`, pulseperiod looks for pulses whose initial transition is positive-going (positive polarity). If you specify `'negative'`, pulseperiod looks for pulses whose initial transition is negative-going (negative polarity).

> **Default:** `'positive'`

StateLevels

Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low and high-state levels, pulseperiod estimates the state levels from the input waveform using the histogram method.

Tolerance

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-765.

> **Default:** 2

Output Arguments

P

Pulse period in seconds. The pulse period is defined as the time between the mid-reference level instants of two consecutive transitions.

INITCROSS

Mid-reference level instant of initial transition.

FINALCROSS

Mid-reference level instant of final transition.

NEXTCROSS

Mid-reference level instant of the first pulse transition after the final transition of the preceding pulse.

MIDLEV

Waveform value that corresponds to the mid-reference level.

Definitions   **Mid-Reference Level**

The mid-reference level in a bilevel waveform with low-state level, *S_1*, and high- state level, *S_2*, is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

Mid-Reference Level Instant

Let $y_{50\%}$ denote the mid-reference level.

Let $t_{50\%-}$ and $t_{50\%+}$ denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.

Let $y_{50\%-}$ and $y_{50\%+}$ denote the waveform values at $t_{50\%-}$ and $t_{50\%+}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

Pulse Polarity

If the initial transition of a pulse is positive-going, the pulse has positive polarity. The following figure shows a positive-polarity pulse.

Positive Polarity Pulse

Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the initial transition of a pulse is negative-going, the pulse has negative polarity. The following figure shows a negative-polarity pulse.

Negative Polarity Pulse

Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \tfrac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

State Levels ± 2% Tolerance

Examples

Pulse Period of Bilevel Waveform

Compute the pulse period of a bilevel waveform with two positive-polarity transitions. The sampling rate is 4 MHz.

pulseperiod

```
load('pulseex.mat', 'x', 't');
p = pulseperiod(x, t);
```

Determine Mid-Reference Level Instants of Pulse Period

Determine the mid-reference level instants, which define the pulse period for a bilevel waveform. Mark the mid-reference level instants on a plot of the data.

```
load('pulseex.mat', 'x', 't');
[p,initcross,~,nextcross,midlev] = pulseperiod(x,t);
fprintf('Pulse period is %2.3f microseconds \n',p*1e6);
plot(t.*1e6,x); hold on;
grid on; axis tight; xlabel('microseconds');
plot(initcross.*1e6,midlev,'ro','markerfacecolor',[1 0 0]);
plot(nextcross.*1e6,midlev,'ro','markerfacecolor',[1 0 0]);
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Standard 181, 2003.

See Also   dutycycle | pulsesep | pulsewidth | statelevels

pulsesep

Purpose   Separation between bilevel waveform pulses

Syntax
```
S = pulsesep(X)
S = pulsesep(X,FS)
S = pulsesep(X,T)
[S,INITCROSS] = pulsesep(...)
[S,INITCROSS,FINALCROSS] = pulsesep(...)
[S,INITCROSS,FINALCROSS,NEXTCROSS] = pulsesep(...)
[S,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulsesep(...)
[S,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] =
pulsesep(...,Name,
    Value)
```

Description   `S = pulsesep(X)` returns the differences, `S`, between the mid-reference level instants of the final negative-going transitions of every positive-polarity pulse and the next positive-going transition. `X` is a bilevel waveform. To determine the transitions that compose each pulse, `pulsesep` estimates the state levels of `X` by a histogram method. `pulsesep` identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See "State-Level Tolerances" on page 3-775. Because `pulsesep` uses interpolation to determine the mid-reference level instants, `S` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`S = pulsesep(X,FS)` specifies the sampling rate, `FS`, in Hz as a positive scalar. The first time instant corresponds to t=0. Because `pulsesep` uses interpolation to determine the mid-reference level instants, `S` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`S = pulsesep(X,T)` specifies the sampling instants, `T`, in a vector equal in length to `X`. Because `pulsesep` uses interpolation to determine the mid-reference level instants, `S` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`[S,INITCROSS] = pulsesep(...)` returns the mid-reference level instants, `INITCROSS`, of the first positive-polarity transitions.

`[S,INITCROSS,FINALCROSS] = pulsesep(...)` returns the mid-reference level instants, `FINALCROSS`, of the final transition of each pulse.

`[S,INITCROSS,FINALCROSS,NEXTCROSS] = pulsesep(...)` returns the mid-reference level instants, `NEXTCROSS`, of the next detected transition after each pulse.

`[S,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulsesep(...)` returns the mid-reference level, `MIDLEV`.

`[S,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulsesep(...,Name,Value)` returns the pulse separations with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

X

Bilevel waveform. If the waveform, `X`, does not contain at least two transitions, `pulsesep` outputs an empty matrix.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of `T` must equal the length of the bilevel waveform, `X`.

Name-Value Pair Arguments

MidPct

Mid-reference level as a percentage of the waveform amplitude.

Default: 50

pulsesep

Polarity

Pulse polarity. Specify the polarity as `'positive'` or `'negative'`. If you specify `'positive'`, `pulsesep` looks for pulses with positive-going (positive polarity) initial transitions. If you specify `'negative'`, `pulsesep` looks for pulses with negative-going (negative polarity) initial transitions. See "Pulse Polarity" on page 3-773.

Default: `'positive'`

StateLevels

Low- and high-state levels. `StateLevels` is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, `pulsesep` estimates the state levels from the input waveform using the histogram method.

Tolerance

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-775.

Default: 2

Output Arguments

S

Pulse separations in seconds. The *pulse separation* is defined as the time between the mid-reference level instants of the final transition of one pulse and the initial transition of the next pulse. See "Pulse Separation" on page 3-777.

INITCROSS

Mid-reference level instants of initial transition.

FINALCROSS

Mid-reference level instants of final transition.

NEXTCROSS

Mid-reference level instants of the initial transition after the final transition of the preceding pulse.

MIDLEV

Waveform value that corresponds to the mid-reference level.

Definitions   Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level, S_1, and high-state level, S_2, is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

Mid-Reference Level Instant

Let $y_{50\%}$ denote the mid-reference level.

Let $t_{50\%-}$ and $t_{50\%+}$ denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.

Let $y_{50\%-}$ and $y_{50\%+}$ denote the waveform values at $t_{50\%-}$ and $t_{50\%+}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

Pulse Polarity

If the pulse has an initial positive-going transition, the pulse has positive polarity. The following figure shows a positive-polarity pulse.

Positive Polarity Pulse

Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has an initial negative-going transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.

Negative Polarity Pulse

Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

State-Level Tolerances

Each state level can have an associated lower- and upper-state boundary. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \tfrac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

State Levels ± 2% Tolerance

Pulse Separation

Pulse separation is the time difference between the mid-reference level instant of the final transition of one pulse and the mid-reference level instant of the initial transition of the next pulse. The following figure illustrates pulse separation.

Examples

Pulse Separation in Bilevel Waveform

Compute the pulse separation in a bilevel waveform with two positive-polarity transitions. The sampling rate is 4 MHz.

```
load('pulseex.mat', 'x', 't');
s = pulsesep(x, t);
```

Determine Mid-Reference Level Instants Defining Pulse Separation

Determine the mid-reference level instants, which define the pulse separation for a bilevel waveform. Mark the mid-reference level instants on a plot of the data.

```
load('pulseex.mat', 'x', 't');
[s,~,finalcross,nextcross,midlev] = pulsesep(x,t);
fprintf('Pulse separation is %2.3f microseconds \n',s*1e6);
plot(t.*1e6,x); hold on;
grid on; axis tight; xlabel('microseconds');
plot(finalcross.*1e6,midlev,'ro','markerfacecolor',[1 0 0]);
plot(nextcross.*1e6,midlev,'ro','markerfacecolor',[1 0 0]);
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Standard 181, 2003.

See Also   dutycycle | pulseperiod | pulsewidth | statelevels

Purpose   Bilevel waveform pulse width

Syntax   W = pulsewidth(X)
                 W = pulsewidth(X,FS)
                 W = pulsewidth(X,T)
                 [W,INITCROSS] = pulsewidth(...)
                 [W,INITCROSS,FINALCROSS] = pulsewidth(...)
                 [W,INITCROSS,FINALCROSS,MIDLEV] = pulsewidth(...)
                 W = pulsewidth(...,Name,Value)

Description W = pulsewidth(X) returns a vector, W, containing the time differences
                 between the mid-reference level instants of the initial and final
                 transitions of each positive-polarity pulse in the bilevel waveforml,
                 X. To determine the transitions, pulsewidth estimates the low- and
                 high-state levels of X by a histogram method. pulsewidth identifies all
                 regions that cross the upper-state boundary of the low state and the
                 lower-state boundary of the high state. The low-state and high-state
                 boundaries are expressed as the state level plus or minus a multiple of
                 the difference between the state levels. See "State-Level Tolerances"
                 on page 3-786. Because pulsewidth uses interpolation to determine
                 the mid-reference level instants, W may contain values that do not
                 correspond to sampling instants of the bilevel waveform, X.

                 W = pulsewidth(X,FS) specifies the sample rate, FS, in hertz as a
                 positive scalar. The first sample in the waveform corresponds to t=0.
                 Because pulsewidth uses interpolation to determine the mid-reference
                 level instants, W may contain values that do not correspond to sampling
                 instants of the bilevel waveform, X.

                 W = pulsewidth(X,T) specifies the sample instants, T, as a vector
                 with the same number of elements as X. Because pulsewidth uses
                 interpolation to determine the mid-reference level instants, W may
                 contain values that do not correspond to sampling instants of the bilevel
                 waveform, X.

                 [W,INITCROSS] = pulsewidth(...) returns a column vector,
                 INITCROSS, whose elements correspond to the mid-reference level
                 instants of the initial transition of each pulse.

pulsewidth

[W,INITCROSS,FINALCROSS] = pulsewidth(...) returns a column vector, FINALCROSS, whose elements correspond to the mid-reference level instants of the final transition of each pulse.

[W,INITCROSS,FINALCROSS,MIDLEV] = pulsewidth(...) returns the waveform value, MIDLEV, which corresponds to the mid-reference level.

W = pulsewidth(...,Name,Value) returns the pulse widths with additional options specified by one or more Name,Value pair arguments.

Input Arguments

X

Bilevel waveform. X is a real-valued row or column vector.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

MidPct

Mid-reference level as percentage of the waveform amplitude. See "Mid-Reference Level" on page 3-783.

> **Default:** 50

Polarity

Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', pulsewidth looks for pulses with positive-going (positive polarity) initial transitions. If you specify 'negative', pulsewidth looks for pulses with negative-going (negative polarity) initial transitions. See "Pulse Polarity" on page 3-784.

Default: `'positive'`

StateLevels

Low- and high-state levels. `StateLevels` is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, `pulsewidth` estimates the state levels from the input waveform using the histogram method.

Tolerance

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-786.

Default: 2

Output Arguments

W

Pulse widths in seconds. The pulse width is the time difference between the initial and final transitions of a pulse. The times of the initial and final transitions are referred to as *transition occurence instants* in [1].

INITCROSS

Mid-reference level instants of the initial transition

FINALCROSS

Mid-reference level instants of the final transition

MIDLEV

Waveform value corresponding to the mid-reference level

Definitions

Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level, S_1, and high- state level, S_2, is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

Mid-Reference Level Instant

Let $y_{50\%}$ denote the mid-reference level.

Let $t_{50\%-}$ and $t_{50\%+}$ denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.

Let $y_{50\%-}$ and $y_{50\%+}$ denote the waveform values at $t_{50\%-}$ and $t_{50\%+}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

Pulse Polarity

If the pulse has a positive-going initial transition, the pulse has positive polarity. The following figure shows a positive-polarity pulse.

Positive Polarity Pulse

Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has a negative-going initial transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.

pulsewidth

Negative Polarity Pulse

Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

State-Level Tolerances

Each state level can have an associated lower- and upper-state boundary. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

State Levels ± 2% Tolerance

Examples

Pulse Width of Bilevel Waveform

Compute the pulse width of a bilevel waveform sampled at 4 MHz.

```
load('pulseex.mat', 'x', 't');
```

```
w = pulsewidth(x, t);
plot(t,x); grid on;
```

First and Second Transition Times for Bilevel Waveform

Compute the initial and final transition occurrences for a bilevel waveform sampled at 4 MHz. Plot the result annotated with the transition occurrences.

```
load('pulseex.mat', 'x', 't');
fs = 4e6;
[w,initcross,finalcross] = pulsewidth(x,fs);
plot(t.*1e6,x);
set(gca,'xtick',[initcross*1e6 finalcross*1e6]);
grid on;
xlabel('microseconds');
```

pulsewidth

Specify State Levels for Bilevel Waveform

Specify the state levels for the bilevel waveform instead of estimating the levels from the data. Use the `'StateLevels'` name-value pair to enter the low-state level as 0 and the high-state level as 5.

```
load('pulseex.mat', 'x', 't');
[w,initcross,finalcross] = pulsewidth(x,fs,'StateLevels',[0 5]);
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

See Also   dutycycle | pulseperiod | pulsesep | statelevels

pulstran

Purpose   Pulse train

Syntax
```
pulstran
y = pulstran(t,d,'func')
pulstran(t,d,'func',p1,p2,...)
pulstran(t,d,p,fs)
pulstran(t,d,p)
pulstran(...,'func')
```

Description pulstran generates pulse trains from continuous functions or sampled prototype pulses.

y = pulstran(t,d,'*func*') generates a pulse train based on samples of a continuous function, '*func*', where '*func*' is

- 'gauspuls', for generating a Gaussian-modulated sinusoidal pulse

- 'rectpuls', for generating a sampled aperiodic rectangle

- 'tripuls', for generating a sampled aperiodic triangle

pulstran is evaluated length(d) times and returns the sum of the evaluations y = func(t-d(1)) + func(t-d(2)) + ...

The function is evaluated over the range of argument values specified in array t, after removing a scalar argument offset taken from the vector d. Note that *func* must be a vectorized function that can take an array t as an argument.

An optional gain factor may be applied to each delayed evaluation by specifying d as a two-column matrix, with the offset defined in column 1 and associated gain in column 2 of d. Note that a row vector will be interpreted as specifying delays only.

pulstran(t,d,'*func*',p1,p2,...) allows additional parameters to be passed to '*func*' as necessary. For example:

func(t-d(1),p1,p2,...) + func(t-d(2),p1,p2,...) + ...

pulstran(t,d,p,fs) generates a pulse train that is the sum of multiple delayed interpolations of the prototype pulse in vector p, sampled at the

rate fs, where p spans the time interval [0,(length(p)-1)/fs], and its samples are identically 0 outside this interval. By default, linear interpolation is used for generating delays.

pulstran(t,d,p) assumes that the sampling rate fs is equal to 1 Hz.

pulstran(...,'*func*') specifies alternative interpolation methods. See interp1 for a list of available methods.

Examples

Example 1

This example generates an asymmetric sawtooth waveform with a repetition frequency of 3 Hz and a sawtooth width of 0.1s. It has a signal length of 1s and a 1 kHz sample rate:

```
t = 0 : 1/1e3 : 1;        % 1 kHz sample freq for 1 sec
d = 0 : 1/3 : 1;          % 3 Hz repetition freq
y = pulstran(t,d,'tripuls',0.1,-1);
plot(t,y)
```

pulstran

Example 2

This example generates a periodic Gaussian pulse signal at 10 kHz, with 50% bandwidth. The pulse repetition frequency is 1 kHz, sample rate is 50 kHz, and pulse train length is 10 msec. The repetition amplitude should attenuate by 0.8 each time:

```
t = 0 : 1/50E3 : 10e-3;
d = [0 : 1/1E3 : 10e-3 ; 0.8.^(0:10)]';
y = pulstran(t,d,'gauspuls',10e3,0.5);
plot(t,y)
```


See Also   chirp | cos | diric | gauspuls | rectpuls | sawtooth | sin | sinc | square | tripuls

Purpose   PSD using Welch's method

Syntax   `[Pxx,w] = pwelch(x)`
`[Pxx,w] = pwelch(x,window)`
`[Pxx,w] = pwelch(x,window,noverlap)`
`[Pxx,w] = pwelch(x,window,noverlap,nfft)`
`[Pxx,w] = pwelch(x,window,noverlap,w)`
`[Pxx,f] = pwelch(x,window,noverlap,nfft,fs)`
`[Pxx,f] = pwelch(x,window,noverlap,f,fs)`
`[...] = pwelch(x,window,noverlap,...,'`*range*`')`
`pwelch(x,...)`

Description   `[Pxx,w] = pwelch(x)` estimates the power spectral density `Pxx` of the
input signal vector `x` using Welch's method. Welch's method splits the
data into overlapping segments, computes modified periodograms of
the overlapping segments, and averages the resulting periodograms to
produce the power spectral density estimate.

- The vector `x` is segmented into eight sections of equal length, each
  with 50% overlap.

- Any remaining (trailing) entries in `x` that cannot be included in the
  eight segments of equal length are discarded.

- Each segment is windowed with a Hamming window (see `hamming`)
  that is the same length as the segment.

The power spectral density is calculated in units of power per radians
per sample. The corresponding vector of frequencies w is computed in
radians per sample, and has the same length as `Pxx`.

A real-valued input vector `x` produces a full power one-sided (in
frequency) PSD (by default), while a complex-valued `x` produces a
two-sided PSD.

In general, the length *N* of the FFT and the values of the input `x`
determine the length of `Pxx` and the range of the corresponding
normalized frequencies. For this syntax, the (default) length *N* of the
FFT is the larger of 256 and the next power of 2 greater than the length

of the segment. The following table indicates the length of Pxx and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	(N/2) +1	[0, π]
Complex-valued	N	[0, 2π)

`[Pxx,w] = pwelch(x,window)` calculates the modified periodogram using either:

- The window length `window` for the Hamming window when `window` is a positive integer

- The window weights specified in `window` when `window` is a vector

With this syntax, the input vector x is divided into an integer number of segments with 50% overlap, and each segment is the same length as the window. Entries in x that are left over after it is divided into segments are discarded. If you specify `window` as the empty vector `[]`, then the signal data is divided into eight segments, and a Hamming window is used on each one.

`[Pxx,w] = pwelch(x,window,noverlap)` divides x into segments according to `window`, and uses the integer `noverlap` to specify the number of signal samples (elements of x) that are common to two adjacent segments. `noverlap` must be less than the length of the window you specify. If you specify `noverlap` as the empty vector `[]`, then pwelch determines the segments of x so that there is 50% overlap (default).

`[Pxx,w] = pwelch(x,window,noverlap,nfft)` uses Welch's method to estimate the PSD while specifying the length of the FFT with the integer `nfft`. If you specify `nfft` as the empty vector `[]`, the number of

points used in the PSD estimate defaults to a maximum of 256 or the next power of two greater than the length of window. For a window length less than or equal to 256, nfft defaults to 256. For a window length greater than 256, nfft defaults to the next power of two.

The length of Pxx and the frequency range for w depend on nfft and the values of the input x. The following table indicates the length of Pxx and the frequency range for w for this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	[0, π]
Real-valued	Odd	(nfft + 1)/2	[0, π)
Complex-valued	Even or odd	nfft	[0, 2π)

[Pxx,w] = pwelch(x,window,noverlap,w) estimates the two-sided PSD at the normalized frequencies specified in the vector w using the Goertzel algorithm. The frequencies of w are rounded to the nearest DFT bin commensurate with the resolution of the signal. The units of w are rad/sample.

[Pxx,f] = pwelch(x,window,noverlap,nfft,fs) uses the sampling frequency fs specified in hertz (Hz) to compute the PSD vector (Pxx) and the corresponding vector of frequencies (f). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify fs as the empty vector [], the sampling frequency defaults to 1 Hz.

The frequency range for f depends on nfft, fs, and the values of the input x. The length of Pxx is the same as in the PSD and Frequency Vector Characteristics on page 3-797 above. The following table indicates the frequency range for f for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pwelch(x,window,noverlap,f,fs)` estimates the two-sided PSD at the normalized frequencies specified in the vector `f` using the Goertzel algorithm. The `f` vector returned is the same vector as the input `f` vector. The frequencies of `f` are rounded to the nearest DFT bin commensurate with the resolution of the signal.

`[...] = pwelch(x,window,noverlap,...,'range')` specifies the range of frequency values. This syntax is useful when `x` is real. The string `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.

  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

  - If you don't specify `fs`, the frequency range is $[0, 2\pi)$.

- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

The string `'range'` can appear anywhere in the syntax after `noverlap`.

`pwelch(x,...)` with no output arguments plots the PSD estimate in dB per unit frequency in the current figure window.

Examples
Estimate the PSD of a signal consisting of a 200-Hz sinusoid plus noise. The sampling frequency is 1 kHz. Use a segment length of 128 samples with a 120-sample overlap. Plot the one-sided PSD estimate.

```
rng default;
Fs = 1000;
t = 0:1/Fs:1-(1/Fs);
% 200Hz cosine + noise
x = cos(2*pi*t*200) + randn(size(t));
pwelch(x,128,120,length(x),Fs,'onesided')
```


Algorithms   pwelch calculates the power spectral density using Welch's method (see References below):

1 The input signal vector x is divided into *k* overlapping segments according to window and noverlap (or their default values). If the window size is larger than the number of FFT points (NFFT) , the signal is divided into NFFT–length segments and then, the last segment is padded with zeros.

2 The specified (or default) window is applied to each segment of x. (No preprocessing is done before applying the window to each segment.)

3 An nfft-point FFT is applied to the windowed data.

4 The (modified) periodogram of each windowed segment is computed.

5 The set of modified periodograms is averaged to form the spectrum estimate $S(e^{j\omega})$.

6 The resulting spectrum estimate is scaled to compute the power spectral density as $S(e^{j\omega})/F$, where *F* is

- 2π when you do not supply the sampling frequency

- fs when you supply the sampling frequency

The number of segments *k* that x is divided into is calculated as:

- Eight if you don't specify window, or if you specify it as the empty vector [ ]

- $k = \dfrac{m - o}{l - o}$ if you specify window as a nonempty vector or a scalar
  In this equation, *m* is the length of the signal vector x, *o* is the number of overlapping samples (noverlap), and *l* is the length of each segment (the window length).

References

[1] Hayes, M., *Statistical Digital Signal Processing and Modeling*, John Wiley & Sons, 1996.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1997, pp. 52-54.

[3] Welch, P.D, "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms," *IEEE Trans. Audio Electroacoustics, Vol. AU-15 (June 1967)*, pp. 70-73.

See Also   dspdata.msspectrum | pburg | pcov | peig | periodogram | pmcov | pmtm | pmusic | pyulear | spectrum.welch

pyulear

Purpose	PSD using Yule-Walker AR method

Syntax

```
Pxx = pyulear(x,p)
Pxx = pyulear(x,p,nfft)
[Pxx,w] = pyulear(...)
[Pxx,w] = pyulear(x,p,w)
Pxx = pyulear(x,p,nfft,fs)
Pxx = pyulear(x,p,f,fs)
[Pxx,f] = pyulear(x,p,nfft,fs)
[Pxx,f] = pyulear(x,p,f,fs)
[Pxx,f] = pyulear(x,p,nfft,fs,'range')
[Pxx,w] = pyulear(x,p,nfft,'range')
pyulear(...)
```

Description   `Pxx = pyulear(x,p)` implements the Yule-Walker algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal. `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD. This estimate is also an estimate of the maximum entropy.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

Pxx = pyulear(x,p,nfft) uses the integer FFT length nfft to calculate the PSD vector Pxx.

[Pxx,w] = pyulear(...) also returns w, a vector of normalized angular frequencies at which the two-sided PSD is estimated. Pxx and w have the same length. The units for w are rad/sample.

The length of Pxx and the frequency range for w depend on nfft and the values of the input x. The following table indicates the length of Pxx and the frequency range for w in this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	$[0, \pi]$
Real-valued	Odd	(nfft + 1)/2	$[0, \pi]$
Complex-valued	Even or odd	nfft	$[0, 2\pi)$

[Pxx,w] = pyulear(x,p,w) uses a vector of normalized frequencies w with two or more elements to compute the PSD at those frequencies and returns a two-sided PSD.

Pxx = pyulear(x,p,nfft,fs)

or

`Pxx = pyulear(x,p,f,fs)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx` or uses the vector of frequencies `f` in Hz and the sampling frequency `fs` to compute the two-sided PSD vector `Pxx` at those frequencies. If you specify `nfft` as the empty vector `[]`, it uses the default value of 256. If you specify `fs` as the empty vector `[]`, the sampling frequency `fs` defaults to 1 Hz. The spectral density produced is calculated in units of power per Hz.

`[Pxx,f] = pyulear(x,p,nfft,fs)`

or

`[Pxx,f] = pyulear(x,p,f,fs)` returns the frequency vector f. In this case, the units for the frequency vector are in Hz. The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pyulear(x,p,nfft,fs,'range')` or

`[Pxx,w] = pyulear(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.

  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

- If you don't specify fs, the frequency range is [0, 2π).

- 'onesided': Compute the one-sided PSD over the frequency ranges specified for real x. This is the default for determining the frequency range for real-valued x. Note that 'onesided' is not valid if you pass in a vector of frequencies (f or w).

Note You can put the string argument '*range*' anywhere in the input argument list after p.

---

pyulear(...) plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output w (or f) for a given set of parameters.

Tips   The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples   Because the Yule-walker method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use freqz to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using pyulear:

```
% AR filter coefficients
a = [1 -2.2137 2.9403 -2.1697 0.9606];
% AR filter frequency response
freqz(1,a)
title('AR System Frequency Response')
```

Now generate the input signal x by filtering white noise through the AR filter. Estimate the PSD of x based on a fourth-order AR prediction model, since in this case, we know that the original AR system model a has order 4:

```
x = filter(1,a,randn(256,1));      % AR system output
pyulear(x,4)                       % Fourth-order estimate
```

Algorithms   Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

pyulear estimates the PSD of an input signal vector using the Yule-Walker AR method. This method, also called the autocorrelation or windowed method, fits an autoregressive (AR) linear prediction filter model to the signal by minimizing the forward prediction error (based on all observations of the in put sequence) in the least squares sense. This formulation leads to the Yule-Walker equations, which are solved by the Levinson-Durbin recursion. The spectral estimate returned by pyulear is the squared magnitude of the frequency response of this AR model.

References   [1] Marple, S.L., *Digital Spectral Analysis*, Prentice-Hall, 1987, Chapter 7.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also   aryule | lpc | pburg | pcov | peig | periodogram | pmcov | pmtm | pmusic | prony | pwelch | spectrum.yulear

realizemdl

Purpose	Simulink subsystem block for filter

Syntax

```
realizemdl(FiltDesign)
realizemdl(FiltDesign,propertyname1,propertyvalue1,...)
```

Description realizemdl(FiltDesign) generates a model of the filter object FiltDesign in a Simulink subsystem block using sum, gain, and delay blocks from Simulink. The properties and values of FiltObject define the resulting subsystem block parameters.

realizemdl requires Simulink. To accurately realize models of quantized filters, use Simulink Fixed Point™.

realizemdl(FiltDesign,propertyname1,propertyvalue1,...) generates the model forFiltDesign with the associated propertyname/propertyvalue pairs, and any other values you set in FiltDesign.

Note Subsystem filter blocks that you use realizemdl to create support sample-based input and output only. You cannot input or output frame-based signals with the block.

Using the optional propertyname/propertyvalue pairs lets you control more fully the way the block subsystem model gets built, such as where the block goes, what the name is, or how to optimize the block structure. Valid properties and values for realizemdl are listed in this table, with the default value noted and descriptions of what the properties do.

Property Name	Property Values	Description
Destination	`'current'` (default) or `'new'`or *Subsystemname*	Specify whether to add the block to your current Simulink model or create a new model to contain the block. If you provide the name of a current subsystem in *subsystemname*, realizemdl adds the new block to the specified subsystem.
Blockname	`'filter'` (default)	Provides the name for the new subsystem block. By default the block is named 'filter'. To enter a name for the block, use the propertyvalue set to a string *'blockname'*.
MapCoeffstoPorts	`'off'` (default) or `'on'`	Specify whether to map the coefficients of the filter to the ports of the block.
MapStates	`'off'` (default) or `'on'`	Specifies whether to apply the current filter states to the realized model. This lets you save states from a filter object you may have used or configured in a specific way. The default setting of `'off'` means the states are not transferred to the model. Setting the property to `'on'` preserves the current filter states in the realized model.

Property Name	Property Values	Description
OverwriteBlock	`'off'` or `'on'`	Specify whether to overwrite an existing block with the same name or create a new block.
OptimizeZeros	`'off'` (default) or `'on'`	Specify whether to remove zero-gain blocks.
OptimizeOnes	`'off'` (default) or `'on'`	Specify whether to replace unity-gain blocks with direct connections.
OptimizeNegOnes	`'off'` (default) or `'on'`	Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.
OptimizeDelayChains	`'off'` (default) or `'on'`	Specify whether to replace cascaded chains of delay blocks with a single integer delay block to provide an equivalent delay.
CoeffNames	`{'Num'}` (default FIR),`{'Num','Den'}` (default direct form IIR),`{'Num','Den','g'}` (default IIR SOS), `{'K'}` (default form lattice)	Specify the coefficient variable names as string variables in a cell array.`MapCoeffsToPorts` must be set to `'on'` for this property to apply.

Property Name	Property Values	Description
InputProcessing	`'columnsaschannels'` (default), , or `'inherited'`	Specify sample-based (`'columnsaschannels'`) or frame-based (`'elementsaschannels'`) processing.
		The Inherited (this choice will be removed - see release notes) option will be removed in a future release. For more information, see "" for frame-based processing in the *DSP System Toolbox Release Notes*.
RateOption	`'enforcesinglerate'` (default) or `'allowmultirate'`	Specify how the block adjusts the rate at the output to accommodate the reduced number of samples. This parameter applies only when InputProcessing is `'columnsaschannels'`.

Examples Realize Simulink model of lowpass Butterworth filter:

```
Hd = fdesign.lowpass('N,F3dB',4,0.25);
d = design(Hd,'butter');
realizemdl(d);
```

Realize Simulink model with coefficients mapped to ports:

```
Hd = fdesign.lowpass('N,F3dB',4,0.25);
d = design(Hd,'butter');
%Realize Simulink model and export coefficients
```

realizemdl

```
realizemdl(d,'MapCoeffsToPorts','on');
```

In this case, the filter is an IIR filter with a direct form II second-order sections structure. Setting `MapCoeffstoPorts` to `'on'` exports the numerator coefficients, the denominator coefficients, and the gains to the MATLAB workspace using the default variable names `Num`, `Den`, and `g`. Each column of `Num` and `Den` represents one second-order section. You can modify the filter coefficients directly in the MATLAB workspace providing tunability to the realized Simulink model.

See Also   block | design | fdesign

Purpose　　Convert reflection coefficients to autocorrelation sequence

Syntax　　`r = rc2ac(k,r0)`

Description　`r = rc2ac(k,r0)` finds the autocorrelation coefficients, r, of the output of the discrete-time prediction error filter from the lattice-form reflection coefficients k and initial zero-lag autocorrelation r0.

Examples　`k = [0.3090    0.9800    0.0031    0.0082   -0.0082];`
`r0 = 0.1;`
`a = rc2ac(k,r0)`

References　[1] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

See Also　`ac2rc` | `poly2ac` | `rc2poly`

rc2is

Purpose	Convert reflection coefficients to inverse sine parameters
Syntax	isin = is2rc(k)
Description	isin = is2rc(k) returns a vector of inverse sine parameters isin from a vector of reflection coefficients k.
Examples	k = [0.3090 0.9801 0.0031 0.0082 -0.0082]; isin = rc2is(k)
References	[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.
See Also	is2rc

Purpose   Convert reflection coefficients to log area ratio parameters

Syntax   g = rc2lar(k)

Description   g = rc2lar(k) returns a vector of log area ratio parameters g from a
                   vector of reflection coefficients k.

Examples   k = [0.3090 0.9801 0.0031 0.0082 -0.0082];
                   g = rc2lar(k)

References   [1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time
                   Processing of Speech Signals*, Prentice-Hall, 1993.

See Also   lar2rc

rc2poly

Purpose	Convert reflection coefficients to prediction filter polynomial
Syntax	a = rc2poly(k) [a,efinal] = rc2poly(k,r0)
Description	a = rc2poly(k) converts the reflection coefficients k corresponding to the lattice structure to the prediction filter polynomial a, with a(1) = 1. The output a is row vector of length length(k)+1. [a,efinal] = rc2poly(k,r0) returns the final prediction error efinal based on the zero-lag autocorrelation, r0.
Examples	Consider a lattice IIR filter given by reflection coefficients k: k = [0.3090 0.9800 0.0031 0.0082 -0.0082]; Its equivalent prediction filter representation is given by a = rc2poly(k)
Algorithms	rc2poly computes output a using Levinson's recursion [1]. The function **1** Sets the output vector a to the first element of k. **2** Loops through the remaining elements of k. For each loop iteration i, a = [a + a(i-1:-1:1)*k(i) k(i)]. **3** Implements a = [1 a].
References	[1] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
See Also	ac2poly	latc2tf	latcfilt	poly2rc	rc2ac	rc2is	rc2lar	tf2latc

Purpose	Real cepstrum and minimum phase reconstruction

Syntax
```
rceps(x)
[y,ym] = rceps(x)
```

Description   The *real cepstrum* is the inverse Fourier transform of the real logarithm of the magnitude of the Fourier transform of a sequence.

Note rceps only works on real data.

---

rceps(x) returns the real cepstrum of the real sequence x. The real cepstrum is a real-valued function.

[y,ym] = rceps(x) returns both the real cepstrum y and a minimum phase reconstructed version ym of the input sequence.

Algorithms   rceps is an implementation of algorithm 7.2 in [2], that is,

```
y = real(ifft(log(abs(fft(x)))));
```

Appropriate windowing in the cepstral domain forms the reconstructed minimum phase signal:

```
w = [1;2*ones(n/2-1,1);ones(1-rem(n,2),1);zeros(n/2-1,1)];
ym = real(ifft(exp(fft(w.*y))));
```

References   [1] Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing*, Englewood Cliffs, NJ, Prentice-Hall, 1975.

[2] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979.

See Also   cceps | fft | hilbert | iceps | unwrap

rectpuls

Purpose   Sampled aperiodic rectangle

Syntax
```
y = rectpuls(t)
y = rectpuls(t,w)
```

Description `y = rectpuls(t)` returns a continuous, aperiodic, unity-height rectangular pulse at the sample times indicated in array `t`, centered about `t = 0` and with a default width of 1. Note that the interval of non-zero amplitude is defined to be open on the right, that is, `rectpuls(-0.5) = 1` while `rectpuls(0.5) = 0`.

`y = rectpuls(t,w)` generates a rectangle of width `w`.

`rectpuls` is typically used in conjunction with the pulse train generating function `pulstran`.

See Also   `chirp` | `cos` | `diric` | `gauspuls` | `pulstran` | `sawtooth` | `sin` | `sinc` | `square` | `tripuls`

Purpose	Rectangular window
Syntax	w = rectwin(L)
Description	w = rectwin(L) returns a rectangular window of length L in the column vector w. This function is provided for completeness; a rectangular window is equivalent to no window at all.
Algorithms	w = ones(L,1);
References	[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing.* Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.
See Also	barthannwin	bartlett	blackmanharris	bohmanwin	nuttallwin	parzenwin	triang	window	wintool	wvtool

resample

Purpose
Change sampling rate by rational factor

Syntax
```
y = resample(x,p,q)
y = resample(x,p,q,n)
y = resample(x,p,q,n,beta)
y = resample(x,p,q,b)
[y,b] = resample(x,p,q)
```

Description
y = resample(x,p,q) resamples the sequence in vector x at p/q times the original sampling rate, using a polyphase filter implementation. p and q must be positive integers. The length of y is equal to ceil(length(x)*p/q). If x is a matrix, resample works down the columns of x.

resample applies an anti-aliasing (lowpass) FIR filter to x during the resampling process. It designs the filter using firls with a Kaiser window.

y = resample(x,p,q,n) uses n terms on either side of the current sample, x(k), to perform the resampling. The length of the FIR filter resample uses is proportional to n; larger values of n provide better accuracy at the expense of more computation time. The default for n is 10. If you let n = 0, resample performs a nearest-neighbor interpolation

```
y(k) = x(round((k-1)*q/p)+1)
```

where y(k) = 0 if the index to x is greater than length(x).

y = resample(x,p,q,n,beta) uses beta as the design parameter for the Kaiser window that resample employs in designing the lowpass filter. The default for beta is 5.

y = resample(x,p,q,b) filters x using the vector of filter coefficients b.

[y,b] = resample(x,p,q) returns the vector b, which contains the coefficients of the filter applied to x during the resampling process.

Examples   Resample a simple linear sequence at 3/2 the original rate:

```
fs1 = 10;            % Original sampling frequency in Hz
t1 = 0:1/fs1:1;      % Time vector
x = t1;              % Define a linear sequence
y = resample(x,3,2); % Now resample it
t2 = (0:(length(y)-1))*2/(3*fs1);  % New time vector
plot(t1,x,'*',t2,y,'o',-0.5:0.01:1.5,-0.5:0.01:1.5,':')
legend('original','resampled'); xlabel('Time')
```


Notice that the last few points of the output y are inaccurate. In its filtering process, resample assumes the samples at times before and after the given samples in x are equal to zero. Thus large deviations from zero at the end points of the sequence x can cause inaccuracies in y at its end points. The following two plots illustrate this side effect of resample:

```
x = [1:10 9:-1:1]; y = resample(x,3,2);
subplot(2,1,1);
plot(1:19,x,'*',(0:28)*2/3 + 1,y,'o');
title('Edge Effects Not Noticeable');
```

resample

```
legend('original','resampled');
x = [10:-1:1 2:10]; y = resample(x,3,2);
subplot(2,1,2);
plot(1:19,x,'*',(0:28)*2/3 + 1,y,'o')
title('Edge Effects Very Noticeable');
legend('original','resampled');
```


Algorithms   resample performs an FIR design using firls, followed by rate changing implemented with upfirdn.

See Also   decimate | downsample | firls | interp | interp1 | intfilt | kaiser | mfilt | spline | upfirdn | upsample

residuez

Purpose　z-transform partial-fraction expansion

Syntax
```
[r,p,k] = residuez(b,a)
[b,a] = residuez(r,p,k)
```

Description　residuez converts a discrete time system, expressed as the ratio of two polynomials, to partial fraction expansion, or residue, form. It also converts the partial fraction expansion back to the original polynomial coefficients.

Note Numerically, the partial fraction expansion of a ratio of polynomials is an ill-posed problem. If the denominator polynomial is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can cause arbitrarily large changes in the resulting poles and residues. You should use state-space (or pole-zero representations instead.

[r,p,k] = residuez(b,a) finds the residues, poles, and direct terms of a partial fraction expansion of the ratio of two polynomials, $b(z)$ and $a(z)$. Vectors b and a specify the coefficients of the polynomials of the discrete-time system $b(z)/a(z)$ in descending powers of z.

$$B(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_m z^{-m}$$
$$A(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}$$

If there are no multiple roots and a > n-1,

$$\frac{B(z)}{A(z)} = \frac{r(1)}{1 - p(1)z^{-1}} + \cdots + \frac{r(n)}{1 - p(n)z^{-1}} + k(1) + k(2)z^{-1} + \cdots + k(m-n+1)z^{-(m-n)}$$

The returned column vector r contains the residues, column vector p contains the pole locations, and row vector k contains the direct terms. The number of poles is

residuez

```
n = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector k is empty if length(b) is less than length(a); otherwise:

```
length(k) = length(b) - length(a) + 1
```

If p(j) = ... = p(j+s-1) is a pole of multiplicity s, then the expansion includes terms of the form

$$\frac{r(j)}{1 - p(j)z^{-1}} + \frac{r(j+1)}{(1 - p(j)z^{-1})^2} + \cdots + \frac{r(j+s_r - 1)}{(1 - p(j)z^{-1})^s}$$

[b,a] = residuez(r,p,k) with three input arguments and two output arguments, converts the partial fraction expansion back to polynomials with coefficients in row vectors b and a.

The residue function in the standard MATLAB language is very similar to residuez. It computes the partial fraction expansion of continuous-time systems in the Laplace domain (see reference [1]), rather than discrete-time systems in the *z*-domain as does residuez.

Algorithms   residuez applies standard MATLAB functions and partial fraction techniques to find r, p, and k from b and a. It finds

- The direct terms a using deconv (polynomial long division) when length(b) > length(a)-1.

- The poles using p = roots(a).

- Any repeated poles, reordering the poles according to their multiplicities.

- The residue for each nonrepeating pole p_i by multiplying $b(z)/a(z)$ by $1/(1 - p_iz^{-1})$ and evaluating the resulting rational function at $z = p_i$.

- The residues for the repeated poles by solving

  ```
  S2*r2 = h - S1*r1
  ```

for r2 using \. h is the impulse response of the reduced $b(z)/a(z)$, S1 is a matrix whose columns are impulse responses of the first-order systems made up of the nonrepeating roots, and r1 is a column containing the residues for the nonrepeating roots. Each column of matrix S2 is an impulse response. For each root p_j of multiplicity s_j, S2 contains s_j columns representing the impulse responses of each of the following systems.

$$\frac{1}{1-pjz^{-1}}, \frac{1}{(1-pjz^{-1})^2}, \cdots, \frac{1}{(1-pjz^{-1})^{s_j}}$$

The vector h and matrices S1 and S2 have n + xtra rows, where n is the total number of roots and the internal parameter xtra, set to 1 by default, determines the degree of over-determination of the system of equations.

References   [1] Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, NJ, 1975, pp. 166-170.

See Also   convmtx | deconv | poly | prony | residue | roots | ss2tf | tf2ss | tf2zp | tf2zpk | zp2ss

risetime

Purpose   Rise time of positive-going bilevel waveform transitions

Syntax
```
R = risetime(X)
R = risetime(X,FS)
R = risetime(X,T)
[R,LT,UT] = risetime(...)
[R,LT,UT,LL,UL] = risetime(...)
[...] = risetime(...,Name,Value)
risetime(...)
```

Description   R = risetime(X) returns a vector, R, containing the time each transition of the input bilevel waveform, X, takes to cross from the 10% to 90% reference levels. To determine the transitions, risetime estimates the state levels of the input waveform by a histogram method. risetime identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See "State-Level Tolerances" on page 3-831. Because risetime uses interpolation, R may contain values that do not correspond to sampling instants of the bilevel waveform, X.

R = risetime(X,FS) specifies the sampling frequency in hertz. The sampling frequency determines the sample instants corresponding to the elements in X. The first sample instant in X corresponds to t=0. Because risetime uses interpolation, R may contain values that do not correspond to sampling instants of the bilevel waveform, X.

R = risetime(X,T) specifies the sample instants, T, as a vector with the same number of elements as X.

[R,LT,UT] = risetime(...) returns vectors, LT and UT, whose elements correspond to the time instants where X crosses the lower- and upper-percent reference levels.

[R,LT,UT,LL,UL] = risetime(...) returns the levels, LL and UL, that correspond to the lower- and upper-percent reference levels.

[...] = risetime(...,Name,Value) returns the rise times with additional options specified by one or more Name,Value pair arguments.

risetime(...) plots the signal and darkens the regions of each transition where rise time is computed. The plot marks the lower and upper crossings and the associated reference levels. The state levels and the corresponding associated lower- and upper-state boundaries are also plotted.

Input Arguments

X

Bilevel waveform. X is a real-valued row or column vector.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

PctRefLevels

Reference levels as a percentage of the waveform amplitude. The low-state level is defined to be 0 percent. The high-state level is defined to be 100 percent. The value of 'PCTREFLEVELS' is a two-element real row vector whose elements correspond to the lower and upper percent reference levels.

Default: [10 90]

StateLevels

Low- and high-state levels. Specifies the levels to use for the low- and high-state levels as a 2-element real row vector. The first element is the low-state level. The second element is the high-state level.

Tolerance

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-831.

> **Default:** 2

Output Arguments

R

Rise times. R is a vector containing the duration of each positive-going transition. If you specify the sampling rate, FS, or the sampling instants, T, rise times are in seconds. If you do not specify a sampling rate, or sampling instants, rise times are in samples.

LT

Instants when positive-going transition crosses the lower-reference level. By default, the lower reference level is the 10% reference level. The upper reference level is the 90% reference level. You can change the default reference levels by specifying the 'PctRefLevels' name-value pair.

UT

Instants when positive-going transition crosses the upper-reference level. By default, the lower reference level is the 10% reference level. The upper reference level is the 90% reference level. You can change the default reference levels by specifying the 'PctRefLevels' name-value pair.

LL

Lower reference level in waveform amplitude units. LL is a vector containing the waveform value corresponding to the lower reference level in each positive-going transition. By default, the lower reference level is the 10% reference level. You can change the default reference levels by specifying the 'PctRefLevels' name-value pair.

UL

Upper reference level in waveform amplitude units. LL is a vector containing the waveform value corresponding to the upper reference level in each positive-going transition. By default, the upper reference level is the 90% reference level. You can change the default reference levels by specifying the `'PctRefLevels'` name-value pair.

Definitions

Positive-Going Transition

A *positive-going transition* in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-polarity (positive-going) pulse has a terminating state more positive than the originating state. If the waveform is differentiable in the neighborhood of the transition, an equivalent definition is a transition with a positive first derivative. The following figure shows a positive-going transition.

$$X(t)$$

$$\frac{dX}{dt} > 0$$

In the preceding figure, the amplitude values of the waveform do not appear because a positive-going transition does not depend on the actual waveform values. A positive-going transition is defined by the direction of the transition.

Percent Reference Levels

If S_1 is the low state, S_2 is the high state, and U is the *upper*-percent reference level. The waveform value corresponding to the upper percent reference level is

$$S_1 + \frac{U}{100}(S_2 - S_1)$$

If *L* is the *lower*-percent reference level, the waveform value corresponding to the lower percent reference level is

$$S_1 + \frac{L}{100}(S_2 - S_1)$$

State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \tfrac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

State Levels ± 2% Tolerance

Examples   **Rise Time in a Bilevel Waveform**

Determine the rise time in samples for a 2.3 V clock waveform.

Load the 2.3 V clock data. Determine the rise time in samples. Use the default [10 90] percent reference levels.

```
load('transitionex.mat','x');
R = risetime(x);
```

The rise time is less than 1, indicating that the transition occurred in a fraction of a sample.

Rise Time with 20% and 80% Reference Levels

Determine the rise time in a 2.3 V clock waveform sampled at 4 MHz. Compute the rise time using the 20% and 80% reference levels.

Load the 2.3 V clock data with sampling instants. Plot the waveform.

```
load('transitionex.mat','x','t');
plot(t,x);
```

Determine the rise time using the 20% and 80% reference levels.

```
R = risetime(x,'PctRefLevels',[2O 80]);
```

Rise Time, Reference-Level Instants, and Reference Levels

Determine the rise time, reference-level instants, and reference levels in a 2.3 V clock waveform sampled at 4 MHz.

Load the 2.3 V clock waveform along with the sampling instants.

```
load('transitionex.mat','x','t');
```

Determine the rise time, reference-level instants, and reference levels.

```
[R,LT,UT,LL,UL] = risetime(x,t);
```

Plot the waveform in microseconds with the lower- and upper-reference levels and reference-level instants. Show that the rise time is the difference between the upper- and lower-reference level instants.

```
plot(t.*1e6,x);
```

```
xlabel('microseconds'); ylabel('Volts');
hold on; grid on;
plot(LT.*1e6,LL,'ro','markerfacecolor',[1 0 0]);
plot(UT.*1e6,UL,'ro','markerfacecolor',[1 0 0]);
fprintf('Rise time is %1.4f microseconds.\n',(UT-LT)*1e6)
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Standard 181, 2003, pp. 15–17.

See Also   falltime | slewrate | statelevels

Purpose   Reverse Levinson-Durbin recursion

Syntax   r = rlevinson(a,efinal)
                 [r,u] = rlevinson(a,efinal)
                 [r,u,k] = rlevinson(a,efinal)
                 [r,u,k,e] = rlevinson(a,efinal)

Description The reverse Levinson-Durbin recursion implements the step-down
                 algorithm for solving the following symmetric Toeplitz system of linear
                 equations for r, where $r = [r(1)\ Lr(p+1)]$ and $r(i)^*$ denotes the complex
                 conjugate of $r(i)$.

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

r = rlevinson(a,efinal) solves the above system of equations for
r given vector a, where a = [1 $a(2)$ L $a(p+1)$]. In linear prediction
applications, r represents the autocorrelation sequence of the input
to the prediction error filter, where r(1) is the zero-lag element. The
figure below shows the typical filter of this type, where $H(z)$ is the

optimal linear predictor, $x(n)$ is the input signal, $\hat{x}(n)$ is the predicted
signal, and $e(n)$ is the prediction error.

Input vector `a` represents the polynomial coefficients of this prediction error filter in descending powers of z.

$$A(z) = 1 + a(2)z^{-1} + \cdots + a(n+1)z^{-p}$$

The filter must be minimum phase to generate a valid autocorrelation sequence. `efinal` is the scalar prediction error power, which is equal to the variance of the prediction error signal, $\sigma^2(e)$.

`[r,u] = rlevinson(a,efinal)` returns upper triangular matrix U from the UDU^* decomposition

$$R^{-1} = UE^{-1}U^*$$

where

$$R = \begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix}$$

and E is a diagonal matrix with elements returned in output `e` (see below). This decomposition permits the efficient evaluation of the inverse of the autocorrelation matrix, R^{-1}.

Output matrix `u` contains the prediction filter polynomial, `a`, from each iteration of the reverse Levinson-Durbin recursion

$$U = \begin{bmatrix} a_1(1)^* & a_2(2)^* & \cdots & a_{p+1}(p+1)^* \\ 0 & a_2(1)^* & \ddots & a_{p+1}(p)^* \\ 0 & 0 & \ddots & a_{p+1}(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{p+1}(1)^* \end{bmatrix}$$

where $a_i(j)$ is the *j*th coefficient of the *i*th order prediction filter polynomial (i.e., step *i* in the recursion). For example, the 5th order prediction filter polynomial is

```
a5 = u(5:-1:1,5)'
```

Note that `u(p+1:-1:1,p+1)'` is the input polynomial coefficient vector `a`.

`[r,u,k] = rlevinson(a,efinal)` returns a vector `k` of length (*p*+1) containing the reflection coefficients. The reflection coefficients are the conjugates of the values in the first row of `u`.

```
k = conj(u(1,2:end))
```

`[r,u,k,e] = rlevinson(a,efinal)` returns a vector of length *p*+1 containing the prediction errors from each iteration of the reverse Levinson-Durbin recursion: `e(1)` is the prediction error from the first-order model, `e(2)` is the prediction error from the second-order model, and so on.

These prediction error values form the diagonal of the matrix *E* in the UDU^* decomposition of R^{-1}.

$$R^{-1} = UE^{-1}U^*$$

References   [1] Kay, S.M., *Modern Spectral Estimation: Theory and Application*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

See Also   levinson | lpc | prony | stmcb

rms

Purpose	Root-mean-square level
Syntax	`Y = rms(X)` `Y = rms(X,DIM)`

Description   `Y = rms(X)` returns the root-mean-square (RMS) level of the input, `X`. If `X` is a row or column vector, `Y` is a real-valued scalar. For matrices, `Y` contains the RMS levels computed along the first nonsingleton dimension. For example, if `X` is an N-by-M matrix with N>1, `Y` is a 1-by-M row vector containing the RMS levels of the columns of `X`.

`Y = rms(X,DIM)` computes the RMS level of `X` along the dimension, `DIM`.

Input Arguments

X

Real or complex-valued input vector or matrix. By default, `rms` acts along the first nonsingleton dimension of `X`.

DIM

Dimension for RMS levels. The optional `DIM` input argument specifies the dimension along which to compute the RMS levels.

> **Default:** First nonsingleton dimension

Output Arguments

Y

Root-mean-square level. For vectors, `Y` is a real-valued scalar. For matrices, `Y` contains the RMS levels computed along the specified dimension `DIM`. By default, `DIM` is the first nonsingleton dimension.

Definitions   **Root-Mean-Square Level**

The root-mean-square level of a vector, *X*, is

$$X_{\mathrm{RMS}} = \sqrt{\frac{1}{N} \sum_{n=1}^{N} |X_n|^2}$$

with the summation performed along the specified dimension.

Examples

RMS Level of Sinusoid

Compute the RMS level of a 100-Hz sinusoid sampled at 1 kHz.

```
t = 0:0.001:1-0.001;
X = cos(2*pi*100*t);
Y = rms(X);
```

RMS Levels of 2-D Matrix

Create a matrix where each column is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the column index.

Compute the RMS levels of the columns.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)';
X = repmat(x,1,4);
amp = 1:4;
amp = repmat(amp,1e3,1);
X = X.*amp;
Y = rms(X);
```

RMS Levels of 2-D Matrix Along Specified Dimension

Create a matrix where each row is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the row index.

Compute the RMS levels of the rows specifying the dimension equal to 2 with the DIM argument.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t);
X = repmat(x,4,1);
amp = (1:4)';
amp = repmat(amp,1,1e3);
X = X.*amp;
```

```
Y = rms(X,2);
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Std 181, 2003.

See Also   mean | peak2rms | std

Purpose   Frequency and power content using eigenvector method

Syntax   ```
[w,pow] = rooteig(x,p)
[f,pow] = rooteig(...,fs)
[w,pow] = rooteig(...,'corr')
```

Description   `[w,pow] = rooteig(x,p)` estimates the frequency content in the
time samples of a signal x, and returns w, a vector of frequencies in
rad/sample, and the corresponding signal power in the vector pow in
units of power, such as volts^2. The input signal x is specified either as:

- A row or column vector representing one observation of the signal

- A rectangular array for which each row of x represents a separate
  observation of the signal (for example, each row is one output of
  an array of sensors, as in array processing), such that x'*x is an
  estimate of the correlation matrix

Note You can use the output of `corrmtx` to generate such an array x.

You can specify the second input argument p as either:

- A scalar integer. In this case, the signal subspace dimension is p.

- A two-element vector. In this case, p(2), the second element of
  p, represents a threshold that is multiplied by λ_{min}, the smallest
  estimated eigenvalue of the signal's correlation matrix. Eigenvalues
  below the threshold λ_{min}*p(2) are assigned to the noise subspace.
  In this case, p(1) specifies the maximum dimension of the signal
  subspace.

The extra threshold parameter in the second entry in p provides you
more flexibility and control in assigning the noise and signal subspaces.

The length of the vector w is the computed dimension of the signal
subspace. For real-valued input data x, the length of the corresponding
power vector pow is given by

rooteig

```
length(pow) = 0.5*length(w)
```

For complex-valued input data x, pow and w have the same length.

[f,pow] = rooteig(...,fs) returns the vector of frequencies f calculated in Hz. You supply the sampling frequency fs in Hz. If you specify fs with the empty vector [], the sampling frequency defaults to 1 Hz.

[w,pow] = rooteig(...,'corr') forces the input argument x to be interpreted as a correlation matrix rather than a matrix of signal data. For this syntax, you must supply a square matrix for x, and all of its eigenvalues must be nonnegative.

Note You can place the string 'corr' anywhere after p.

Examples   Find the frequency content in a signal composed of three complex exponentials in noise. Use the modified covariance method to estimate the correlation matrix used by the eigenvector method:

```
n=0:99;
s = exp(i*pi/2*n)+2*exp(i*pi/4*n)+...
  exp(i*pi/3*n)+randn(1,100);
% Estimate correlation matrix using
%  modified covariance method.
X=corrmtx(s,12,'mod');
[W,P] = rooteig(X,3)
```

Algorithms   The eigenvector method used by rooteig is the same as that used by peig. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content.

The difference between peig and rooteig is:

• peig returns the pseudospectrum at all frequency samples.

- `rooteig` returns the estimated discrete frequency spectrum, along with the corresponding signal power estimates.

`rooteig` is most useful for frequency estimation of signals made up of a sum of sinusoids embedded in additive white Gaussian noise.

See Also   `corrmtx` | `peig` | `pmusic` | `spectrum` | `rootmusic` | `spectrum.eigenvector`

rootmusic

Purpose   Root MUSIC algorithm

Syntax
```
W = rootmusic(X,P)
[W,POW] = rootmusic(X,P)
[F, POW] = rootmusic(...,Fs)
[W,POW] = rootmusic(...,'corr')
```

Description   `W = rootmusic(X,P)` returns the frequencies in radians/sample for the `P` complex exponentials (sinusoids) that make up the signal X.

The input X is specified either as:

- A row or column vector representing one realization of signal

- A rectangular array for which each row of X represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that `X'*X` is an estimate of the correlation matrix

---

Note You can use the output of `corrmtx` to generate such an array X.

`[W,POW] = rootmusic(X,P)` returns the estimated absolute value squared amplitudes of the sinusoids at the frequencies W.

The second input argument, P is the number of complex sinusoids in X. You can specify P as either:

- A positive integer. In this case, the signal subspace dimension is P.

- A two-element vector. In this case, `P(2)`, the second element of P, represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}*`P(2)` are assigned to the noise subspace. In this case, `P(1)` specifies the maximum dimension of the signal subspace.

The extra threshold parameter in the second entry in P provides you more flexibility and control in assigning the noise and signal subspaces.

The length of the vector W is the computed dimension of the signal subspace. For real-valued input data X, the length of the corresponding power vector POW is given by

```
length(POW) = 0.5*length(W)
```

For complex-valued input data X, POW and W have the same length.

[*F*, *POW*] = rootmusic(...,*Fs*) returns the vector of frequencies F calculated in Hz. You supply the sampling frequency Fs in Hz. If you specify Fs with the empty vector [], the sampling frequency defaults to 1 Hz.

[*W*,*POW*] = rootmusic(...,'corr') forces the input argument X to be interpreted as a correlation matrix rather than a matrix of signal data. For this syntax, you must supply a square matrix for X, and all of its eigenvalues must be nonnegative.

Note You can place the string 'corr' anywhere after P.

Examples   Estimate the amplitudes for 2 sinusoids in noise. The separation between the sinusoids is less than the resolution of the periodogram, 2π/N radians/sample.

```
s1 = RandStream.create('mrg32k3a');
n=0:99; % N is equal to 100. Periodogram resolution is (2*pi)/100
freqs = [pi/4 pi/4+0.06];
s = 2*exp(1j*freqs(1)*n)+1.5*exp(1j*freqs(2)*n)+...
0.5*randn(s1,1,100)+1j*0.5*randn(s1,1,100);
X = corrmtx(s,12,'mod');
[W,P] = rootmusic(X,2);
```

Algorithms   The MUSIC algorithm used by rootmusic is the same as that used by pmusic. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content.

rootmusic

The difference between pmusic and rootmusic is:

- pmusic returns the pseudospectrum at all frequency samples.

- rootmusic returns the estimated discrete frequency spectrum, along with the corresponding signal power estimates.

rootmusic is most useful for frequency estimation of signals made up of a sum of sinusoids embedded in additive white Gaussian noise.

Diagnostics   If the input signal, x is real and an odd number of sinusoids, p is specified, this error message is displayed

```
Real signals require an even number p of complex sinusoids.
```

See Also   corrmtx | peig | pmusic | spectrum | rooteig | spectrum.music

Purpose   Root-sum-of-squares level

Syntax   Y = rssq(X)
                  Y = rssq(X,DIM)

Description   Y = rssq(X) returns the root-sum-of-squares (RSS) level, Y, of the
                  input, X. If X is a row or column vector, Y is a real-valued scalar.
                  For matrices, Y contains the RSS levels computed along the first
                  nonsingleton dimension. For example, if Y is an N-by-M matrix with
                  N>1, Y is a 1-by-M row vector containing the RSS levels of the columns
                  of Y.

                  Y = rssq(X,DIM) computes the RSS level of X along the dimension,
                  DIM.

Input   **X**
Arguments
                  Real- or complex-valued input vector or matrix. By default, rssq acts
                  along the first nonsingleton dimension of X.

                  DIM

                  Dimension for root-sum-of-squares (RSS) level. The optional DIM input
                  argument specifies the dimension along which to compute the RSS level.

                      Default: First nonsingleton dimension

Output   **Y**
Arguments
                  Root-sum-of-squares level. For vectors, Y is a real-valued scalar. For
                  matrices, Y contains the RSS levels computed along the specified
                  dimension, DIM. By default, DIM is the first nonsingleton dimension.

Definitions   **Root-Sum-of-Squares Level**

                  The root-sum-of-squares (RSS) level of a vector, *X*, is

$$X_{\text{RSS}} = \sqrt{\sum_{n=1}^{N} |X_n|^2}$$

with the summation performed along the specified dimension. The RSS is also referred to as the ℓ^2 norm.

Examples   **RSS Level of Sinusoid**

Compute the RSS level of a 100-Hz sinusoid sampled at 1 kHz.

```
t = 0:0.001:1-0.001;
X = cos(2*pi*100*t);
Y = rssq(X);
```

RSS Level of 2-D Matrix

Create a matrix where each column is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the column index.

Compute the RSS level of the columns.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)';
X = repmat(x,1,4);
amp = 1:4;
amp = repmat(amp,1e3,1);
X = X.*amp;
Y = rssq(X);
```

RSS Level of 2-D Matrix Along Specified Dimension

Create a matrix where each row is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the row index.

Compute the RSS level of the rows specifying the dimension equal to 2 with the DIM argument.

```
t = 0:0.001:1-0.001;
```

```
x = cos(2*pi*100*t);
X = repmat(x,4,1);
amp = (1:4)';
amp = repmat(amp,1,1e3);
X = X.*amp;
Y = rssq(X,2);
```

References [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

sawtooth

Purpose   Sawtooth or triangle wave

Syntax
```
sawtooth(t)
sawtooth(t,width)
```

Description   `sawtooth(t)` generates a sawtooth wave with period 2π for the elements of time vector `t`. `sawtooth(t)` is similar to `sin(t)`, but creates a sawtooth wave with peaks of -1 and 1 instead of a sine wave. The sawtooth wave is defined to be -1 at multiples of 2π and to increase linearly with time with a slope of $1/\pi$ at all other times.

`sawtooth(t,width)` generates a modified triangle wave where `width`, a scalar parameter between 0 and 1, determines the point between 0 and 2π at which the maximum occurs. The function increases from -1 to 1 on the interval 0 to 2π`*width`, then decreases linearly from 1 to -1 on the interval 2π`*width` to 2π. Thus a parameter of 0.5 specifies a standard triangle wave, symmetric about time instant π with peak-to-peak amplitude of 1. `sawtooth(t,1)` is equivalent to `sawtooth(t)`.

See Also   chirp | cos | diric | gauspuls | pulstran | rectpuls | sin | sinc | square | tripuls

Purpose   Compute reflection coefficients from autocorrelation sequence

Syntax
```
k = schurrc(r)
[k,e] = schurrc(r)
```

Description   `k = schurrc(r)` uses the Schur algorithm to compute a vector `k` of reflection coefficients from a vector `r` representing an autocorrelation sequence. `k` and `r` are the same size. The reflection coefficients represent the lattice parameters of a prediction filter for a signal with the given autocorrelation sequence, `r`. When `r` is a matrix, `schurrc` treats each column of `r` as an independent autocorrelation sequence, and produces a matrix `k`, the same size as `r`. Each column of `k` represents the reflection coefficients for the lattice filter for predicting the process with the corresponding autocorrelation sequence `r`.

`[k,e] = schurrc(r)` also computes the scalar `e`, the prediction error variance. When `r` is a matrix, `e` is a column vector. The number of rows of `e` is the same as the number of columns of `r`.

Examples   Create an autocorrelation sequence from the MATLAB speech signal contained in `mtlb.mat`, and use the Schur algorithm to compute the reflection coefficients of a lattice prediction filter for this autocorrelation sequence:

```
load mtlb
r = xcorr(mtlb(1:5),'unbiased');
k = schurrc(r(5:end))
k =
   -0.7583
    0.1384
    0.7042
   -0.3699
```

References   [1] Proakis, J. and D. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, Third edition, Prentice-Hall, 1996, pp. 868-873.

schurrc

See Also   levinson

Purpose   Settling time for bilevel waveform

Syntax   ```
S = settlingtime(X,D)
S = settlingtime(X,FS,D)
S = settlingtime(X,T,D)
[S,SLEV,SINST] = settlingtime(...)
[S,SLEV,SINST] = settlingtime(...,Name,Value)
settlingtime(...)
```

**Description**  `S = settlingtime(X,D)` returns the time, `S`, from the mid-reference level instant to the time instant each transition enters and remains within a 2% tolerance region of the final state over the duration, `D`. `D` is a positive scalar. Because `settlingtime` uses interpolation to determine the mid-reference level instant, `S` may contain values that do not correspond to sampling instants. The length of `S` is equal to the number of detected transitions in the input signal, `X`. If for any transition, the level of the waveform does not remain within the lower and upper tolerance boundaries, the requested duration is not present, or an intervening transition is detected, `settlingtime` marks the corresponding element in `S` as `NaN`. See "Settle Seek Duration" on page 3-858 for cases in which `settlingtime` returns a `NaN`. To determine the transitions, `settlingtime` estimates the state levels of the input waveform by a histogram method. `settlingtime` identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See "State-Level Tolerances" on page 3-857.

`S = settlingtime(X,FS,D)` specifies the sampling rate for the bilevel waveform, `X` in hertz. The first sample instant in `X` is equal to t=0. Because `settlingtime` uses interpolation to determine the mid-reference level instant, `S` may contain values that do not correspond to sampling instants.

`S = settlingtime(X,T,D)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

# settlingtime

[S,SLEV,SINST] = settlingtime(...) returns vectors, SLEV, and SINST, whose elements correspond to the levels and sample instants of the settling points for each transition.

[S,SLEV,SINST] = settlingtime(...,Name,Value) returns the settling times, levels, and corresponding sample instants with additional options specified by one or more Name,Value pair arguments.

settlingtime(...) plots the signal and darkens the regions of each transition where settling time is computed. The plot marks the location of the settling time of each transition, the mid-crossings, and the associated reference levels. The plot also displays the state levels with the corresponding lower and upper tolerance boundaries.

## Input Arguments

### X

Bilevel waveform. X is a real-valued row or column vector.

### D

Settle-seek duration. D is a positive scalar, which defines the duration after the mid-reference level instant that settlingtime looks for a settling time. If no settling time occurs in D seconds after the mid-reference level instant, settlingtime returns a NaN. See "Settling Time" on page 3-855 and "Settle Seek Duration" on page 3-858.

### FS

Sample rate in hertz.

### T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

### Name-Value Pair Arguments

#### MidPct

Mid-reference level as a percentage of the waveform amplitude. See "Mid-Reference Level" on page 3-856.

**Default:** 50

#### StateLevels

Low and high-state levels. `StateLevels` is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low and high-state levels, `settlingtime` estimates the state levels from the input waveform using the histogram method.

#### Tolerance

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-857.

**Default:** 2

## Output Arguments

#### S

The time from the mid-reference level instant to the time instant each transition enters and remains within a 2% tolerance region of the final state over duration, `D`.

#### SLEV

Waveform values at the settling points.

#### SINST

Time instants of the settling points.

## Definitions

### Settling Time

The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the 2%-tolerance region around the state level. The settling time is illustrated in the following figure. The low- and high-state levels are the dashed black lines. The 2% tolerances above and below the state levels are shown by the red dashed lines and the settling time is indicated by the yellow circle.

### Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level, $S\_1$, and high- state level, $S\_2$, is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

### Mid-Reference Level Instant

Let $y_{50\%}$ denote the mid reference level.

Let $t_{50\%_-}$ and $t_{50\%_+}$ denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.

Let $y_{50\%_-}$ and $y_{50\%_+}$ denote the waveform values at $t_{50\%_-}$ and $t_{50\%_+}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

### State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \tfrac{\alpha}{100}(S_2 - S_1)$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.

**Settle Seek Duration**

The settle seek duration defines the interval of time after the
mid-reference level instant that settlingtime looks for a settling point.
If settlingtime does not find a settling point within the settle seek

duration, `settlingtime` returns `NaN` for the settling time. The following figure illustrates a settle seek duration of 10 samples.



`settlingtime` may fail to find a settling point in the specified settle seek duration if any one of the following conditions occurs:

- The last waveform value in the settle seek interval is not within the upper- and lower-state boundaries determined by the specified tolerance. The following figure illustrates this condition for a settle seek duration of 8 samples and a 2% tolerance region.

In the preceding figure, you see that the last sample in the settle seek interval exceeds the upper state boundary. In this example, reducing or increasing the settle seek duration can result in a valid settling time.

- There is an insufficient number of waveform samples for the specified settle seek duration. The following figure illustrates this condition for a settle seek duration of 20 samples. The settle seek duration extends beyond the final sample of the waveform.

- An intervening transition is detected before the end of the specified settle seek duration. The following figure illustrates this condition for a settle seek duration of 22 samples. An intervening transition is detected before the end of the 22–sample settle seek duration.

↑ Settle Seek Duration

**Examples**     **Determine Settling Point and Settling Level**

Determine the settling point and corresponding waveform value for a
bilevel waveform. Plot the waveform and mark the settling point.

```
load('transitionex.mat', 'x');
[S,SLEV,SINST] = settlingtime(x,10);
plot(x); hold on;
plot(SINST,SLEV,'ro','markerfacecolor',[1 0 0]);
```

Determine Settling Points for a Three-Transition Bilevel Waveform

Determine the settling points for a three-transition bilevel waveform. The data is sampled at 4 MHz. Use a one-microsecond settle-seek duration. Plot the settling points.

```
load('transitionex.mat', 'x');
y = [x; fliplr(x)];
fs = 4e6;
t = 0:1/fs:(length(y)*1/fs)-1/fs;
```

```
[S,SLEV,SINST] = settlingtime(y,fs,1e-6);
% equivalent to [S,SLEV,SINST] = settlingtime(y,t);
plot(t,y); hold on;
plot(SINST,SLEV,'ro','markerfacecolor',[1 O O]);
legend('Bilevel Waveform','Settling Points','Location','SouthWest');
```


References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Standard 181, 2003, pp. 23–24.

See Also   falltime | midcross | pulsewidth | risetime | statelevels

Purpose   Compute period of sequence

Syntax   p = seqperiod(x)
                 [p,num] = seqperiod(x)

Description p = seqperiod(x) returns the integer p that corresponds to the period
                 of the sequence in a vector x. The period p is computed as the minimum
                 length of a subsequence x(1:p) of x that repeats itself continuously
                 every p samples in x. The length of x does not have to be a multiple of
                 p, so that an incomplete repetition is permitted at the end of x. If the
                 sequence x is not periodic, then p = length(x).

- If x is a matrix, then seqperiod checks for periodicity along each
  column of x. The resulting output p is a row vector with the same
  number of columns as x.

- If x is a multidimensional array, then seqperiod checks for
  periodicity along the first nonsingleton dimension of x. In this case:

  - p is a multidimensional array of integers with a leading singleton
    dimension.

  - The lengths of the remaining dimensions of p correspond to those
    of the dimensions of x after the first nonsingleton one.

 [p,num] = seqperiod(x) also returns the number num of repetitions
 of x(1:p) in x. num might not be an integer.

Examples   x = [4 0 1 6;
                     2 0 2 7;
                     4 0 1 5;
                     2 0 5 6];
                p = seqperiod(x)
                p =
                     2   1   4   3

The result implies:

- The first column of x has period 2.

- The second column of x has period 1.

- The third column of x is not periodic, so p(3) is just the number of rows of x.

- The fourth column of x has period 3, although the last (second) repetition of the periodic sequence is incomplete.

Purpose   Specifications for filter specification object

Syntax   setspecs(D,specvalue1,specvalue2,...)
setspecs(D,Specification,specvalue1,specvalue2,...)
setspecs(...Fs)
setspecs(...,MAGUNITS)

Description   setspecs(D,specvalue1,specvalue2,...) sets the specifications
in filter specification object, D, in the same order they appear in the
Specification property.

setspecs(D,Specification,specvalue1,specvalue2,...) changes
the specifications for an existing filter specification object and sets
values for the new Specification property.

setspecs(...Fs) specifies the sampling frequency, Fs, in Hz. The
sampling frequency must be a scalar trailing all other specifications.
Entering a sampling frequency causes all other frequency specifications
to be in Hz.

setspecs(...,MAGUNITS) specifies the units for any magnitude
specifications. MAGUNITS can be one of the following: 'linear', 'dB',
or 'squared'. The default is 'dB'. The magnitude specifications are
always converted and stored in dB regardless of how the units are
specified.

Use SET(D,'SPECIFICATION') to get the list of all available
specification types for the filter specification object, D.

Examples   Construct a lowpass filter with specifications for the filter order and
cutoff frequency (-6 dB). Use setspecs after construction to set the
values of the filter order and cutoff frequency. Display the values in
the MATLAB command window.

```
D = fdesign.lowpass('N,Fc');
setspecs(D,10,0.2);
D.FilterOrder
D.Fcutoff
```

Construct a highpass filter with specifications for the numerator order, denominator order, and 3-dB frequency. Assume the sampling frequency is 1 kHz. Use setspecs to set the numerator and denominator orders to 6. Set the 3-dB frequency to 250 Hz. In order to use frequency specifications in Hz, specify the sampling frequency as a trailing scalar.

```
D = fdesign.highpass('Nb,Na,F3dB');
setspecs(D,6,6,250,1000);
```

See Also   design | designmethods | designopts | fdesign

Purpose   Savitzky-Golay filter design

Syntax   b = sgolay(k,f)
b = sgolay(k,f,w)
[b,g] = sgolay(...)

Description   b = sgolay(k,f) designs a Savitzky-Golay FIR smoothing filter b. The polynomial order k must be less than the frame size, f, which must be odd. If k = f-1, the designed filter produces no smoothing. The output, b, is an f-by-f matrix whose rows represent the time-varying FIR filter coefficients. In a smoothing filter implementation (for example, sgolayfilt), the last (f-1)/2 rows (each an FIR filter) are applied to the signal during the startup transient, and the first (f-1)/2 rows are applied to the signal during the terminal transient. The center row is applied to the signal in the steady state.

b = sgolay(k,f,w) specifies a weighting vector w with length f, which contains the real, positive-valued weights to be used during the least-squares minimization.

[b,g] = sgolay(...) returns the matrix g of differentiation filters. Each column of g is a differentiation filter for derivatives of order p-1 where p is the column index. Given a signal x of length f, you can find an estimate of the pth order derivative, xp, of its middle value from:

xp((f+1)/2) = (factorial(p)) * g(:,p+1)' * x

Tips   Savitzky-Golay smoothing filters (also called digital smoothing polynomial filters or least squares smoothing filters) are typically used to "smooth out" a noisy signal whose frequency span (without noise) is large. In this type of application, Savitzky-Golay smoothing filters perform much better than standard averaging FIR filters, which tend to filter out a significant portion of the signal's high frequency content along with the noise. Although Savitzky-Golay filters are more effective at preserving the pertinent high frequency components of the signal, they are less successful than standard averaging FIR filters at rejecting noise when noise levels are particularly high. The particular formulation of Savitzky-Golay filters preserves various moment orders

better than other smoothing methods, which tend to preserve peak widths and heights better than Savitzky-Golay.

Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to each frame of noisy data.

Examples   Use `sgolay` to smooth a noisy sinusoid and compare the resulting first and second derivatives to the first and second derivatives computed using `diff`. Notice how using `diff` amplifies the noise and generates useless results.

```
N = 4;                  % Order of polynomial fit
F = 21;                 % Window length
[b,g] = sgolay(N,F);    % Calculate S-G coefficients

dx = .2;
xLim = 200;
x = 0:dx:xLim-1;

y = 5*sin(0.4*pi*x)+randn(size(x));  % Sinusoid with noise

HalfWin  = ((F+1)/2) -1;
for n = (F+1)/2:996-(F+1)/2,
  % Zero-th derivative (smoothing only)
  SG0(n) =  dot(g(:,1), y(n - HalfWin: n + HalfWin));

  % 1st differential
  SG1(n) =  dot(g(:,2), y(n - HalfWin: n + HalfWin));

  % 2nd differential
  SG2(n) = 2*dot(g(:,3)', y(n - HalfWin: n + HalfWin))';
end

SG1 = SG1/dx;         % Turn differential into derivative
SG2 = SG2/(dx*dx);    % and into 2nd derivative

% Scale the "diff" results
```

```
DiffD1 = (diff(y(1:length(SG0)+1)))/ dx;
DiffD2 = (diff(diff(y(1:length(SG0)+2)))) / (dx*dx);

subplot(3,1,1);
plot([y(1:length(SG0))', SG0'])
legend('Noisy Sinusoid','S-G Smoothed sinusoid')

subplot(3, 1, 2);
plot([DiffD1',SG1'])
legend('Diff-generated 1st-derivative', ...
'S-G Smoothed 1st-derivative')

subplot(3, 1, 3);
plot([DiffD2',SG2'])
legend('Diff-generated 2nd-derivative',...
'S-G Smoothed 2nd-derivative')
```

Note The figures below are zoomed in each figure window panel to show more detail.

3-871

References   [1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also   fir1 | firls | filter | sgolayfilt

Purpose   Savitzky-Golay filtering

Syntax
```
y = sgolayfilt(x,k,f)
y = sgolayfilt(x,k,f,w)
y = sgolayfilt(x,k,f,w,dim)
```

Description   `y = sgolayfilt(x,k,f)` applies a Savitzky-Golay FIR smoothing filter to the data in vector `x`. If `x` is a matrix, `sgolayfilt` operates on each column. The polynomial order `k` must be less than the frame size, `f`, which must be odd. If `k = f-1`, the filter produces no smoothing.

`y = sgolayfilt(x,k,f,w)` specifies a weighting vector `w` with length `f`, which contains the real, positive-valued weights to be used during the least-squares minimization. If `w` is not specified or if it is specified as empty, `[ ]`, `w` defaults to an identity matrix.

`y = sgolayfilt(x,k,f,w,dim)` specifies the dimension, `dim`, along which the filter operates. If `dim` is not specified, `sgolayfilt` operates along the first non-singleton dimension; that is, dimension 1 for column vectors and nontrivial matrices, and dimension 2 for row vectors.

Tips   Savitzky-Golay smoothing filters (also called digital smoothing polynomial filters or least-squares smoothing filters) are typically used to "smooth out" a noisy signal whose frequency span (without noise) is large. In this type of application, Savitzky-Golay smoothing filters perform much better than standard averaging FIR filters, which tend to filter out a significant portion of the signal's high frequency content along with the noise. Although Savitzky-Golay filters are more effective at preserving the pertinent high frequency components of the signal, they are less successful than standard averaging FIR filters at rejecting noise.

Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to frames of noisy data.

Examples   Smooth the `mtlb` signal by applying a cubic Savitzky-Golay filter to data frames of length 41:

sgolayfilt

```
load mtlb                          % Load data
smtlb = sgolayfilt(mtlb,3,41);     % Apply 3rd-order filter
subplot(2,1,1)
plot([1:2000],mtlb(1:2000)); axis([0 2000 -4 4]);
title('mtlb'); grid;
subplot(2,1,2)
plot([1:2000],smtlb(1:2000)); axis([0 2000 -4 4]);
title('smtlb'); grid;
```


References   [1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also   medfilt1 | filter | sgolay | sosfilt

Purpose   Shift data to operate on specified dimension

Syntax   `[x,perm,nshifts] = shiftdata(x,dim)`

Description `[x,perm,nshifts] = shiftdata(x,dim)` shifts data `x` to permute dimension `dim` to the first column using the same permutation as the built-in `filter` function. The vector `perm` returns the permutation vector that is used.

If `dim` is missing or empty, then the first non-singleton dimension is shifted to the first column, and the number of shifts is returned in `nshifts`.

`shiftdata` is meant to be used in tandem with `unshiftdata`, which shifts the data back to its original shape. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

Examples   ### Example 1

This example shifts `x`, a 3-x-3 magic square, permuting dimension `2` to the first column. `unshiftdata` shifts `x` back to its original shape.

1. Create a 3-x-3 magic square:

```
x = fi(magic(3))
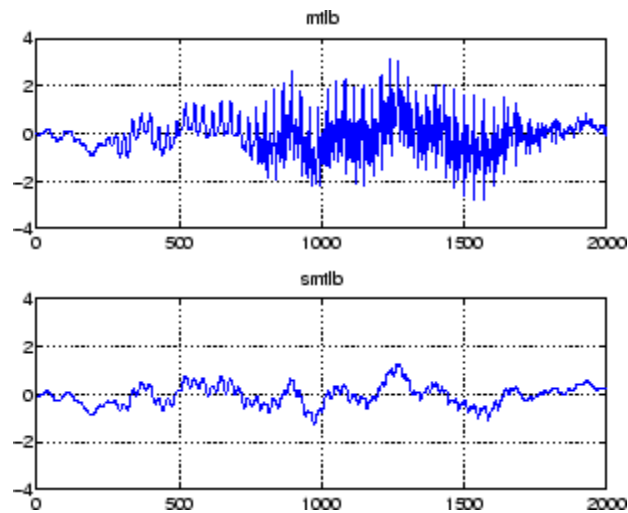
x =

    8    1    6
    3    5    7
    4    9    2
```

2. Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

The permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`:

```
x =

     8     3     4
     1     5     9
     6     7     2


perm =

     2     1


nshifts =

     []
```

3. Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

     8     1     6
     3     5     7
     4     9     2
```

Example 2

This example shows how `shiftdata` and `unshiftdata` work when you define `dim` as empty.

1. Define `x` as a row vector:

```
x = 1:5
```

```
x =

     1     2     3     4     5
```

2. Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])
```

`x` is returned as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts:

```
x =

     1
     2
     3
     4
     5


perm =

     []


nshifts =

     1
```

3. Using `unshiftdata`, restore `x` to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =

    1    2    3    4    5
```

See Also   permute | shiftdim | unshiftdata

Purpose   Signal processing window object

Syntax   w=sigwin.*window*

Description   w=sigwin.*window* returns a window object, w, of type *window*. Each window type takes one or more inputs. If you specify a sigwin.*window* with no inputs, a default window of length 64 is created.

Note You must specify a *window* type with sigwin.

Constructors

window for sigwin specifies the type of window. The following table lists the supported window functions with links to the corresponding class reference page for the window object.

Window	Window object
Modified Bartlett-Hanning Window	sigwin.barthannwin
Bartlett Window	sigwin.bartlett
Blackman Window	sigwin.blackman
Blackman-Harris Window	sigwin.blackmanharris
Bohman Window	sigwin.bohmanwin
Dolph-Chebyshev Window	sigwin.chebwin
Flat Top Window	sigwin.flattopwin
Gaussian Window	sigwin.gausswin
Hamming Window	sigwin.hamming
Hann (Hanning) Window	sigwin.hann
Kaiser Window	sigwin.kaiser

Window	Window object
Nuttall defined 4–term Blackman-Harris Window	`sigwin.nuttallwin`
Parzen Window	`sigwin.parzenwin`
Rectangular Window	`sigwin.rectwin`
Taylor Window	`sigwin.taylorwin`
Triangular Window	`sigwin.triang`
Tukey Window	`sigwin.tukeywin`

Methods

Methods provide ways of performing functions directly on your `sigwin` object without having to specify the window parameters again. You can apply this method directly on the variable you assigned to your `sigwin` object.

Method	Description
`generate`	Returns a column vector of values representing the window.
`info`	Returns information about the window object.
`winwrite`	Writes an ASCII file that contains window weights for a single window object or a vector of window objects. Default filename is `untitled.wf`.
	`winwrite(Hd,filename)` writes to a disk file named `filename` in the current working directory. The `.wf` extension is added automatically.

Viewing Object Parameters

As with any object, you can use `get` to view a `sigwin` object's parameters. To see a specific parameter,

```
 get(w,'parameter')
```

or to see all parameters for an object,

```
get(w)
```

Changing Object Parameters

To set specific parameters,

```
set(w,'parameter1',value,'parameter2',value,...)
```

Note that you must use single quotation marks around the parameter name.

Examples

Create a default Bartlett window and view the results in the Window Visualization Tool (`wvtool`). See `bartlett` for information on Bartlett windows:

```
w=sigwin.bartlett

w =
    Length: 64
      Name: 'Bartlett'
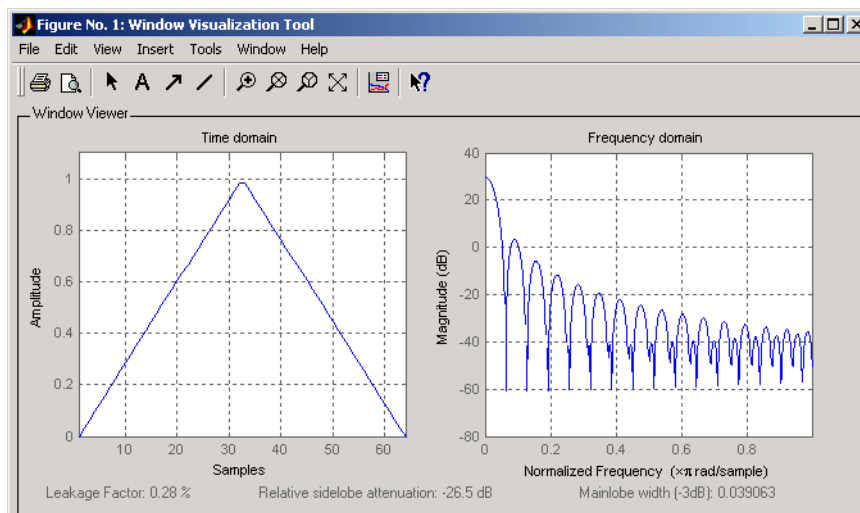
wvtool(w)
```

Create a 128-point Chebyshev window with 100 dB of sidelobe
attenuation. (See chebwin for information on Chebyshev windows.)
View the results of this and the above Bartlett window in the Window
Design and Analysis Tool (wintool):

```
w1=sigwin.chebwin(128,100)

w1 =

            Length: 128
              Name: 'Chebyshev'
     SidelobeAtten: 100
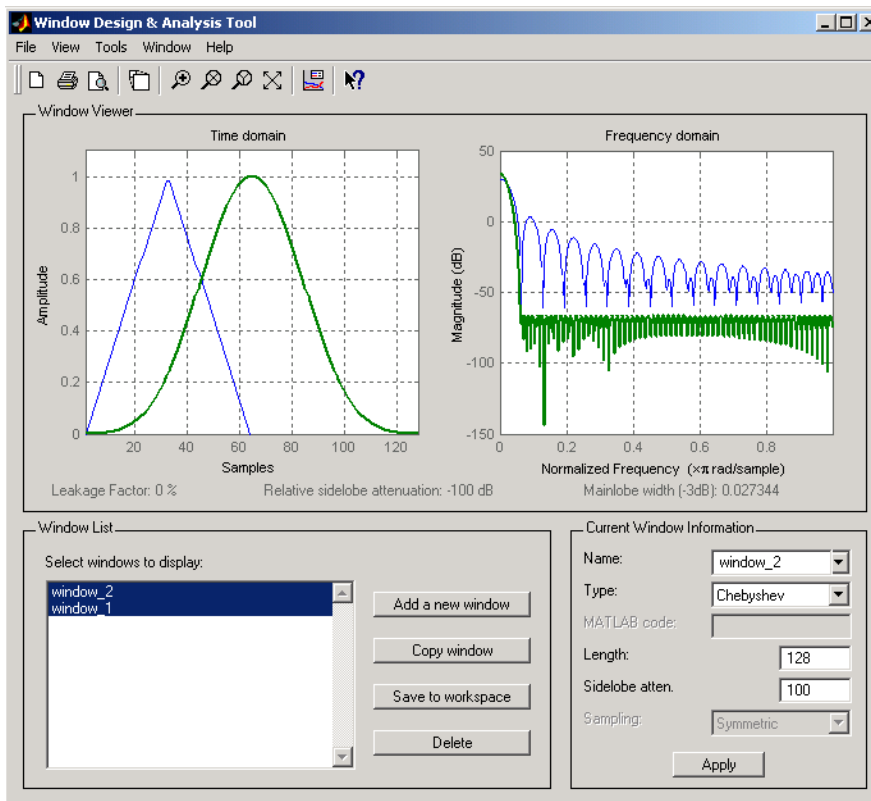
wintool(w,w1)
```

To save the window values in a vector, use:

```
d = generate(w);
```

See Also   window | wintool | wvtool

sigwin.barthannwin

Purpose Construct Bartlett-Hanning window object

Description `sigwin.barthannwin` creates a handle to a Bartlett-Hanning window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines a modified Bartlett-Hanning window of length *N*:

$$w(x) = 0.62 - 0.48\,|\,x\,| + 0.38\cos(2\pi x) \quad -1/2 \le x \le 1/2$$

where *x* is an *N*-point linearly spaced vector over the interval [1/2, 1/2].

Construction `H = sigwin.barthannwin` returns a modified Bartlett-Hanning window object H of length 64.

`H = sigwin.barthannwin(`*Length*`)` returns a modified Bartlett-Hanning window object H of length *Length*. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Properties **Length**

Modified Bartlett-Hanning window length. The window length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Methods

generate	Generates modified Bartlett-Hanning window
info	Display information about modified Bartlett-Hanning window object
winwrite	Save Bartlett window object values in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Default length N=64 modified Bartlett-Hanning window:

```
H=sigwin.barthannwin;
wvtool(H);
```


Generate length N=128 modified Bartlett-Hanning window, return values, and write ASCII file with window values:

sigwin.barthannwin

```
H=sigwin.barthannwin(128);
% Return window with generate
win=generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'barthannwin_128')
```

References   Yeong, H.H., and Pearce, J.A. "A New Window and Comparison to Standard Windows," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 37, 1989, pp. 298–301.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

                   • Property Attributes

Purpose   Generates modified Bartlett-Hanning window

Syntax   win = generate(H)

Description   win = generate(H) returns the values of the modified Bartlett-Hanning
                   window object H as a double-precision column vector.

Examples   Extract values from modified Bartlett-Hanning window object:

```
H=sigwin.barthannwin(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.barthannwin.info

Purpose   Display information about modified Bartlett-Hanning window object

Syntax   info(H)
                   info_win = info(H)

Description   info(H) displays length information for the modified Bartlett-Hanning
                   window object H.

                   info_win = info(H) returns length information for the modified
                   Bartlett-Hanning window object H in the character array info_win.

Examples   Return information about a modified Bartlett-Hanning window object:

                   H = sigwin.barthannwin(256);
                   info_win = info(H);

Purpose   Save Bartlett window object values in ASCII file

Syntax   winwrite(H)
                   winwrite(H,'filename')

Description   winwrite(H) opens a dialog box that enables you to export the values of
                   the modified Bartlett-Hanning window object H to an ASCII file with
                   filename extension wf .

                   winwrite(H,'filename') saves the values of the modified
                   Bartlett-Hanning window object H in the current folder as a column
                   vector in the ASCII file 'filename'. The filename extension is wf.

Examples   Write modified Bartlett-Hanning window values to ASCII file:

                   ```
                   H = sigwin.barthannwin;
                   % Open dialog box for ASCII file
                   winwrite(H);
                   ```

sigwin.bartlett

Purpose   Construct Bartlett window object

Description   `sigwin.bartlett` creates a handle to a Bartlett window object for use in
                     spectral analysis and filtering by the window method. Object methods
                     enable workspace import and ASCII file export of the window values.

                     For *N* even, the following equation defines the Bartlett window:

$$w(n) = \begin{cases} \dfrac{2n}{N-1} & 0 \le n \le N/2-1 \\ 2 - \dfrac{2n}{N-1} & N/2 \le n \le N-1 \end{cases}$$

                     For *N* odd, the equation for the Bartlett window is:

$$w(n) = \begin{cases} \dfrac{2n}{N-1} & 0 \le n \le (N-1)/2 \\ 2 - \dfrac{2n}{N-1} & (N-1)/2+1 \le n \le N-1 \end{cases}$$

Construction   `H = sigwin.bartlett` returns a Bartlett window object `H` of length 64.

                     `H = sigwin.bartlett(`*Length*`)` returns a Bartlett window object `H` of
                     length *Length*. *Length* must be a positive integer. Entering a positive
                     noninteger value for *Length* rounds the length to the nearest integer.
                     Entering a 1 for *Length* results in a window with a single value of 1.

Properties   **Length**

                         Bartlett window length. The length requires a positive integer.
                         Entering a positive noninteger value for *Length* rounds the length
                         to the nearest integer. Entering a 1 for *Length* results in a
                         window with a single value of 1.

Methods

generate	Generates Bartlett window
info	Display information about Bartlett window object
winwrite	Save Bartlett window object values in ASCII file

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Create default length N=64 Bartlett window:

```
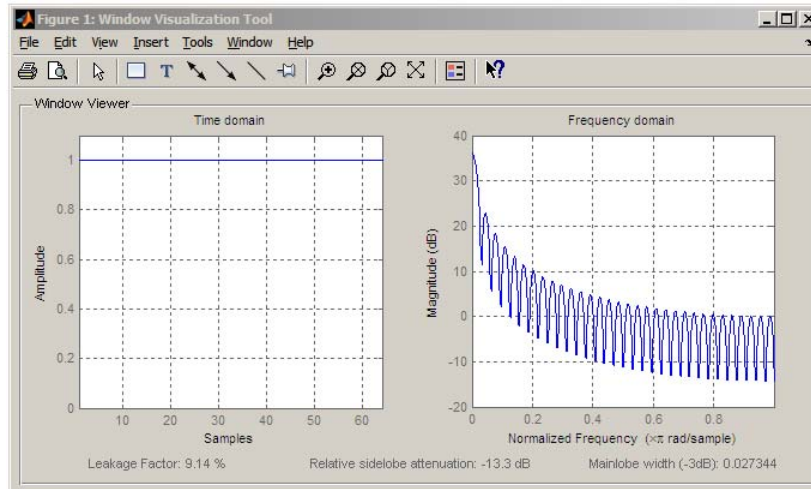H = sigwin.bartlett;
wvtool(H);
```


Generate length N=128 Bartlett window, return values, and write ASCII file with window values:

```
H = sigwin.bartlett(128);
% Return window with generate
```

```
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'bartlett_128')
```

References   Oppenheim, A.V., and Schafer, R.W. *Discrete-time Signal Processing*,
Upper Saddle River, N.J: Prentice Hall, 1989, pp. 444–447.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

    • Property Attributes

Purpose Generates Bartlett window

Syntax `win = generate(H)`

Description `win = generate(H)` returns the values of the Bartlett window object `H` as a double-precision column vector.

Examples Extract values from Bartlett window object:

```
H = sigwin.bartlett(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.bartlett.info

Purpose	Display information about Bartlett window object
Syntax	info(H) info_win = info(H)
Description	info(H) displays length information for the Bartlett window object H. info_win = info(H) returns length information for the Bartlett window object H in the character array info_win.
Examples	Return information about a Bartlett window object: H = sigwin.bartlett(256); info_win = info(H);

Purpose   Save Bartlett window object values in ASCII file

Syntax   winwrite(H)
                  winwrite(H,'filename')

Description   winwrite(H) opens a dialog box that enables you to export the values of
                  the Bartlett window object H to an ASCII file with filename extension wf.

                  winwrite(H,'filename') saves the values of the Bartlett window
                  object H in the current folder as a column vector in the ASCII file
                  'filename'. The filename extension is wf.

Examples   Write Bartlett window values to ASCII file:

                  H=sigwin.bartlett;
                  % Open dialog box for ASCII file
                  winwrite(H);

sigwin.blackman

Purpose   Construct Blackman window object

Description   `sigwin.blackman` creates a handle to a Blackman window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Blackman window of length *N*:

$$w(n) = 0.42 - 0.5\cos(2\pi n/(N-1)) + 0.08\cos(4\pi n/(N-1)) \quad 0 \le n \le M-1$$

where *M* is *N/2* for *N* even and *(N+1)/2* for *N* odd.

In the symmetric case, the second half of the Blackman window $M \le n \le N-1$ is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Blackman window in FIR filter design.

The periodic Blackman window is constructed by extending the desired window length by one sample to *N+1*, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a Blackman window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

Construction   `H = sigwin.blackman` returns a Blackman window object H of length 64 with symmetric sampling.

`H = sigwin.blackman(Length)` returns a Blackman window object H of length *Length* with symmetric sampling. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.blackman(Length,SamplingFlag)` returns a Blackman window object H with sampling *Sampling_Flag*. The *Sampling_Flag* can be either `'symmetric'` or `'periodic'`.

Properties

Length

Blackman window length. Must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

SamplingFlag

'symmetric' is the default and forces exact symmetry between the first and second halves of the Blackman window. A symmetric window is preferred in FIR filter design by the window method.

'periodic' designs a symmetric Blackman window of length *Length+1* and truncates the window to length *Length*. This design is preferred in spectral analysis where the window is treated as one period of a *Length*-point periodic sequence.

Methods

generate	Generates Blackman window
info	Display information about Blackman window object
winwrite	Save Blackman window in ASCII file

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Default length N=64 symmetric Blackman window:

```
H = sigwin.blackman;
wvtool(H);
```

sigwin.blackman

Generate length N=128 periodic Blackman window, return values, and write ASCII file:

```
H = sigwin.blackman(128,'periodic');
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'blackman_128')
```

References   Oppenheim, A.V. and Schafer, R.W. *Discrete-time Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1989, pp. 444–447.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes
              • Property Attributes

Purpose   Generates Blackman window

Syntax   win = generate(H)

Description   win = generate(H) returns the values of the Blackman window object
                 H as a double-precision column vector.

Examples   Extract values from Blackman window object:

```
H = sigwin.blackman(128);
% Extract window values as column vector
win = generate(H);
```

sigwin.blackman.info

Purpose   Display information about Blackman window object

Syntax   info(H)
                info_win = info(H)

Description info(H) displays length and sampling information about the Blackman window object H.

info_win = info(H) returns length and sampling information about the Blackman window object H in the character array info_win.

Examples   Return information about a Blackman window object:

```
H = sigwin.blackman(256);
info_win = info(H);
```

Purpose   Save Blackman window in ASCII file

Syntax   winwrite(H)
                     winwrite(H,'filename')

Description   winwrite(H) opens a dialog box that enables you to export the values of
                    the Blackman window object H to an ASCII file with filename extension
                    wf .

                    winwrite(H,'filename') saves the values of the Blackman window
                    object H in the current folder as a column vector in the ASCII file
                    'filename' with filename extension wf.

Examples   Write Blackman window values to ASCII file:

                    ```
                    H=sigwin.blackman;
                    % Open dialog box for ASCII file
                    winwrite(H);
                    ```

sigwin.blackmanharris

Purpose   Construct Blackman–Harris window object

Description   `sigwin.blackmanharris` creates a handle to a Blackman-Harris
window object for use in spectral analysis and FIR filtering by the
window method. Object methods enable workspace import and ASCII
file export of the window values.

The following equation defines the **symmetric** Blackman-Harris
window of length *N*:

$$w(n) = a_0 - a_1 \cos(\tfrac{2\pi n}{N-1}) + a_2 \cos(\tfrac{4\pi n}{N-1}) - a_3 \cos(\tfrac{6\pi n}{N-1}) \quad 0 \le n \le N - 1$$

The following equation defines the **periodic** Blackman-Harris window
of length *N*:

$$w(n) = a_0 - a_1 \cos(\tfrac{2\pi n}{N}) + a_2 \cos(\tfrac{4\pi n}{N}) - a_3 \cos(\tfrac{6\pi n}{N}) \quad 0 \le n \le N - 1$$

The following table lists the coefficients:

Coefficient	Value
a0	0.35875
a1	0.48829
a2	0.14128
a3	0.01168

Construction   `H = sigwin.blackmanharris` returns a Blackman-Harris window
object H of length 64.

`H = sigwin.blackmanharris(Length)` returns a Blackman-Harris
window object H of length `Length`. `Length` must be a positive integer.
Entering a positive noninteger value for `Length` rounds the length to
the nearest integer. Entering a 1 for `Length` results in a window with
a single value of 1.

Properties   **Length**

Blackman-Harris window length. The window length requires a positive integer. Entering a positive noninteger value for Length rounds the length to the nearest integer. Entering a 1 for Length results in a window with a single value of 1.

SamplingFlag

The type of window returned as one of `'symmetric'` or `'periodic'`. The default is `'symmetric'`. A symmetric window exhibits perfect symmetry between halves of the window. Setting the SamplingFlag property to `'periodic'` results in a N-periodic window. The equations for the Blackman-Harris window differ slightly based on the value of the SamplingFlag property. See "Description" on page 3-902 for details.

Methods

generate	Generates Blackman–Harris window
info	Display information about Blackman–Harris window object
winwrite	Save Blackman–Harris window in ASCII file

Copy Semantics   Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples   Default length N=64 Blackman-Harris window:

```
H=sigwin.blackmanharris;
wvtool(H);
```

sigwin.blackmanharris

Generate length N=128 periodic Blackman-Harris window, return values, and write ASCII file:

```
H=sigwin.blackmanharris(128);
H.SamplingFlag = 'periodic';
% Return window with generate
win=generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'blackmanharris_128')
```

References   Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE..* Vol. 66, 1978.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

• Property Attributes

Purpose   Generates Blackman–Harris window

Syntax   win = generate(H)

Description   win = generate(H) returns the values of the Blackman–Harris
                     window object H as a double-precision column vector.

Examples   Extract values from Blackman–Harris window object:

```
H=sigwin.blackmanharris(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.blackmanharris.info

Purpose	Display information about Blackman–Harris window object
Syntax	info(H) info_win = info(H)
Description	info(H) displays length information for the Blackman–Harris window object H. info_win = info(H) returns length information for the Blackman–Harris window object H in the character array info_win.
Examples	Return information about a Blackman–Harris window object: H = sigwin.blackmanharris(256); info_win = info(H);

Purpose   Save Blackman–Harris window in ASCII file

Syntax   ```
winwrite(H)
winwrite(H,'filename')
```

Description   winwrite(H) opens a dialog box that enables you to export the values of
the Blackman–Harris window object H to an ASCII file with filename
extension wf.

winwrite(H,'filename') saves the values of the Blackman–Harris
window object H in the current folder as a column vector in the ASCII
file 'filename' with filename extension wf.

Examples   Write Blackman–Harris window values to ASCII file:

```
H=sigwin.blackmanharris;
% Open dialog box for ASCII file
winwrite(H);
```

sigwin.bohmanwin

Purpose   Construct Bohman window object

Description   `sigwin.bohmanwin` creates a handle to a Bohman window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Bohman window of length *N*:

$$w(x) = (1 - |x|)\cos(\pi\,|x|) + \frac{1}{\pi}\sin(\pi\,|x|) \quad -1 \le x \le 1$$

where *x* is a length *N* vector of linearly spaced values generated using `linspace`. The first and last elements of the Bohman window are forced to be identically zero.

Construction   `H = sigwin.bohmanwin` returns a Bohman window object `H` of length 64.

`H = sigwin.bohmanwin(`*Length*`)` returns a Bohman window object `H` of length *Length*. *Length* is a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Properties   **Length**

      Bohman window length. Must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Methods

generate	Generates Bohman window
info	Display information about Bohman window object
winwrite	Save Bohman window object values in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Default length N=64 Bohman window:

```
H=sigwin.bohmanwin;
wvtool(H);
```
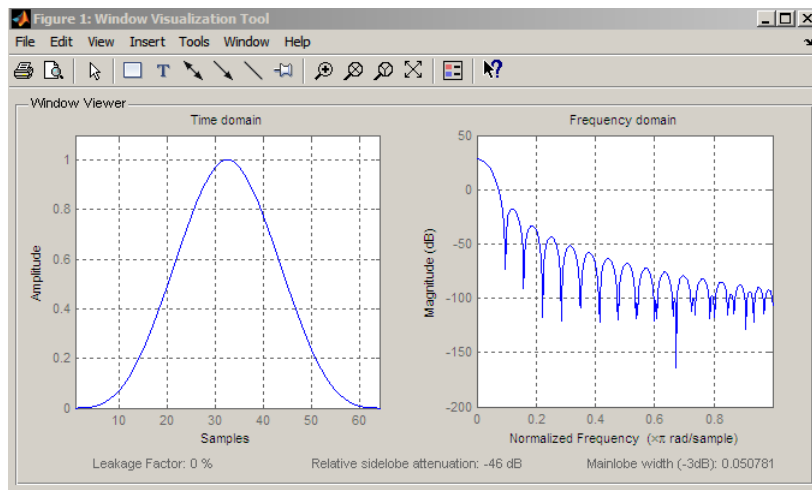

Generate length N=128 Bohman window, return values, and write ASCII file:

```
H=sigwin.bohmanwin(128);
% Return window with generate
win=generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'bohmanwin_128')
```

sigwin.bohmanwin

References   Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform."*Proceedings of the IEEE*.Vol. 66, 1978, pp. 51–83.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

• Property Attributes

Purpose   Generates Bohman window

Syntax   `win = generate(H)`

Description   `win = generate(H)` returns the values of the Bohman window object as a double-precision column vector.

Examples   Extract values from Bohman window object:

```
H=sigwin.bohmanwin(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.bohmanwin.info

Purpose	Display information about Bohman window object
Syntax	info(H) info_win = info(H)
Description	info(H) displays length information for the Bohman window object H. info_win = info(H) returns length information for the Bohman window object H in the character array info_win.
Examples	Return information for a Bohman window object: H=sigwin.bohmanwin(256); info_win=info(H);

Purpose   Save Bohman window object values in ASCII file

Syntax   ```
winwrite(H)
winwrite(H,'filename')
```

**Description**   winwrite(H) opens a dialog to export the values of the Bohman window object H to an ASCII file. The file extension .wf is automatically appended.

winwrite(H,'filename') saves the values of the Bohman window object H in the current folder as a column vector in the ASCII file 'filename'. The file extension .wf is automatically appended to filename.

**Examples**      Write Bohman window values to ASCII file:

```
H=sigwin.bohmanwin;
% Open dialog box for ASCII file
winwrite(H);
```

sigwin.chebwin

Purpose Construct Dolph-Chebyshev window object

Description sigwin.chebwin creates a handle to a Dolph–Chebyshev window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The Dolph-Chebyshev window is constructed in the frequency domain by taking samples of the window's Fourier transform:

$$\hat{W}(k) = (-1)^k \frac{\cos[N \cos^{-1}[\beta \cos(\pi k / N)]]}{\cosh[N \cosh^{-1}(\beta)]} \quad 0 \le k \le N-1$$

where

$$\beta = \cos[1/N \cosh^{-1}(10^{\alpha})]$$

α determines the level of the sidelobe attenuation. The level of the sidelobe attenuation is equal to -20α. For example, 100 dB of attenuation results from setting $\alpha = 5$

The discrete-time Dolph-Chebyshev window is obtained by taking the inverse DFT of $\hat{W}(k)$ and scaling the result to have a peak value of 1.

Construction H = sigwin.chebwin returns a Dolph-Chebyshev window object H of length 64 with relative sidelobe attenuation of 100 dB.

H = sigwin.chebwin(*Length*) returns a Dolph–Chebyshev window object H of length *Length* with relative sidelobe attenuation of 100 dB. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. A window length of 1 results in a window with a single value equal to 1.

H = sigwin.chebwin(*Length*,*SidelobeAtten*) returns a Dolph-Chebyshev window object with relative sidelobe attenuation of *atten_param* dB.

Properties   **Length**

Dolph-Chebyshev window length.

SidelobeAtten

The attenuation parameter in dB. The attenuation parameter is a positive real number that determines the relative sidelobe attenuation of the window.

Methods

generate	Generates Dolph-Chebyshev window
info	Display information about Dolph–Chebyshev window object
winwrite	Save Dolph-Chebyshev window object values in ASCII file

Copy Semantics   Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples   Default length N=64 Dolph–Chebyshev window with 100 dB relative sidelobe attenuation:

```
H=sigwin.chebwin;
wvtool(H);
```

sigwin.chebwin

Generate length N=128 Chebyshev window with 120 dB attenuation, return values, and write ASCII file:

```
H=sigwin.chebwin(128,120);
% Return window with generate
win=generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'chebwin_128_100')
```

References   Harris.F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, 1978, pp. 51–83.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

      • Property Attributes

Purpose   Generates Dolph-Chebyshev window

Syntax   win = generate(H)

Description win = generate(H) returns the values of the Dolph-Chebyshev window
object H as a double-precision column vector.

Examples   Extract values from Dolph-Chebyshev window object:

```
H=sigwin.chebwin(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.chebwin.info

Purpose	Display information about Dolph–Chebyshev window object
Syntax	info(H) info_win = info(H)
Description	info(H) displays length and relative sidelobe attenuation information for the Dolph-Chebyshev window object H. info_win = info(H) returns length information for the Dolph-Chebyshev window object H in the character array info_win.
Examples	Return information about a Dolph-Chebyshev window object: H=sigwin.chebwin(256); info_win=info(H);

Purpose   Save Dolph-Chebyshev window object values in ASCII file

Syntax   winwrite(H)
                   winwrite(H,'filename')

Description   winwrite(H) opens a dialog to export the values of the Dolph-Chebyshev
                   window object H to an ASCII file. The file extension .wf is automatically
                   appended.

                   winwrite(H,'filename') saves the values of the Dolph-Chebyshev
                   window object H in the current folder as a column vector in the ASCII
                   file 'filename'. The file extension .wf is automatically appended to
                   filename.

Examples   Write Dolph-Chebyshev window values to ASCII file:

```
H=sigwin.chebwin;
% Open dialog box for ASCII file
winwrite(H);
```

sigwin.flattopwin

Purpose Construct flat top window object

Description `sigwin.flattopwin` creates a handle to a flat top window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

Construction `H = sigwin.flattopwin` returns a flat top window object `H` of length 64 with symmetric sampling.

`H = sigwin.flattopwin(`*Length*`)` returns a flat top window object of length *Length* with symmetric sampling. *Length* must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.flattopwin(`*Length*`,`*SamplingFlag*`)` returns a flat top window object `H` of length *Length* with sampling *SamplingFlag*. The *SamplingFlag* can be either `'symmetric'` or `'periodic'`.

Properties **Length**

Flat top window length. Must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

SamplingFlag

`'symmetric'` is the default and forces exact symmetry between the first and second halves of the flat top window. A symmetric window is preferred in FIR filter design.

`'periodic'` designs a symmetric flat top window of length *Length+1* and truncates the window to length *Length*. This design is preferred in spectral analysis where the window is treated as one period of a *Length*-point periodic sequence.

Methods	generate	Generates flat top window
	info	Display information about flat top window object
	winwrite	Save flat top window in ASCII file

Definitions The following equation defines the flat top window of length *N*:

$$w(n) = a_0 - a_1 \cos(2\pi n / (N-1)) + a_2 \cos(4\pi n / (N-1)) - a_3 \cos(6\pi n / (N-1)) + a_4 \text{ co}$$

where *M* is *N/2* for *N* even and *(N+1)/2* for *N* odd.

The second half of the symmetric flat top window $M \le n \le N-1$ is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a flat top window in FIR filter design by the window method.

The periodic flat top window is constructed by extending the desired window length by one sample, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a flat top window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

The coefficients are listed in the following table:

Coefficient	Value
a0	0.21557895
a1	0.41663158
a2	0.277263158
a3	0.083578947
a4	0.006947368

sigwin.flattopwin

Copy Semantics   Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples   Default length N=64 symmetric flat top window:

```
H=sigwin.flattopwin;
wvtool(H);
```


Generate length N=128 periodic flat top window, return values, and write ASCII file:

```
H=sigwin.flattopwin(128,'periodic');
% Return window with generate
win=generate(H);
% Write ascii file in current directory
% with window values
winwrite(H,'flattopwin_128')
```

References   Oppenheim, A.V. and Schafer, R.W. *Discrete-time Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1989, pp. 444–447.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

                    • Property Attributes

sigwin.flattopwin.generate

Purpose   Generates flat top window

Syntax   win = generate(H)

Description win = generate(H) returns the values of the flat top window object as
a double-precision column vector.

Examples   Extract values from flat top window object:

```
H=sigwin.flattopwin(128);
% Extract window values as column vector
win=generate(H);
```

Purpose   Display information about flat top window object

Syntax   info(H)
                 info_win = info(H)

Description info(H) displays length and sampling information for the flat top
                 window object H.

                 info_win = info(H) returns length and sampling information for the
                 flat top window object H in the character array info_win.

Examples   Return information about a flat top window object:

                 H=sigwin.flattopwin(256);
                 info_win=info(H);

sigwin.flattopwin.winwrite

Purpose	Save flat top window in ASCII file
Syntax	winwrite(H) winwrite(H,'filename')
Description	winwrite(H) opens a dialog to export the flat top window values to an ASCII file. The file extension .wf is automatically appended. winwrite(H,'filename') saves the values of the flat top window object H in the current folder as a column vector in the ASCII file 'filename'. The file extension .wf is automatically appended to filename.
Examples	Write flat top window values to ASCII file: H=sigwin.flattopwin; % Open dialog for ASCII file winwrite(H);

Purpose   Construct Gaussian window object

Description   `sigwin.gausswin` creates a handle to a Gaussian window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Gaussian window of length *N*:

$$w(x) = e^{-1/2(\alpha^2 x^2/M^2)} \quad -M \leq x \leq M$$

where *M=(N-1)/2* and *x* is a linearly spaced vector of length *N*.

Equating α with the usual standard deviation of a Gaussian value, σ, note:

$$\alpha = \frac{(N-1)}{2\sigma}$$

Construction   `H = sigwin.gausswin` returns a Gaussian window object H of length 64 and dispersion parameter *alpha* of 2.5.

`H = sigwin.gausswin(`*Length*`)` returns a Gaussian window object H of length *Length* and dispersion parameter *alpha* of 2.5. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.gausswin(`*Length*`,`*Alpha*`)` returns a Gaussian window object with dispersion parameter *alpha*. *alpha* requires a nonnegative real number and is inversely proportional to the standard deviation of a Gaussian value.

Properties   **Length**

Gaussian window length. The window length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

sigwin.gausswin

Alpha

Width of Gaussian window. `Alpha` is inversely proportional to the standard deviation of a Gaussian. Larger values of `Alpha` produce Gaussian windows with inflection points closer to the peak value, or narrower windows. In the frequency domain, larger values of `Alpha` produce a Gaussian window with increased spread of the main lobe in frequency but decreased sidelobe energy.

Methods

generate	Generates Gaussian window
info	Display information about Gaussian window object
winwrite	Save Gaussian window in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Compare two Gaussian windows with different `alpha` values:

```
H=sigwin.gausswin(64,4);
H1=sigwin.gausswin(64,2.5);
% Plot comparison
fwvt=wvtool(H,H1);
legend(get(fwvt,'currentaxes'),'\alpha=4','\alpha=2.5');
```

The main lobe is wider for `alpha=4` but the window, with `alpha=4`, demonstrates reduced sidelobe energy.

References   Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE*. Vol. 66, 1978, pp. 51–83.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes
              • Property Attributes

sigwin.gausswin.generate

Purpose   Generates Gaussian window

Syntax   win = generate(H)

Description win = generate(H) returns the values of the Gaussian window object H
                 as a double-precision column vector.

Examples   Extract values from Gaussian window object:

```
H=sigwin.gausswin(128,4);
% Extract window values as column vector
win=generate(H);
```

Purpose   Display information about Gaussian window object

Syntax   info(H)
                   info_win = info(H)

Description   info(H) displays length and dispersion information for the Gaussian
                   window object H.

                   info_win = info(H) returns length and dispersion information for the
                   Gaussian window object H in the character array info_win.

Examples   Return information about a Gaussian window object:

                   H=sigwin.gausswin(256);
                   info_win=info(H);

sigwin.gausswin.winwrite

Purpose	Save Gaussian window in ASCII file
Syntax	`winwrite(H)` `winwrite(H,'filename')`
Description	`winwrite(H)` opens a dialog to export the values of Gaussian window object `H` to an ASCII file. The file extension `.wf` is automatically appended. `winwrite(H,'filename')` saves the values of the Gaussian window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

Examples	Write Gaussian window values to ASCII file:
	
	 ```H=sigwin.gausswin;```
	 ```% Open dialog for ASCII file```
	 ```winwrite(H);```

Purpose   Construct Hamming window object

Description sigwin.hamming creates a handle to a Hamming window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Hamming window of length *N*:

$$w(n) = 0.54 - 0.46\cos(2\pi n / N - 1) \quad 0 \le n \le M - 1$$

where *M* is *N/2* for *N* even and *(N+1)/2* for *N* odd.

The second half of the symmetric Hamming window $M \le n \le N - 1$ is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Hamming window in FIR filter design.

The periodic Hamming window is constructed by extending the desired window length by one sample, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a Hamming window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

Construction H = sigwin.hamming returns a symmetric Hamming window object H of length 64.

H = sigwin.hamming(*Length*) returns a symmetric Hamming window object with length *Length* . *Length* must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

H = sigwin.hamming(*Length*,*SamplingFlag*) returns a Hamming window with sampling *Sampling_Flag*. The *SamplingFlag* can be either 'symmetric' or 'periodic'.

sigwin.hamming

Properties   **Length**

Hamming window length. The window length must be a positive
integer. Entering a positive noninteger value for *Length* rounds
the length to the nearest integer. Entering a 1 for *Length* results
in a window with a single value of 1.

SamplingFlag

`'symmetric'` is the default and forces exact symmetry between
the first and second halves of the Hamming window. A symmetric
window is preferred in FIR filter design by the window method.

`'periodic'` designs a symmetric Hamming window of length
Length+1 and truncates the window to length *Length*. This design
is preferred in spectral analysis where the window is treated as
one period of a *Length*-point periodic sequence.

Methods

generate	Generates Hamming window
info	Display information about Hamming window object
winwrite	Save Hamming window in ASCII file

Copy Semantics   Handle. To learn how copy semantics affect your use of the class,
see Copying Objects in the MATLAB Programming Fundamentals
documentation.

Examples   Default length N=64 symmetric Hamming window:

```
H=sigwin.hamming;
wvtool(H);
```

Generate a length N=128 periodic Hamming window, return the values, and write ASCII file:

```
H=sigwin.hamming(128,'periodic');
% Return window values with generate
win=generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'hamming_128')
```

References   Oppenheim, A.V. and Schafer, R.W. *Discrete-time Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1989, pp. 444–447.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

• Property Attributes

sigwin.hamming.generate

Purpose   Generates Hamming window

Syntax   win = generate(H)

Description win = generate(H) returns the values of the Hamming window object
as a double-precision column vector.

Examples   Extract values from Hamming window object:

```
H=sigwin.hamming(128);
% Extract window values as column vector
win=generate(H);
```

Purpose   Display information about Hamming window object

Syntax   info(H)
info_win = info(H)

Description info(H) displays length and sampling information for the Hamming window object H.

info_win = info(H) returns length and sampling information for the Hamming window object H in the character array info_win.

Examples   Return information about a Hamming window object:

H=sigwin.hamming(256);
info_win=info(H);

sigwin.hamming.winwrite

Purpose   Save Hamming window in ASCII file

Syntax
```
winwrite(H)
winwrite(H,'filename')
```

Description   winwrite(H) opens a dialog to export the Hamming window values to an ASCII file. The file extension .wf is automatically appended.

winwrite(H,'filename') saves the values of the Hamming window object H in the current folder as a column vector in the ASCII file 'filename'. The file extension .wf is automatically appended to filename.

Examples   Write Hamming window values to ASCII file:

```
H=sigwin.hamming;
% Open dialog box for ASCII file
winwrite(H);
```

Purpose   Construct Hann (Hanning) window object

Description sigwin.hann creates a handle to a Hann window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The symmetric Hann window of length *N* is defined as:

$$w(n) = 0.5(1 - \cos(2\pi n / N - 1)) \quad 0 \le n \le M - 1$$

where *M* is *N/2* for *N* even and *(N+1)/2* for *N* odd.

The second half of the symmetric Hann window $M \le n \le N - 1$ is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Hann window in FIR filter design.

The periodic Hann window is constructed by extending the desired window length by one sample, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a Hann window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

Construction H = sigwin.hann returns a symmetric Hann window object H of length 64.

H = sigwin.hann(*Length)* returns a symmetric Hann window object with length *Length*. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

H = sigwin.hann(*Length,SamplingFlag*) returns a Hann window object with sampling *Sampling_Flag*. The *SamplingFlag* can be either 'symmetric' or 'periodic'.

sigwin.hann

Properties   **Length**

Hann window length. Must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

SamplingFlag

`'symmetric'` is the default and forces exact symmetry between the first and second halves of the Hann window. A symmetric window is preferred in FIR filter design by the window method.

`'periodic'` designs a symmetric Hann window of length *Length+1* and truncates the window to length *Length*. This design is preferred in spectral analysis where the window is treated as one period of a *Length*-point periodic sequence.

Methods

generate	Generates Hann window
info	Display information about Hann window object
winwrite	Save Hann window object values in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Default length N=64 symmetric Hann window:

```
H=sigwin.hann;
wvtool(H);
```

Generate length N=128 periodic Hann window, return values, and write ASCII file:

```
H=sigwin.hann(128,'periodic');
% Return window with generate
win=generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'hann_128')
```

References   Oppenheim, A.V. and Schafer, R.W. *Discrete-time Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1989, pp. 444–447.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

              • Property Attributes

sigwin.hann.generate

Purpose   Generates Hann window

Syntax   win = generate(H)

Description   win = generate(H) returns the values of the Hann window object H as
                   a double-precision column vector.

Examples   Extract values from Hann window object:

```
H=sigwin.hann(128);
% Extract window values as column vector
win=generate(H);
```

Purpose   Display information about Hann window object

Syntax   info(H)
                  info_win = info(H)

Description   info(H) displays length and sampling information for the Hann
                  window object H.

                  info_win = info(H) returns length and sampling information for the
                  Hann window object H in the character array info_win.

Examples   Return information about a Hann window object:

                  H=sigwin.hann(256);
                  info_win=info(H);

sigwin.hann.winwrite

Purpose	Save Hann window object values in ASCII file
Syntax	`winwrite(H)` `winwrite(H,'filename')`
Description	`winwrite(H)` opens a dialog to export the values of the Hann window object `H` to an ASCII file. The file extension `.wf` is automatically appended.
	`winwrite(H,'filename')` saves the values of the Hann window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.
Examples	Write Hann window values to ASCII file:

```
H=sigwin.hann;
% Open dialog box for ASCII file
winwrite(H);
```

Purpose   Construct Kaiser window object

Description   `sigwin.kaiser` creates a handle to a Kaiser window object for use
in spectral analysis and FIR filtering by the window method. Object
methods enable workspace import and ASCII file export of the window
values.

The following equation defines the Kaiser window of length *N*:

$$w(x) = I_0\left(\beta\sqrt{1 - \frac{4x^2}{(N-1)^2}}\right) / I_0(\beta) \quad -(N-1)/2 \le x \le (N-1)/2$$

where *x* is linearly spaced *N*-point vector and $I_0()$ is the modified zero-th
order Bessel function of the first kind. β is the attenuation parameter.

Construction   `H = sigwin.kaiser` returns a Kaiser window object H of length 64 and
attenuation parameter *beta* of 0.5.

`H = sigwin.kaiser(`*Length*`)` returns a Kaiser window object H of
length *Length* and attenuation parameter *beta* of 0.5. *Length* requires
a positive integer. Entering a positive noninteger value for *Length*
rounds the length to the nearest integer. Entering a 1 for *Length*
results in a window with a single value of 1.

`H = sigwin.kaiser(`*Length*`,`*Beta*`)` returns a Kaiser window object
with real-valued attenuation parameter *beta*.

Properties   **Length**

Kaiser window length. The window length requires a positive
integer. Entering a positive noninteger value for *Length* rounds
the length to the nearest integer. Entering a 1 for *Length* results
in a window with a single value of 1.

Beta

Attenuation parameter. `Beta` requires a real number. Larger
absolute values of `Beta` result in greater stopband attenuation,

or equivalently greater attenuation between the main lobe and first side lobe.

Methods

generate	Generates Kaiser window
info	Display information about Kaiser window object
winwrite	Save Kaiser window in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Compare two Kaiser windows with different Beta values:

```
H = sigwin.kaiser(128,1.5);
% Kaiser window with Beta=4.5
H1 = sigwin.kaiser(128,4.5);
% Plot comparison
fwvt = wvtool(H,H1);
legend(get(fwvt,'currentaxes'),'\beta=1.5','\beta=4.5');
```

References   Oppenheim, A.V., and Schafer, R.W. *Discrete-time Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1989, pp. 444–447.

See Also   besseli | sigwin | window | wvtool |

Tutorials   • "Windows"

How To   • Class Attributes

• Property Attributes

sigwin.kaiser.generate

Purpose   Generates Kaiser window

Syntax   win = generate(H)

Description   win = generate(H) returns the values of the Kaiser window object as a double-precision column vector.

Examples   Extract values from Kaiser window object:

```
H=sigwin.kaiser(128,4);
% Extract window values as column vector
win=generate(H);
```

Purpose   Display information about Kaiser window object

Syntax
```
info(H)
info_win = info(H)
```

Description   info(H) displays length and attenuation information for the Kaiser window object H.

info_win = info(H) returns length and attenuation information for the Kaiser window object H in the character array info_win.

Examples   Return information about a Kaiser window object:

```
H=sigwin.kaiser(256);
info_win=info(H);
```

sigwin.kaiser.winwrite

Purpose   Save Kaiser window in ASCII file

Syntax   winwrite(H)
                 winwrite(H,'filename')

Description winwrite(H) opens a dialog to export the Kaiser window values to an
                 ASCII file. The file extension .wf is automatically appended.

                 winwrite(H,'filename') saves the values of the Kaiser window object
                 H in the current folder as a column vector in the ASCII file 'filename'.
                 The file extension .wf is automatically appended to filename.

Examples   Write Kaiser window values to ASCII file:

                 H=sigwin.kaiser;
                 % Open dialog box for ASCII file
                 winwrite(H);

Purpose Construct Nuttall defined 4–term Blackman-Harris window object

Description `sigwin.nuttallwin` creates a handle to a Nuttall defined 4–term Blackman-Harris window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

Construction `H = sigwin.nuttallwin` returns a Nuttall defined 4–term Blackman-Harris window object window object `H` of length 64.

`H = sigwin.nuttallwin(Length)` returns a Nuttall defined 4–term Blackman-Harris window object `H` of length `Length`. Entering a positive noninteger value for `Length` rounds the length to the nearest integer. Entering a 1 for `Length` results in a window with a single value of 1. The `SamplingFlag` property defaults to `'symmetric'`.

Properties **Length**

Nuttall defined 4–term Blackman-Harris window length. The window length must be a positive integer. Entering a positive noninteger value for `Length` rounds the length to the nearest integer. Entering a 1 for `Length` results in a window with a single value of 1.

SamplingFlag

The type of window returned as one of `'symmetric'` or `'periodic'`. The default is `'symmetric'`. A symmetric window exhibits perfect symmetry between halves of the window. Setting the `SamplingFlag` property to `'periodic'` results in a N-periodic window. The equations for the Nuttall defined 4–term Blackman-Harris window differ slightly based on the value of the `SamplingFlag` property. See "Definitions" on page 3-952 for details.

sigwin.nuttallwin

Methods

generate	Generates Nuttall defined 4–term Blackman-Harris window
info	Display information about Nuttall defined 4–term Blackman-Harris window object
winwrite	Save Nuttall defined 4-term Blackman-Harris window object values in ASCII file

Definitions

The following equation defines the symmetric Nuttall defined 4–term Blackman-Harris window of length *N*.

$$w(n) = a_0 - a_1 \cos(\tfrac{2\pi n}{N-1}) + a_2 \cos(\tfrac{4\pi n}{N-1}) - a_3 \cos(\tfrac{6\pi n}{N-1}) \quad 0 \le n \le N-1$$

The following equation defines the periodic Nuttall defined 4–term Blackman-Harris window of length *N*.

$$w(n) = a_0 - a_1 \cos(\tfrac{2\pi n}{N}) + a_2 \cos(\tfrac{4\pi n}{N}) - a_3 \cos(\tfrac{6\pi n}{N}) \quad 0 \le n \le N-1$$

The following table lists the coefficients:

Coefficient	Value
a0	0.3635819
a1	0.4891775
a2	0.1365995
a3	0.0106411

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples   Construct a length N=64 symmetric Nuttall defined 4–term
Blackman-Harris window:

```
H=sigwin.nuttallwin;
wvtool(H);
```


Generate a length N=128 periodic Nuttall defined 4–term
Blackman-Harris window, return values, and write ASCII file:

```
H=sigwin.nuttallwin(128);
H.SamplingFlag = 'periodic';
% Return window with generate
win=generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'nuttallwin_128')
```

References   Nuttall, A.H. "Some Windows with Very Good Sidelobe Behavior."
IEEE Transactions on Acoustics, Speech, and Signal Processing. Vol.
29, 1981, pp. 84–91.

sigwin.nuttallwin

See Also   `sigwin | window | wvtool`

Tutorials   • "Windows"

How To   • Class Attributes

                • Property Attributes

Purpose   Generates Nuttall defined 4–term Blackman-Harris window

Syntax   win = generate(H)

Description win = generate(H) returns the values of the Nuttall defined 4–term Blackman-Harris window object as a double-precision column vector.

Examples   Extract values from Nuttall defined 4–term Blackman-Harris window object:

```
H=sigwin.nuttallwin(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.nuttalwin.info

Purpose   Display information about Nuttall defined 4–term Blackman-Harris
                   window object

Syntax   info(H)
                   info_win = info(H)

Description   info(H) displays length information about the Nuttall defined 4–term
                   Blackman-Harris window object H.

                   info_win = info(H) returns length information about the Nuttall
                   defined 4–term Blackman-Harris window object H in the character
                   array info_win.

Examples   Return information about Nuttall defined 4–term Blackman-Harris
                   window object:

                   H=sigwin.nuttallwin(256);
                   info_win=info(H);

Purpose   Save Nuttall defined 4-term Blackman-Harris window object values in ASCII file

Syntax   ```
winwrite(H)
winwrite(H,'filename')
```

Description winwrite(H) opens a dialog to export the values of the Nuttall defined 4-term Blackman-Harris window object H to an ASCII file. The file extension .wf is automatically appended.

winwrite(H,'filename') saves the values of the Nuttall defined 4-term Blackman-Harris window object H in the current folder as a column vector in the ASCII file 'filename'. The file extension .wf is automatically appended to filename.

Examples   Write Nuttall defined 4-term Blackman-Harris window values to ASCII file:

```
H=sigwin.nuttallwin;
% Open dialog box for ASCII file
winwrite(H);
```

sigwin.parzenwin

Purpose   Construct Parzen window object

Description   `sigwin.parzenwin` creates a handle to a Parzen window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the *N*–point Parzen window over the interval $-\frac{(N-1)}{2} \leq n \leq \frac{(N-1)}{2}$:

$$
w(n) = \begin{cases} 1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3 & 0 \leq |n| \leq (N-1)/4 \\ 2\left(1 - \frac{|n|}{N/2}\right)^3 & (N-1)/4 < |n| \leq (N-1)/2 \end{cases}
$$

Construction   `H = sigwin.parzenwin` returns a Parzen window object `H` of length 64.

`H = sigwin.parzenwin(`*Length*`)` returns a Parzen window object `H` of length *Length*. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Properties   **Length**

        Length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Methods

generate	Generate Parzen window
info	Display information about Parzen window object
winwrite	Save Parzen window in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Default length N=64 Parzen window:

```
H=sigwin.parzenwin;
wvtool(H);
```
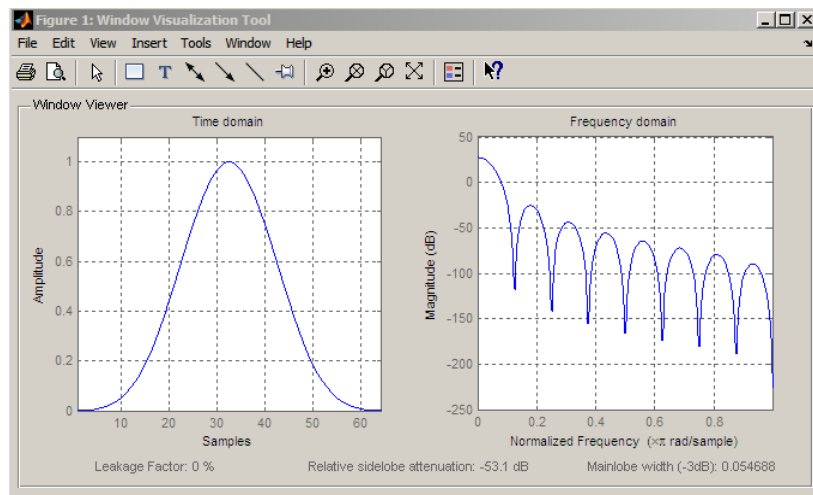

Generate length N=128 Parzen window object, return values, and write ASCII file:

```
H=sigwin.parzenwin(128);
% Return window with generate
win=generate(H);
% Write ascii file in current directory
% with window values
winwrite(H,'parzenwin_128')
```

sigwin.parzenwin

References   Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*, Vol. 66. 1978, pp. 51–83.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

• Property Attributes

Purpose   Generate Parzen window

Syntax   win = generate(H)

Description   win = generate(H) returns the values of the Parzen window object as
                   a double-precision column vector.

Examples   Extract values from Parzen window object:

```
H=sigwin.parzenwin(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.parzenwin.info

Purpose   Display information about Parzen window object

Syntax   info(H)
info_win=info(H)

Description   info(H) displays length information about the Parzen window object H.

info_win=info(H) returns length information about the Parzen window object H in the character array info_win.

Examples   Return information about a Parzen window object:

```
% 256-point Parzen window
H=sigwin.parzenwin(256);
info_win=info(H);
```

Purpose   Save Parzen window in ASCII file

Syntax   winwrite(H)
              winwrite(H,'filename')

Description   winwrite(H) opens a dialog to export the values of the Parzen window object H to an ASCII file. The file extension .wf is automatically appended.

winwrite(H,'filename') saves the values of the Parzen window object H in the current folder as a column vector in the ASCII file 'filename'. The file extension .wf is automatically appended to filename.

Examples   Write Parzen window values to ASCII file:

```
H=sigwin.parzenwin;
% Open dialog box for ASCII file
winwrite(H);
```

sigwin.rectwin

Purpose   Construct rectangular window object

Description   `sigwin.rectwin` creates a handle to a rectangular window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the rectangular window of length *N*:

$$w(n) = 1 \quad 0 \leq n \leq N - 1$$

Construction   `H = sigwin.rectwin` returns a rectangular window object H of length 64.

`H = sigwin.rectwin(`*Length*`)` returns a rectangular window object H of length *Length*. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Properties   **Length**

Rectangular window length. The window length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Methods

generate	Generates rectangular window
info	Display information about rectangular window object
winwrite	Save rectangular window in ASCII file

Copy Semantics   Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples   Create default length N=64 rectangular window:

```
H=sigwin.rectwin;
wvtool(H);
```


Generate length N=128 rectangular window, return values, and write
ASCII file:

```
H=sigwin.rectwin(128);
% Return window with generate
win=generate(H);
% Write ascii file in current directory
% with window values
winwrite(H,'rectwin_128')
```

References   Oppenheim, A.V., and Schafer, R.W. *Discrete-time Signal Processing*,
Upper Saddle River, N.J: Prentice Hall, 1989, pp. 444–447.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

sigwin.rectwin

How To
- Class Attributes
- Property Attributes

Purpose   Generates rectangular window

Syntax   win = generate(H)

Description   win = generate(H) returns the values of the rectangular window
                   object H as a double-precision column vector.

Examples   Extract values from rectangular window object:

```
H=sigwin.rectwin(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.rectwin.info

Purpose	Display information about rectangular window object
Syntax	info(H) info_win = info(H)
Description	info(H) displays length information for the rectangular window object H. info_win = info(H) returns length information for the rectangular window object H in the character array info_win.
Examples	Return information about a rectangular window object: H=sigwin.rectangular(256); info_win=info(H);

Purpose   Save rectangular window in ASCII file

Syntax
```
winwrite(H)
winwrite(H,'filename')
```

Description   winwrite(H) opens a dialog to export the values of the rectangular window object H to an ASCII file. The file extension .wf is automatically appended.

winwrite(H,'filename') saves the values of the rectangular window object H in the current folder as a column vector in the ASCII file 'filename'. The file extension .wf is automatically appended to filename.

Examples   Write rectangular window values to ASCII file:

```
H=sigwin.rectwin;
% Open dialog box for ASCII file
winwrite(H);
```

sigwin.taylorwin

Purpose   Construct Taylor window object

Description   `sigwin.taylorwin` creates a handle to a Taylor window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

Taylor windows are similar to Dolph-Chebyshev windows. The Taylor window approximates the minimization of the main lobe width in the Dolph-Chebyshev window, but allows the sidelobe levels to decrease beyond a certain frequency. Taylor windows are typically used in radar applications, such as weighting synthetic aperature radar images and antenna design.

Construction   `H = sigwin.taylorwin` returns a Taylor window object `H` of length 64, with a maximum sidelobe level of 30 dB and 4 constant-level sidelobes adjacent to the main lobe.

`H = sigwin.taylorwin(`*Length*`)` returns a Taylor window object `H` of length *Length* with a maximum sidelobe level of 30 dB and 4 constant-level sidelobes adjacent to the main lobe. *Length* must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.taylorwin(`*Length*`,`*Nbar*`)` returns a Taylor window object with *Nbar* nearly constant-level sidelobes adjacent to the main lobe. *Nbar* must be a positive integer.

`H = sigwin.taylorwin(`*Length*`,`*Nbar*`,`*SidelobeLevel*`)` returns a Taylor window object with a maximum sidelobe level *SidelobeLevel* dB below the main lobe level.

Properties   **Length**

      Taylor window length. The window length must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Nbar

> Number of nearly constant-level sidelobes. Must be a positive integer.

SidelobeLevel

> Maximum sidelobe level relative to the main lobe peak. The maximum sidelobe level is a nonnegative number which gives side lobes `SidelobeLevel` dB down from the main lobe peak.

Methods	generate	Generates Taylor window
	info	Display information about Taylor window object
	winwrite	Save Taylor window object values in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Default length N=64 Taylor window:

```
H=sigwin.taylorwin;
wvtool(H);
```
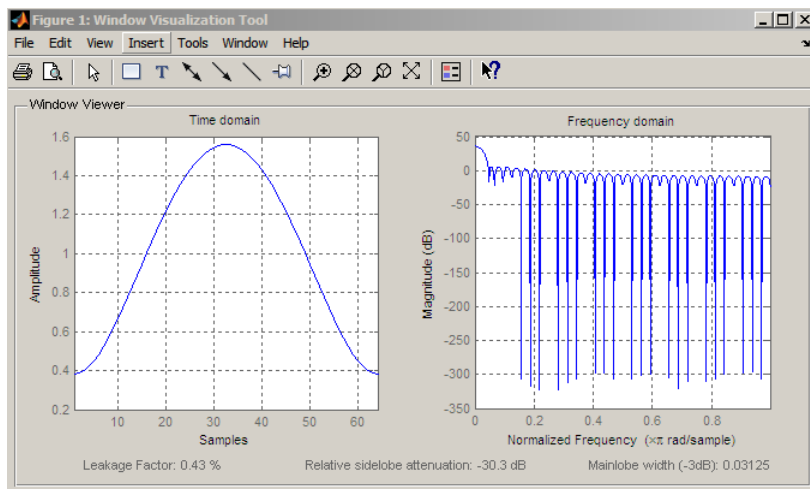
sigwin.taylorwin

Generate length N=128 Taylor window, return values, and write ASCII file with window values:

```
H=sigwin.taylorwin(128);
% Return window with generate
win=generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'taylorwin_128')
```

References   Carrara, W.G., R.M. Majewski and R.S. Goodman. *Spotlight Synthetic Aperature Radar: Signal Processing Algorithms*, Artech House Publishers, Boston, 1995, Appendix D.2.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

     • Property Attributes

Purpose   Generates Taylor window

Syntax   win = generate(H)

Description   win = generate(H) returns the values of the Taylor window object H as
                   a double-precision column vector.

Examples   Extract values from Taylor window object:

```
H=sigwin.taylorwin(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.taylorwin.info

Purpose   Display information about Taylor window object

Syntax   info(H)
info_win = info(H)

Description   info(H) displays length and sidelobe information for the Taylor window object H.

info_win = info(H) returns length and sidelobe information for the Taylor window object H in the character array info_win.

Examples   Return information about a Taylor window object:

H=sigwin.taylorwin(256);
info_win=info(H);

Purpose   Save Taylor window object values in ASCII file

Syntax   winwrite(H)
                   winwrite(H,'filename')

Description   winwrite(H) opens a dialog to export the values of the Taylor window
                   object H to an ASCII file. The file extension .wf is automatically
                   appended.

                   winwrite(H,'filename') saves the values of the Taylor window object
                   H in the current folder as a column vector in the ASCII file 'filename'.
                   The file extension .wf is automatically appended to filename.

Examples   Write Taylor window values to ASCII file:

                   H=sigwin.taylorwin;
                   % Open dialog box for ASCII file
                   winwrite(H);

sigwin.triang

Purpose   Construct triangular window object

Description   `sigwin.triang` is a triangular window object.

`sigwin.triang` creates a handle to a triangular window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

For *L* odd, the triangular window is defined as:

$$w(n) = \begin{cases} \dfrac{2n}{L+1} & 1 \le n \le (L+1)/2 \\ 2 - \dfrac{2n}{L+1} & (L+1)/2 + 1 \le n \le L \end{cases}$$

For *L* even, the triangular window is defined as:

$$w(n) = \begin{cases} \dfrac{(2n-1)}{L} & 1 \le n \le L/2 \\ 2 - \dfrac{(2n-1)}{L} & L/2 + 1 \le n \le L \end{cases}$$

Construction   `H = sigwin.triang` returns a triangular window object H of length 64.

`H = sigwin.triang(Length)` returns a triangular window object H of length *Length*. Entering a positive non-integer value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Properties   **Length**

Triangular window length. The window length requires a positive integer. Entering a positive non-integer value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Methods	generate	Generates triangular window
	info	Display information about triangular window
	winwrite	Save triangular window in ASCII file

Copy Semantics   Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples   Default length L = 64 triangular window:

```
H=sigwin.triang;
wvtool(H);
```
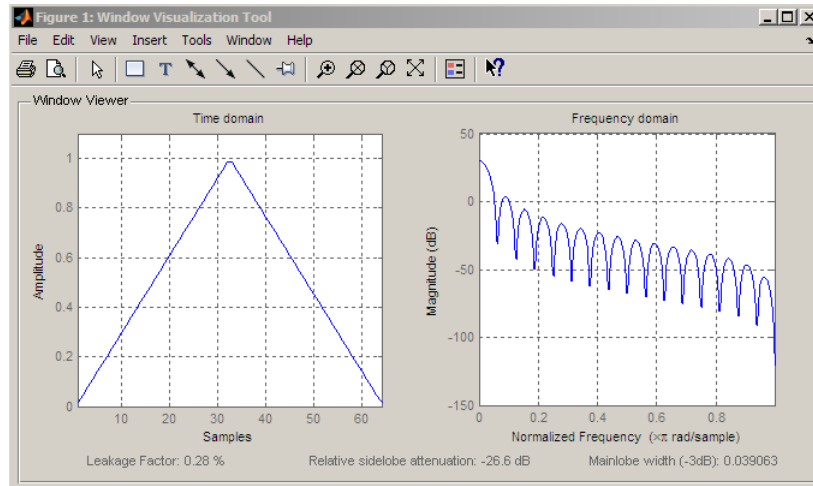

Generate length L = 128 triangular window, return values, and write ASCII file:

```
H=sigwin.triang(128);
```

```
% Return window with generate
win=generate(H);
% Write ascii file in current directory
% with window values
winwrite(H,'triang_128')
```

References   Oppenheim, A.V., and Schafer, R.W. *Discrete-time Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1989, pp. 444–447.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

• Property Attributes

Purpose   Generates triangular window

Syntax   `win = generate(H)`

Description   `win = generate(H)` returns the values of the triangular window object `H` as a double-precision column vector.

Examples   Extract values from triangular window object:

```
H=sigwin.triang(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.triang.info

Purpose   Display information about triangular window

Syntax   info(H)
                   info_array = info(H)

Description   info(H) displays length information for the triangular window object H.

                   info_array = info(H) returns length information for the triangular
                   window object H in the character array info_array.

Examples   Return information about a triangular window object:

                   H=sigwin.triangular(256);
                   info_win=info(H);

Purpose   Save triangular window in ASCII file

Syntax   winwrite(H)
                winwrite(H,'filename')

Description winwrite(H) opens a dialog to export the values of the triangular
                window object H to an ASCII file. The file extension .wf is automatically
                appended.

                winwrite(H,'filename') saves the values of the triangular window
                object H as a column vector in the ASCII file 'filename' in the current
                folder. The file extension .wf is automatically appended to filename.

Examples   Write triangular window values to ASCII file:

                ```
                H=sigwin.triang;
                % Open dialog box for ASCII file
                winwrite(H);
                ```

sigwin.tukeywin

Purpose   Construct Tukey window object

Description   sigwin.tukeywin creates a handle to a Tukey window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the *N*–point Tukey window:

$$w(x) = \begin{cases} \frac{1}{2}\{1 + \cos(\frac{2\pi}{\alpha}[x - \alpha/2])\} & 0 \le x < \frac{\alpha}{2} \\ 1 & \frac{\alpha}{2} \le x < 1 - \frac{\alpha}{2} \\ \frac{1}{2}\{1 + \cos(\frac{2\pi}{\alpha}[x - 1 + \alpha/2])\} & 1 - \frac{\alpha}{2} \le x \le 1 \end{cases}$$

where *x* is a *N*–point linearly spaced vector generated using linspace. The parameter α is the ratio of cosine-tapered section length to the entire window length with $0 \le \alpha \le 1$. For example, setting α=0.5 produces a Tukey window where 1/2 of the entire window length consists of segments of a phase-shifted cosine with period *2a=1*. If you specify α≤0, an *N*-point rectangular window is returned. If you specify α≥1, a von Hann window (sigwin.hann) is returned.

Construction H = sigwin.tukeywin returns a Tukey or cosine-tapered window object H of length 64 with *Alpha* parameter equal to 0.5.

H = sigwin.tukeywin(*Length*) returns a Tukey window object H of length *Length* with *Alpha* parameter equal to 0.5. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer.

H = sigwin.tukeywin(*Length,Alpha*) returns a Tukey window object with the ratio of the tapered section length to the entire window length *Alpha*. *Alpha* defaults to 0.5. As *Alpha* approaches zero, the Tukey window approaches a rectangular window. As *Alpha* approaches one, the Tukey window approaches a Hann window.

Properties

Length

Tukey window length. The window length must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Alpha

The ratio of tapered window section to constant section. As a ratio, Alpha satisfies the inequality $0 \le \alpha \le 1$. As *Alpha* approaches zero, the Tukey window approaches a rectangular window. As *Alpha* approaches one, the Tukey window approaches a Hann window. Specifying *Alpha* less than zero or greater than one replaces *Alpha* with 0 and 1 respectively.

Methods

generate	Generates Tukey window
info	Display information about Tukey window object
winwrite	Save Tukey window in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Default length N=64 Tukey window:

```
H=sigwin.tukeywin;
wvtool(H);
```
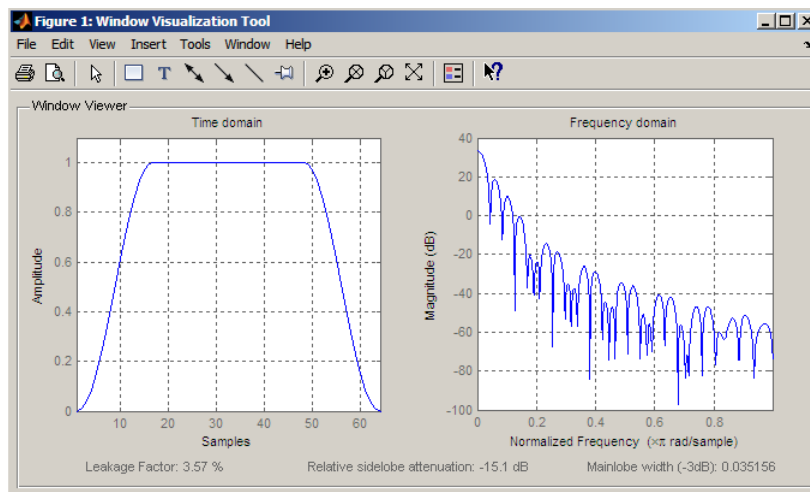
sigwin.tukeywin

Generate length N=128 Tukey window, return values, and write ASCII file:

```
H=sigwin.tukeywin(128,1/4);
% Return window with generate
win=generate(H);
% Write ascii file in current directory
% with window values
winwrite(H,'tukeywin_128')
```

References   [1] Bloomfield P. *Fourier Analysis of Time Series: An Introduction*, New York: Wiley-Interscience, 2000, p.69.

See Also   sigwin | window | wvtool

Tutorials   • "Windows"

How To   • Class Attributes

• Property Attributes

Purpose   Generates Tukey window

Syntax   win = generate(H)

Description win = generate(H) returns the values of the Tukey window object H as
                 a double-precision column vector.

Examples   Extract values from Tukey window object:

```
H=sigwin.tukeywin(128);
% Extract window values as column vector
win=generate(H);
```

sigwin.tukeywin.info

Purpose   Display information about Tukey window object

Syntax   info(H)
                 info_win = info(H)

Description info(H) displays length and tapered–to–constant section ratio
                 information for the Tukey window object H.

                 info_win = info(H) returns length and tapered–to–constant section
                 ratio information for the Tukey window object H in the character array
                 info_win.

Examples   Return information about a Tukey window object:

                 H=sigwin.tukey(256);
                 info_win=info(H);

Purpose   Save Tukey window in ASCII file

Syntax   winwrite(H)
                winwrite(H,'filename')

Description winwrite(H) opens a dialog to export the values of the Tukey window
                object to an ASCII file. The file extension .wf is automatically appended.

                winwrite(H,'filename') saves the values of the Tukey window object
                H in the current folder as a column vector in the ASCII file 'filename'.
                The file extension .wf is automatically appended to filename.

Examples   Write Tukey window values to ASCII file:

```
H=sigwin.tukeywin;
% Open dialog box for ASCII file
winwrite(H);
```

sinc

Purpose　Sinc vector or matrix

Syntax　`y = sinc(x)`

Description　`sinc` computes the sinc function of an input vector or array, where the sinc function is

$$\mathrm{sinc}(t) = \begin{cases} 1, & t = 0 \\ \dfrac{\sin(\pi t)}{\pi t} & t \neq 0 \end{cases}$$

This function is the continuous inverse Fourier transform of the rectangular pulse of width 2π and height 1.

$$\mathrm{sinc}(t) = \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{j\omega t} d\omega$$

`y = sinc(x)` returns an array `y` the same size as `x`, whose elements are the sinc function of the elements of `x`.

The space of functions bandlimited in the frequency range $\omega \in [-\pi, \pi]$ is spanned by the infinite (yet countable) set of sinc functions shifted by integers. Thus any such bandlimited function $g(t)$ can be reconstructed from its samples at integer spacings.

$$g(t) = \sum_{n=-\infty}^{\infty} g(n)\mathrm{sinc}(t - n)$$

Examples　Perform ideal bandlimited interpolation by assuming that the signal to be interpolated is 0 outside of the given time interval and that it has been sampled at exactly the Nyquist frequency:

```
t = (1:10)';              % Column vector of time samples
x = randn(size(t));       % Column vector of data
ts = linspace(-5,15,600)'; % Times at which to interpolate
```

```
y = sinc(ts(:,ones(size(t))) - t(:,ones(size(ts)))')*x;
plot(t,x,'o',ts,y)
```

See Also   chirp | cos | diric | gauspuls | pulstran | rectpuls | sawtooth |
sin | square | tripuls

slewrate

Purpose   Slew rate of bilevel waveform

Syntax
```
S = slewrate(X)
S = slewrate(X,Fs)
S = slewrate(X,T)
[S,LT,UT] = slewrate(...)
[S,LT,UT,LL,UL] = slewrate(...)
S = slewrate(...,Name,Value)
slewrate(...)
```

Description   S = slewrate(X) returns the slew rate for all transitions found in the
bilevel waveform, X. The slew rate is the slope of the line connecting the
10% and 90% reference levels. The sample instants of X are the indices
of the vector. To determine the transitions, slewrate estimates the
state levels of the input waveform by a histogram method. slewrate
identifies all regions that cross the upper-state boundary of the low
state and the lower-state boundary of the high state. The low-state and
high-state boundaries are expressed as the state level plus or minus a
multiple of the difference between the state levels. See "State-Level
Tolerances" on page 3-993.

S = slewrate(X,Fs) specifies the sample rate, Fs, in hertz. The first
time instant in X corresponds to t=0.

S = slewrate(X,T) specifies the sample instants in the vector, T. The
length of T must equal the length of X.

[S,LT,UT] = slewrate(...) returns the time instants when
the waveform crosses the lower-percent reference level, LT, and
upper-percent reference level, UT. If you do not specify lower- and
upper-percent reference levels, the levels default to 10% and 90%.

[S,LT,UT,LL,UL] = slewrate(...) returns the waveform values
that correspond to the lower-reference levels, LL, and upper-reference
levels, UL.

S = slewrate(...,Name,Value) returns the slew rate for all
transitions with additional options specified by one or more
Name,Value pair arguments.

slewrate(...) plots the bilevel waveform and darkens the regions of each transition where the slew rate is computed. The plot marks the lower- and upper-reference level crossings and associated reference levels. The plot indicates the state levels and associated lower and upper tolerances.

Input Arguments

X

Bilevel waveform as a real-valued column or row vector. If the input waveform does not have at least one transition, slewrate returns an empty matrix.

Fs

Sampling rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

PctRefLevels

Percent reference levels. See "Percent Reference Levels" on page 3-992 for a definition.

Default: [10,90]

StateLevels

Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, slewrate estimates the state levels from the input waveform using the histogram method.

Tolerance

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 3-993.

Default: 2

Output Arguments

S

Slew rates as real-valued scalars. A positive slew rate indicates that the upper-percent reference level occurs later than the lower-percent reference level. A negative slew rate indicates that the upper-percent reference level occurs before the lower-percent reference level.

LT

Time instants when signal crosses the lower percent reference level. If you do not specify the lower percent reference levels with the `'PctRefLevels'` name-value pair, the lower percent reference level is 10%.

UT

Time instants when signal crosses the upper-percent reference level. If you do not specify the upper-percent reference levels with the `'PctRefLevels'` name-value pair, the upper-percent reference level is 90%.

LL

Waveform values at the lower-reference level.

UL

Waveform values at the upper-reference level.

Definitions

Percent Reference Levels

If S_1 is the low state, S_2 is the high state, and U is the *upper*-percent reference level. The waveform value corresponding to the upper-percent reference level is

$$S_1 + \frac{U}{100}(S_2 - S_1)$$

If *L* is the *lower*-percent reference level, the waveform value corresponding to the lower percent reference level is

$$S_1 + \frac{L}{100}(S_2 - S_1)$$

Slew Rate

The slew rate is the slope of a line connecting the upper- and lower-percent reference levels. Let t_L denote the time instant when the waveform crosses the lower reference level and t_U denote the time instant when the waveform crosses the upper percent reference level. Using the definitions for the upper and lower percent reference levels given in "Percent Reference Levels" on page 3-992, the slew rate is

$$\frac{S_1 + \frac{U}{100}(S_2 - S_1) - \{S_1 + \frac{L}{100}(S_2 - S_1)\}}{t_U - t_L}$$

$$\frac{\frac{U-L}{100}(S_2 - S_1)}{t_U - t_L}$$

When t_L occurs earlier than t_U, the slew rate is positive. When t_U occurs earlier than t_L, the slew rate is negative.

State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

State Levels ± 2% Tolerance

Examples

Slew Rate For One-Transition Waveform

Use `slewrate` with no output arguments to plot the slew rate information for a step waveform sampled at 4 MHz.

slewrate

Load the `transitionex.mat` file and compute the slew rate.

```
load('transitionex.mat', 'x', 't');
slewrate(x, t)
```
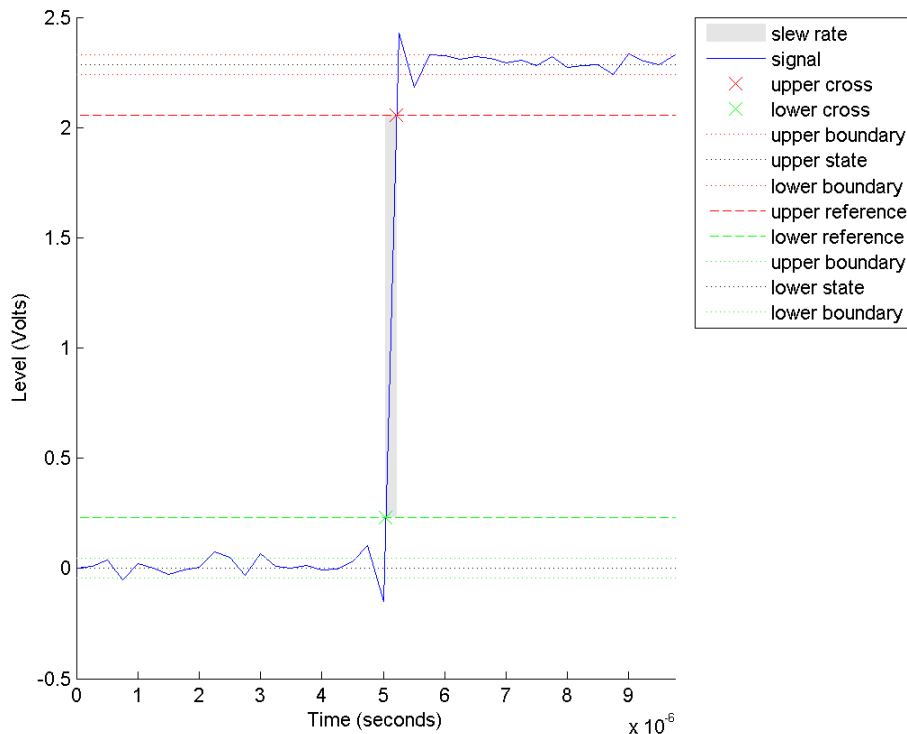

Slew Rates for Three-Transition Waveform

Use the waveform in "Slew Rate For One-Transition Waveform" on page 3-995 to create a three-transition (two positive and one negative) bilevel waveform. Obtain the slew rates for the three transitions.

```
load('transitionex.mat', 'x');
```

```
y = [x ; fliplr(x)];
t = 0:1/4e6:(length(y)*(1/4e6))-1/4e6;
S = slewrate(y, t);
```

Lower and Upper Transition Times

Return the lower- and upper-transition times for the three-transition waveform in"Slew Rates for Three-Transition Waveform" on page 3-996 .

```
load('transitionex.mat', 'x');
y = [x ; fliplr(x)];
t = 0:1/4e6:(length(y)*(1/4e6))-1/4e6;
[S,LT,UT] = slewrate(y, t);
% or [S,LT,UT] = slewrate(y,4e6);
```

Lower and Upper Reference Levels

Return the waveform values corresponding to the lower- and upper-reference levels for the three-transition waveform in"Slew Rates for Three-Transition Waveform" on page 3-996 . Compute these values for the default 10% and 90% and for 20% and 80%.

```
load('transitionex.mat', 'x');
y = [x ; fliplr(x)];
t = 0:1/4e6:(length(y)*(1/4e6))-1/4e6;
[~,LT_1090,UT_1090,LL_1090,UL_1090] = slewrate(y, t);
[~,LT_2080,UT_2080,LL_2080,UL_2080] = slewrate(y, t,...
    'PctRefLevels',[20 80]);
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

See Also   falltime | midcross | pulsewidth | risetime | settlingtime | statelevels

sos2cell

Purpose Convert second-order sections matrix to cell array

Syntax
```
c = sos2cell(m)
c = sos2cell(m,g)
```

Description `c = sos2cell(m)` changes an *L*-by-6 second-order section matrix m generated by `tf2sos` into a 1-by-*L* cell array of 1-by-2 cell arrays c. You can use c to specify a quantized filter with *L* cascaded second-order sections.

The matrix m should have the form

```
m = [b1 a1;b2 a2; ... ;bL aL]
```

where both `bi` and `ai`, with $i = 1, ..., L$, are 1-by-3 row vectors. The resulting c is a 1-by-*L* cell array of cells of the form

```
c = { {b1 a1} {b2 a2} ... {bL aL} }
```

`c = sos2cell(m,g)` with the optional gain term g, prepends the constant value g to c. When you use the added gain term in the command, c is a 1-by-*L* cell array of cells of the form

```
c = {{g,1} {b1,a1} {b2,a2}...{bL,aL} }
```

Examples Use `sos2cell` to convert the 2-by-6 second-order section matrix produced by `tf2sos` into a 1-by-2 cell array c of cells. Display the second entry in the first cell in c:

```
[b,a] = ellip(4,0.5,20,0.6);
m = tf2sos(b,a);
c = sos2cell(m);
c{1}{2}
ans =
    1.0000    0.1677    0.2575
```

See Also tf2sos | cell2sos

Purpose	Convert digital filter second-order section parameters to state-space form
Syntax	`[A,B,C,D] = sos2ss(sos)` `[A,B,C,D] = sos2ss(sos,g)`
Description	sos2ss converts a second-order section representation of a given digital filter to an equivalent state-space representation.

`[A,B,C,D] = sos2ss(sos)` converts the system sos, in second-order section form, to a single-input, single-output state-space representation.

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] \quad = Cx[n] + Du[n]$$

The discrete transfer function in second-order section form is given by

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

sos is a L-by-6 matrix organized as

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

The entries of sos must be real for proper conversion to state space. The returned matrix A is size N-by-N, where $N = L -$ B is a length $N-1$ column vector, C is a length $N-1$ row vector, and D is a scalar.

`[A,B,C,D] = sos2ss(sos,g)` converts the system sos in second-order section form with gain g.

$$H(z) = g \prod_{k=1}^{L} H_k(z)$$

Examples   Compute the state-space representation of a simple second-order section system with a gain of 2:

```
sos = [1  1  1  1  0 -1; -2  3  1  1 10  1];
[A,B,C,D] = sos2ss(sos)
A =
   -10     0    10     1
     1     0     0     0
     0     1     0     0
     0     0     1     0
B =
     1
     0
     0
     0
C =
    21     2   -16    -1
D =
    -2
```

Algorithms   sos2ss first converts from second-order sections to transfer function using sos2tf, and then from transfer function to state-space using tf2ss.

See Also   sos2tf | sos2zp | ss2sos | tf2ss | zp2ss

Purpose   Convert digital filter second-order section data to transfer function form

Syntax   `[b,a] = sos2tf(sos)`
                     `[b,a] = sos2tf(sos,g)`

Description   `sos2tf` converts a second-order section representation of a given digital filter to an equivalent transfer function representation.

`[b,a] = sos2tf(sos)` returns the numerator coefficients `b` and denominator coefficients `a` of the transfer function that describes a discrete-time system given by `sos` in second-order section form. The second-order section format of $H(z)$ is given by

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

`sos` is an L-by-6 matrix that contains the coefficients of each second-order section stored in its rows.

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

Row vectors `b` and `a` contain the numerator and denominator coefficients of $H(z)$ stored in descending powers of z.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \cdots + b_{n+1} z^{-n}}{a_1 + a_2 z^{-1} + \cdots + a_{m+1} z^{-m}}$$

`[b,a] = sos2tf(sos,g)` returns the transfer function that describes a discrete-time system given by `sos` in second-order section form with gain `g`.

sos2tf

$$H(z) = g \prod_{k=1}^{L} H_k(z)$$

Examples   Compute the transfer function representation of a simple second-order section system:

```
sos = [1  1  1  1  0 -1; -2  3  1  1 10  1];
[b,a] = sos2tf(sos)
b =
   -2    1    2    4    1
a =
    1   10    0  -10   -1
```

Algorithms   sos2tf uses the conv function to multiply all of the numerator and denominator second-order polynomials together. For higher order filters (possibly starting as low as order 8), numerical problems due to roundoff errors may occur when forming the transfer function.

See Also   latc2tf | sos2ss | sos2zp | ss2tf | tf2sos | zp2tf

Purpose   Convert digital filter second-order section parameters to zero-pole-gain form

Syntax   ```
[z,p,k] = sos2zp(sos)
[z,p,k] = sos2zp(sos,g)
```

**Description**  sos2zp converts a second-order section representation of a given digital filter to an equivalent zero-pole-gain representation.

[z,p,k] = sos2zp(sos) returns the zeros z, poles p, and gain k of the system given by sos in second-order section form. The second-order section format of $H(z)$ is given by

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

sos is an $L$-by-6 matrix that contains the coefficients of each second-order section in its rows.

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

Column vectors z and p contain the zeros and poles of the transfer function $H(z)$.

$$H(z) = k \frac{(z - z_1)(z - z_2)\cdots(z - z_n)}{(p - p_1)(p - p_2)\cdots(p - p_m)}$$

where the orders $n$ and $m$ are determined by the matrix sos.

[z,p,k] = sos2zp(sos,g) returns the zeros z, poles p, and gain k of the system given by sos in second-order section form with gain g.

# sos2zp

$$H(z) = g \prod_{k=1}^{L} H_k(z)$$

**Examples**     Compute the poles, zeros, and gain of a simple system in second-order section form:

```
sos = [1  1  1  1  0 -1; -2  3  1  1 10  1];
[z,p,k] = sos2zp(sos)
z =
  -0.5000 + 0.8660i
  -0.5000 - 0.8660i
   1.7808
  -0.2808
p =
  -1.0000
   1.0000
  -9.8990
  -0.1010
k =
    -2
```

Algorithms   sos2zp finds the poles and zeros of each second-order section by repeatedly calling tf2zp.

See Also   sos2ss | sos2tf | ss2zp | tf2zp | tf2zpk | zp2sos

Purpose   Second-order (biquadratic) IIR digital filtering

Syntax   ```
y = sosfilt(sos,x)
y = sosfilt(sos,x,dim)
```

Description   `y = sosfilt(sos,x)` applies the second-order section digital filter `sos` to the vector `x`. The output, `y`, is the same length as `x`.

`sos` represents the second-order section digital filter $H(z)$

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

by an *L*-by-6 matrix containing the coefficients of each second-order section in its rows.

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

If `x` is a matrix, `sosfilt` applies the filter to each column of `x` independently. The output `y` is a matrix of the same size, containing the filtered data corresponding to each column of `x`.

If `x` is a multidimensional array, `sosfilt` filters along the first nonsingleton dimension. The output `y` is a multidimensional array of the same size as `x`, containing the filtered data corresponding to each row and column of `x`

`y = sosfilt(sos,x,dim)` operates along the dimension `dim`.

References   [1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also   filter | medfilt1 | sgolayfilt

spectrogram

Purpose   Spectrogram using short-time Fourier transform

Syntax
```
S = spectrogram(x)
S = spectrogram(x,window)
S = spectrogram(x,window,noverlap)
S = spectrogram(x,window,noverlap,nfft)
S = spectrogram(x,window,noverlap,nfft,fs)
[S,F,T] = spectrogram(x,window,noverlap,F)
[S,F,T] = spectrogram(x,window,noverlap,F,fs)
[S,F,T,P] = spectrogram(...)
spectrogram(...,FREQLOCATION)
spectrogram(...)
```

Description spectrogram, when used without any outputs, plots a spectrogram or, when used with an S output, returns the short-time Fourier transform of the input signal. To create a spectrogram from the returned short-time Fourier transform data, refer to the [S,F,T,P] syntax described below.

S = spectrogram(x) returns S, the short time Fourier transform of the input signal vector x. By default, x is divided into eight segments. If x cannot be divided exactly into eight segments, it is truncated. These default values are used.

- window is a Hamming window of length nfft.

- noverlap is the number of samples that each segment overlaps. The default value is the number producing 50% overlap between segments.

- nfft is the FFT length and is the maximum of 256 or the next power of 2 greater than the length of each segment of x. Instead of nfft, you can specify a vector of frequencies, F. See below for more information.

- fs is the sampling frequency, which defaults to normalized frequency.

Each column of S contains an estimate of the short-term, time-localized frequency content of x. Time increases across the columns of S and frequency increases down the rows.

If x is a length Nx complex signal, S is a complex matrix with nfft rows and k columns, where for a scalar window

```
k = fix((Nx-noverlap)/(window-noverlap))
```

or if window is a vector

```
k = fix((Nx-noverlap)/(length(window)-noverlap))
```

For real x, the output S has (nfft/2+1) rows if nfft is even, and (nfft+1)/2 rows if nfft is odd.

S = spectrogram(x,window) uses the window specified. If window is an integer, x is divided into segments equal to that integer value and a Hamming window is used. If window is a vector, x is divided into segments equal to the length of window and then the segments are windowed using the window functions specified in the window vector. For a list of available windows see "Windows".

Note To obtain the same results for the removed specgram function, specify a 'Hann' window of length 256.

S = spectrogram(x,window,noverlap) overlaps noverlap samples of each segment. noverlap must be an integer smaller than window or if window is a vector, smaller than the length of window.

S = spectrogram(x,window,noverlap,nfft) uses the nfft number of sampling points to calculate the discrete Fourier transform. nfft must be a scalar.

S = spectrogram(x,window,noverlap,nfft,fs) uses fs sampling frequency in Hz. If fs is specified as empty [], it defaults to 1 Hz.

[S,F,T] = spectrogram(x,window,noverlap,F) uses a vector F of frequencies in Hz. F must be a vector with at least two elements. This case computes the spectrogram at the frequencies in F using the Goertzel algorithm. The specified frequencies are rounded to the

nearest DFT bin commensurate with the signal's resolution. In all other syntax cases where nfft or a default for nfft is used, the short-time Fourier transform is used. The F vector returned is a vector of the rounded frequencies. T is a vector of times at which the spectrogram is computed. The length of F is equal to the number of rows of S. The length of T is equal to k, as defined above and each value corresponds to the center of each segment.

[S,F,T] = spectrogram(x,window,noverlap,F,fs) uses a vector F of frequencies in Hz as above and uses the fs sampling frequency in Hz. If fs is specified as empty [], it defaults to 1 Hz.

[S,F,T,P] = spectrogram(...) returns a matrix P containing the power spectral density (PSD) of each segment. For real x, P contains the one-sided modified periodogram estimate of the PSD of each segment. For complex x and when you specify a vector of frequencies F, P contains the two-sided PSD.

spectrogram(...,FREQLOCATION) specifies which axis to use as the frequency axis in displaying the spectrogram. Specify FREQLOCATION as a trailing string argument. Valid options are 'xaxis' or 'yaxis'. The strings are not case sensitive. If you do not specify FREQLOCATION, spectrogram uses the x-axis as the frequency axis by default.

The elements of the PSD matrix P are given by $P(i,j) = k \, | \, S(i,j) \, |^2$ where k is a real-valued scalar defined as follows

- For the one-sided PSD,

$$k = \frac{2}{Fs \sum_{n=1}^{L} | \, w(n) \, | \, 2}$$

  where $w(n)$ denotes the window function (Hamming by default) and Fs is the sampling frequency. At zero and the Nyquist frequencies, the factor of 2 in the numerator is replaced by 1.

- For the two-sided PSD,

$$k = \frac{1}{Fs\sum\limits_{n=1}^{L}|w(n)|^2}$$

at all frequencies.

- If the sampling frequency is not specified, Fs is replaced in the denominator by 2π.

spectrogram(...) plots the PSD estimate for each segment on a surface in a figure window. The plot is created using

```
surf(T,F,10*log10(abs(P)));
axis tight;
view(0,90);
```

Using spectrogram(...,'*freqloc*') syntax and adding a '*freqloc*' string (either 'xaxis' or 'yaxis') controls where the frequency axis is displayed. Using 'xaxis' displays the frequency on the *x*-axis. Using 'yaxis' displays frequency on the *y*-axis and time on the *x*-axis. The default is 'xaxis'. If you specify both a '*freqloc*' string and output arguments, '*freqloc*' is ignored.

Examples   Compute and display the PSD of each segment of a quadratic chirp, which starts at 100 Hz and crosses 200 Hz at t = 1 sec.

```
T = 0:0.001:2;
X = chirp(T,100,1,200,'q');
spectrogram(X,128,120,128,1E3);
title('Quadratic Chirp');
```
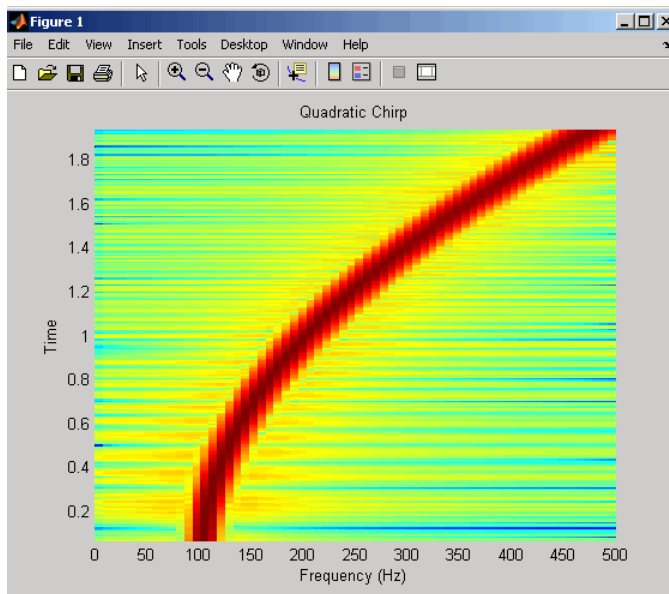
Compute and display the PSD of each segment of a linear chirp, which starts at DC and crosses 150 Hz at t = 1 sec.

```
T = 0:0.001:2;
X = chirp(T,0,1,150);
[S,F,T,P] = spectrogram(X,256,250,256,1E3);
surf(T,F,10*log10(P),'edgecolor','none'); axis tight;
view(0,90);
xlabel('Time (Seconds)'); ylabel('Hz');
```
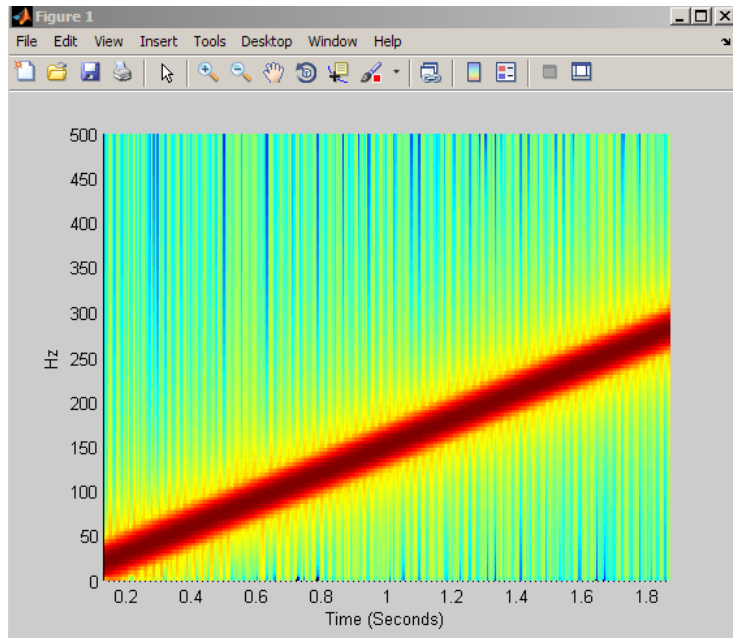
References [1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 713-718.

[2] Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

See Also goertzel | periodogram | pwelch | spectrum.periodogram | spectrum.welch

How To • "Windows"

spectrum

Purpose	Spectral estimation
Syntax	Hs = spectrum.*estmethod*(input1,...)
Description	Hs = spectrum.*estmethod*(input1,...) returns a spectral estimation object Hs of type *estmethod*. This object contains all the parameter information needed for the specified estimation method. Each estimation method takes one or more inputs, which are described on the individual reference pages.

Estimation Methods

Estimation methods for spectrum specify the type of spectral estimation method to use. Available estimation methods for spectrum are listed below.

Note You must use a spectral *estmethod* with spectrum.

Spectrum Estimation Methods

spectrum.estmethod	Description	Corresponding Function
spectrum.burg	Burg	pburg
spectrum.cov	Covariance	pcov
spectrum.eigenvector	Eigenvector	peig
spectrum.mcov	Modified covariance	pmcov
spectrum.mtm	Thompson multitaper	pmtm
spectrum.music	Multiple Signal Classification	pmusic
spectrum.periodogram	Periodogram	periodogram
spectrum.welch	Welch	pwelch
spectrum.yulear	Yule-Walker	pyulear

For more information on each estimation method, use the syntax
`help spectrum.`*`estmethod`* at the MATLAB prompt or refer to its
reference page.

Note For estimation methods that use overlap and window length
inputs, you specify the number of overlap samples as a percent overlap
and you specify the segment length instead of the window length.

For estimation methods that use windows, if the window uses an
additional parameter, a property is dynamically added to the spectrum
object for that parameter. You can change that property using `set` (see
"Changing Object Properties" on page 3-1023).

Methods

Methods provide ways of performing functions directly on your
`spectrum` object without having to specify the spectral estimation
parameters again. You can apply these methods directly on the variable
you assigned to your `spectrum` object. For more information on any of
these methods, use the syntax `help spectrum/`*`method`* at the MATLAB
prompt or refer to the table below.

Spectrum Methods

Method	Description
msspectrum	Note that the msspectrum method is only available for the periodogram and welch spectrum estimation objects.
	The mean-squared spectrum is intended for discrete spectra (from periodic, discrete-time signals). The distribution of the mean square value across frequency is the msspectrum. Unlike the power spectral density (see psd below), the peaks in the mean-square spectrum reflect the power in the signal at a given frequency. For the PSD, the power is reflected as the area in a frequency band. The units of the mean-squared spectrum are units of power.

Spectrum Methods (Continued)

Method	Description
	Hmss = msspectrum(Hs,X) returns a mean-square spectrum object containing the mean-square (power) estimate of the discrete-time signal X using the spectrum object Hs. Default for real X is the 'onesided' Nyquist frequency range and for complex X the default is the 'twosided' Nyquist frequency range.
	Hmss contains a vector of normalized frequencies W, at which the mean-square spectrum is estimated. For real signals, the range of W is [0,π] if the number of FFT points (NFFT) is even, and [0,π) if NFFT is odd. For complex signals, the range of W is [0,2π). To estimate the spectrum on a vector of specific frequencies, see FreqPoints property below.
	The msspectrum method includes these properties, which you can set using this msspectrum method or via the msspectrumopts method. These properties are listed here and described in the msspectrumopts section below:
	SpectrumType — 'onesided' or 'twosided' NormalizedFrequency – normalizes frequency between 0 and 1 Fs — sampling frequency in Hz NFFT — number of FFT points CenterDC — shifts data and frequencies to center DC component FreqPoints — 'All' or 'User Defined' FrequencyVector — frequencies at which to compute spectrum ConfLevel — confidence level to calculate the confidence interval. Value must be from 0 to 1.
	For example, Hmss = msspectrum(Hs,X,'FreqPoints','User Defined', FreqVector,fvect) returns a mean-square spectrum object where the spectrum is calculated only on the frequency points defined in the frequency vector, fvect.

Spectrum Methods (Continued)

Method	Description
	msspectrum(...) with no output arguments plots the mean-square spectrum in dB.
msspectrumopts	Hopts = msspectrumopts(Hs) returns an object that contains options for the spectrum object Hs. Hopts = msspectrumopts(Hs,X) returns an object with data-specific options and defaults. You can pass an Hopts options object as an argument to the msspectrum method. Any individual option you specify after the Hopts object overrides the value in Hopts. For example, Hmss = msspectrum(Hs,X,Hopts, 'SpectrumType', 'twosided') overrides the default SpectrumType value in Hopts. The following properties apply to both msspectrumopts and msspectrum methods. Hmss = msspectrum (..., 'SpectrumType', 'twosided') returns the two-sided mean-square spectrum. The spectrum length (NFFT) is computed over [0,2π), or if Fs is specified, [0,Fs) . Entering 'onesided' returns the one-sided mean-square spectrum, which contains the total signal power in half the Nyquist range. Default is 'onesided'. Hmss = msspectrum(Hs,X,'NormalizedFrequency',true) returns a mean-square spectrum object with frequency values normalized between 0 and 1. Default is true. Hmss = msspectrum(Hs,X,'Fs',Fs) returns a mean-square spectrum object computed as a function of frequency, where Fs is the sampling frequency in Hz. Note that you can set Fs only if NormalizedFrequency is set to false. Hmss = msspectrum(...,'NFFT',nfft) specifies the number of FFT points to use. Valid values are a positive integer, 'Nextpow2' or 'Auto'. 'Nextpow2' uses the next power of 2 greater than the

Spectrum Methods (Continued)

Method	Description
	input length or 256, whichever is greater. `'Auto'` uses the input length or 256, whichever is greater. Default is `'Nextpow2'`. Note that for `spectrum.welch`, `'Nextpow2'` and `'Auto'` are compared to the `SegmentLength` instead of the input length.
	`Hmss = msspectrum (..., 'Centerdc', true)` shifts the data and frequency values so that the DC component is at the center of the spectrum. Default is `false`.
	To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to `'User Defined'`, which replaces the NFFT property of `msspectrum` with a `FrequencyVector` property. `Hopts.FreqPoints = 'User Defined'` (Note that the default for `FreqPoints` is `'All'`, which causes `msspectrum` to use the NFFT property as described above.)
	Then, specify the frequency vector F to use. `Hopts.FrequencyVector = F` (Note that the default value for `FrequencyVector` is `'Auto'`. In this case, the number of frequency points used follows the same rule as described for NFFT `'Auto'` above.)
	`Hmms = msspectrum(...,'ConfLevel',p)` specifies the confidence level p for computing the confidence interval, which is an estimate of the error in the calculated mean-squared spectrum. The confidence level (p) is between 0 and 1. For example, `Hmss = msspectrum(Hs,X,'ConfLevel',0.95)` returns the 95% confidence interval.

Spectrum Methods (Continued)

Method	Description
psd	Note that `music` and `eigenvector` spectrum objects do not support the `psd` method. See the `pseudospectrum` method below.
	The power spectral density (PSD) is intended for continuous spectra. The integral of the PSD over a given frequency band computes the average power in the signal in that frequency band. In contrast to the `msspectrum`, the peaks in this spectra do not reflect the power at a given frequency. The units of the PSD are power per unit of frequency. See the `avgpower` method of `dspdata` for more information.
	`Hpsd = psd (Hs,X)` returns a power spectral density object containing the power spectral density estimate of the discrete-time signal X using the spectrum object Hs. The PSD is the distribution of power per unit frequency. Default for real X is `'onesided'` and for complex X is `'twosided'`.
	`Hpsd` contains a vector of normalized frequencies W, at which the PSD is estimated. For real signals, the range of W is $[0,\pi]$ if the number of FFT points (NFFT) is even, and $[0,\pi]$ if NFFT is odd. For complex signals, the range of W is $[0,2\pi]$.
	The `psd` method includes these properties, which you can set using this `psd` method or via the `psdopts` method. These properties are listed here and described in the `psdopts` section below:

	`SpectrumType` — `'onesided'` or `'twosided'`
	`NormalizedFrequency` — normalizes frequency between 0 and 1
	`Fs` — sampling frequency in Hz
	`NFFT` — number of FFT points
	`CenterDC` — shifts data and frequencies to center DC component
	`FreqPoints` — `'All'` or `'User Defined'`
	`FrequencyVector` – frequencies at which to compute spectrum
	`ConfLevel` — confidence level to calculate the confidence interval. Value must be from 0 to 1.

Spectrum Methods (Continued)

Method	Description
	For example, `Hmss = psd(Hs,X,'FreqPoints','User Defined', FreqVector,fvect)` returns a PSD object where the spectrum is calculated only on the frequency points defined in the frequency vector, `fvect`.
	`psd(...)` with no output arguments plots PSD in dB per unit frequency.
psdopts	`Hopts = psdopts(Hs)` returns an object that contains options for the spectrum object `Hs`.
	`Hopts = psdopts(Hs,X)` returns an object with data-specific options and defaults.
	You can pass an `Hopts` options object as an argument to the `psd` method. Any individual option you specify after the `Hopts` object overrides the value in `Hopts`. For example, `Hpsd = psd(Hs,X,Hopts,'SpectrumType', 'twosided')` overrides the `SpectrumType` value in `Hopts`.
	The following properties apply to both `psdmopts` and `psd` methods.
	`Hpsd = psd (Hs,X,'SpectrumType','twosided')` returns the two-sided power spectral density of `X`. The spectrum length is NFFT and is computed over $[0,2\pi]$ if `Fs` is not specified or $[0,Fs]$ if `Fs` is specified. Entering `'onesided'` returns the one-sided PSD, which contains the total signal power.
	`Hmss = psd(Hs,X,'NormalizedFrequency',true)` returns a power spectral density object with frequency values normalized between 0 and 1. Default is `true`.
	`Hpsd = psd (...,'Fs',Fs)` returns a power spectral density object computed as a function of frequency, where `Fs` is the sampling frequency in Hz.

Spectrum Methods (Continued)

Method	Description
	Hmss = psd(...,'NFFT',nfft) specifies the number of FFT points to use. Valid values are a positive integer, 'Nextpow2' or 'Auto'. 'Nextpow2' uses the next power of 2 greater than the input length or 256, whichever is greater. 'Auto' uses the input length or 256, whichever is greater. Default is 'Nextpow2'. Note that for spectrum.welch, 'Nextpow2' and 'Auto' are compared to the SegmentLength instead of the input length.
	Hmss = psd (..., 'Centerdc', true) shifts the data and frequency values so that the DC component is at the center of the spectrum. Default is false.
	To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to 'User Defined', which replaces the NFFT property of psd with a FrequencyVector property. Hopts.FreqPoints = 'User Defined' (Note that the default for FreqPoints is 'All' which causes psd to use the NFFT property as described above.)
	Hmms = psd(...,'ConfLevel',p) specifies the confidence level p for computing the confidence interval, which is an estimate of the error in the calculated PSD. The confidence level (p) is between 0 and 1. For example, Hmss = psd(Hs,X,'ConfLevel',0.95) returns the 95% confidence interval.

Spectrum Methods (Continued)

Method	Description
pseudospectrum	Note that this method is used for only music or eigenvector spectrum objects.
	Hps = pseudospectrum(Hs,X) returns an object containing the pseudospectrum estimate of the discrete-time signal X using the spectrum object Hs. Hs must be a music or eigenvector object. Default for real X is 'half' and for complex X is the 'whole' Nyquist frequency range.
	Hps contains a vector of normalized frequencies W, at which the pseudospectrum is estimated. For real signals, the range of W is $[0,\pi]$ if the number of FFT points (NFFT) is even, and $[0,\pi]$ if NFFT is odd. For complex signals, the range of W is $[0,2\pi)$.
	The pseudospectrum method includes these properties, which you can set using this pseudospectrum method or via the pseudospectrumopts method. These properties are described below: SpectrumRange — 'half' or 'whole' NormalizedFrequency — normalizes frequency between 0 and 1 Fs — sampling frequency in Hz NFFT — number of FFT points CenterDC — shifts data and frequencies to center DC component FreqPoints — 'All' or 'User Defined' FrequencyVector — frequencies at which to compute spectrum
	For example, Hmss = psd(Hs,X,'FreqPoints','User Defined', FreqVector,fvect) returns a PSD object where the spectrum is calculated only on the frequency points defined in the frequency vector, fvect.
	pseudospectrum(...) with no output arguments plots the pseudospectrum in dB.

Spectrum Methods (Continued)

Method	Description
pseudo-spectrumopts	Hopts = pseudospectrumopts(Hs) returns an object that contains options for the spectrum object Hs.
	Hopts = pseudospectrumopts(Hs,X) returns an object with data-specific options and defaults. You can pass an Hopts options object as an argument to the pseudospectrum method. Any individual option you specify after the Hopts object overrides the value in Hopts. For example, Hpseudospectrum= pseudospectrum(Hs,X, Hopts,'SpectrumRange', 'whole') overrides the SpectrumRange value in Hopts.
	Hmps = pseudospectrum (..., 'SpectrumRange', 'whole') returns the pseudospectrum over the whole Nyquist range. The spectrum length is NFFT and is computed over [0,2π) if Fs is not specified or [0,Fs) if Fs is specified. Entering 'half' returns the pseudospectrum calculated over half the Nyquist range.
	Hmss = pseudospectrum(Hs,X,'NormalizedFrequency',true) returns a pseudospectrum object with frequency values normalized between 0 and 1. Default is true.
	Hps = pseudospectrum(Hs,X,'Fs',Fs) returns a pseudospectrum object computed as a function of frequency, where Fs is the sampling frequency in Hz.
	Hps = pseudospectrum(...,'NFFT',nfft) specifies the number of FFT points to use. Valid values are a positive integer, 'Nextpow2' or 'Auto'. 'Nextpow2' uses the next power of 2 greater than the input length or 256, whichever is greater. 'Auto' uses the input length or 256, whichever is greater. Default is 'Nextpow2'.
	Hps = pseudospectrum(...,'Centerdc',true) shifts the data and frequency values so that the DC component is at the center of the spectrum. The default value is false.
	To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to 'User Defined', which replaces

Spectrum Methods (Continued)

Method	Description
	the NFFT property of `pseudospectrum` with a `FrequencyVector` property. `Hopts.FreqPoints = 'User Defined'` (Note that the default for `FreqPoints` is `'All'`, which causes `pseudospectrum` to use the NFFT property as described above.)
powerest	Note that `powerest` is available only for `music` and `eigenvector` spectrum objects. `POW = powerest(Hs,X)` returns a vector `POW` containing estimates of the powers of the complex sinusoids in `X`. The input `X` can be a vector or a matrix. If it is a matrix it can be a data matrix, where $X^{'*}X = R$ or a correlation matrix R. The value the `InputType` property of `Hs` determines how `X` is interpreted. `Hs` must be a `music` or `eigenvector` spectrum object. `[POW,W]=powerest(Hs,X)` returns `POW` and a vector `W` of the frequencies in rad/sample of the sinusoids in `X`. `[POW,F]=powerest(Hs,X,Fs)` returns `POW` and a vector `F` of the frequencies in Hz of the sinusoids in `X`. `Fs` is the sampling frequency.

Viewing Object Properties

As with any object, you can use `get` to view a `spectrum` object's properties. To see a specific property, use

```
get(Hs,'property')
```

where `'property'` is the specific property name.

To see all properties for an object, use

```
get(Hs)
```

Changing Object Properties

To set specific properties, use

```
set(Hs,'property1',value, 'property2',value,...)
```

where `'property1'`, `'property2'`, etc. are the specific property names.

To view the options for a property use `set` without specifying a value

```
set(Hs,'property')
```

Note that you must use single quotation marks around the property name. For example, to change the order of a Burg `spectrum` object `Hs` to 6, use

```
set(Hs,'order',6)
```

Another example of using `set` to change an object's properties is this example of changing the dynamically created window property of a periodogram `spectrum` object.

```
Hs=spectrum.periodogram        % Create periodogram object

Hs =

    EstimationMethod: 'Periodogram'
         WindowName: 'Rectangular'

set(Hs,'WindowName','Chebyshev')    % Change window type
Hs                                  % View changed object

Hs =

    EstimationMethod: 'Periodogram'
         WindowName: 'Chebyshev'  % Note changed property
       SidelobeAtten: 100

set(Hs,'SidelobeAtten',150)    % Change dynamic property
```

```
Hs                              % View changed object

Hs =

    EstimationMethod: 'Periodogram'
          WindowName: 'Chebyshev'
       SidelobeAtten: 150
```

All `spectrum` object properties can be changed using the `set` command, except for the `EstimationMethod` property.

Another way to change an object's properties is by using the `inspect` command which opens the Property Inspector window where you can edit any property, except dynamic properties, such as those used with windows.

```
inspect(Hs)
```

Copying an Object

To create a copy of an object, use the `copy` method.

```
H2 = copy(Hs)
```

Note Using the syntax `H2 = Hs` copies only the object handle and does not create a new object.

Examples   Define a cosine of 200 Hz, add some noise and then view its power spectral density estimate generated with the periodogram algorithm.

```
Fs = 1000;
t = 0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.periodogram;
psd(Hs,x,'Fs',Fs)
```
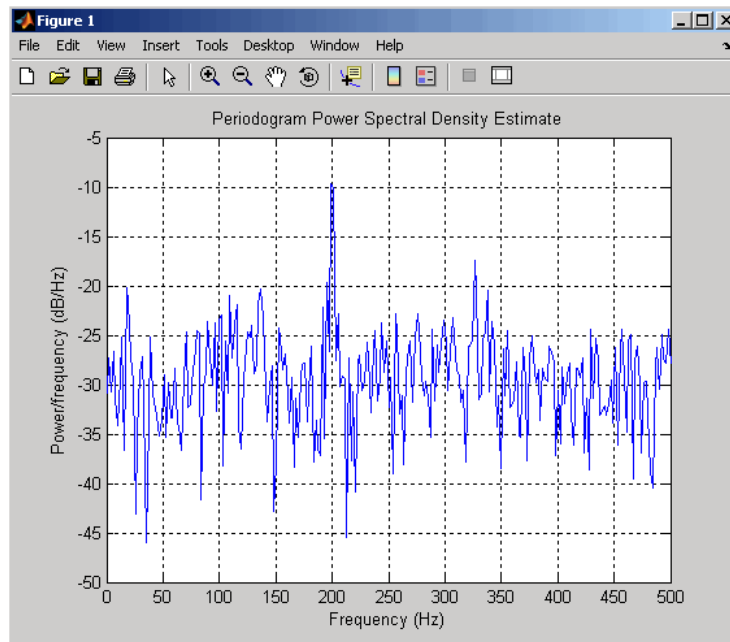
Refer to the reference pages for each estimation method for more examples.

See Also   dspdata | spectrum.burg | spectrum.cov | spectrum.mcov | spectrum.yulear | spectrum.periodogram | spectrum.welch | spectrum.mtm | spectrum.eigenvector | spectrum.music

spectrum.burg

Purpose   Burg spectrum

Syntax
```
Hs = spectrum.burg
Hs = spectrum.burg(order)
```

Description   `Hs = spectrum.burg` returns a default Burg spectrum object, `Hs`, that defines the parameters for the Burg parametric spectral estimation algorithm. The Burg algorithm estimates the spectral content by fitting an autoregressive (AR) linear prediction filter model of a given order to the signal.

`Hs = spectrum.burg(order)` returns a spectrum object, `Hs` with the specified `order`. The default value for `order` is 4.

Note See pburg for more information on the Burg algorithm.

Examples   Define a fourth order autoregressive model and view its power spectral density using the Burg algorithm.

```
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);    % 4th order AR filter
Hs=spectrum.burg;                      % 4th order AR model
psd(Hs,x,'NFFT',512)
```

See Also   dspdata | spectrum | spectrum.cov | spectrum.mcov | spectrum.yulear | spectrum.periodogram | spectrum.welch | spectrum.mtm | spectrum.eigenvector | spectrum.music

Purpose   Covariance spectrum

Syntax   Hs = spectrum.cov
                     Hs = spectrum.cov(order)

Description   Hs = spectrum.cov returns a default covariance spectrum object, Hs,
                     that defines the parameters for the covariance spectral estimation
                     algorithm. The covariance algorithm estimates the spectral content
                     by fitting an autoregressive (AR) linear prediction model of a given
                     order to the signal.

                     Hs = spectrum.cov(order) returns a spectrum object, Hs with the
                     specified order. The default value for order is 4.

                     Note See pcov for more information on the covariance algorithm.

Examples   Define a fourth order autoregressive model and view its power spectral
                     density using the covariance algorithm.

```
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);    % 4th order AR filter
Hs=spectrum.cov;                       % 4th order AR model
psd(Hs,x,'NFFT',512)
```

See Also   dspdata | spectrum | spectrum.burg | spectrum.mcov |
                     spectrum.yulear | spectrum.periodogram | spectrum.welch |
                     spectrum.mtm | spectrum.eigenvector | spectrum.music

spectrum.eigenvector

Purpose   Eigenvector spectrum

Syntax
```
Hs = spectrum.eigenvector
Hs = spectrum.eigenvector(NSinusoids)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)
```

Description   `Hs = spectrum.eigenvector` returns a default eigenvector spectrum object, `Hs`, that defines the parameters for an eigenanalysis spectral estimation method. This object uses the following default values:

Default Values

Property Name	Default Value	Description
NSinusoids	2	Number of complex sinusoids
SegmentLength	4	Length of each of the time-based segments into which the input signal is divided.
OverlapPercent	50	Percent overlap between segments

Default Values (Continued)

Property Name	Default Value	Description
WindowName	`'Rectangular'`	Window name string or `'User Defined'` (see `window` for valid window names). For more information on each window, refer to its reference page.
		This argument can also be a cell array containing the window name string or `'User Defined'` and, if used for the particular window, an optional parameter value. The syntax is {wname,wparam}.
		You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).
SubspaceThreshold	0	Threshold is the cutoff for signal and noise separation. The threshold is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (λ_{min}`*threshold`) are assigned to the noise subspace.
InputType	`'Vector'`	Type of input that will be used with this spectrum object. Valid values are `'Vector'`, `'DataMatrix'` and `'CorrelationMatrix'`.

spectrum.eigenvector

Hs = spectrum.eigenvector(NSinusoids) returns a spectrum object, Hs, with the specified number of sinusoids and default values for all other properties. Refer to the table above for default values.

Hs = spectrum.eigenvector(NSinusoids,SegmentLength) returns a spectrum object, Hs, with the specified segment length.

Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent) returns a spectrum object, Hs, with the specified overlap between segments.

Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName) returns a spectrum object, Hs, with the specified window.

Note Window names must be enclosed in single quotes, such as spectrum.eigenvector(3,32,50,'chebyshev') or spectrum.eigenvector(3,32,50,{'chebyshev',60}).

Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold) returns a spectrum object, Hs, with the specified subspace threshold.

Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType) returns a spectrum object, Hs, with the specified input type.

Note See peig for more information on the eigenanalysis algorithm.

Examples Define a complex signal with three sinusoids, add noise, and view its pseudospectrum using eigenanalysis. Set the FFT length to 128.

```
n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
Hs=spectrum.eigenvector(3,32,95,'rectangular',5);
```

```
pseudospectrum(Hs,s,'NFFT',128)
```

References   [1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978).

See Also   dspdata | spectrum | spectrum.music | spectrum.burg | spectrum.cov | spectrum.mcov | spectrum.yulear | spectrum.periodogram | spectrum.welch | spectrum.mtm

spectrum.mcov

Purpose   Modified covariance spectrum

Syntax   Hs = spectrum.mcov
                Hs = spectrum.mcov(order)

Description Hs = spectrum.mcov returns a default modified covariance spectrum
                object, Hs, that defines the parameters for the modified covariance
                spectral estimation algorithm. The modified covariance algorithm
                estimates the spectral content by fitting an autoregressive (AR) linear
                prediction filter model of a given order to the signal.

                Hs = spectrum.mcov(order) returns a spectrum object, Hs with the
                specified order. The default value for order is 4.

                Note See pmcov for more information on the modified covariance
                algorithm.

Examples   Define a fourth order autoregressive model and view its power spectral
                density using the modified covariance algorithm.

```
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);   % 4th order AR filter
Hs=spectrum.mcov;                     % 4th order AR model
psd(Hs,x,'NFFT',512)
```

See Also   dspdata | spectrum | spectrum.burg | spectrum.cov |
                spectrum.yulear | spectrum.periodogram | spectrum.welch |
                spectrum.mtm | spectrum.eigenvector | spectrum.music

Purpose   Thomson multitaper spectrum

Syntax   Hs = spectrum.mtm
                   Hs = spectrum.mtm(TimeBW)
                   Hs = spectrum.mtm(DPSS,Concentrations)
                   Hs = spectrum.mtm(...,CombineMethod)

Description   Hs = spectrum.mtm returns a default Thomson multitaper spectrum
                   object, Hs that defines the parameters for the Thomson multitaper
                   spectral estimation algorithm, which uses a linear or nonlinear
                   combination of modified periodograms. The periodograms are computed
                   using a sequence of orthogonal tapers (windows in the frequency
                   domain) specified from discrete prolate spheroidal sequences (dpss).
                   This object uses the following default values:

Property Name	Default Value	Description
TimeBW	4	Product of time and bandwidth for the discrete prolate spheroidal sequences (or Slepian sequences) used as data windows
CombineMethod	'adaptive'	Algorithm for combining the individual spectral estimates. Valid values are 'adaptive' — adaptive (nonlinear) 'unity' — unity weights (linear) 'eigenvector' — Eigenvalue weights (linear)

Hs = spectrum.mtm(TimeBW) returns a spectrum object, Hs with the
specified time-bandwidth product.

Hs = spectrum.mtm(DPSS,Concentrations) returns a spectrum
object, Hs with the specified dpss data tapers and their concentrations.

spectrum.mtm

> **Note** You can either specify the time-bandwidth product (`TimeBW`) or the `DPSS` data tapers and their `Concentrations`. See `dpss` and `pmtm` for more information.

`Hs = spectrum.mtm(...,CombineMethod)` returns a spectrum object, `Hs`, with the specified method for combining the spectral estimates. Refer to the table above for valid `CombineMethod` values.

Examples Define a cosine of 200 Hz, add noise and view its power spectral density using the Thomson multitaper algorithm with a time-bandwidth product of 3.5.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.mtm(3.5);
psd(Hs,x,'Fs',Fs)
```
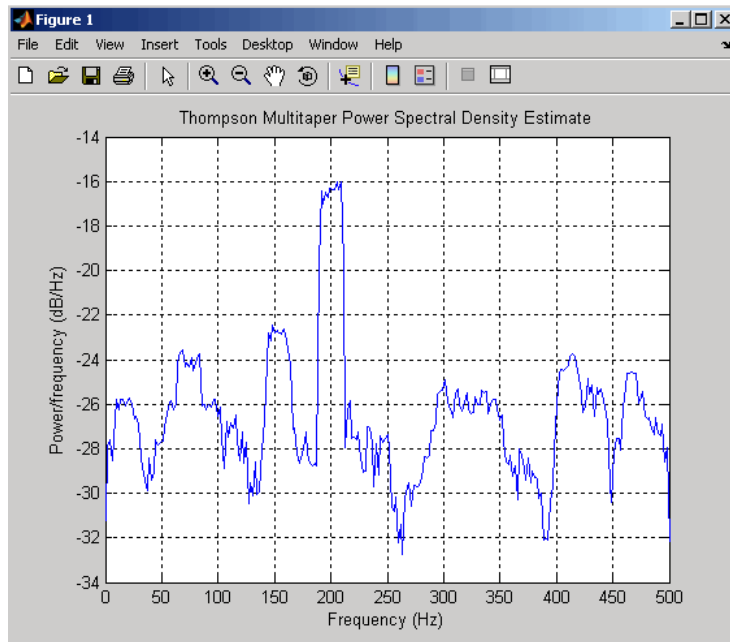
The above example could be done by specifying the data tapers and concentrations instead of the time-bandwidth product.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
[e,v]=dpss(length(x),3.5);
Hs=spectrum.mtm(e,v);
psd(Hs,x,'Fs',Fs)
```

See Also   dspdata | spectrum | spectrum.periodogram | spectrum.welch | spectrum.burg | spectrum.cov | spectrum.mcov | spectrum.yulear | spectrum.eigenvector | spectrum.music

spectrum.music

Purpose　　　　Multiple signal classification spectrum

Syntax　　　　　```
Hs = spectrum.music
Hs = spectrum.music(NSinusoids)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)
```

**Description**　　Hs = spectrum.music returns a default multiple signal classification
(MUSIC) spectrum object, Hs, that defines the parameters for the
MUSIC spectral estimation algorithm, which uses Schmidt's eigenspace
analysis algorithm. This object uses the following default values.

### Default Values

| Property Name | Default Value | Description |
|---|---|---|
| NSinusoids | 2 | Number of complex sinusoids |
| SegmentLength | 4 | Length of each of the time-based segments into which the input signal is divided. |
| OverlapPercent | 50 | Percent overlap between segments |

**Default Values (Continued)**

| Property Name | Default Value | Description |
|---|---|---|
| WindowName | 'Rectangular' | Window name string or 'User Defined' (see window for valid window names). For more information on each window, refer to its reference page). |
| | | This argument can also be a cell array containing the window name string or 'User Defined' and, if used for the particular window, an optional parameter value. The syntax is {wname,wparam}. |
| | | You can use set to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see spectrum for information on using set). |

**Default Values (Continued)**

| Property Name | Default Value | Description |
|---|---|---|
| SubspaceThreshold | O | Threshold is the cutoff for signal and noise separation. The threshold is multiplied by $\lambda_{min}$, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold ($\lambda_{min}$*threshold) are assigned to the noise subspace. |
| InputType | 'Vector' | Type of input that will be used with this spectrum object. Valid values are 'Vector', 'DataMatrix' and 'CorrelationMatrix'. |

Hs = spectrum.music(NSinusoids) returns a spectrum object, Hs, with the specified number of sinusoids and default values for all other properties. Refer to the table above for default values.

Hs = spectrum.eigenvector(NSinusoids,SegmentLength) returns a spectrum object, Hs, with the specified segment length.

Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent) returns a spectrum object, Hs, with the specified overlap between segments.

Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName) returns a spectrum object, Hs, with the specified window.

> **Note** Window names must be enclosed in single quotes,
> such as spectrum.music(3,32,50,'chebyshev') or
> spectrum.music(3,32,50,{'chebyshev',60})

Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold) returns a
spectrum object, Hs, with the specified subspace threshold.

Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)
returns a spectrum object, Hs, with the specified input type.

> **Note** See pmusic for more information on the MUSIC algorithm.

**Examples**   Define a complex signal with three sinusoids, add noise, and estimate
its pseudospectrum using the MUSIC algorithm.

```
n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
Hs=spectrum.music(3,20);
pseudospectrum(Hs,s)
```

References   [1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with
the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66
(January 1978).

See Also   dspdata | spectrum | spectrum.eigenvector | spectrum.burg
| spectrum.cov | spectrum.mcov | spectrum.yulear |
spectrum.periodogram | spectrum.welch | spectrum.mtm

spectrum.periodogram

Purpose   Periodogram spectrum

Syntax   Hs = spectrum.periodogram
                     Hs = spectrum.periodogram(winname)
                     Hs = spectrum.periodogram({winname,winparameter})

Description   Hs = spectrum.periodogram returns a default periodogram spectrum
                     object, Hs, that defines the parameters for the periodogram spectral
                     estimation method. This default object uses a rectangular window and
                     a default FFT length equal to the next power of 2 (NextPow2) that is
                     greater than the input length.

                     Hs = spectrum.periodogram(winname) returns a spectrum object,
                     Hs, that uses the specified window. If the window uses an optional
                     associated window parameter, it is set to the default value. This object
                     uses the default FFT length.

                     Hs = spectrum.periodogram({winname,winparameter}) returns
                     a spectrum object, Hs, that uses the specified window and optional
                     associated window parameter, if any. You specify the window and
                     window parameter in a cell array with a windowname string and the
                     parameter value. This object uses the default FFT length.

                     Valid windowname strings are:

                     'Bartlett'
                     'Bartlett-Hanning'
                     'Blackman'
                     'Blackman-Harris'
                     'Bohman'
                     'Chebyshev'
                     'Flat Top'
                     'Gaussian'
                     'Hamming'
                     'Hann'
                     'Kaiser'
                     'Nuttall'
                     'Parzen'

'Rectangular'
'Triangular'
'Tukey'
'User Defined'

See `window` and the corresponding window function page for window parameter information.

You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).

Note Window names must be enclosed in single quotes, such as `spectrum.periodogram('tukey')` or `spectrum.periodogram({'tukey',0.7})`.

Note See `periodogram` for more information on the periodogram algorithm.

Examples Define a cosine of 200 Hz, add noise and view its spectral content using the periodogram spectral estimation technique.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.periodogram;        % Use default values
psd(Hs,x,'Fs',Fs)
```

References   [1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978).

See Also   dspdata | spectrum | spectrum.welch | spectrum.mtm | spectrum.burg | spectrum.cov | spectrum.mcov | spectrum.yulear | spectrum.eigenvector | spectrum.music

Purpose   Welch spectrum

Syntax
```
Hs = spectrum.welch
Hs = spectrum.welch(WindowName)
Hs = spectrum.welch(WindowName,SegmentLength)
Hs = spectrum.welch(WindowName,SegmentLength,OverlapPercent)
```

Description   `Hs = spectrum.welch` returns a default Welch spectrum object, `Hs`, that defines the parameters for Welch's averaged, modified periodogram spectral estimation method. The object uses these default values.

Property Name	Default Value	Description

`{WindowName,`
 	
`winparam}`
 	
Cell array containing `WindowName` and optional window parameter
	`'Hamming'`,
	
	`SamplingFlag:`
	symmetric
		Cell array containing the window name string or `'User Defined'` and, if used for the particular window, an optional parameter value. (See `window` for valid window names and for more information on each window, refer to its reference page.)

		You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window. (See `spectrum` for information on using `set`.)
`WindowName`	`'Hamming'`, `SamplingFlag:` symmetric	Valid `windowname` strings are: `'Bartlett'`

Property Name	Default Value	Description

		`'Bartlett-Hanning'`
		`'Blackman'`
		`'Blackman-Harris'`
		`'Bohman'`
		`'Chebyshev'`
		`'Flat Top'`
		`'Gaussian'`
		`'Hamming'`
		`'Hann'`
		`'Kaiser'`
		`'Nuttall'`
		`'Parzen'`
		`'Rectangular'`
		`'Triangular'`
		`'Tukey'`
		`'User Defined'`
		
		Window names must be enclosed in single quotes, such as `spectrum.welch('tukey')` or `spectrum.welch({'tukey',0.7})`. See `window` and the corresponding window function page for window parameter information. You can use `set` to change the value of the additional window parameter or to define the MATLAB expression and parameters for a user-defined window (see

Property Name	Default Value	Description
		spectrum for information on using set).
SegmentLength	64	Length of each of the time-based segments into which the input signal is divided. A modified periodogram is computed on each segment and the average of the periodograms forms the spectral estimate. Choosing the segment length is a compromise between estimate reliability (shorter segments) and frequency resolution (longer segments). A long segment length produces better resolution while a short segment length produces more averages, and therefore a decrease in the variance.
OverlapPercent	50%	Percent overlap between segments

Hs = spectrum.welch(WindowName) returns a spectrum object, Hs, using Welch's method with the specified window and the default values for all other parameters. To specify parameters for a window, use a cell array formatted as spectrum.welch({WindowName,winparam}).

Hs = spectrum.welch(WindowName,SegmentLength) returns a spectrum object, Hs with the specified segment length.

spectrum.welch

```
Hs = spectrum.welch(WindowName,SegmentLength,OverlapPercent)
```
returns a spectrum object, Hs with the specified percentage overlap between segments.

Note See pwelch for more information on the Welch algorithm.

Examples   Define a cosine of 200 Hz, add noise and view its spectral content using the Welch algorithm.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.welch;
psd(Hs,x,'Fs',Fs)
```

The following example produces a result similar to the obsoleted
spectrum function, which used a Hann window as the default.

```
Fs = 1000;
t = 0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
window=33;
noverlap=32;
nfft=4097;
h = spectrum.welch('Hann',window,100*noverlap/window);
hpsd = psd(h,x,'NFFT',nfft,'Fs',Fs);
Pw = hpsd.Data;
Fw = hpsd.Frequencies;
```

References   [1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with
the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66
(January 1978).

See Also   dspdata | spectrum | spectrum.periodogram | spectrum.mtm |
spectrum.burg | spectrum.cov | spectrum.mcov | spectrum.yulear
| spectrum.eigenvector | spectrum.music

spectrum.yulear

Purpose   Yule-Walker spectrum object

Syntax   Hs = spectrum.yulear
                 Hs = spectrum.yulear(order)

Description Hs = spectrum.yulear returns a default Yule-Walker spectrum object,
                 Hs, that defines the parameters for the Yule-Walker spectral estimation
                 algorithm. This method is also called the auto-correlation or windowed
                 method. The Yule-Walker algorithm estimates the spectral content by
                 fitting an autoregressive (AR) linear prediction filter model of a given
                 order to the signal. This leads to a set of Yule-Walker equations, which
                 are solved using Levinson-Durbin recursion.

                 Hs = spectrum.yulear(order) returns a spectrum object, Hs, with the
                 specified order. The default value for order is 4.

                 Note See pyulear for more information on the Yule-Walker algorithm.

Examples   Define a fourth order autoregressive model and view its spectral content
                 using the Yule-Walker algorithm.

```
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);    % 4th order AR filter
Hs=spectrum.yulear;                    % 4th order AR model
psd(Hs,x,'NFFT',512)
```

See Also   dspdata | spectrum | spectrum.burg | spectrum.cov |
                 spectrum.mcov | spectrum.periodogram | spectrum.welch |
                 spectrum.mtm | spectrum.eigenvector | spectrum.music

Purpose   Open interactive digital signal processing tool

Syntax   sptool

Description   The command, sptool, opens SPTool, a suite of four tools: Signal Browser, Filter Design and Analysis Tool, FVTool, and Spectrum Viewer. These tools provide access to many of the signal, filter, and spectral analysis functions in the toolbox. When you type sptool at the command line, the SPTool suite opens.

Using SPTool, you can:

- Analyze signals listed in the **Signals** list box with the Signal Browser.

- Design or edit filters with the Filter Design and Analysis Tool (includes a Pole/Zero Editor).

- Analyze filter responses for filters listed in the **Filters** list box with FVTool.

- Apply filters in the **Filters** list box to signals in the **Signals** list box.

- Create and analyze signal spectra with the Spectrum Viewer.

- Print the Signal Browser, Filter Design and Analysis Tool, and Spectrum Viewer.

You can activate all four integrated signal processing tools from SPTool.

Signal Browser

The Signal Browser allows you to view, measure, and analyze the time-domain information of one or more signals. To activate the Signal Browser, press the **View** button under the **Signals** list box in SPTool.

See the following sections for more information on the Signal Browser:

- "Displaying Multiple Signals" on page 3-1052
- "Signal Display" on page 3-1054
- "Toolbar" on page 3-1057
- "Measurements Panels" on page 3-1060
- "Visuals — Time Domain Options" on page 3-1083
- "Style Dialog Box" on page 3-1088

sptool

Displaying Multiple Signals

Multiple Signal Input

Select more than one signal in the **Signals** list box to show multiple signals within the same display or on separate displays. By default, the signals appear as different-colored lines on the same display. The signals can have different dimensions, sample rates, and data types. Each signal can be either real or complex valued.

Multiple Displays

You can display multiple channels of data on different displays in the Scope window. In the Scope toolbar, select **View > Layout**, or select the Layout button (⊞). This feature allows you to tile the window into a number of separate displays, up to a grid of 4 rows and 4 columns. For example, if there are three inputs to the Scope, you can display the signals in separate displays by selecting row 3, column 1, as shown in the following figure.

After you select row 3, column 1, the Scope window has three separate displays, as shown in the following figure.

Signal Display

The Signal Browser uses the longest time length of all the input signals selected in the **Signals** list box for the time range. To communicate the array of times that corresponds to the current display, the scope uses the **Minimum time-axis limit**, **Time units**, and **Maximum time-axis limit** indicators on the scope window. The following figure highlights these aspects of the Signal Browser window.

- **Minimum time-axis limit** — The Signal Browser sets the minimum *time*-axis limit to 0.

- **Maximum time-axis limit** — The Signal Browser sets the maximum *time*-axis limit to the final time step of the longest input signal.

- **Time units** — The units used to describe the *time*-axis. The Signal Browser sets the time units using the value of the **Time Units** parameter on the **Main** tab of the Visuals:Time Domain Options dialog box. By default, this parameter is set to `Metric (based on Time Span)` and displays in metric units such as microseconds, milliseconds, minutes, days, etc. You can change the unit of measure to `Seconds` to always display the *time*-axis values in units of seconds. You can change it to `None` to suppress the display of units of measure on the *time*-axis. When you set this parameter to `None`, then the Signal Browser shows only the word `Time` on the *time*-axis.

  To hide both the word `Time` and the values on the *time*-axis, set the **Show time-axis labels** parameter to `None`. To hide both the word `Time` and the values on the *time*-axis in all displays except the bottom ones in each column of displays, set this parameter to `Bottom Displays Only`. This behavior differs from that of the Simulink Scope block, which always shows the values but never shows a label on the *x*-axis.

Signal Names and Legend Strings

Signal Browser uses the names of the signals in the SPTool as the strings displayed in the legends. If you change the name of any selected

signal in the **Signals** list box, its corresponding legend string in Signal Browser changes immediately. To change the name of any selected signal, from the SPTool menu, select **Edit > Name**. Signal Browser automatically updates the legend string to reflect the new signal name you entered. Similarly, if you modify any string in a legend in Signal Browser, SPTool updates the corresponding signal name in the **Signals** list box.

Axes Maximization

You can specify whether to display the Signal Browser in maximized axes mode. In this mode, the axes are expanded to fill the entire display. In each display, there is no space to show titles or axis labels. The minimum and maximum *time*-axis limits are located at the far-left and far-right edges of the display. The values at the axis tick marks appear as grid lines on top of the axes. The following figure highlights how three displays appear in maximized axes mode in the Signal Browser window.

To enable or disable this mode, in the Signal Browser menu, select **View > Properties** to bring up the Visuals:Time Domain Options

dialog box. In the **Main** pane, you can set the **Maximize axes** parameter to one of the following options:

- Auto — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.

- On — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.

- Off — In this mode, none of the axes appear maximized.

See the "Visuals — Time Domain Options" on page 3-1083 section for more information.

Toolbar
The Signal Browser toolbar contains the following buttons.

Print Button

Button	Menu Location	Shortcut Keys	Description
🖨	**File > Print**	**Ctrl+P**	Print the current scope window. To print the current scope window to a figure rather than sending it to your printer, select **File > Print to figure**.

Axes Control Buttons

⊕	**Tools > Zoom In**	N/A	When this tool is active, you can zoom in on the scope window. To do so, click in the center of your area of interest, or click and drag your cursor to draw a rectangular area of interest inside the scope window.
⋈Q⋈	**Tools > Zoom X**	N/A	When this tool is active, you can zoom in on the *x*-axis. To do so, click inside the scope window, or click and drag your cursor along the *x*-axis over your area of interest.
Ǫ̯	**Tools > Zoom Y**	N/A	When this tool is active, you can zoom in on the *y*-axis. To do so, click inside the scope window, or click and drag your cursor along the *y*-axis over your area of interest.
⤢	**Tools > Scale Axes Limits**	**Ctrl+A**	Click this button to scale the axes in the active scope window. Alternatively, you can enable automatic axes scaling by selecting one of the following options from the **Tools** menu: • **Automatically Scale Axes Limits** — When you select this option, the scope scales the axes as needed during simulation. • **Scale Axes Limits at Stop** — When you select this option, the scope scales the axes each time the simulation is stopped.

Measurements Buttons

⊞	**Tools > Measurements > Cursor Measurements**	N/A	Open or close the **Cursor Measurements** panel. This panel puts screen cursors on all the displays. See the "Cursor Measurements Panel" on page 3-1062 section for more information.
⩕	**Tools > Measurements > Signal Statistics**	N/A	Open or close the **Signal Statistics** panel. This panel displays the maximum, minimum, peak-to-peak difference, mean, median, RMS values of a selected signal, and the times at which the maximum and minimum occur. See the "Signal Statistics Panel" on page 3-1064 section for more information.
⊓ᷟ	**Tools > Measurements > Bilevel Measurements**	N/A	Open or close the **Bilevel Measurements** panel. This panel displays information about a selected signal's transitions, overshoots or undershoots, and cycles. See the "Bilevel Measurements Panel" on page 3-1066 section for more information.
⩕	**Tools > Measurements > Peak Finder**	N/A	Open or close the **Peak Finder** panel. This panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified.

			See the "Peak Finder Panel" on page 3-1080 section for more information.

Other Buttons

🔊	**Tools > Play Selected Signal**	N/A	Play an audio signal. The function soundsc is used to play the signal.
🔲	**View > Show All Legends**	N/A	Show a legend that matches each line style to a signal name in every display.
⊞	**View > Layout**	N/A	Arrange the layout of displays in the Signal Browser. This feature allows you to tile your screen into a number of separate displays, up to a grid of 4 rows and 4 columns. You may find multiple displays useful when you select multiple input signals in SPTool. The default display is 1 row and 1 column. See the "Multiple Displays" on page 3-1052 section for more information.

You can control whether this toolbar appears in the Signal Browser window. From the Signal Browser menu, select **View > Toolbar**.

Measurements Panels The Measurements panels are the five panels that appear at the right side of the Signal Browser. These panels are labeled **Trace selection**, **Cursor measurements**, **Signal statistics**, **Bilevel measurements**, and **Peak finder**.

Measurements Panel Buttons

Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

Button	Description
![top]	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
![collapse]	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide all of its panes to conserve space. After you click this button, it becomes the expand button ![expand].
![expand]	Expand the current panel. This button appears after you click the collapse button to hide the panes in the current panel. Click this button to display the panes in the current panel and show measurements again. After you click this button, it becomes the collapse button ![collapse] again.
![undock]	Undock the current panel. This button lets you move the current panel into a separate window that can be relocated anywhere on your screen. After you click this button, it becomes the dock button ![dock] in the new window.
![dock]	Dock the current panel. This button appears only after you click the undock button. Click this button to put the current panel back into the right side of the Scope window. After you click this button, it becomes the undock button ![undock] again.
![close]	Close the current panel. This button lets you remove the current panel from the right side of the Scope window.

Some panels have their measurements separated by category into a number of panes. Click the pane expand button ![expand] to show each pane that is hidden in the current panel. Click the pane collapse button ![collapse] to hide each pane that is shown in the current panel.

Trace Selection Panel

When you use the Scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click on any of the other Measurements panels. Choose the signal name

for which you would like to display time domain measurements. See the following figure.

You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools > Measurements > Trace Selection**.

Cursor Measurements Panel

The **Cursor Measurements** panel displays screen cursors. You can choose to hide or display the **Cursor Measurements** panel. In the Scope menu, select **Tools > Measurements > Cursor Measurements**. Alternatively, in the Scope toolbar, click the Cursor Measurements button.

Settings Pane

The **Settings** pane enables you to modify the type of screen cursors used to calculate time and value measurements.

- **Screen Cursors**— Shows screen cursors.

- **Horizontal**— Shows horizontal screen cursors.

- **Vertical**— Shows vertical screen cursors.

- **Waveform Cursors**— Shows cursors that attach to the input signals.

- **Lock Cursor Spacing**— Locks the time difference between the two cursors.

Measurements Pane

The **Measurements** pane shows the time and value measurements.

- **1 |**— Shows or enables you to modify the time or value at cursor number one, or both.

- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.

- **Δt**— Shows the absolute value of the difference in the times between cursor number one and cursor number two.

- **ΔV**— Shows the absolute value of the difference in signal amplitudes between cursor number one and cursor number two.

- **1/Δt**— Shows the rate, the reciprocal of the absolute value of the difference in the times between cursor number one and cursor number two.

- **ΔV/Δt**— Shows the scope, the ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times between cursors.

Signal Statistics Panel

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the times at which the maximum and minimum values occur. You can choose to hide or display the **Signal Statistics** panel. In the Scope menu, select **Tools > Measurements > Signal Statistics**. Alternatively, in the Scope toolbar, click the Signal Statistics button.

⊼ ▼ Signal Statistics		↗ ✕
	Value	Time (us)
Max	2.343	22.500
Min	-57.337 m	5.000
Peak to Peak	2.400	
Mean	1.150	
Median	1.152	
RMS	1.628	

Signal Statistics Measurements

The **Signal Statistics** panel shows statistics about the portion of the input signal within the *x*-axis and *y*-axis limits of the active display. The statistics shown are:

- **Max** — Shows the maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.

- **Min** — Shows the minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.

- **Peak to Peak** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.

- **Mean** — Shows the average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.

- **Median** — Shows the median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.

- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the Scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Bilevel Measurements Panel

The **Bilevel Measurements** panel shows information about a selected signal's transitions, overshoots or undershoots, and cycles. You can choose to hide or display the **Bilevel Measurements** panel. In the Scope menu, select **Tools > Measurements > Bilevel Measurements**. Alternatively, in the Scope toolbar, you can select the Bilevel Measurements button.

The **Bilevel Measurements** panel is separated into four panes, labeled **Settings**, **Transitions**, **Overshoots / Undershoots**, and **Cycles**. You can expand each pane to see the available options.

Settings Pane

The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level, as shown in the following figure.

- **Auto State Level** — When this check box is selected, the Bilevel measurements panel autodetects the high- and low- state levels of a bilevel waveform. For more information on the algorithm this option uses, see the Signal Processing Toolbox `statelevels` function

reference. When this check box is cleared, you may enter in values for the high- and low- state levels manually.

- **High** — Used to manually specify the value that denotes a positive polarity, or high-state level, as shown in the following figure.

Positive Polarity Pulse

- **Low** — Used to manually specify the value that denotes a negative polarity, or low-state level, as shown in the following figure.

Negative Polarity Pulse

- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. In the following figure, the mid-reference level is shown as the horizontal line, and its corresponding mid-reference level instant is shown as the vertical line.

- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, D, which you can set when you run the settlingtime function. The settling time is displayed in the **Overshoots/Undershoots** pane.

Transitions Pane

The **Transitions** pane displays calculated measurements associated with the input signal changing between its two possible state level values, high and low.

A positive-going transition, or *rising edge*, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. The following figure shows a positive-going transition.

X(t)

$$\frac{dX}{dt} > 0$$

Whenever there is a plus sign (+) next to a text label, this symbol refers to measurement associated with a rising edge, a transition from a low-state level to a high-state level.

A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. The following figure shows a negative-going transition.

Whenever there is a minus sign (–) next to a text label, this symbol refers to measurement associated with a falling edge, a transition from a high-state level to a low-state level.

The Transition measurements assume that the amplitude of the input signal is in units of volts. You must convert all input signals to volts for the Transition measurements to be valid.

- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals:Time Domain Options dialog box. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox statelevels function reference.

- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals:Time Domain Options dialog box. For

more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.

- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.

- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.

- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `risetime` function reference.

- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray in the following figure.

For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

- **– Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.

- **– Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `falltime` function reference.

- **– Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. For more information on the algorithm

this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

Overshoots/Undershoots

The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. *Overshoot* and *undershoot* refer to the amount that a signal respectively exceeds and falls below its final steady-state value. *Preshoot* refers to the amount prior to a transition that a signal varies from its initial steady-state value. This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.

The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.

- **+ Preshoot** — Average lowest aberration in the region immediately preceding each rising transition.

- **+ Overshoot** — Average highest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.

- **+ Undershoot** — Average lowest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- **+ Settling Time** — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.

You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

- – **Preshoot** — Average highest aberration in the region immediately preceding each falling transition.

- – **Overshoot** — Average highest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.

- – **Undershoot** — Average lowest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- – **Settling Time** — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can

modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

Cycles

The **Cycles** pane displays calculated measurements pertaining to repetitions or trends in the displayed portion of the input signal.

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots in the following figure.

For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulseperiod` function reference.

- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.

- **+ Pulses** — Number of positive-polarity pulses counted.

- **+ Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.

- **+ Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

- **– Pulses** — Number of negative-polarity pulses counted.

- **– Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.

- **– Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that particular rising edge. However, this feature does not apply to the **High** and **Low** measurements.

Peak Finder Panel

The **Peak Finder** panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified. You can choose to hide or display the **Peak Finder** panel. In the Scope menu, select **Tools > Measurements > Peak Finder**. Alternatively, in the Scope toolbar, select the Peak Finder button.

The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

Settings Pane

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the MINPEAKHEIGHT parameter, which you can set when you run the findpeaks function.

- **Max Num of Peaks** — The maximum number of peaks to show. This setting is equivalent to the NPEAKS parameter, which you can set when you run the findpeaks function.

- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the MINPEAKDISTANCE parameter, which you can set when you run the findpeaks function.

- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. Peak excursion is illustrated alongside peak threshold in the following figure.

Whereas the peak threshold is a minimum value necessary for a sample value to be a peak, the peak excursion is the minimum difference between a peak sample and the samples to its left and right in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than

the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the THRESHOLD parameter, which you can set when you run the findpeaks function.

- **Label peaks** — When this check box is selected, the Peak Finder displays the calculated peak values directly on the plot. By default, this check box is cleared and no values are displayed on the plot.

- **Sort by time** — When this check box is selected, the Peak Finder puts the largest calculated peak values in the time range shown in the display in time-sorted order. The Peak Finder displays the peak values in the **Peaks** pane. The **Max Num of Peaks** parameter still controls the number of peaks listed. By default, this check box is cleared and the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height.

Peaks Pane

The **Peaks** pane displays all of the largest calculated peak values and the times at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list. Using the **Sort by time** check box, you set the order in which the peaks are listed.

The numerical values displayed in the **Value** column are equivalent to the pks output argument returned when you run the findpeaks function. The numerical values displayed in the **Time** column are similar to the locs output argument returned when you run the findpeaks function.

When you use the zoom options in the Scope, the peak locations automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one peak to make the **Peak Finder** panel display information about only that particular peak.

Visuals — Time Domain Options

The Visuals:Time Domain Options dialog box controls the visual configuration settings of the Scope displays. From the Scope menu, select **View > Properties** to open this dialog box.

Main Pane

The **Main** pane of the Visuals:Time Domain Options dialog box appears as follows.

Time units

Specify the units used to describe the *time*-axis. The default setting is Metric. You can select one of the following options:

- Metric — In this mode, the Scope converts the times on the *time*-axis to some metric units such as milliseconds, microseconds, days, etc. The Scope chooses the appropriate metric units, based on the

minimum *time*-axis limit and the maximum *time*-axis limit of the scope window.

- Seconds — In this mode, the Scope always displays the units on the *time*-axis as seconds.

- None — In this mode, the Scope displays no units on the *time*-axis. The Scope shows only the word Time on the *time*-axis.

This parameter is Tunable.

Show time-axis labels

Specify how to display the time units used to describe the *time*-axis. The default setting is All. You can select one of the following options:

- All — In this mode, the *time*-axis labels appear in all displays.

- None — In this mode, the *time*-axis labels do not appear in the displays.

- Bottom Displays Only — In this mode, the *time*-axis labels appear in only the bottom row of the displays.

Tunable.

Maximize axes

Specify whether to display the Scope in maximized axes mode. In this mode, each of the axes are expanded to fit into the entire display. In each display, there is no space to show labels. Tick mark values are shown on top of the plotted data. The default setting is Auto. You can select one of the following options:

- Auto — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.

- On — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.

- Off — In this mode, none of the axes appear maximized.

Display Pane

The **Display** pane of the Visuals:Time Domain Options dialog box appears as follows.

Select display

Specify the active display as a number, where a display number corresponds to the index of the input signal. The index of a display is its column-wise placement index. The default setting is 1. Set this parameter to control the display for which you intend to modify the title, legend, grid, *y*-axis limits, *y*-axis label, and magnitude/phase display parameters. Tunable.

Title

Specify the active display title as a string. By default, the active display has no title. Tunable.

Show legend

Select this check box to show the legend in the active display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn the legend off, clear the **Show legend** check box. Tunable.

You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal named `MySignal` would have the following default names in the channel legend: `MySignal:1`, `MySignal:2`.

The Time Scope does not display signal names that were labeled within an unmasked subsystem. You must label all input signals to the Scope block that originate from an unmasked subsystem.

To change the appearance of any channel of any input signal in the Scope window, from the menu, select **View > Style**.

Show grid

When you select this check box, a grid appears in the active display of the Scope. To hide the grid, clear this check box. Tunable.

Plot signal(s) as magnitude and phase

When you select this check box, the Scope splits the active display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal is complex valued, the Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display, as shown in the following figure.

Selecting this check box and clicking the **Apply** or **OK** button changes the active display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.

This feature is particularly useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box always produces the same behavior. The phase is 0 for nonnegative input and 180 degrees for negative input. Tunable.

Minimum Y-limit

Specify the minimum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of –180 degrees.

Maximum Y-limit

Specify the maximum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

Y-axis label

Specify the text for the scope to display to the left of the *y*-axis. Tunable.

This parameter becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that parameter, the *y*-axis label always appears as Magnitude on the top axes and Phase on the bottom axes.

Style Dialog Box

In the **Style** dialog box, you can customize the style of displays. You can change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. From the Signal Browser menu, select **View > Style** to open this dialog box.

Properties

The **Style** dialog box allows you to modify the following properties of the Signal Browser:

Figure color

Specify the color that you want to apply to the background of the Signal Browser. By default, the figure color is gray.

Plot type

Specify the type of plot to use. The default setting is Line. Valid values for **Plot type** are:

- Line — Displays input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB line or plot function.

- Stairs — Displays input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB stairs function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.

This parameter is Tunable.

Select display

Specify the active display as a number, where a display number corresponds to the index of the input signal. The number of a display corresponds to its column-wise placement index. The default setting is 1. Set this parameter to control which display should have its axes colors, line properties, marker properties, and visibility changed. Tunable.

Axes colors

Specify the color that you want to apply to the background of the axes for the active display.

Properties for line

Specify the signal for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected signal on the active display should be visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected signal on the active display.

Marker

Specify marks for the selected signal on the active display to show at data points. This parameter is similar to the Marker property for the MATLAB Handle Graphics plot objects. You can choose any of the marker symbols from the following table.

Specifier	Marker Type
none	No marker (default)
○	Circle
□	Square
×	Cross
•	Point
+	Plus sign
✳	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

Filter Design and Analysis Tool

The Filter Design and Analysis Tool fdatool allows you to design and edit FIR and IIR filters. To launch fdatool, press either the **New** button or the **Edit** button under the **Filters** list box in SPTool.

Note When you open FDATool from SPTool, a reduced version of FDATool that is compatible with SPTool opens.

The Filter Design and Analysis Tool has a Pole/Zero Editor you can access by selecting the icon in the left column of FDATool.

Filter Visualization Tool

The Filter Visualization Tool (fvtool) allows you to view the characteristics of a designed or imported filter, including its magnitude response, phase response, group delay, phase delay, pole-zero plot, impulse response, and step response. To activate FVTool, click the **View** button under the **Filters** list box in SPTool.

Spectrum Viewer

The Spectrum Viewer allows you to analyze frequency-domain data graphically using a variety of methods of spectral density estimation, including the Burg method, the FFT method, the multitaper method, the MUSIC eigenvector method, Welch's method, and the Yule-Walker autoregressive method. To activate the Spectrum Viewer:

• Click the **Create** button under the **Spectra** list box to compute the power spectral density for a signal selected in the **Signals** list box in SPTool. You may need to click **Apply** to view the spectra.

- Click the **View** button to analyze spectra selected under the **Spectra** list box in SPTool.

- Click the **Update** button under the **Spectra** list box in SPTool to modify a selected power spectral density signal.

In addition, you can right-click in any plot display area to modify signal properties.

See Also   fdatool | findpeaks | fvtool

Purpose   Square wave

Syntax   x = square(t)
               x = square(t,duty)

Description   x = square(t) generates a square wave with period 2π for the
                 elements of time vector t. square(t) is similar to sin(t), but creates a
                 square wave with peaks of ±1 instead of a sine wave.

                 x = square(t,duty) generates a square wave with specified duty
                 cycle, duty, which is a number between 0 and 100. The *duty cycle* is the
                 percent of the period in which the signal is positive.

See Also   chirp | cos | diric | gauspuls | pulstran | rectpuls | sawtooth |
               sin | tripuls

ss2sos

Purpose　　Convert digital filter state-space parameters to second-order sections form

Syntax
```
[sos,g] = ss2sos(A,B,C,D)
[sos,g] = ss2sos(A,B,C,D,iu)
[sos,g] = ss2sos(A,B,C,D,'order')
[sos,g] = ss2sos(A,B,C,D,iu,'order')
[sos,g] = ss2sos(A,B,C,D,iu,'order','scale'')
sos = ss2sos(...)
```

Description　ss2sos converts a state-space representation of a given digital filter to an equivalent second-order section representation.

[sos,g] = ss2sos(A,B,C,D) finds a matrix sos in second-order section form with gain g that is equivalent to the state-space system represented by input arguments A, B, C, and D. The input system must be single output and real. sos is an *L*-by-6 matrix

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

$$H(z) = g \prod_{k=1}^{L} H_k(z) = g \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

[sos,g] = ss2sos(A,B,C,D,iu) specifies a scalar iu that determines which input of the state-space system A, B, C, D is used in the conversion. The default for iu is 1.

[sos,g] = ss2sos(A,B,C,D,'order') and

[sos,g] = ss2sos(A,B,C,D,iu,'order') specify the order of the rows in sos, where 'order' is

- 'down', to order the sections so the first row of sos contains the poles closest to the unit circle

- 'up', to order the sections so the first row of sos contains the poles farthest from the unit circle (default)

The zeros are always paired with the poles closest to them.

[sos,g] = ss2sos(A,B,C,D,iu,'*order*','*scale*'') specifies the desired scaling of the gain and the numerator coefficients of all second-order sections, where '*scale*' is

- 'none', to apply no scaling (default)

- 'inf', to apply infinity-norm scaling

- 'two', to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

Note Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

---

sos = ss2sos(...) embeds the overall system gain, g, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

Note Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use ss2sos with two outputs.

---

ss2sos

Examples

Find a second-order section form of a Butterworth lowpass filter:

```
[A,B,C,D] = butter(5,0.2);
sos = ss2sos(A,B,C,D)
sos =
    0.0013    0.0013         0    1.0000   -0.5095         0
    1.0000    2.0008    1.0008    1.0000   -1.0966    0.3554
    1.0000    1.9979    0.9979    1.0000   -1.3693    0.6926
```

Algorithms

ss2sos uses a four-step algorithm to determine the second-order section representation for an input state-space system:

1 It finds the poles and zeros of the system given by A, B, C, and D.

2 It uses the function zp2sos, which first groups the zeros and poles into complex conjugate pairs using the cplxpair function. zp2sos then forms the second-order sections by matching the pole and zero pairs according to the following rules:

   a Match the poles closest to the unit circle with the zeros closest to those poles.

   b Match the poles next closest to the unit circle with the zeros closest to those poles.

   c Continue until all of the poles and zeros are matched.

   ss2sos groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.

3 It orders the sections according to the proximity of the pole pairs to the unit circle. ss2sos normally orders the sections with poles closest to the unit circle last in the cascade. You can tell ss2sos to order the sections in the reverse order by specifying the 'down' flag.

4 ss2sos scales the sections by the norm specified in the '*scale*' argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\| H \|_p = \left[\frac{1}{2\pi} \int\limits_{0}^{2\pi} | H(\omega) |^p \ d\omega \right]^{\frac{1}{p}}$$

where p can be either ∞ or 2. See the references for details. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

Diagnostics   If there is more than one input to the system, ss2sos gives the following error message:

```
State-space system must have only one input.
```

References   [1] Jackson, L.B., *Digital Filters and Signal Processing, 3rd ed.*, Kluwer Academic Publishers, Boston, 1996. Chapter 11.

[2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998. Chapter 9.

[3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

See Also   cplxpair | sos2ss | ss2tf | ss2zp | tf2sos | zp2sos

ss2tf

Purpose   Convert state-space filter parameters to transfer function form

Syntax   `[b,a] = ss2tf(A,B,C,D,iu)`

Description   `ss2tf` converts a state-space representation of a given system to an equivalent transfer function representation.

`[b,a] = ss2tf(A,B,C,D,iu)` returns the transfer function

$$H(s) = \frac{B(s)}{A(s)} = C(sI - A)^{-1}B + D$$

of the system

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

from the `iu`-th input. Vector `a` contains the coefficients of the denominator in descending powers of *s*. The numerator coefficients are returned in array `b` with as many rows as there are outputs *y*. `ss2tf` also works with systems in discrete time, in which case it returns the *z*-transform representation.

The `ss2tf` function is part of the standard MATLAB language.

Algorithms   The `ss2tf` function uses `poly` to find the characteristic polynomial det(*sI*–*A*) and the equality:

$$H(s) = C(sI - A)^{-1}B = \frac{\det(sI - A + BC) - \det(sI - A)}{\det(sI - A)}$$

See Also   `latc2tf` | `sos2tf` | `ss2sos` | `ss2zp` | `tf2ss` | `zp2tf`

Purpose   Convert state-space filter parameters to zero-pole-gain form

Syntax   `[z,p,k] = ss2zp(A,B,C,D,i)`

Description   ss2zp converts a state-space representation of a given system to an equivalent zero-pole-gain representation. The zeros, poles, and gains of state-space systems represent the transfer function in factored form.

`[z,p,k] = ss2zp(A,B,C,D,i)` calculates the transfer function in factored form

$$H(s) - \frac{Z(s)}{P(s)} - k\frac{(s-z_1)(s-z_2)\cdots(s-z_n)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

of the continuous-time system

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

from the ith input (using the ith columns of B and D). The column vector p contains the pole locations of the denominator coefficients of the transfer function. The matrix z contains the numerator zeros in its columns, with as many columns as there are outputs *y* (rows in C). The column vector k contains the gains for each numerator transfer function.

ss2zp also works for discrete time systems. The input state-space system must be real.

The ss2zp function is part of the standard MATLAB language.

Examples   Here are two ways of finding the zeros, poles, and gains of a discrete-time transfer function:

$$H(z) = \frac{2 + 3z^{-1}}{1 + 0.4z^{-1} + z^{-2}}$$

```
b = [2 3 0];
```

```
a = [1 0.4 1];
[z,p,k] = tf2zp(b,a)
z =
     0.0000
    -1.5000
p =
    -0.2000 + 0.9798i
    -0.2000 - 0.9798i
k =
     2
[A,B,C,D] = tf2ss(b,a);
[z,p,k] = ss2zp(A,B,C,D,1)
z =
     0.0000
    -1.5000
p =
    -0.2000 + 0.9798i
    -0.2000 - 0.9798i
k =
     2
```

Algorithms   ss2zp finds the poles from the eigenvalues of the A array. The zeros are
the finite solutions to a generalized eigenvalue problem:

```
z = eig([A B;C D], diag([ones(1,n) 0]));
```

In many situations this algorithm produces spurious large, but finite,
zeros. ss2zp interprets these large zeros as infinite.

ss2zp finds the gains by solving for the first nonzero Markov
parameters.

References   [1] Laub, A.J., and B.C. Moore, "Calculation of Transmission Zeros
Using QZ Techniques," *Automatica 14* (1978), p. 557.

See Also   sos2zp | ss2sos | ss2tf | tf2zp | tf2zpk | zp2ss

Purpose   State-level estimation for bilevel waveform with histogram method

Syntax
```
LEVELS = statelevels(X)
LEVELS = statelevels(X,NBINS)
LEVELS = statelevels(X,NBINS,METHOD)
[LEVELS,HISTOGRAM] = statelevels(...)
[LEVELS,HISTOGRAM,BINLEVELS] = statelevels(...)
statelevels(...)
```

Description   LEVELS = statelevels(X) estimates the low- and high-state levels in the bilevel waveform, X, using the histogram method. See "Algorithms" on page 3-1108.

LEVELS = statelevels(X,NBINS) specifies the number of bins to use in the histogram as a positive scalar. If unspecified, NBINS defaults to 100.

LEVELS = statelevels(X,NBINS,METHOD) estimates state levels using the mean or mode of the subhistograms. Valid entries for METHOD are 'mean' or 'mode'. METHOD defaults to 'mode'. See "Algorithms" on page 3-1108.

[LEVELS,HISTOGRAM] = statelevels(...) returns the histogram, HISTOGRAM, of the values in X.

[LEVELS,HISTOGRAM,BINLEVELS] = statelevels(...) returns the centers of the histogram bins.

statelevels(...) displays a plot of the signal and the corresponding computed histogram.

Input Arguments

X

Bilevel waveform. X is a real-valued row or column vector.

NBINS

Number of histogram bins

Default: 100

METHOD

State-level estimation method in the subhistograms. METHOD is a string indicating the statistic to use for the estimation of the low- and high-state levels. Valid entries for METHOD are `'mode'` or `'mean'`. See "Algorithms" on page 3-1108.

> **Default:** `'mode'`

Output Arguments

LEVELS

Levels of low and high states. LEVELS is a 1-by-2 row vector of state levels estimated by the histogram method. The first element of LEVELS is the low-state level. The second element of LEVELS is the high-state level.

HISTOGRAM

Histogram counts (frequencies). HISTOGRAM is a column vector with NBINS elements containing the number of data values in each histogram bin.

BINLEVELS

Histogram bin centers. BINLEVELS is a column vector containing the bin centers for the histogram counts in HISTOGRAM.

Definitions

State

A particular level, which can be associated with an upper- and lower-state boundary. States are ordered from the most negative to the most positive. In a bilevel waveform, the most negative state is the low state. The most positive state is the high state.

State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small

number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \tfrac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

State Levels ± 2% Tolerance

Examples　**Display State Levels and Subhistograms**

Estimate the low- and high-state levels of 2.3 V underdamped clock data. Plot the data with the estimated state levels and subhistograms.

```
load('clockex.mat', 'x');
statelevels(x);
```

State Levels with 100 Bins and Modes of Subhistograms

Estimate the low and high-state levels of 2.3 V underdamped clock data sampled at 4 MHz.

Use the default number of bins and modes of the subhistograms to estimate the state levels. Plot the clock data with the lines indicating the estimated low and high-state levels.

```
load('clockex.mat', 'x', 't');
LEVELS = statelevels(x);
plot(t,x);
hold on;
plot(t,LEVELS(1).*ones(length(x)),'r--');
plot(t,LEVELS(2).*ones(length(x)),'r--');
```

State Levels Using Means of Subhistograms

Estimate the low and high-state levels of 2.3 V underdamped clock data sampled at 4 MHz.

Use the default number of bins and means of the subhistograms to estimate the state levels. Plot the clock data with the lines indicating the estimated low and high-state levels.

```
load('clockex.mat', 'x', 't');
LEVELS = statelevels(x,1e3,'mean');
```

Histogram Counts and Histogram Bin Centers

Estimate the low- and high-state levels of 2.3 V underdamped clock data sampled at 4 MHz. Return the histogram counts and histogram bin centers used in the histogram method.

```
load('clockex.mat', 'x', 't');
[LEVELS,HISTOGRAM,BINLEVELS] = statelevels(x);
```

statelevels

Algorithms statelevels uses the histogram method to estimate the states of a bilevel waveform. The histogram method is described in [1]. To summarize the method:

1 Determine the maximum and minimum amplitudes and amplitude range of the data.

2 For the specified number of histogram bins, determine the bin width as the ratio of the amplitude range to the number of bins.

3 Sort the data values into the histogram bins.

4 Identify the lowest-indexed histogram bin, i_{low}, and highest-indexed histogram bin, i_{high}, with nonzero counts.

5 Divide the histogram into two subhistograms.

The indices of the lower histogram bins are $i_{low} \leq i \leq 1/2(i_{high} - i_{low})$.

The indices of the upper histogram bins are $i_{low} + 1/2(i_{high} - i_{low}) \leq i \leq i_{high}$.

6 Compute the state levels by determining the mode or mean of the lower and upper histograms.

References [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

See Also midcross | overshoot | risetime | undershoot

Purpose   Step response of digital filter

Syntax
```
[h,t] = stepz(b,a)
[h,t] = stepz(b,a,n)
[h,t] = stepz(b,a,n,fs)
stepz(b,a)
stepz(Hd)
```

Description   [h,t] = stepz(b,a) computes the step response of the filter with numerator coefficients b and denominator coefficients a. stepz chooses the number of samples and returns the response in the column vector h and sample times in the column vector t (where t = [0:n-1]', and n = length(t) is computed automatically).

[h,t] = stepz(b,a,n) computes the first n samples of the step response when n is an integer (t = [0:n-1]'). I

[h,t] = stepz(b,a,n,fs) computes n samples and produces a vector t of length n so that the samples are spaced 1/fs units apart. fs is assumed to be in Hz.

stepz(b,a) with no output arguments plots the step response in the current figure window.

stepz(Hd) plots the step response of the filter and displays the plot in fvtool. The input Hd is a dfilt filter object or an array of dfilt filter objects. Note that if you have DSP System Toolbox product installed and are using a dfilt object with fixed-point properties, its filter internals are not used when calculating the step response.

stepz works for both real and complex input systems.

Examples   **Example 1**

Plot the step response of a Butterworth filter:

```
[b,a] = butter(3,.4);
stepz(b,a)
```

The same example using a dfilt object and displaying the result in the Filter Visualization Tool (fvtool) is

```
[b,a] = butter(3,.4);
Hd=dfilt.df1(b,a);
stepz(Hd)
```

Figure No. 1: Filter Visualization Tool - Step Response

Step Response

Example 2

Plot the first 50 samples of the step response of a fourth-order lowpass elliptic filter with cutoff frequency of 0.4 times the Nyquist frequency:

```
[b,a] = ellip(4,0.5,20,0.4);
stepz(b,a,50)
```

The same example using a dfilt object and displaying the result in the Filter Visualization Tool (fvtool) is

```
[b,a] = ellip(4,0.5,20,0.4);
Hd=dfilt.df1(b,a);
stepz(Hd,50)
```

Algorithms   stepz filters a length n step sequence using

```
filter(b,a,ones(1,n))
```

and plots the results using stem.

To compute n in the auto-length case, stepz either uses n = length(b) for the FIR case or first finds the poles using p = roots(a), if length(a) is greater than 1.

If the filter is unstable, n is chosen to be the point at which the term from the largest pole reaches 10^6 times its original value.

If the filter is stable, n is chosen to be the point at which the term due to the largest amplitude pole is 5*10^-5 of its original amplitude.

If the filter is oscillatory (poles on the unit circle only), `stepz` computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms, n is chosen to equal five periods of the slowest oscillation or the point at which the term due to the largest (nonunity) amplitude pole is `5*10^-5` of its original amplitude, whichever is greater.

`stepz` also allows for delays in the numerator polynomial. The number of delays is incorporated into the computation for the number of samples.

See Also   `freqz | grpdelay | impz | phasez | zplane`

Purpose   Compute linear model using Steiglitz-McBride iteration

Syntax   ```
[b,a] = stmcb(h,nb,na)
[b,a] = stmcb(y,x,nb,na)
[b,a] = stmcb(h,nb,na,niter)
[b,a] = stmcb(y,x,nb,na,niter)
[b,a] = stmcb(h,nb,na,niter,ai)
[b,a] = stmcb(y,x,nb,na,niter,ai)
```

Description Steiglitz-McBride iteration is an algorithm for finding an IIR filter with
a prescribed time domain impulse response. It has applications in both
filter design and system identification (parametric modeling).

[b,a] = stmcb(h,nb,na) finds the coefficients b and a of the system
$b(z)/a(z)$ with approximate impulse response h, exactly nb zeros, and
exactly na poles.

[b,a] = stmcb(y,x,nb,na) finds the system coefficients b and a of
the system that, given x as input, has y as output. x and y must be
the same length.

[b,a] = stmcb(h,nb,na,niter) and

[b,a] = stmcb(y,x,nb,na,niter) use niter iterations. The default
for niter is 5.

[b,a] = stmcb(h,nb,na,niter,ai) and

[b,a] = stmcb(y,x,nb,na,niter,ai) use the vector ai as the initial
estimate of the denominator coefficients. If ai is not specified, stmcb
uses the output argument from [b,ai] = prony(h,0,na) as the vector
ai.

stmcb returns the IIR filter coefficients in length nb+1 and na+1 row
vectors b and a. The filter coefficients are ordered in descending powers
of z.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb}}{a(1) + a(2)z^{-1} + \cdots + a(na+1)z^{-na}}$$

stmcb

Examples   Approximate the impulse response of a Butterworth filter with a system of lower order:

```
[b,a] = butter(6,0.2);
h = filter(b,a,[1 zeros(1,100)]);
freqz(b,a,128)
```
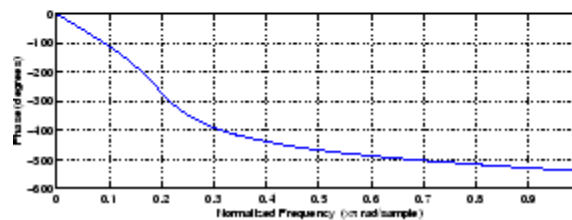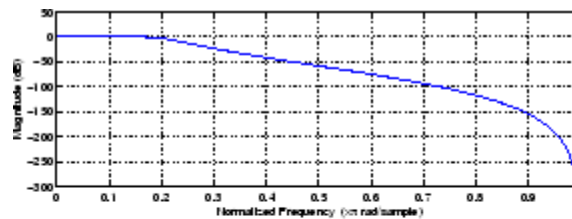


```
[bb,aa] = stmcb(h,4,4);
freqz(bb,aa,128)
```
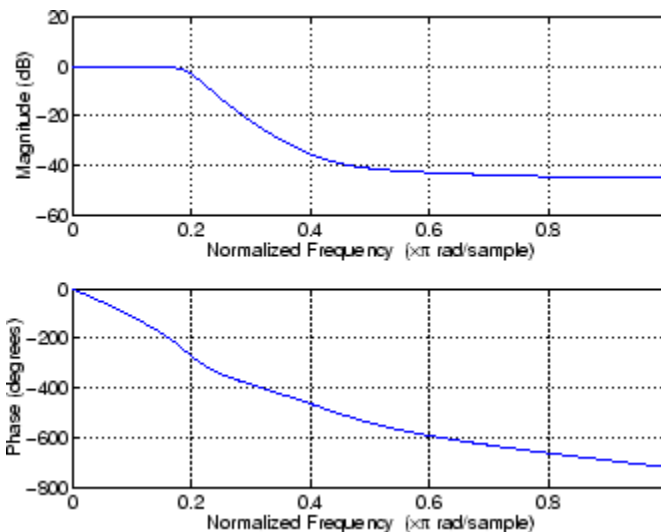
Algorithms   stmcb attempts to minimize the squared error between the impulse response *h* of *b(z)/a(z)* and the input signal *x*.

$$\min_{a,b} \sum_{i=0}^{\infty} |x(i) - h(i)|^2$$

stmcb iterates using two steps:

1 It prefilters h and x using $1/a(z)$.

2 It solves a system of linear equations for b and a using \.

stmcb repeats this process niter times. No checking is done to see if the b and a coefficients have converged in fewer than niter iterations.

Diagnostics   If x and y have different lengths, stmcb produces this error message,

```
Input signal X and output signal Y must
have the same length.
```

stmcb

References   [1] Steiglitz, K., and L.E. McBride, "A Technique for the Identification of Linear Systems," *IEEE Trans. Automatic Control, Vol. AC-10* (1965), pp. 461-464.

[2] Ljung, L., *System Identification: Theory for the Use*r, Prentice-Hall, Englewood Cliffs, NJ, 1987, p. 297.

See Also   levinson | lpc | aryule | prony

Purpose   Strip plot

Syntax   strips(x)
                   strips(x,n)
                   strips(x,sd,fs)
                   strips(x,sd,fs,scale)

Description   strips(x) plots vector x in horizontal strips of length 250. If x is
                   a matrix, strips(x) plots each column of x. The left-most column
                   (column 1) is the top horizontal strip.

                   strips(x,n) plots vector x in strips that are each n samples long.

                   strips(x,sd,fs) plots vector x in strips of duration sd seconds, given
                   a sampling frequency of fs samples per second.

                   strips(x,sd,fs,scale) scales the vertical axes.

                   If x is a matrix, strips(x,n), strips(x,sd,fs), and
                   strips(x,sd,fs,scale) plot the different columns of x on the same
                   strip plot.

                   strips ignores the imaginary part of complex-valued x.

Examples   Plot two seconds of a frequency modulated sinusoid in 0.25 second strips:

```
fs = 1000;                          % Sampling frequency
t = 0:1/fs:2;                       % Time vector
x = vco(sin(2*pi*t),[10 490],fs);   % FM waveform
strips(x,0.25,fs)
```

strips

See Also plot | stem

Purpose   Taylor window

Syntax   w = taylorwin(n)
                   w = taylorwin(n,nbar)
                   w = taylorwin(n,nbar,sll)

Description   Taylor windows are similar to Chebyshev windows. While a Chebyshev
                   window has the narrowest possible mainlobe for a specified sidelobe
                   level, a Taylor window allows you to make tradeoffs between the
                   mainlobe width and sidelobe level. The Taylor distribution avoids edge
                   discontinuities, so Taylor window sidelobes decrease monotonically.
                   Taylor window coefficients are not normalized. Taylor windows are
                   typically used in radar applications, such as weighting synthetic
                   aperature radar images and antenna design.

                   w = taylorwin(n) returns an n-point Taylor window in a column vector
                   w. The values in this vector are the window weights or coefficients.
                   n must be a positive integer. The default value for the number of
                   approximately equal height sidelobes (nbar) is 4 and for the maximum
                   sidelobe level (sll) is -30.

                   w = taylorwin(n,nbar) returns an n-point Taylor window with
                   nbar nearly constant-level sidelobes adjacent to the mainlobe. These
                   sidelobes are "nearly constant-level" because some decay occurs in the
                   transition region. nbar must be a positive integer.

                   w = taylorwin(n,nbar,sll) returns an n-point Taylor window with
                   a maximum sidelobe level of sll dB relative to the mainlobe peak.sll
                   must be a negative value, such as -30, which produces sidelobes with
                   peaks 30 dB down from the mainlobe peak.

Examples   Generate a 64-point Taylor window with four nearly constant-level
                   sidelobes and a peak sidelobe level of -35 dB relative to the mainlobe
                   peak.

                   ```
                   w = taylorwin(64,4,-35);
                   wvtool(w);
                   ```
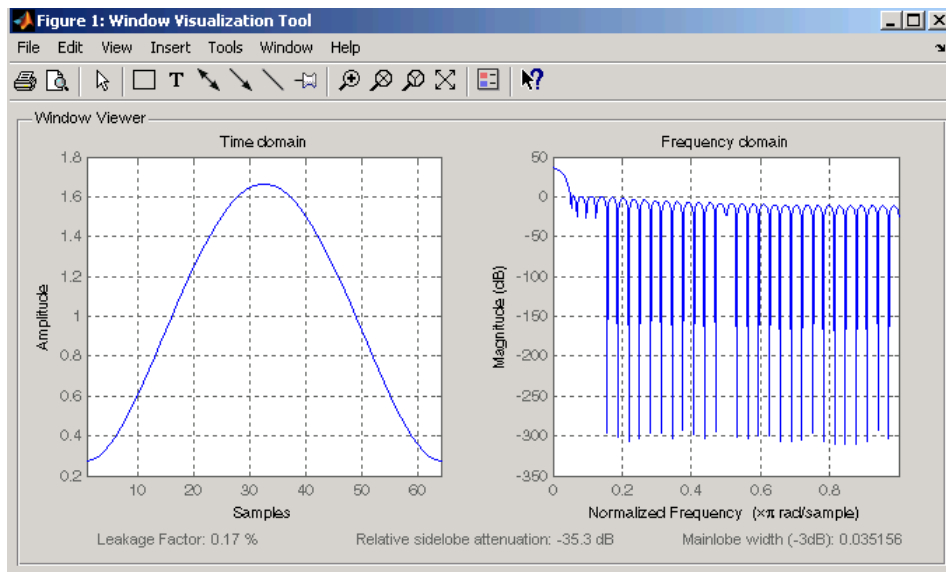
References   [1] Carrara, W.G., R.M. Majewski and R.S. Goodman, *Spotlight Synthetic Aperature Radar: Signal Processing Algorithms*, Artech House Publishers, Boston, 1995, Appendix D.2.

[2] Brookner, Eli, *Practical Phased Array Antenna Systems*, Lex Book, Lexington, MA, 1991.

Purpose   Convert transfer function filter parameters to lattice filter form

Syntax   ```
                     [k,v] = tf2latc(b,a)
                     k = tf2latc(1,a)
                     [k,v] = tf2latc(1,a)
                     k = tf2latc(b)
                     k = tf2latc(b, 'phase')
                     ```

**Description**      `[k,v] = tf2latc(b,a)` finds the lattice parameters k and the ladder
                     parameters v for an IIR (ARMA) lattice-ladder filter, normalized
                     by a(1). Note that an error is generated if one or more of the lattice
                     parameters are exactly equal to 1.

                     `k = tf2latc(1,a)` finds the lattice parameters k for an IIR all-pole
                     (AR) lattice filter.

                     `[k,v] = tf2latc(1,a)` returns the scalar ladder coefficient at the
                     correct position in vector v. All other elements of v are zero.

                     `k = tf2latc(b)` finds the lattice parameters k for an FIR (MA) lattice
                     filter, normalized by b(1).

                     `k = tf2latc(b, 'phase')` specifies the type of FIR (MA) lattice filter,
                     where '*phase*' is

                     • 'max', for a maximum phase filter.

                     • 'min', for a minimum phase filter.

**See Also**         latc2tf | latcfilt | tf2sos | tf2ss | tf2zp | tf2zpk

# tf2sos

**Purpose**        Convert digital filter transfer function data to second-order sections form

**Syntax**
```
[sos,g] = tf2sos(b,a)
[sos,g] = tf2sos(b,a,'order')
[sos,g] = tf2sos(b,a,'order','scale')
sos = tf2sos(...)
```

**Description**    tf2sos converts a transfer function representation of a given digital filter to an equivalent second-order section representation.

[sos,g] = tf2sos(b,a) finds a matrix sos in second-order section form with gain g that is equivalent to the digital filter represented by transfer function coefficient vectors a and b.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \cdots + b_{n+1} z^{-n}}{a_1 + a_2 z^{-1} + \cdots + a_{m+1} z^{-m}}$$

sos is an *L*-by-6 matrix

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients $b_{ik}$ and $a_{ik}$ of the second-order sections of $H(z)$.

$$H(z) = g \prod_{k=1}^{L} H_k(z) = g \prod_{k=1}^{L} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}}$$

[sos,g] = tf2sos(b,a,'order') specifies the order of the rows in sos, where 'order' is

- 'down', to order the sections so the first row of sos contains the poles closest to the unit circle

• 'up', to order the sections so the first row of sos contains the poles farthest from the unit circle (default)

[sos,g] = tf2sos(b,a,'*order*','*scale*') specifies the desired scaling of the gain and numerator coefficients of all second-order sections, where '*scale*' is:

• 'none', to apply no scaling (default)

• 'inf', to apply infinity-norm scaling

• 'two', to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

---

**Note** Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

---

sos = tf2sos(...) embeds the overall system gain, g, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

---

**Note** Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use ss2sos with two outputs.

---

**Algorithms**    tf2sos uses a four-step algorithm to determine the second-order section representation for an input transfer function system:

**1** It finds the poles and zeros of the system given by b and a.

**2** It uses the function `zp2sos`, which first groups the zeros and poles into complex conjugate pairs using the `cplxpair` function. `zp2sos` then forms the second-order sections by matching the pole and zero pairs according to the following rules:

**a** Match the poles closest to the unit circle with the zeros closest to those poles.

**b** Match the poles next closest to the unit circle with the zeros closest to those poles.

**c** Continue until all of the poles and zeros are matched.

`tf2sos` groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.

**3** It orders the sections according to the proximity of the pole pairs to the unit circle. `tf2sos` normally orders the sections with poles closest to the unit circle last in the cascade. You can tell `tf2sos` to order the sections in the reverse order by specifying the `'down'` flag.

**4** `tf2sos` scales the sections by the norm specified in the `'scale'` argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\| H \|_p = \left[ \frac{1}{2\pi} \int\limits_0^{2\pi} |H(\omega)|^p \, d\omega \right]^{\frac{1}{p}}$$

where $p$ can be either $\infty$ or 2. See the references for details on the scaling. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

**References**

[1] Jackson, L.B., *Digital Filters and Signal Processing, 3rd ed.*, Kluwer Academic Publishers, Boston, 1996, Chapter 11.

[2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998, Chapter 9.

[3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

**See Also**    cplxpair | sos2tf | ss2sos | tf2ss | tf2zp | tf2zpk | zp2sos

# tf2ss

**Purpose**        Convert transfer function filter parameters to state-space form

**Syntax**        `[A,B,C,D] = tf2ss(b,a)`

**Description**   `tf2ss` converts the parameters of a transfer function representation of a given system to those of an equivalent state-space representation.

`[A,B,C,D] = tf2ss(b,a)` returns the A, B, C, and D matrices of a state space representation for the single-input transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \cdots + b_{n-1}s + b_n}{a_1 s^{m-1} + \cdots + a_{m-1}s + a_m} = C(sI - A)^{-1}B + D$$

in controller canonical form

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

The input vector a contains the denominator coefficients in descending powers of *s*. The rows of the matrix b contain the vectors of numerator coefficients (each row corresponds to an output). In the discrete-time case, you must supply b and a to correspond to the numerator and denominator polynomials with coefficients in descending powers of *z*.

For discrete-time systems you must make b have the same number of columns as the length of a. You can do this by padding each numerator represented in b (and possibly the denominator represented in the vector a) with trailing zeros. You can use the function `eqtflength` to accomplish this if b and a are vectors of unequal lengths.

The `tf2ss` function is part of the standard MATLAB language.

**Examples**      Consider the system:

$$H(s) = \frac{\left[ \dfrac{2s+3}{s^2+2s+1} \right]}{s^2+0.4s+1}$$

To convert this system to state-space, type

```
b = [0 2 3; 1 2 1];
a = [1 0.4 1];
[A,B,C,D] = tf2ss(b,a)
A =
   -0.4000    -1.0000
    1.0000         0
B =
     1
     0
C =
    2.0000     3.0000
    1.6000         0
D =
     0
     1
```

Note There is disagreement in the literature on naming conventions for the canonical forms. It is easy, however, to generate similarity transformations that convert these results to other forms.

See Also   sos2ss | ss2tf | tf2sos | tf2zp | tf2zpk | zp2ss

tf2zp

Purpose   Convert transfer function filter parameters to zero-pole-gain form

Syntax   `[z,p,k] = tf2zp(b,a)`

Description `tf2zp` finds the zeros, poles, and gains of a continuous-time transfer function.

> **Note** You should use `tf2zp` when working with positive powers ($s^2 + s + 1$), such as in continuous-time transfer functions. A similar function, `tf2zpk`, is more useful when working with transfer functions expressed in inverse powers ($1 + z^{-1} + z^{-2}$), which is how transfer functions are usually expressed in DSP.

`[z,p,k] = tf2zp(b,a)` finds the matrix of zeros `z`, the vector of poles `p`, and the associated vector of gains `k` from the transfer function parameters `b` and `a`:

- The numerator polynomials are represented as columns of the matrix `b`.

- The denominator polynomial is represented in the vector `a`.

Given a SIMO continuous-time system in polynomial transfer function form

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \cdots + b_{n-1}s + b_n}{a_1 s^{m-1} + \cdots + a_{m-1}s + a_m}$$

you can use the output of `tf2zp` to produce the single-input, multioutput (SIMO) factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

The following describes the input and output arguments for `tf2zp`:

- The vector a specifies the coefficients of the denominator polynomial $A(s)$ (or $A(z)$) in descending powers of s (z^{-1}).

- The ith row of the matrix b represents the coefficients of the ith numerator polynomial (the ith row of $B(s)$ or $B(z)$). Specify as many rows of b as there are outputs.

- For continuous-time systems, choose the number nb of columns of b to be less than or equal to the length na of the vector a.

- For discrete-time systems, choose the number nb of columns of b to be equal to the length na of the vector a. You can use the function eqtflength to provide equal length vectors in the case that b and a are vectors of unequal lengths. Otherwise, pad the numerators in the matrix b (and, possibly, the denominator vector a) with zeros.

- The zero locations are returned in the columns of the matrix z, with as many columns as there are rows in b.

- The pole locations are returned in the column vector p and the gains for each numerator transfer function in the vector k.

The tf2zp function is part of the standard MATLAB language.

Examples   Find the zeros, poles, and gains of this continuous-time system:

$$H(s) = \frac{2s^2 + 3s}{s^2 + 0.4s + 1}$$

```
b = [2 3];
a = [1 0.4 1];
[b,a] = eqtflength(b,a);      % Make lengths equal
[z,p,k] = tf2zp(b,a)          % Obtain zero-pole-gain form
z =
          0
    -1.5000
p =
   -0.2000 + 0.9798i
   -0.2000 - 0.9798i
```

```
k =
   2
```

See Also   sos2zp | ss2zp | tf2sos | tf2ss | tf2zpk | zp2tf

Purpose   Convert transfer function filter parameters to zero-pole-gain form

Syntax   `[z,p,k] = tf2zpk(b,a)`

Description   `tf2zpk` finds the zeros, poles, and gains of a discrete-time transfer function.

Note You should use `tf2zpk` when working with transfer functions expressed in inverse powers $(1 + z^{-1} + z^{-2})$, which is how transfer functions are usually expressed in DSP. A similar function, `tf2zp`, is more useful for working with positive powers $(s^2 + s + 1)$, such as in continuous-time transfer functions.

`[z,p,k] = tf2zpk(b,a)` finds the matrix of zeros `z`, the vector of poles `p`, and the associated vector of gains `k` from the transfer function parameters `b` and `a`:

- The numerator polynomials are represented as columns of the matrix `b`.

- The denominator polynomial is represented in the vector `a`.

Given a single-input, multiple output (SIMO) discrete-time system in polynomial transfer function form

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} \cdots + b_{n-1} z^{-n} + b_n z^{-n-1}}{a_1 + a_2 z^{-1} \cdots + a_{m-1} z^{-m} + a_m z^{-m-1}}$$

you can use the output of `tf2zpk` to produce the single-input, multioutput (SIMO) factored transfer function form

$$H(z) = \frac{Z(z)}{P(z)} = k \frac{(z - z_1)(z - z_2) \cdots (z - z_m)}{(z - p_1)(z - p_2) \cdots (z - p_n)}$$

The following describes the input and output arguments for `tf2zpk`:

- The vector a specifies the coefficients of the denominator polynomial $A(z)$ in descending powers of z.

- The ith row of the matrix b represents the coefficients of the ith numerator polynomial (the ith row of $B(s)$ or $B(z)$). Specify as many rows of b as there are outputs.

- The zero locations are returned in the columns of the matrix z, with as many columns as there are rows in b.

- The pole locations are returned in the column vector p and the gains for each numerator transfer function in the vector k.

Examples   Find the poles, zeros, and gain of a Butterworth filter:

```
[b,a] = butter(3,.4);
[z,p,k] = tf2zpk(b,a)
z =
  -1.0000
  -1.0000 + 0.0000i
  -1.0000 - 0.0000i

p =
   0.2094 + 0.5582i
   0.2094 - 0.5582i
   0.1584

k =
    0.0985
```

See Also   sos2zp | ss2zp | tf2sos | tf2ss | tf2zp | zp2tf

Purpose   Transfer function estimate

Syntax   Txy = tfestimate(x,y)
                   Txy = tfestimate(x,y,window)
                   Txy = tfestimate(x,y,window,noverlap)
                   [Txy,W] = tfestimate(x,y,window,noverlap,nfft)
                   [Txy,F] = tfestimate(x,y,window,noverlap,nfft,fs)
                   [...] = tfestimate(x,y,...,'twosided')
                   tfestimate(...)

Description   Txy = tfestimate(x,y) finds a transfer function estimate Txy given
                   input signal vector x and output signal vector y. Vectors x and y must
                   be the same length. The relationship between the input x and output
                   y is modeled by the linear, time-invariant transfer function Txy. The
                   transfer function is the quotient of the cross power spectral density
                   (*Pyx*) of x and y and the power spectral density (*Pxx*) of x.

$$T_{xy}(f) = \frac{P_{yx}(f)}{P_{xx}(f)}$$

If x is real, tfestimate estimates the transfer function at positive
frequencies only; in this case, the output Txy is a column vector of
length nfft/2+1 for nfft even and (nfft+1)/2 for nfft odd. If x or y is
complex, tfestimate estimates the transfer function for both positive
and negative frequencies and Txy has length nfft.

tfestimate uses the following default values:

Default Values

Parameter	Description	Default Value
nfft	FFT length which determines the frequencies at which the PSD is estimated For real x and y, the length of Txy is (nfft/2+1) if nfft is even or (nfft+1)/2 if nfft is odd. For complex x or y, the length of Txy is nfft.	Maximum of 256 or the next power of 2 greater than the length of each section of x or y
fs	Sampling frequency	1
window	Windowing function and number of samples to use to section x and y	Periodic Hamming window with length equal to the signal segment length that results from dividing the signal x into eight sections and then applying the default or specified overlap.
noverlap	Number of samples by which the sections overlap	Value to obtain 50% overlap

Note You can use the empty matrix [] to specify the default value for any input argument except x or y. For example, Txy = tfestimate(x,y,[],[],128) uses a Hamming window with default length, as described above, default noverlap to obtain 50% overlap, and the specified 128 nfft.

Txy = tfestimate(x,y,window) specifies a windowing function, divides x and y into overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for window, Txy uses a Hamming window of that length. The length of the window must be less than or equal to nfft. If the length of the window exceeds nfft, tfestimate zero pads the sections. To replicate the output of the obsoleted tfe function, specify 'hanning(nfft)' as the window.

Txy = tfestimate(x,y,window,noverlap) overlaps the sections of x by noverlap samples. noverlap must be an integer smaller than the length of window.

[Txy,W] = tfestimate(x,y,window,noverlap,nfft) uses the specified FFT length nfft in estimating the PSD and CPSD estimates for the transfer function. It also returns W, which is the vector of normalized frequencies (inrad/sample) at which the tfestimate is estimated. For real signals, the range of W is [0, π] when nfft is even and [0, π) when nfft is odd. For complex signals, the range of W is [0, 2π).

[Txy,F] = tfestimate(x,y,window,noverlap,nfft,fs) returns Txy as a function of frequency and a vector F of frequencies at which tfestimate estimates the transfer function. fs is the sampling frequency in Hz. F is the same size as Txy, so plot(f,Txy) plots the transfer function estimate versus properly scaled frequency. For real signals, the range of F is [0, fs/2] when nfft is even and [0, fs/2] when nfft is odd. For complex signals, the range of F is [0, fs).

[...] = tfestimate(x,y,...,'twosided') returns a transfer function estimate with frequencies that range over the entire interval from 0 to the sampling frequency, [0,Fs]. Specifying 'onesided' uses from 0 to the Nyquist frequency.

tfestimate(...) with no output arguments plots the transfer function estimate in the current figure window.

Examples Compute and plot the transfer function estimate between two colored noise sequences x and y:

tfestimate

```
h = fir1(30,0.2,rectwin(31));
x = randn(16384,1);
y = filter(h,1,x);
tfestimate(x,y,1024,[],[],512)
```


Algorithms   tfestimate uses Welch's averaged periodogram method. See pwelch for details.

See Also   cpsd | mscohere | periodogram | pwelch | spectrum

Purpose   Triangular window

Syntax   triang(L)

Description   triang(L) returns an L-point triangular window in the column vector w. The coefficients of a triangular window are:

For L odd:

$$w(n) = \begin{cases} \dfrac{2n}{L+1} & 1 \le n \le (L+1)/2 \\[2mm] 2 - \dfrac{2n}{L+1} & (L+1)/2 + 1 \le n \le L \end{cases}$$

For L even:

$$w(n) = \begin{cases} \dfrac{(2n-1)}{L} & 1 \le n \le L/2 \\[2mm] 2 - \dfrac{(2n-1)}{L} & L/2 + 1 \le n \le L \end{cases}$$

The triangular window is very similar to a Bartlett window. The Bartlett window always ends with zeros at samples 1 and L, while the triangular window is nonzero at those points. For L odd, the center L-2 points of triang(L-2) are equivalent to bartlett(L).

Examples   Create a 200-point triangular window and plot the result using WVTool.

```
L = 200;
wvtool(triang(L))
```

triang

References   [1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 447-448.

See Also   barthannwin | bartlett | blackmanharris | bohmanwin | nuttallwin | parzenwin | rectwin | window | wintool | wvtool

tripuls

Purpose   Sampled aperiodic triangle

Syntax   y = tripuls(T)
                   y = tripuls(T,w)
                   y = tripuls(T,w,s)

Description   y = tripuls(T) returns a continuous, aperiodic, symmetric,
                   unity-height triangular pulse at the times indicated in array T, centered
                   about T=0 and with a default width of 1.

                   y = tripuls(T,w) generates a triangular pulse of width w.

                   y = tripuls(T,w,s) generates a triangular pulse with skew s, where
                   -1 < s < 1. When s is 0, a symmetric triangular pulse is generated.

Examples   Create a triangular pulse with width 0.4.

```
fs = 10000;
t = -1:1/fs:1;
w = .4;
x = tripuls(t,w);
figure,plot(t,x)
xlabel('Time (sec)');ylabel('Amplitude');
title('Triangular Aperiodic Pulse')
```

tripuls

See Also   chirp | cos | diric | gauspuls | pulstran | rectpuls | sawtooth | sin | square | tripuls

Purpose   Tukey (tapered cosine) window

Syntax   w = tukeywin(L,r)

Description   w = tukeywin(L,r) returns an L-point Tukey window in the column
vector w. A Tukey window is a rectangular window with the first
and last r/2 percent of the samples equal to parts of a cosine. See
"Definitions" on page 3-1144 for the equation of the Tukey window. r is
a real number between 0 and 1. If you input $r \leq 0$, you obtain a rectwin
window. If you input $r \geq 1$, you obtain a hann window. r defaults to 0.5.

Examples   Compute 128-point Tukey windows with five different values for r and
display the results using WVTool.

```
L=128;
t0=tukeywin(L,0);          % Equivalent to rectangular window
t25=tukeywin(L,0.25);
t5=tukeywin(L);            % r=0.5
t75=tukeywin(L,0.75);
t1=tukeywin(L,1);          % Equivalent to Hann window
wvtool(t0,t25,t5,t75,t1)
```

tukeywin

Definitions The following equation defines the *L*–point Tukey window:

$$
w(x) = \begin{cases} \dfrac{1}{2}\{1 + \cos(\frac{2\pi}{r}[x - r/2])\} & 0 \le x < \dfrac{r}{2} \\[2ex] 1 & \dfrac{r}{2} \le x < 1 - \dfrac{r}{2} \\[2ex] \dfrac{1}{2}\{1 + \cos(\frac{2\pi}{r}[x - 1 + r/2])\} & 1 - \dfrac{r}{2} \le x \le 1 \end{cases}
$$

where *x* is a *L*–point linearly spaced vector generated using `linspace`. The parameter *r* is the ratio of cosine-tapered section length to the entire window length with $0 \le r \le 1$. For example, setting *r* =0.5 produces a Tukey window where 1/2 of the entire window length consists of segments of a phase-shifted cosine with period *2r =1*. If you specify

r≤0, an *L*-point rectangular window is returned. If you specify r≥1, an *L*-point von Hann window (hann) is returned.

References   [1] Bloomfield P. *Fourier Analysis of Time Series: An Introduction*, New York: Wiley-Interscience, 2000, p.69.

See Also   chebwin | gausswin | kaiser | window | wintool | wvtool

udecode

Purpose Decode 2^n-level quantized integer inputs to floating-point outputs

Syntax
y = udecode(u,n)
y = udecode(u,n,v)
y = udecode(u,n,v,'*SaturateMode*')

Description y = udecode(u,n) inverts the operation of uencode and reconstructs quantized floating-point values from an encoded multidimensional array of integers u. The input argument n must be an integer between 2 and 32. The integer n specifies that there are 2^n quantization levels for the inputs, so that entries in u must be either:

- Signed integers in the range $[-2^n/2, (2^n/2) - 1]$

- Unsigned integers in the range $[0, 2^n-1]$

Inputs can be real or complex values of any integer data type (uint8, uint16, uint32, int8, int16, int32). Overflows (entries in u outside of the ranges specified above) are saturated to the endpoints of the range interval. The output y has the same dimensions as u. Its entries have values in the range [-1,1].

y = udecode(u,n,v) decodes u such that the output y has values in the range [-v,v], where the default value for v is 1.

y = udecode(u,n,v,'*SaturateMode*') decodes u and treats input overflows (entries in u outside of [-v,v]) according to the string '*saturatemode*', which can be one of the following:

- 'saturate': Saturate overflows. This is the default method for treating overflows.

    - Entries in signed inputs u whose values are outside of the range $[-2^n/2, (2^n/2) - 1]$ are assigned the value determined by the closest endpoint of this interval.

    - Entries in unsigned inputs u whose values are outside of the range $[0, 2^n-1]$ are assigned the value determined by the closest endpoint of this interval.

- 'wrap': Wrap all overflows according to the following:

- Entries in signed inputs u whose values are outside of the range $[-2^n/2, (2^n/2) - 1]$ are wrapped back into that range using modulo 2^n arithmetic (calculated using $u = \mathtt{mod(u+2\char94n/2,2\char94n)-(2\char94n/2)}$).

- Entries in unsigned inputs u whose values are outside of the range $[0, 2^n-1]$ are wrapped back into the required range before decoding using modulo 2^n arithmetic (calculated using $u = \mathtt{mod(u,2\char94n)}$).

Examples

```
% Create signed 8-bit integer string
u = int8([-1 1 2 -5]);
% Decode with 3 bits
ysat = udecode(u,3)
ysat =
   -0.2500    0.2500    0.5000   -1.0000
```

Notice the last entry in u saturates to 1, the default peak input magnitude. Change the peak input magnitude:

```
ysatv = udecode(u,3,6)    % Set peak input magnitude to 6
ysatv =
   -1.5000    1.5000    3.0000   -6.0000
```

The last input entry still saturates. Try wrapping the overflows:

```
ywrap = udecode(u,3,6,'wrap')
ywrap =
   -1.5000    1.5000    3.0000    4.5000
```

Try adding more quantization levels:

```
yprec = udecode(u,5)
yprec =
   -0.0625    0.0625    0.1250   -0.3125
```

Algorithms

The algorithm adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701. Integer input values are uniquely mapped (decoded) from one of 2^n uniformly spaced integer values to quantized floating-point values in the range [-v,v]. The smallest

integer input value allowed is mapped to -v and the largest integer input value allowed is mapped to v. Values outside of the allowable input range are either saturated or wrapped, according to specification.

The real and imaginary components of complex inputs are decoded independently.

References   [1] *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993.

See Also   uencode

Purpose   Quantize and encode floating-point inputs to integer outputs

Syntax
```
y = uencode(u,n)
y = uencode(u,n,v)
y = uencode(u,n,v,'SignFlag')
```

Description   y = uencode(u,n) quantizes the entries in a multidimensional array of floating-point numbers u and encodes them as integers using 2^n-level quantization. n must be an integer between 2 and 32 (inclusive). Inputs can be real or complex, double- or single-precision. The output y and the input u are arrays of the same size. The elements of the output y are unsigned integers with magnitudes in the range $[0, 2^n$-1]. Elements of the input u outside of the range [-1,1] are treated as overflows and are saturated.

- For entries in the input u that are less than -1, the value of the output of uencode is 0.

- For entries in the input u that are greater than 1, the value of the output of uencode is 2^n-1.

y = uencode(u,n,v) allows the input u to have entries with floating-point values in the range [-v,v] before saturating them (the default value for v is 1). Elements of the input u outside of the range [-v,v] are treated as overflows and are saturated:

- For input entries less than -v, the value of the output of uencode is 0.

- For input entries greater than v, the value of the output of uencode is 2^n-1.

y = uencode(u,n,v,'SignFlag') maps entries in a multidimensional array of floating-point numbers u whose entries have values in the range [-v,v] to an integer output y. Input entries outside this range are saturated. The integer type of the output depends on the string 'SignFlag' and the number of quantization levels 2^n. The string 'SignFlag' can be one of the following:

- 'signed': Outputs are signed integers with magnitudes in the range $[-2^n/2, (2^n/2) - 1]$.

uencode

- `'unsigned'` (default): Outputs are unsigned integers with magnitudes in the range $[0, 2^n\text{-}1]$.

The output data types are optimized for the number of bits as shown in the table below.

n	Unsigned Integer	Signed Integer
2 to 8	uint8	int8
9 to 16	uint16	int16
17 to 32	uint32	int32

Examples Map floating-point scalars in [-1, 1] to `uint8` (unsigned) integers, and produce a staircase plot. Note that the horizontal axis plots from -1 to 1 and the vertical axis plots from 0 to 7 (2^3-1):

```
u = [-1:0.01:1];
y = uencode(u,3);
plot(u,y,'.')
```


Now look at saturation effects when you under specify the peak value for the input:

```
u = [-2:0.5:2];
y = uencode(u,5,1)
y =
     0     0     0     8    16    24    31    31    31
```

Now look at the output for

```
u = [-2:0.5:2];
y = uencode(u,5,2,'signed')
y =
   -16   -12    -8    -4     0     4     8    12    15
```

Algorithms uencode maps the floating-point input value to an integer value determined by the requirement for 2^n levels of quantization. This encoding adheres to the definition for uniform encoding specified in ITU-T Recommendation G.701. The input range [-v,v] is divided into 2^n evenly spaced intervals. Input entries in the range [-v,v] are first quantized according to this subdivision of the input range, and then mapped to one of 2^n integers. The range of the output depends on whether or not you specify that you want signed integers.

References [1] *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Term*s, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993.

See Also udecode

unshiftdata

Purpose   Inverse of `shiftdata`

Syntax   `y = unshiftdata(x,perm,nshifts)`

Description `y = unshiftdata(x,perm,nshifts)` restores the orientation of the data that was shifted with `shiftdata`. The permutation vector is given by `perm`, and `nshifts` is the number of shifts that was returned from `shiftdata`.

`unshiftdata` is meant to be used in tandem with `shiftdata`. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

Examples   ### Example 1

This example shifts `x`, a `3`-`by`-`3` magic square, permuting dimension `2` to the first column. `unshiftdata` shifts `x` back to its original shape.

1. Create a `3`-`by`-`3` magic square:

```
x = fi(magic(3))

x =

     8     1     6
     3     5     7
     4     9     2

```

2. Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

This command returns the permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`:

```
x =
```

```
    8    3    4
    1    5    9
    6    7    2


perm =

    2    1


nshifts =

    []
```

3. Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

    8    1    6
    3    5    7
    4    9    2
```

Example 2

This example shows how shiftdata and unshiftdata work when you define dim as empty.

1. Define x as a row vector:

```
x = 1:5

x =

    1    2    3    4    5
```

2. Define dim as empty to shift the first non-singleton dimension of x to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])
```

This command returns x as a column vector, along with perm, the permutation vector, and nshifts, the number of shifts:

```
x =

    1
    2
    3
    4
    5


perm =

    []


nshifts =

    1
```

3. Using unshiftdata, restore x to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

    1    2    3    4    5
```

See Also   ipermute | shiftdata | shiftdim

upfirdn

Purpose   Upsample, apply FIR filter, and downsample

Syntax   *yout* = upfirdn(*xin*,*h*)
                   yout = upfirdn(*xin*,*h*,*p*)
                   yout = upfirdn(*xin*,*h*,*p*,*q*)

Description   upfirdn performs a cascade of three operations:

1 Upsampling the input data in the matrix xin by a factor of the
integer p (inserting zeros)

2 FIR filtering the upsampled signal data with the impulse response
sequence given in the vector or matrix h

3 Downsampling the result by a factor of the integer q (throwing away
samples)

upfirdn has been implemented as a MEX-file for maximum speed,
so only the outputs actually needed are computed. The FIR filter is
usually a lowpass filter, which you must design using another function
such as firpm or fir1.

Note The function resample performs an FIR design using firls,
followed by rate changing implemented with upfirdn.

---

yout = upfirdn(*xin*,*h*) filters the input signal xin with the FIR filter
having impulse response h. If xin is a row or column vector, then it
represents a single signal. If xin is a matrix, then each column is
filtered independently. If h is a row or column vector, then it represents
one FIR filter. If h is a matrix, then each column is a separate FIR
impulse response sequence. If yout is a row or column vector, then
it represents one signal. If yout is a matrix, then each column is a
separate output. No upsampling or downsampling is implemented with
this syntax.

upfirdn

yout = upfirdn(*xin,h,p*) specifies the integer upsampling factor p, where p has a default value of 1.

yout = upfirdn(*xin,h,p,q*) specifies the integer downsampling factor q, where q has a default value of 1. The length of the output, *yout*, is ceil(((length(xin)-1)*p+length(h))/q)

Note Since upfirdn performs convolution and rate changing, the yout signals have a different length than xin. The number of rows of yout is approximately p/q times the number of rows of xin.

Tips   Usually the inputs xin and the filter h are vectors, in which case only one output signal is produced. However, when these arguments are arrays, each column is treated as a separate signal or filter. Valid combinations are:

1 xin is a vector and h is a vector.

There is one filter and one signal, so the function convolves xin with h. The output signal yout is a row vector if xin is a row; otherwise, yout is a column vector.

2 xin is a matrix and h is a vector.

There is one filter and many signals, so the function convolves h with each column of xin. The resulting yout will be an matrix with the same number of columns as xin.

3 xin is a vector and h is a matrix.

There are many filters and one signal, so the function convolves each column of h with xin. The resulting yout will be an matrix with the same number of columns as h.

4 xin is a matrix and h is a matrix, both with the same number of columns.

There are many filters and many signals, so the function convolves corresponding columns of xin and h. The resulting yout is an matrix with the same number of columns as xin and h.

Examples   Change the sampling rate by a factor of 147/160. This factor is used to convert from 48kHz (DAT rate) to 44.1kHz (CD sampling rate).

```
L = 147; M = 160;     % Interpolation/decimation factors.
N = 24*L;
h = fir1(N-1,1/M,kaiser(N,7.8562));
h = L*h; % Passband gain = L
Fs = 48e3;            % Original sampling frequency-48kHz
n = 0:10239;          % 10240 samples, 0.213 seconds long
x = sin(2*pi*1e3/Fs*n); % Original signal, sinusoid @ 1kHz
y = upfirdn(x,h,L,M);   % 9430 samples, still .213 seconds

% Overlay original (48kHz) with resampled
% signal (44.1kHz) in red.

stem(n(1:49)/Fs,x(1:49)); hold on
stem(n(1:45)/(Fs*L/M),y(13:57),'r','filled');
xlabel('Time (sec)');ylabel('Signal value');
```

upfirdn

Algorithms　　upfirdn uses a polyphase interpolation structure. The number of multiply-add operations in the polyphase structure is approximately $(L_h L_x - p L_x)/q$ where L_h and L_x are the lengths of $h[n]$ and $x[n]$, respectively.

A more accurate flops count is computed in the program, but the actual count is still approximate. For long signals $x[n]$, the formula is often exact.

Diagnostics　　If p and q are large and do not have many common factors, you may see this message:

```
Filter length is too large - reduce problem complexity.
```

Instead, you should use an interpolation function, such as interp1, to perform the resampling and then filter the input.

References　　[1] Crochiere, R.E., and L.R. Rabiner, *Multi-Rate Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1983, pp. 88-91.

[2] Crochiere, R.E., "A General Program to Perform Sampling Rate Conversion of Data by Rational Ratios," *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, pp. 8.2-1 to 8.2-7.

See Also   conv | decimate | downsample | filter | interp | intfilt | resample | upsample

upsample

Purpose	Increase sampling rate by integer factor

Syntax

```
y = upsample(x,n)
y = upsample(x,n,phase)
```

Description   y = upsample(x,n) increases the sampling rate of x by inserting n-1 zeros between samples. x can be a vector or a matrix. If x is a matrix, each column is considered a separate sequence. The upsampled y has x*n samples.

y = upsample(x,n,phase) specifies the number of samples by which to offset the upsampled sequence. phase must be an integer from 0 to n-1.

Examples   Increase the sampling rate of a sequence by 3:

```
x = [1 2 3 4];
y = upsample(x,3);
x,y
x =
    1   2   3   4
y =
    1   0   0   2   0   0   3   0   0   4   0   0
```

Increase the sampling rate of the sequence by 3 and add a phase offset of 2:

```
x = [1 2 3 4];
y = upsample(x,3,2);
x,y
x =
    1   2   3   4
y =
    0   0   1   0   0   2   0   0   3   0   0   4
```

Increase the sampling rate of a matrix by 3:

```
x = [1 2; 3 4; 5 6;];
y = upsample(x,3);
```

```
x,y
x =
     1     2
     3     4
     5     6
y =
     1     2
     0     0
     0     0
     3     4
     0     0
     0     0
     5     6
     0     0
     0     0
```

See Also   decimate | downsample | interp | interp1 | resample | spline | upfirdn

undershoot

Purpose　　　Undershoot metrics of bilevel waveform transitions

Syntax　　　　US = undershoot(X)
US = undershoot(X,FS)
US = undershoot(X,T)
[US,USLEV,USINST] = undershoot(...)
[...] = undershoot(...,Name,Value)
undershoot(...)

Description　　US = undershoot(X) returns the greatest deviations below the
final state levels of each transition in the bilevel waveform, X. The
undershoots, US, are expressed as a percentage of the difference between
the state levels. See "Undershoot" on page 3-1165. The length of US
corresponds to the number of transitions detected in the input signal.
The sample instants in X correspond to the vector indices. To determine
the transitions, undershoot estimates the state levels of the input
waveform by a histogram method. undershoot identifies all regions
that cross the upper-state boundary of the low state and the lower-state
boundary of the high state. The low-state and high-state boundaries are
expressed as the state level plus or minus a multiple of the difference
between the state levels. See "State-Level Tolerances" on page 3-1167.

US = undershoot(X,FS) specifies the sampling frequency, FS, in hertz.
The sampling frequency determines the sample instants corresponding
to the elements in X. The first sample instant in X corresponds to t=0.

US = undershoot(X,T) specifies the sample instants, T, as a vector
with the same number of elements as X.

[US,USLEV,USINST] = undershoot(...) returns the levels, USLEV,
and sample instants, USINST, of the undershoots for each transition.

[...] = undershoot(...,Name,Value) returns the greatest
deviations below the final state level with additional options specified
by one or more Name,Value pair arguments.

undershoot(...) plots the bilevel waveform and marks the location
of the undershoot of each transition as well as the lower- and upper
reference-level instants and the associated reference levels. undershoot

also plots the state levels and associated lower- and upper-state boundaries.

Input Arguments

X

Bilevel waveform. X is a real-valued row or column vector.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

PctRefLevels

Reference levels as a percentage of the waveform amplitude. The lower-state level is defined to be 0 percent. The upper-state level is defined to be 100 percent. The value of 'PCTREFLEVELS' is a 2-element real row vector whose elements correspond to the lower and upper percent reference levels.

> **Default:** [10 90]

Region

Specify the region over which to perform the undershoot computation. Valid values for 'Region' are 'Preshoot' or 'Postshoot'. If you specify 'Preshoot', the end of the pretransition aberration region is defined as the last instant when the signal exits the first state. If you specify 'Postshoot', the start of the posttransition aberration region is defined as the instant when the signal enters the second state.

> **Default:** 'Postshoot'

SeekFactor

Aberration region duration. Specifies the duration of the region over which to compute the undershoot for each transition as a multiple of the corresponding transition duration. The edge of the waveform may be reached, or a complete intervening transition may be detected, before the duration aberration region duration elapses. In such cases, the duration is truncated to the edge of the waveform or the start of the intervening transition.

Default: 3

StateLevels

Lower- and upper-state levels. Specify the levels to use for the lower- and upper-state levels as a 2-element real row vector whose first and second elements correspond to the lower- and upper-state levels of the input waveform.

Tolerance

Specify the tolerance that the initial and final levels of each transition must be within the respective state levels. The 'Tolerance' value is a scalar expressing a percentage of the difference between the upper- and lower-state levels. See "State-Level Tolerances" on page 3-1167.

Default: 2

Output Arguments

US

Undershoots expressed as a percentage of the state levels. The undershoot percentages are computed based on the greatest deviation from the final state level in each transition. By default undershoots are computed for posttransition aberration regions. See "Undershoot" on page 3-1165.

USLEV

Level of the pretransition or posttransition undershoot.

USINST

Sample instants of pretransition or posttransition undershoots. If you specify the sampling frequency or sampling instants, the undershoot instants are in seconds. If you do not specify the sampling frequency or sampling instants, the undershoot instants are the indices of the input vector.

Definitions

Undershoot

For a positive-going (positive-polarity) pulse, undershoot expressed as a percentage is

$$100\frac{(S_2 - U)}{(S_2 - S_1)}$$

where U is the greatest deviation below the high-state level, S_2 is the high state, and S_1 is the low state.

For a negative-going (negative-polarity) pulse, undershoot expressed as a percentage is

$$100\frac{(S_1 - U)}{(S_2 - S_1)}$$

The following figure illustrates the calculation of undershoot for a positive-going transition.

undershoot

The red dashed lines indicate the estimated state levels. The double-sided black arrow depicts the difference between the high- and low-state levels. The solid black line indicates the difference between the high-state level and the undershoot value.

State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

undershoot

Examples   **Undershoot Percentage in Posttransition Aberration Region**

Determine the maximum percent undershoot relative to the high-state level in a 2.3 V clock waveform.

Load the 2.3 V clock data. Plot the waveform. In this example, you see that the maximum undershoot in the posttransition region occurs near index 23.

```
load('transitionex.mat', 'x');
plot(x);
set(gca,'xtick',[1 5 12 19 23 30 40]);
grid on;
```

Determine the maximum percent undershoot.

```
us = undershoot(x);
```

Undershoot Percentage, Levels, and Sample Instant in Posttransition Aberration Region

Determine the maximum percent undershoot relative to the high-state level, the level of the undershoot, and the sample instant in a 2.3 V clock waveform.

Load the 2.3 V clock data with sampling instants. Plot the waveform. The clock data is sampled at 4 MHz.

```
load('transitionex.mat', 'x','t');
plot(t,x);
```

Determine the maximum percent undershoot, the level of the undershoot in volts, and the sampling instant where the maximum undershoot occurs. Plot the result.

```
[us,uslev,usinst] = undershoot(x,t);
plot(t.*1e6,x); xlabel('Microseconds');
hold on; grid on;
plot(usinst*1e6,uslev,'ro','markerfacecolor',[1 0 0]);
```

Undershoot Percentage, Levels, and Sample Instant in Pretransition Aberration Region

Determine the maximum percent undershoot relative to the low-state level, the level of the undershoot, and the sample instant in a 2.3

undershoot

V clock waveform. Specify the `'Region'` as `'Preshoot'` to output pre-transition metrics.

Load the 2.3 V clock data with sampling instants. Plot the waveform. The clock data is sampled at 4 MHz.

```
load('transitionex.mat', 'x','t');
plot(t,x);
```

Determine the maximum percent undershoot, the level of the undershoot in volts, and the sampling instant where the maximum undershoot occurs. Plot the result.

```
load('transitionex.mat', 'x','t');
[us,uslev,usinst] = undershoot(x,t,'Region','Preshoot');
plot(t.*1e6,x); xlabel('Microseconds');
hold on; grid on;
plot(usinst*1e6,uslev,'ro','markerfacecolor',[1 0 0]);
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Standard 181, 2003, pp. 15–17.

See Also   overshoot | settlingtime | statelevels

Purpose	Structures for specification object with design method

Syntax
```
filtstruct = validstructures(D)
C = validstructures(D,METHOD)
Cs = validstructures(D,...,'SystemObject',sysobjflag)
```

Description `filtstruct = validstructures(D)` returns a structure array containing all valid filter structures for the filter specification object, `D`, organized by design method. Each design method is a field in the structure array, `filtstruct`. The fields contain a cell array of strings.

`C = validstructures(D,METHOD)` returns the valid structures for the filter specification object, `D`, and the design method, `METHOD`, in a cell array of strings.

`Cs = validstructures(D,...,'SystemObject',sysobjflag)` returns the valid structures for designing a filter System object when *sysobjflag* is `true`. To use System objects, you must have the DSP System Toolbox product installed. When *sysobjflag* is `false`, the function returns valid structures for designing `dfilt` and `mfilt` objects, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for `dfilt` and `mfilt` objects.

Examples Design a default lowpass filter specification object. Return all valid design methods and structures in a structure array. Display the fieldnames to see all valid design methods. Display the valid filter structures for the `equiripple` field.

```
D = fdesign.lowpass;
filtstruct = validstructures(D);
fieldnames(filtstruct)
filtstruct.equiripple
```

Create a highpass filter of order 50 with a 3-dB frequency of 0.2. Obtain the available structures for a Butterworth design.

validstructures

```
D = fdesign.highpass('N,F3dB',50,0.2);
C = validstructures(D,'butter');
```

If you have DSP System Toolbox software installed, use the 'SystemObject',*sysobjflag* syntax to return valid structures for a filter System object:

```
Cs = validstructures(D,'butter','SystemObject',true);
```

See Also   design | designmethods | designopts | fdesign

Purpose   Voltage controlled oscillator

Syntax   y = vco(x,fc,fs)
                  y = vco(x,[Fmin Fmax],fs)

Description   y = vco(x,fc,fs) creates a signal that oscillates at a frequency
                  determined by the real input vector or array x with sampling frequency
                  fs. fc is the carrier or reference frequency; when x is 0, y is an fc Hz
                  cosine with amplitude 1 sampled at fs Hz. x ranges from -1 to 1, where
                  x = -1 corresponds to 0 frequency output, x = 0 corresponds to fc, and
                  x = 1 corresponds to 2*fc. Output y is the same size as x.

                  y = vco(x,[Fmin Fmax],fs) scales the frequency modulation range
                  so that ±1 values of x yield oscillations of Fmin Hz and Fmax Hz
                  respectively. For best results, Fmin and Fmax should be in the range 0
                  to fs/2.

                  By default, fs is 1 and fc is fs/4.

                  If x is a matrix, vco produces a matrix whose columns oscillate
                  according to the columns of x.

Examples   Generate two seconds of a signal sampled at 10,000 samples/second
                  whose instantaneous frequency is a triangle function of time:

```
fs = 10000;
t = 0:1/fs:2;
x = vco(sawtooth(2*pi*t,0.75),[0.1 0.4]*fs,fs);
```

                  Plot the spectrogram of the generated signal:

```
spectrogram(x,kaiser(256,5),220,512,fs,'yaxis')
```

Algorithms   vco performs FM modulation using the `modulate` function.

Diagnostics   If any values of `x` lie outside [-1, 1], vco gives the following error message.

```
X outside of range [-1,1].
```

See Also   `demod` | `modulate`

Purpose   Window function gateway

Syntax   ```
window
w = window(fhandle,n)
w = window(fhandle,n,winopt)
```
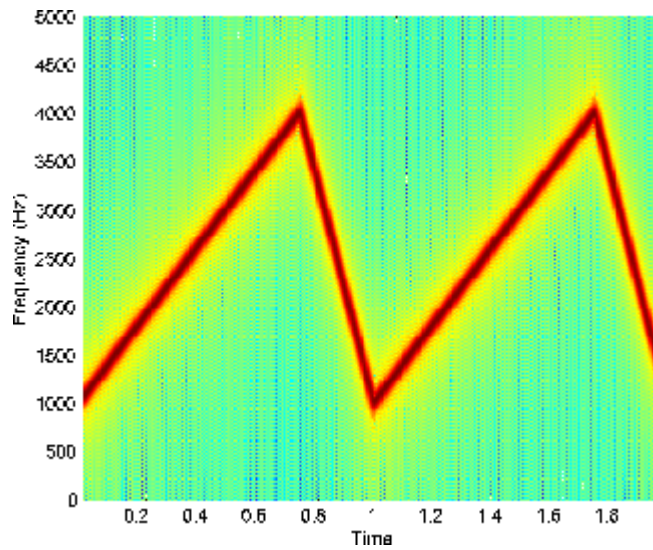
Description   window opens the Window Design and Analysis Tool (wintool).

w = window(fhandle,n) returns the n-point window, specified by its function handle, fhandle, in column vector w. Function handles are window function names preceded by an @.

    @barthannwin
    @bartlett
    @blackman
    @blackmanharris
    @bohmanwin
    @chebwin
    @flattopwin
    @gausswin
    @hamming
    @hann
    @kaiser
    @nuttallwin
    @parzenwin
    @rectwin
    @taylorwin
    @triang
    @tukeywin

window

w = window(fhandle,n,*winopt*) returns the window specified by its function handle, fhandle, and its *winopt* value or sampling flag string. For chebwin, kaiser, and tukeywin, you must enter a *winopt* value. For the other windows listed below, *winopt* values are optional.

Window	winopt Description	winopt Value
blackman	sampling flag string	'periodic'or 'symmetric'
chebwin	sidelobe attenuation relative to mainlobe	numeric
flattopwin	sampling flag string	'periodic'or 'symmetric'
gausswin	alpha value (reciprocal of standard deviation)	numeric
hamming	sampling flag string	'periodic'or 'symmetric'
hann	sampling flag string	'periodic'or 'symmetric'
kaiser	beta value	numeric
taylorwin	1. number of sidelobes 2. maximum sidelobe level in dB relative to mainlobe peak	1. integer greater than or equal to 1 2. negative value
tukeywin	ratio of taper to constant sections	numeric

Examples   Create Blackman Harris, Hamming, and Gaussian windows and plot
them in the same WVTool.

```
N = 65;
w = window(@blackmanharris,N);
w1 = window(@hamming,N);
w2 = window(@gausswin,N,2.5);
wvtool(w,w1,w2)
```
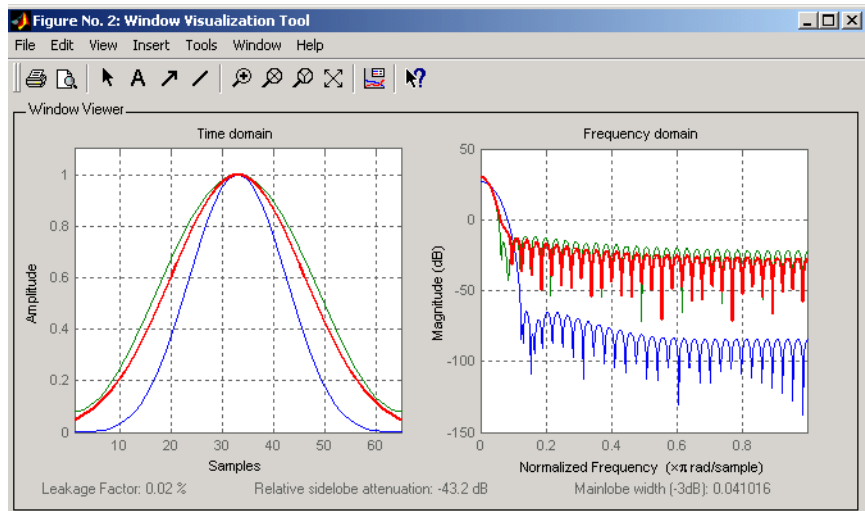

See Also   barthannwin | bartlett | blackman | blackmanharris | bohmanwin
| chebwin | flattopwin | gausswin | hamming | hann | kaiser |
nuttallwin | parzenwin | rectwin | triang | taylorwin | tukeywin

window (filter design method)

Purpose   FIR filter using windowed impulse response

Syntax   h = window(d,'window',fcnhndl)
                 h = window(d,win)

description

> **Note** This is a description of the overloaded method used in conjunction
> with fdesign to design a filter from a filter specification object. To
> access the window function gateway see window.

h = window(d,'window',fcnhndl) designs an FIR filter using
the specifications in filter specification object d. Depending on the
specification type of d, the returned filter is either a single-rate digital
filter — a dfilt, or a multirate digital filter — an mfilt.

fcnhndl is a handle to a filter design function that returns a window
vector, such as the hamming or blackman functions. fcnarg is an
optional argument that returns a window. You pass the function to
window. Refer to example 1 in the following section to see the function
argument used to design the filter.

h = window(d,win) designs a filter using the vector you supply in win.
The length of vector win must be the same as the impulse response of
the filter, which is equal to the filter order plus one.

Examples   Construct a lowpass filter specification object of order 10 with a cutoff
frequency of 12 kilohertz. We use a sampling frequency of 48 kilohertz.
Next we use a function handle to the function Kaiser to provide the
window.

```
d=fdesign.lowpass('n,fc',10,12000,48000);
Hd=window(d,'window',@kaiser);
fvtool(Hd);
```
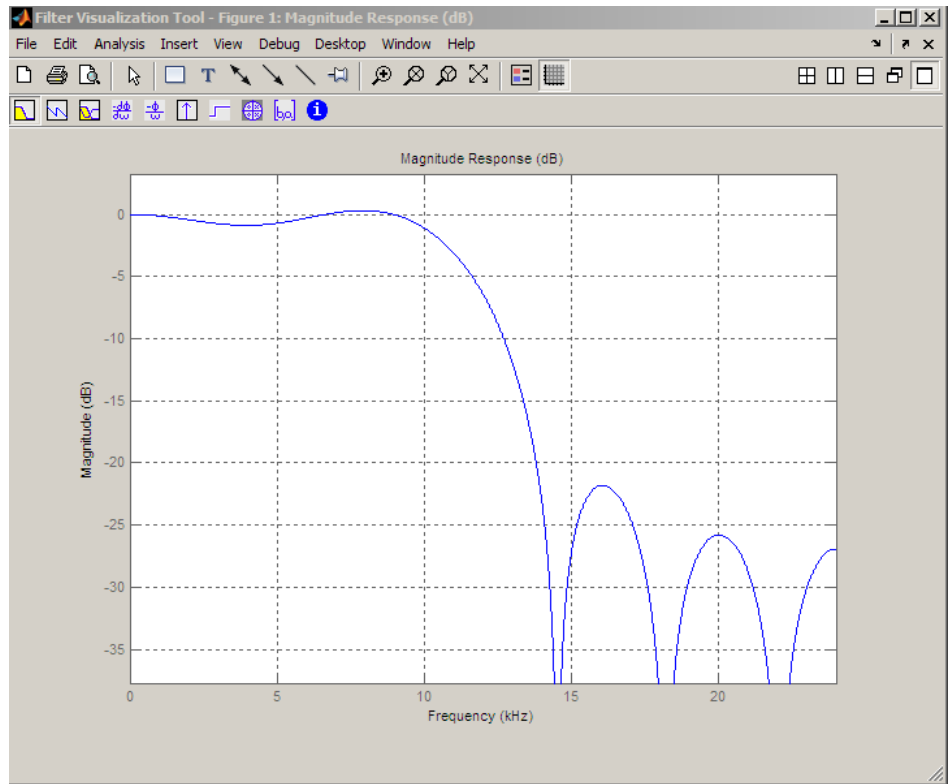
See Also   design | designmethods | fdesign

wintool

Purpose	Open Window Design and Analysis Tool
Syntax	`wintool` `wintool(obj1,obj2,...)`
Description	`wintool` opens the Window Design and Analysis Tool (WinTool), a graphical user interface (GUI) for designing and analyzing spectral windows. It opens with a default 64-point Hamming window.

`wintool(obj1,obj2,...)` opens WinTool with the `sigwin` window object(s) specified in `obj1`, `obj2`, etc.

Note A related tool, `wvtool`, is available for displaying, annotating, or printing windows.

---
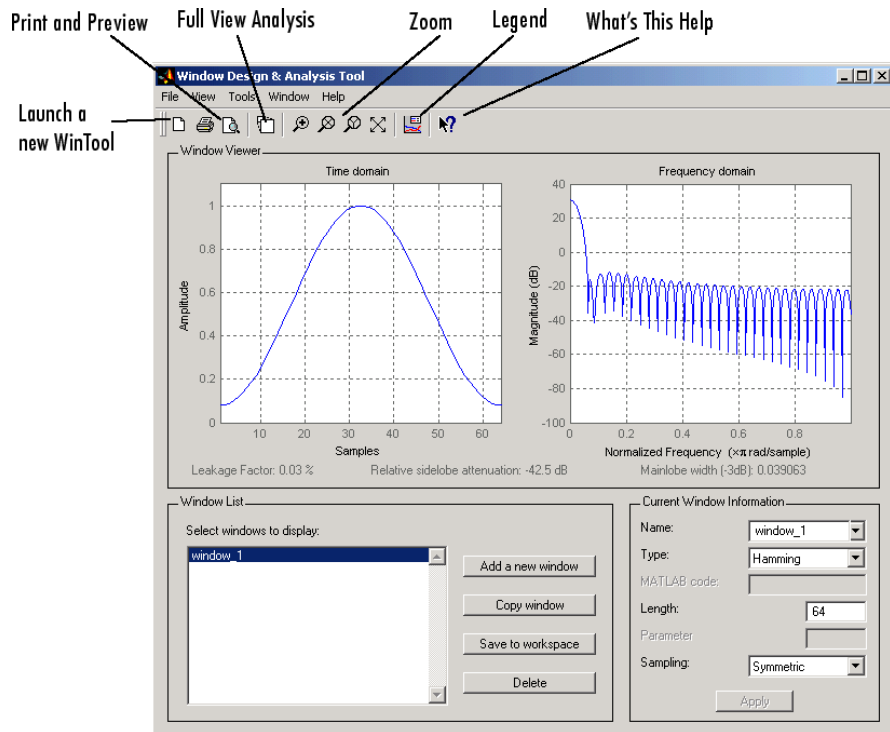
wintool has three panels:

- Window Viewer displays the time domain and frequency domain representations of the selected window(s). The currently active window is shown in bold. Three window measurements are shown below the plots.

  - Leakage factor — ratio of power in the sidelobes to the total window power

  - Relative sidelobe attenuation — difference in height from the mainlobe peak to the highest sidelobe peak

  - Mainlobe width (-3dB) — width of the mainlobe at 3 dB below the mainlobe peak

3-1181

- Window List lists the windows available for display in the Window Viewer. Highlight one or more windows to display them. The Window List buttons are:

  - **Add a new window** — Adds a default Hamming window with length 64 and symmetric sampling. You can change the information for this window by applying changes made in the **Current Window Information** panel.

  - **Copy window** — Copies the selected window(s).

  - **Save to workspace** — Saves the selected window(s) as vector(s) to the MATLAB workspace. The name of the window in `wintool` is used as the vector name.

  - **Delete** — Removes the selected window(s) from the window list.

- *Current Window Information* displays information about the currently active window. The active window name is shown in the `Name` field. To make another window active, select its name from the **Name** menu.

Window Parameters

Each window is defined by the parameters in the Current Window Information panel. You can change the current window's characteristics by changing its parameters and clicking **Apply**. The parameters of the current window are

- **Name** — Name of the window. The name is used for the legend in the Window Viewer, in the Window List, and for the vector saved to the workspace. You can either select a name from the menu or type the desired name in the edit box.

- **Type** — Algorithm for the window. Select the type from the menu. All Signal Processing Toolbox windows are available.

- **MATLAB code** — Any valid MATLAB expression that returns a vector defining the window if `Type = User Defined`.

- **Length** — Number of samples.

- **Parameter** — Additional parameter for windows that require it, such as Chebyshev, which requires you to specify the sidelobe attenuation. Note that the title "Parameter" changes to the appropriate parameter name.

- **Sampling** — Type of sampling to use for generalized cosine windows (Hamming, Hann, and Blackman) — `Periodic` or `Symmetric`. `Periodic` computes a length `n+1` window and returns the first `n` points, and `Symmetric` computes and returns the `n` points specified in `Length`.

WinTool Menus

In addition to the usual menus items, `wintool` contains these `wintool`-specific menu commands:

File menu:

- **Export** — Exports window coefficient vectors or `sigwin` window objects to the MATLAB workspace, a text file, or a MAT-file.

  In the **Window List** in WinTool, highlight the window(s) you want to export and then select **File > Export**. For exporting to the workspace or a MAT-file, specify the variable name for each window coefficient or object. To overwrite variables in the workspace, select the Overwrite variables check box.

- **Full View Analysis** — Copies the windows shown in both plots to a separate `wvtool` figure window. This is useful for printing and annotating. This option is also available with the Full View Analysis toolbar button.

View menu:

- **Time domain** — Select to show the time domain plot in the Window Viewer panel.

- **Frequency domain** — Select to show the frequency domain plot in the Window Viewer panel.
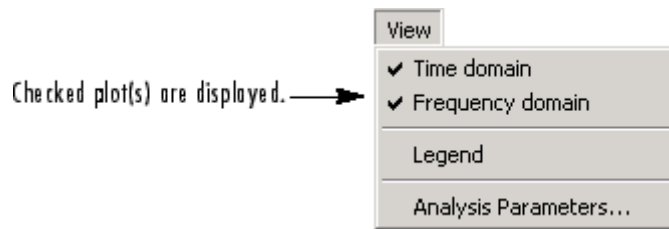
Checked plot(s) are displayed.

- **Legend** — Toggles the window name legend on and off. This option is also available with the Legend toolbar button.

- **Analysis Parameters** — Controls the response plot parameters, including number of points, range, *x*- and *y*-axis units, sampling frequency, and normalized magnitude.

  You can also access the Analysis Parameters by right-clicking the *x*-axis label of a plot in the Window Viewer panel. The *x*-axis units for the time domain plot depend on the selected Sampling Frequency units.

Frequency Domain	Time Domain
Hz	sec
kHz	ms
MHz	μs
GHz	picosec

Tools menu:

- **Zoom In** — Zooms in along both *x*- and *y*-axes.

- **Zoom X** — Zooms in along the *x*-axis only. Drag the mouse in the *x* direction to select the zoom area.

- **Zoom Y** — Zooms in along the y-axis only. Drag the mouse in the *y* direction to select the zoom area.

- **Full View** — Returns to full view.

See Also   window | wvtool

wvtool

Purpose
Open Window Visualization Tool

Syntax
```
wvtool(WindowVector)
wvtool(WinObj)
wvtool(WindowVector1,WinObj1,...,WinObjN,WindowVectorN)
H = wvtool(...)
```

Description
wvtool(WindowVector) opens the Window Visualization Tool (WVTool) with time and frequency domain plots of the window vector specified in WindowVector. WindowVector must be a real-valued row or column vector. By default, the frequency domain plot is the magnitude squared of the Fourier transform of the window vector in decibels (dB). You can generate window vectors for a number of common window functions using the Signal Processing Toolbox software. See window for a list of supported window functions.

wvtool(WinObj) opens WVTool with time and frequency domain plots of the sigwin object WinObj. See sigwin for a list of supported signal processing window objects.

wvtool(WindowVector1,WinObj1,...,WinObjN,WindowVectorN) opens WVTool with time and frequency domain plots of the window vectors or window objects specified in WindowVector1,WinObj1,...,WinObjN, WindowVectorN.

H = wvtool(...) returns the figure handle H.

Note A related tool, wintool, is available for designing and analyzing windows.
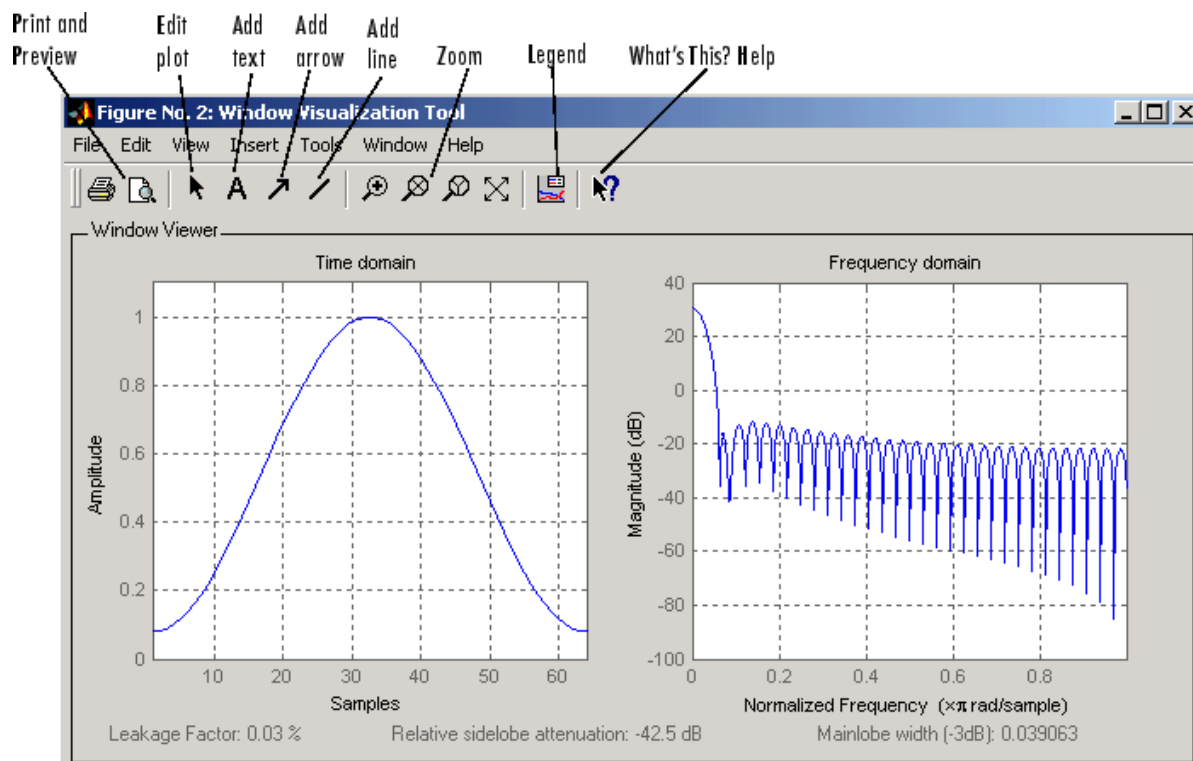
Note If you launch WVTool from FDATool, an **Add/Replace** icon, which controls how new windows are added from FDATool, appears on the toolbar.

WVTool Menus

In addition to the usual menus items, wvtool contains these wvtool-specific menu commands:

File menu:

• **Export** — Exports the displayed plot(s) to a graphic file.

Edit menu:

- **Copy figure** — Copies the displayed plot(s) to the clipboard (available only on Windows platforms).

- **Copy options** — Displays the Preferences dialog box (available only on Windows platforms).

- **Figure**, **Axes**, and **Current Object Properties** — Displays the Property Editor.

View menu:

- **Time domain** — Check to show the time domain plot.

- **Frequency domain** — Check to show the frequency domain plot.
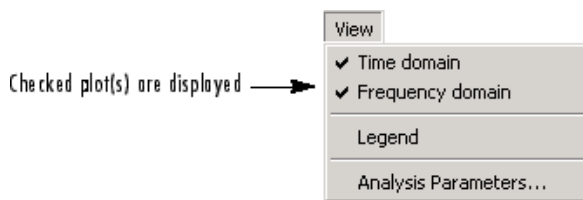
Checked plot(s) are displayed

- **Legend** — Toggles the window name legend on and off. This option is also available with the **Legend** toolbar button.

- *Analysis Parameters* — Controls the response plot parameters, including number of points, range, *x*- and *y*-axis units, sampling frequency, and normalized magnitude.

  You can also access the Analysis Parameters by right-clicking the *x*-axis label of a plot in the Window Viewer panel.

- **Insert** menu:

You use the **Insert** menu to add labels, titles, arrows, lines, text, and axes to your plots.

Tools menu:

- **Edit Plot** — Turns on plot editing mode

- **Zoom In** — Zooms in along both *x*- and *y*-axes.

- **Zoom X** — Zooms in along the *x*-axis only. Drag the mouse in the *x* direction to select the zoom area.

- **Zoom Y** — Zooms in along the y-axis only. Drag the mouse in the *y* direction to select the zoom area.

- **Full View** — Returns to full view.

Examples   Compare Hamming, Hann, and Gaussian windows:

```
wvtool(hamming(64),hann(64),gausswin(64))
```
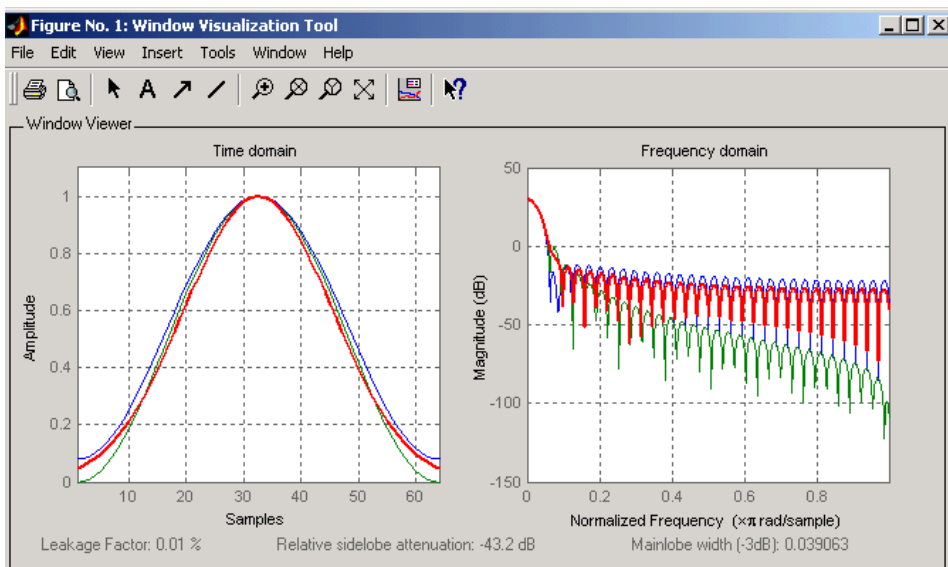

Compare Kaiser window objects with different beta values:

```
H = sigwin.kaiser(128,1.5);
% Kaiser window with beta=4.5
H1 = sigwin.kaiser(128,4.5);
wvtool(H,H1)
```

See Also   fdatool | sigwin | window | wintool

Purpose   Cross-correlation

Syntax
```
c = xcorr(x,y)
c = xcorr(x)
c = xcorr(x,y,'option')
c = xcorr(x,'option')
c = xcorr(x,y,maxlags)
c = xcorr(x,maxlags)
c = xcorr(x,y,maxlags,'option')
c = xcorr(x,maxlags,'option')
[c,lags] = xcorr(...)
[c,lags] = xcorr(gpuArrayX,gpuArrayY,maxlags,'option')
```

Description   xcorr estimates the cross-correlation sequence of a random process. Autocorrelation is handled as a special case.

The true cross-correlation sequence is

$$R_{xy}(m) = E\{x_{n+m}y_n^*\} = E\{x_n y_{n-m}^*\}$$

where x_n and y_n are jointly stationary random processes, $-\infty < n < \infty$, and $E\{\cdot\}$ is the expected value operator. xcorr must estimate the sequence because, in practice, only a finite segment of one realization of the infinite-length random process is available.

c = xcorr(x,y) returns the cross-correlation sequence in a length 2*N-1 vector, where x and y are length N vectors (N>1). If x and y are not the same length, the shorter vector is zero-padded to the length of the longer vector.

By default, xcorr computes raw correlations with no normalization.

$$\hat{R}xy(m) = \begin{cases} \displaystyle\sum_{n=0}^{N-m-1} x_{n+m}y_n^* & m \geq 0 \\ \hat{R}yx^*(-m) & m < 0 \end{cases}$$

The output vector c has elements given by:

$$c(m) = \hat{R}_{xy}(m - N) \quad m = 1, 2, \ldots 2N - 1$$

In general, the correlation function requires normalization to produce an accurate estimate (see below).

c = xcorr(x) is the autocorrelation sequence for the vector x. If x is an N-by-P matrix, c is a matrix with $2N$-1 rows whose P^2 columns contain the cross-correlation sequences for all combinations of the columns of x. For more information on matrix processing with xcorr, see "Multiple Channels".

xcorr produces correlations identically equal to 1.0 at zero lag only when you perform an autocorrelation and only when you set the 'coeff' option. For example,

```
x=0:0.01:10;
X = sin(x);
[r,lags]=xcorr(X,'coeff');
max(r)
```

c = xcorr(x,y,'*option*') specifies a normalization option for the cross-correlation, where '*option*' is

- 'biased': Biased estimate of the cross-correlation function

$$\hat{R}_{xy,biased}(m) = \frac{1}{N}\hat{R}_{xy}(m)$$

- 'unbiased': Unbiased estimate of the cross-correlation function

$$\hat{R}_{xy,unbiased}(m) = \frac{1}{N-|m|}\hat{R}_{xy}(m)$$

- 'coeff': Normalizes the sequence so the autocorrelations at zero lag are identically 1.0.

- `'none'`, to use the raw, unscaled cross-correlations (default)

See [1] for more information on the properties of biased and unbiased correlation estimates.

`c = xcorr(x,'option')` specifies one of the above normalization options for the autocorrelation.

`c = xcorr(x,y,maxlags)` returns the cross-correlation sequence over the lag range `[-maxlags:maxlags]`. Output `c` has length `2*maxlags+1`.

`c = xcorr(x,maxlags)` returns the autocorrelation sequence over the lag range `[-maxlags:maxlags]`. Output `c` has length `2*maxlags+1`. If x is an N-by-P matrix, `c` is a matrix with `2*maxlags+1` rows whose P^2 columns contain the autocorrelation sequences for all combinations of the columns of x.

`c = xcorr(x,y,maxlags,'option')` specifies both a maximum number of lags and a scaling option for the cross-correlation.

`c = xcorr(x,maxlags,'option')` specifies both a maximum number of lags and a scaling option for the autocorrelation.

`[c,lags] = xcorr(...)` returns a vector of the lag indices at which `c` was estimated, with the range `[-maxlags:maxlags]`. When `maxlags` is not specified, the range of `lags` is `[-N+1:N-1]`.

In all cases, the cross-correlation or autocorrelation computed by xcorr has the zeroth lag in the middle of the sequence, at element or row `maxlags+1` (element or row `N` if `maxlags` is not specified).

`[c,lags] = xcorr(gpuArrayX,gpuArrayY,maxlags,'option')` returns the autocorrelation or cross-correlation sequence for input objects of class `gpuArray`. See "Using gpuArray" for details on gpuArray objects. Using xcorr with gpuArray objects requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See http://www.mathworks.com/products/parallel-computing/requirements.html for details. The returned autocorrelation or cross-correlation sequence, `c`, is a gpuArray object.

"GPU Acceleration for Autocorrelation Sequence Estimation" on page 3-1194 shows you how to compute the autocorrelation sequence on the GPU.

Examples

The second output, lags, is useful for plotting the cross-correlation or autocorrelation. For example, the estimated autocorrelation of zero-mean Gaussian white noise $c_{ww}(m)$ can be displayed for $-10 \leq m \leq 10$ using:

```
ww = randn(1000,1);
[c_ww,lags] = xcorr(ww,10,'coeff');
stem(lags,c_ww)
```

Swapping the x and y input arguments reverses (and conjugates) the output correlation sequence. For row vectors, the resulting sequences are reversed left to right; for column vectors, up and down. The following example illustrates this property (mat2str is used for a compact display of complex numbers):

```
x = [1,2i,3]; y = [4,5,6];
[c1,lags] = xcorr(x,y);
c1 = mat2str(c1,2), lags
c2 = conj(fliplr(xcorr(y,x)));
c2 = mat2str(c2,2)
```

For the case where input argument x is a matrix, the output columns are arranged so that extracting a row and rearranging it into a square array produces the cross-correlation matrix corresponding to the lag of the chosen row. For example, the cross-correlation at zero lag can be retrieved by:

```
X = randn(2,2);
[M,P] = size(X);
c = xcorr(X);
c0 = zeros(P); c0(:) = c(M,:)    % Extract zero-lag row
```

You can calculate the matrix of correlation coefficients that the MATLAB function `corrcoef` generates by substituting:

```
c = xcov(X,'coef')
```

in the last example. The function `xcov` subtracts the mean and then calls `xcorr`.

Use `fftshift` to move the second half of the sequence starting at the zeroth lag to the front of the sequence. `fftshift` swaps the first and second halves of a sequence.

GPU Acceleration for Autocorrelation Sequence Estimation

The following example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See http://www.mathworks.com/products/parallel-computing/requirements.html for details.

Create a signal consisting of a 10-Hz sine wave in additive noise. Use `gpuArray` to create a gpuArray object stored on your computer's GPU.

```
t = 0:0.001:10-0.001;
x = cos(2*pi*10*t)+randn(size(t));
X = gpuArray(x);
```

Compute the normalized autocorrelation sequence to lag 200.

```
[xc,lags] = xcorr(X,200,'coeff');
```

The output, xc, is a gpuArray object. Use `gather` to transfer the data from the GPU to the MATLAB workspace as a double-precision vector.

```
xc = gather(xc);
```

Algorithms   For more information on estimating covariance and correlation functions, see [1].

References   [1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also   conv | corrcoef | cov | xcorr2 | xcov

xcorr2

Purpose   2–D cross-correlation

Syntax   C = xcorr2(A,B)
                 A = xcorr2(A)
                 C = xcorr2(gpuArrayA,gpuArrayB)

Description C = xcorr2(A,B) returns the cross-correlation of matrices A and B with
                 no scaling. xcorr2 is the two-dimensional version of xcorr. It has its
                 maximum value when the two matrices are aligned so that they are
                 shaped as similarly as possible.

                 If matrix *A* has dimensions (*Ma, Na*) and matrix *B* has dimensions (*Mb,
                 Nb*), The equation for the two-dimensional discrete cross-correlation is

                 $$C(i,j) = \sum_{m=0}^{(Ma-1)} \sum_{n=0}^{(Na-1)} A(m,n) \cdot conj(B(m+i,n+j))$$

                 where $0 \le i < Ma + Mb - 1$ and $0 \le j < Na + Nb - 1$.

                 A = xcorr2(A) is the autocorrelation matrix of input matrix A. It is
                 identical to xcorr2(A,A).

                 C = xcorr2(gpuArrayA,gpuArrayB) returns the cross-correlation of
                 the input matrices of class gpuArray. See "Using gpuArray"
                 for details on gpuArray objects. Using xcorr2 with gpuArray
                 objects requires Parallel Computing Toolbox software and a
                 CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See
                 http://www.mathworks.com/products/parallel-computing/requirements.html
                 for details. The output cross-correlation matrix, C, is a gpuArray object.
                 See "GPU Acceleration for Cross-Correlation Matrix Computation" on
                 page 3-1201 for an example of using the GPU to compute the
                 cross-correlation.

Examples   **Output Matrix Size**

                 If matrix I1 has dimensions (4,3) and matrix I2 has dimensions (2,2),
                 the following equations determine the number of rows and columns of
                 the output matrix:

$$C_{\text{full}_{\text{rows}}} = I1_{\text{rows}} + I2_{\text{rows}} - 1 = 4 + 2 - 1 = 5$$

$$C_{\text{full}_{\text{columns}}} = I1_{\text{columns}} + I2_{\text{columns}} - 1 = 3 + 2 - 1 = 4$$

The resulting matrix is

$$C_{\text{full}} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \\ c_{40} & c_{41} & c_{42} & c_{43} \end{bmatrix}$$

Computing a Specific Element

$$C_{\text{valid}_{\text{columns}}} = I1_{\text{columns}} - I2_{\text{columns}} + 1 = 2$$

In cross-correlation, the value of an output element is computed as a weighted sum of neighboring elements. For example, suppose the first input matrix represents an image and is defined as

```
I1 = [17   24    1    8   15
      23    5    7   14   16
       4    6   13   20   22
      10   12   19   21    3
      11   18   25    2    9]
```

The second input matrix also represents an image and is defined as

```
I2 = [8    1    6
      3    5    7
      4    9    2]
```

The following figure shows how to compute the (2,4) output element (zero-based indexing) using these steps:

1 Slide the center element of I2 so that lies on top of the (1,3) element of I1.

2 Multiply each weight in I2 by the element of I1 underneath.

3 Sum the individual products from step 2.

The (2,4) output element from the cross-correlation is

$$1 \cdot 8 + 8 \cdot 1 + 15 \cdot 6 + 7 \cdot 3 + 14 \cdot 5 + 16 \cdot 7 + 13 \cdot 4 + 20 \cdot 9 + 22 \cdot 2 = 585$$
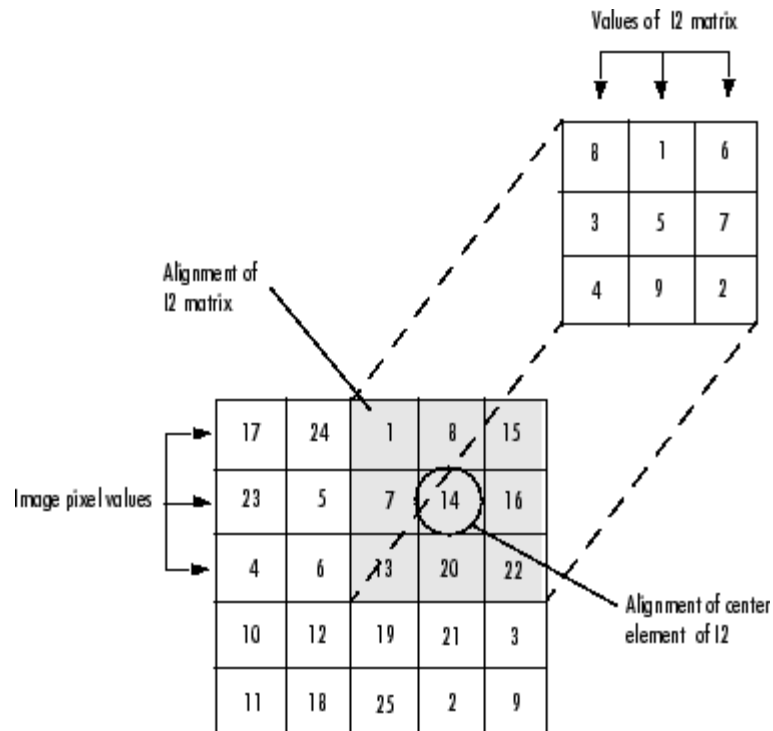
The normalized cross-correlation of the (2,4) output element is

585/sqrt(sum(dot(I1p,I1p))*sum(dot(I2,I2))) = 0.8070

where I1p = [1 8 15; 7 14 16; 13 20 22].

Recovery of Template Shift with Cross-Correlation

Shift a template by a known amount and recover the shift using cross-correlation.

Create a template in an 11-by-11 matrix. Create a 22-by-22 matrix and shift the original template by 8 along the row dimension and 6 along the column dimension.

```
template = .2*ones(11);
template(6,3:9) = .6;
template(3:9,6) = .6;
offsetTemplate = .2*ones(22);
offset = [8 6];
offsetTemplate( (1:size(template,1))+offset(1),...
                (1:size(template,2))+offset(2) ) = template;
```

Plot the original and shifted templates.

```
imagesc(offsetTemplate); colormap gray;
hold on;
imagesc(template);
```
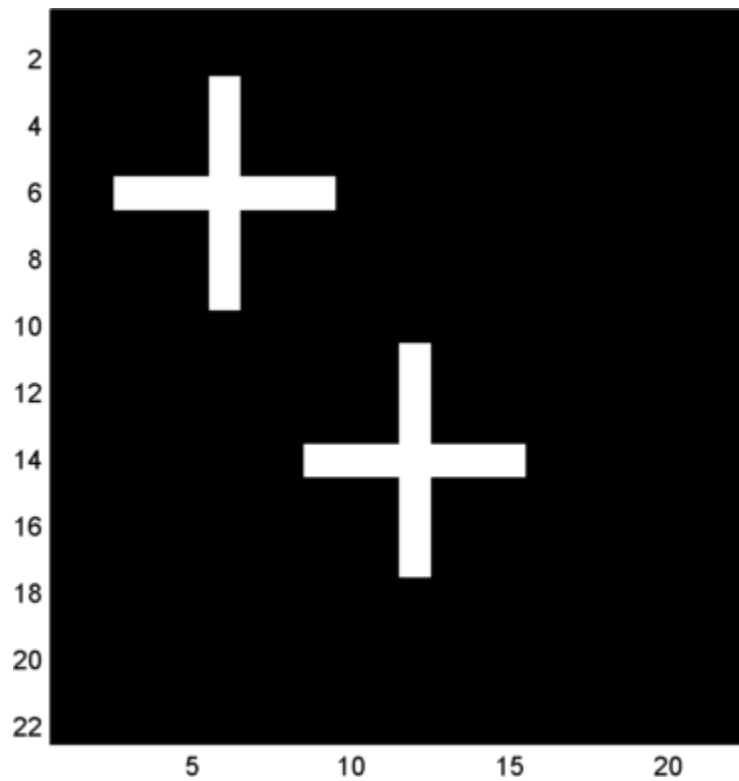
Cross-correlate the two matrices and find the maximum absolute value of the cross-correlation. Use the position of the maximum absolute value to determine the shift in the template. Check the result against the known shift.

```
cc = xcorr2(offsetTemplate,template);
[max_cc, imax] = max(abs(cc(:)));
[ypeak, xpeak] = ind2sub(size(cc),imax(1));
corr_offset = [ (ypeak-size(template,1)) (xpeak-size(template,2)) ];
isequal(corr_offset,offset)
```

The returned 1 indicates that the shift obtained the cross-correlation equals the known the template shift in both the row and column dimension.

GPU Acceleration for Cross-Correlation Matrix Computation

The following example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See http://www.mathworks.com/products/parallel-computing/requirements.html for details.

Repeat the example "Recovery of Template Shift with Cross-Correlation" on page 3-1199. For convenience, the code to create the original and shifted templates is repeated.

```
template = .2*ones(11);
template(6,3:9) = .6;
template(3:9,6) = .6;
offsetTemplate = .2*ones(22);
offset = [8 6];
offsetTemplate( (1:size(template,1))+offset(1),...
                (1:size(template,2))+offset(2) ) = template;
```

Put the original and shifted template matrices on your GPU using `gpuArray` objects.

```
template = gpuArray(template);
offsetTemplate = gpuArray(offsetTemplate);
```

Compute the cross-correlation on the GPU.

```
cc = xcorr2(offsetTemplate,template);
```

Return the result to the MATLAB workspace using `gather`, use the maximum absolute value of the cross-correlation to determine the shift, and compare the result with the known shift.

```
cc = gather(cc);
[max_cc, imax] = max(abs(cc(:)));
```

```
[ypeak, xpeak] = ind2sub(size(cc),imax(1));
corr_offset = [ (ypeak-size(template,1)) (xpeak-size(template,2)) ];
isequal(corr_offset,offset)
```

See Also   conv2 | filter2 | xcorr

Purpose   Cross-covariance

Syntax
```
v = xcov(x,y)
v = xcov(x)
v = xcov(x,'option')
[c,lags] = xcov(x,y,maxlags)
[c,lags] = xcov(x,maxlags)
[c,lags] = xcov(x,maxlags)
[c,lags] = xcov(x,y,maxlags,'option')
[c,lags] = xcov(gpuArrayX,gpuArrayY,maxlags,'option')
```

Description   xcov estimates the cross-covariance sequence of random processes.
Autocovariance is handled as a special case.

The true cross-covariance sequence is the cross-correlation of
mean-removed sequences

$$\phi_{xy}(m) = E\{(x_{n+m} - \mu_x)(y_n - \mu_y)^*)\}$$

where μ_x and μ_y are the mean values of the two stationary random
processes, $*$ denotes the complex conjugate, and $E\{\}$ is the expected
value operator. xcov estimates the sequence because, in practice, access
is available to only a finite segment of the infinite-length random
process.

v = xcov(x,y) returns the cross-covariance sequence in a length 2*N*-1
vector, where x and y are length *N* vectors. For information on how
arrays are processed with xcov, see "Multiple Channels".

v = xcov(x) is the autocovariance sequence for the vector x. Where x
is an *N*-by-*P* array, v = xcov(x) returns an array with 2*N*-1 rows whose
P^2 columns contain the cross-covariance sequences for all combinations
of the columns of x.

By default, xcov computes raw covariances with no normalization. For
a length *N* vector

$$c_{xy}(m) = \begin{cases} \displaystyle\sum_{n=0}^{N-|m|-1} \left(x(n+m) - \frac{1}{N}\sum_{i=0}^{N-1} x_i \right)\left(y_n^* - \frac{1}{N}\sum_{i=0}^{N-1} y_i^* \right) & m \geq 0 \\ c_{yx}^*(-m) & m < 0 \end{cases}$$

The output vector c has elements given by c(m) = c_{xy}(m-N), m = 1, ..., 2N-1.

The covariance function requires normalization to estimate the function properly.

v = xcov(x,'*option*') specifies a scaling option, where '*option*' is

- 'biased', for biased estimates of the cross-covariance function

- 'unbiased', for unbiased estimates of the cross-covariance function

- 'coeff', to normalize the sequence so the auto-covariances at zero lag are identically 1.0

- 'none', to use the raw, unscaled cross-covariances (default)

See [1] for more information on the properties of biased and unbiased correlation and covariance estimates.

[c,lags] = xcov(x,y,maxlags) where x and y are length m vectors, returns the cross-covariance sequence in a length 2*maxlags+1 vector c. lags is a vector of the lag indices where c was estimated, that is, [-maxlags:maxlags].

[c,lags] = xcov(x,maxlags) is the autocovariance sequence over the range of lags [-maxlags:maxlags].

[c,lags] = xcov(x,maxlags) where x is an m-by-p array, returns array c with 2*maxlags+1 rows whose P^2 columns contain the cross-covariance sequences for all combinations of the columns of x.

[c,lags] = xcov(x,y,maxlags,'*option*') specifies a scaling option, where '*option*' is the last input argument.

[c,lags] = xcov(gpuArrayX,gpuArrayY,maxlags,'*option*') returns the autocovariance or cross-covariance sequence for input

objects of class `gpuArray`. See "Using gpuArray" for details on gpuArray objects. Using `xcov` with `gpuArray` objects requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See http://www.mathworks.com/products/parallel-computing/requirements.html for details. The returned autocovariance or cross-covariance sequence, `c`, is a gpuArray object.

"Autocovariance using the GPU" on page 3-1205 shows you how to compute the autocovariance sequence on the GPU.

In all cases, `xcov` gives an output such that the zeroth lag of the covariance vector is in the middle of the sequence, at element or row `maxlag+1` or at `m`.

Examples The second output `lags` is useful when plotting. For example, the estimated autocovariance of white Gaussian noise $c_{ww}(m)$ can be displayed for $-10 \le m \le 10$ using:

```
ww = randn(1000,1);      % White Gaussian noise
[cov_ww,lags] = xcov(ww,10,'coeff');
stem(lags,cov_ww)
```

Autocovariance using the GPU

The following example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See http://www.mathworks.com/products/parallel-computing/requirements.html for details.

Create a signal consisting of a 10-Hz sine wave in additive noise. Use `gpuArray` to create a gpuArray object stored on your computer's GPU.

```
t = 0:0.001:10-0.001;
x = cos(2*pi*10*t)+randn(size(t));
X = gpuArray(x);
```

Compute the autocovariance sequence to lag 200.

```
[xc,lags] = xcov(X,200);
```

The output, `xc`, is a gpuArray object. Use `gather` to transfer the data from the GPU to the MATLAB workspace as a double-precision vector.

```
xc = gather(xc);
```

Algorithms   `xcov` computes the mean of its inputs, subtracts the mean, and then calls `xcorr`. For more information on estimating covariance and correlation functions, see [1].

Diagnostics   `xcov` does not check for any errors other than the correct number of input arguments. Instead, it relies on the error checking in `xcorr`.

References   [1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also   conv | corrcoef | cov | xcorr | xcorr2

yulewalk

Purpose	Recursive digital filter design

Syntax

```
[b,a] = yulewalk(n,f,m)
```

Description

yulewalk designs recursive IIR digital filters using a least-squares fit to a specified frequency response.

[b,a] = yulewalk(n,f,m) returns row vectors b and a containing the n+1 coefficients of the order n IIR filter whose frequency-magnitude characteristics approximately match those given in vectors f and m:

- f is a vector of frequency points, specified in the range between 0 and 1, where 1 corresponds to half the sample frequency (the Nyquist frequency). The first point of f must be 0 and the last point 1, with all intermediate points in increasing order. Duplicate frequency points are allowed, corresponding to steps in the frequency response.

- m is a vector containing the desired magnitude response at the points specified in f.

- f and m must be the same length.

- plot(f,m) displays the filter shape.

The output filter coefficients are ordered in descending powers of z.

$$\frac{B(z)}{A(z)} = \frac{b(1)+b(2)z^{-1}+\cdots+b(n+1)z^{-n}}{a(1)+a(2)z^{-1}+\cdots+a(n+1)z^{-n}}$$

When specifying the frequency response, avoid excessively sharp transitions from passband to stopband. You may need to experiment with the slope of the transition region to get the best filter design.

Examples

Design an 8th-order lowpass filter and overplot the desired frequency response with the actual frequency response:

```
f = [0 0.6 0.6 1];
m = [1 1 0 0];
[b,a] = yulewalk(8,f,m);
```

```
[h,w] = freqz(b,a,128);
plot(f,m,w/pi,abs(h),'--')
legend('Ideal','yulewalk Designed')
title('Comparison of Frequency Response Magnitudes')
```
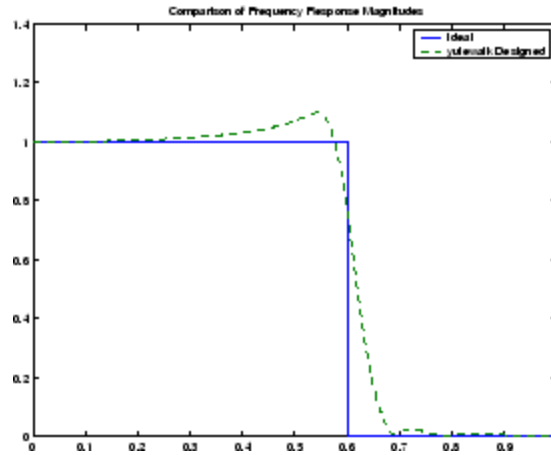

Algorithms yulewalk performs a least-squares fit in the time domain. It computes the denominator coefficients using modified Yule-Walker equations, with correlation coefficients computed by inverse Fourier transformation of the specified frequency response. To compute the numerator, yulewalk takes the following steps:

1 Computes a numerator polynomial corresponding to an additive decomposition of the power frequency response.

2 Evaluates the complete frequency response corresponding to the numerator and denominator polynomials.

3 Uses a spectral factorization technique to obtain the impulse response of the filter.

4 Obtains the numerator polynomial by a least-squares fit to this impulse response.

References   [1] Friedlander, B., and B. Porat, "The Modified Yule-Walker Method of ARMA Spectral Estimation," *IEEE Transactions on Aerospace Electronic Systems, AES-20, No. 2* (March 1984), pp. 158-173.

See Also   butter | cheby1 | cheby2 | ellip | fir2 | firls | maxflat | firpm

zerophase

Purpose   Zero-phase response of digital filter

Syntax   ```
[Hr,w] = zerophase(b,a)
[Hr,w] = zerophase(b,a,nfft)
[Hr,w] = zerophase(b,a,nfft,'whole')
[Hr,w] = zerophase(b,a,w)
[Hr,f] = zerophase(...,fs)
[Hr,w,phi] = zerophase(...)
zerophase(...)
```

**Description**   [Hr,w] = zerophase(b,a) returns the zero-phase response Hr, and the
frequency vector w (in radians/sample) at which Hr is computed, given
a filter defined by numerator b and denominator a. For FIR filters
where a=1, you can omit the value a from the command. The zero-phase
response is evaluated at 512 equally spaced points on the upper half of
the unit circle.

The zero-phase response, Hr(w), is related to the frequency response,
H(w) by

$$H(e^{j\omega}) = Hr(\omega)e^{j\varphi(\omega)}$$

where $H(e^{j\omega})$ is the frequency response, $Hr(\omega)$ is the zero-phase response
and [[PHI1]]($w$) is the continuous phase.

---

**Note** The zero-phase response is always real, but it is not the equivalent
of the magnitude response. The former can be negative while the latter
cannot be negative.

---

[Hr,w] = zerophase(b,a,nfft) returns the zero-phase response Hr
and frequency vector w (radians/sample), using nfft frequency points
on the upper half of the unit circle.

[Hr,w] = zerophase(b,a,nfft,'whole') returns the zero-phase response Hr and frequency vector w (radians/sample), using nfft frequency points around the whole unit circle.

[Hr,w] = zerophase(b,a,w) returns the zero-phase response Hr and frequency vector w (radians/sample) at frequencies in vector w.

[Hr,f] = zerophase(...,fs) returns the zero-phase response Hr and frequency vector f (Hz), using the sampling frequency fs (in Hz), to determine the frequency vector f (in Hz) at which Hr is computed.

[Hr,w,phi] = zerophase(...) returns the zero-phase response Hr, frequency vector w (rad/sample), and the continuous phase component, phi. (Note that this quantity is not equivalent to the phase response of the filter when the zero-phase response is negative.)

zerophase(...) with no output arguments, plots the zero-phase response versus frequency.

**Examples**

**Example 1**

Plot the zero-phase response of a constrained least squares FIR filter:

```
b=fircls1(54,.3,.02,.008);
zerophase(b);
```

Example 2

Plot the zero-phase response of an elliptic filter:

```
[b,a]=ellip(10,.5,20,.4);
zerophase(b,a,512,'whole');
```

See Also   freqs | freqz | fvtool | grpdelay | invfreqz | phasedelay | phasez

zp2sos

Purpose   Convert zero-pole-gain filter parameters to second-order sections form

Syntax
```
[sos,g] = zp2sos(z,p,k)
[sos,g] = zp2sos(z,p,k,'order')
[sos,g] = zp2sos(z,p,k,'order','scale')
[sos,g] = zp2sos(z,p,k,'order','scale',zeroflag)
sos = zp2sos(...)
```

Description   zp2sos converts a discrete-time zero-pole-gain representation of a given digital filter to an equivalent second-order section representation.

[sos,g] = zp2sos(z,p,k) creates a matrix sos in second-order section form with gain g equivalent to the discrete-time zero-pole-gain filter represented by input arguments z, p, and k. Vectors z and p contain the zeros and poles of the filter's transfer function $H(z)$, not necessarily in any particular order.

$$H(z) = k\frac{(z-z_1)(z-z_2)\cdots(z-z_n)}{(z-p_1)(z-p_2)\cdots(z-p_m)}$$

where n and m are the lengths of z and p, respectively, and k is a scalar gain. The zeros and poles must be real or complex conjugate pairs. sos is an L-by-6 matrix

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

$$H(z) = g\prod_{k=1}^{L} H_k(z) = g\prod_{k=1}^{L}\frac{b_{0k}+b_{1k}z^{-1}+b_{2k}z^{-2}}{1+a_{1k}z^{-1}+a_{2k}z^{-2}}$$

The number *L* of rows of the matrix sos is the closest integer greater than or equal to the maximum of *n*/2 and *m*/2.

[sos,g] = zp2sos(z,p,k,'*order*') specifies the order of the rows in sos, where '*order*' is

- 'down', to order the sections so the first row of sos contains the poles closest to the unit circle

- 'up', to order the sections so the first row of sos contains the poles farthest from the unit circle (default)

[sos,g] = zp2sos(z,p,k,'*order*','*scale*') specifies the desired scaling of the gain and the numerator coefficients of all second-order sections, where '*scale*' is

- 'none', to apply no scaling (default)

- 'inf', to apply infinity-norm scaling

- 'two', to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

Note Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

[sos,g] = zp2sos(z,p,k,'*order*','*scale*',zeroflag) specifies whether to keep together real zeros that are the negatives of each other instead of ordering them according to proximity to poles. Setting zeroflag to true keeps the zeros together and results in a numerator with a middle coefficient equal to zero. The default for zeroflag is false.

sos = zp2sos(...) embeds the overall system gain, g, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

> **Note** Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use ss2sos with two outputs.

Examples

Find a second-order section form of a Butterworth lowpass filter:

```
[z,p,k] = butter(5,0.2);
sos = zp2sos(z,p,k);
```

Algorithms

zp2sos uses a four-step algorithm to determine the second-order section representation for an input zero-pole-gain system:

1 It groups the zeros and poles into complex conjugate pairs using the cplxpair function.

2 It forms the second-order section by matching the pole and zero pairs according to the following rules:

   a Match the poles closest to the unit circle with the zeros closest to those poles.

   b Match the poles next closest to the unit circle with the zeros closest to those poles.

   c Continue until all of the poles and zeros are matched.

   zp2sos groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.

3 It orders the sections according to the proximity of the pole pairs to the unit circle. zp2sos normally orders the sections with poles closest to the unit circle last in the cascade. You can tell zp2sos to order the sections in the reverse order by specifying the down flag.

4 zp2sos scales the sections by the norm specified in the '*scale*' argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\|H\|_p = \left[\frac{1}{2\pi}\int\limits_0^{2\pi} |H(\omega)|^p\ d\omega\right]^{\frac{1}{p}}$$

where p can be either ∞ or 2. See the references for details on the scaling. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

References

[1] Jackson, L.B., *Digital Filters and Signal Processing*, 3rd ed., Kluwer Academic Publishers, Boston, 1996, Chapter 11.

[2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998, Chapter 9.

[3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

See Also   cplxpair | filternorm | sos2zp | ss2sos | tf2sos | zp2ss | zp2tf

zp2ss

Purpose
Convert zero-pole-gain filter parameters to state-space form

Syntax
`[A,B,C,D] = zp2ss(z,p,k)`

Description
`zp2ss` converts a zero-pole-gain representation of a given system to an equivalent state-space representation.

`[A,B,C,D] = zp2ss(z,p,k)` finds a single input, multiple output, state-space representation

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

given a system in factored transfer function form.

$$H(s) = \frac{Z(s)}{P(s)} = k \frac{(s-z_1)(s-z_2)\cdots(s-z_n)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

Column vector `p` specifies the pole locations, and matrix `z` the zero locations with as many columns as there are outputs. The gains for each numerator transfer function are in vector `k`. The A, B, C, and D matrices are returned in controller canonical form.

`Inf` values may be used as place holders in `z` if some columns have fewer zeros than others.

Algorithms
`zp2ss`, for single-input systems, groups complex pairs together into two-by-two blocks down the diagonal of the A matrix. This requires the zeros and poles to be real or complex conjugate pairs.

See Also
`sos2ss` | `ss2zp` | `tf2ss` | `zp2sos` | `zp2tf`

Purpose   Convert zero-pole-gain filter parameters to transfer function form

Syntax   `[b,a] = zp2tf(z,p,k)`

Description `zp2tf` forms transfer function polynomials from the zeros, poles, and gains of a system in factored form.

`[b,a] = zp2tf(z,p,k)` finds a rational transfer function

$$\frac{B(s)}{A(s)} = \frac{b_1 s^{(n-1)} + \cdots + b_{(n-1)}s + b_n}{a_1 s^{(m-1)} + \cdots + a_{(m-1)}s + a_m}$$

given a system in factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s - z_1)(s - z_2)\cdots(s - z_m)}{(s - p_1)(s - p_2)\cdots(s - p_n)}$$

Column vector p specifies the pole locations, and matrix z specifies the zero locations, with as many columns as there are outputs. The gains for each numerator transfer function are in vector k. The zeros and poles must be real or come in complex conjugate pairs. The polynomial denominator coefficients are returned in row vector a and the polynomial numerator coefficients are returned in matrix b, which has as many rows as there are columns of z.

Inf values can be used as place holders in z if some columns have fewer zeros than others.

Algorithms   The system is converted to transfer function form using poly with p and the columns of z.

See Also   sos2tf | ss2tf | tf2zp | tf2zpk | zp2sos | zp2ss

zplane

Purpose	Zero-pole plot

Syntax

```
zplane(z,p)
zplane(b,a)
zplane(Hd)
[hz,hp,ht] = zplane(z,p)
```

Description

This function displays the poles and zeros of discrete-time systems.

zplane(z,p) plots the zeros specified in column vector z and the poles specified in column vector p in the current figure window. The symbol 'o' represents a zero and the symbol 'x' represents a pole. The plot includes the unit circle for reference. If z and p are arrays, zplane plots the poles and zeros in the columns of z and p in different colors.

You can override the automatic scaling of zplane using

```
axis([xmin xmax ymin ymax])
```

or

```
set(gca,'ylim',[ymin ymax])
```

or

```
set(gca,'xlim',[xmin xmax])
```

after calling zplane. This is useful in the case where one or a few of the zeros or poles have such a large magnitude that the others are grouped tightly around the origin and are hard to distinguish.

zplane(b,a) where b and a are row vectors, first uses roots to find the zeros and poles of the transfer function represented by numerator coefficients b and denominator coefficients a. The transfer function is defined in terms of z^{-1}:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}}$$

zplane(Hd) finds the zeros and poles of the transfer function represented by the dfilt filter object Hd. The pole-zero plot is displayed in fvtool.

[hz,hp,ht] = zplane(z,p) returns vectors of handles to the zero lines, hz, and the pole lines, hp. ht is a vector of handles to the axes/unit circle line and to text objects, which are present when there are multiple zeros or poles. If there are no zeros or no poles, hz or hp is the empty matrix [].

Examples   For data sampled at 1000 Hz, plot the poles and zeros of a 4th-order elliptic lowpass digital filter with cutoff frequency of 200 Hz, 3 dB of ripple in the passband, and 30 dB of attenuation in the stopband:

```
[z,p,k] = ellip(4,3,30,200/500);
zplane(z,p);
title('4th-Order Elliptic Lowpass Digital Filter');
```


To generate the same plot with a transfer function representation of the filter, use:

```
[b,a] = ellip(4,3,30,200/500);    % Transfer function
```

```
zplane(b,a)
```

To generate the same plot using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) use:

```
[b,a] = ellip(4,3,30,200/500);
Hd=dfilt.df1(b,a);
zplane(Hd)
```


See Also   freqz

Index

G

H

I