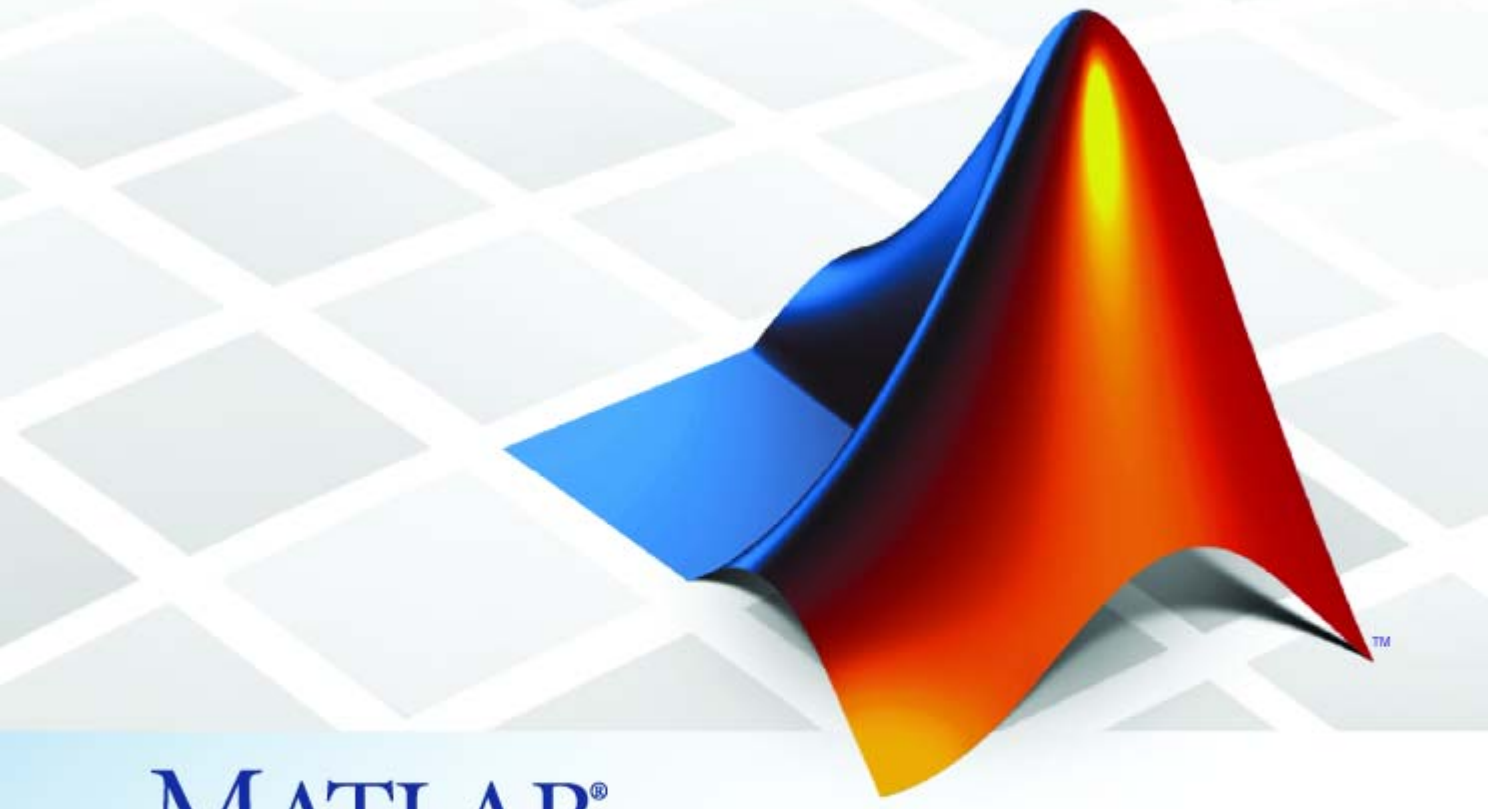
Filter Design Toolbox™ 4

Reference Guide



MATLAB®

The MathWorks™

Accelerating the pace of engineering and science

How to Contact The 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.

Filter Design Toolbox™ Reference Guide

Revision History

March 2007	Online only	New for Version 4.1 (Release 2007a)
September 2007	Online only	New for Version 4.2 (Release 2007b)
March 2008	Online only	New for Version 4.3 (Release 2008a)

Contents

1 Function Reference

Function Reference

Adaptive Filter Constructors

Least Mean Squares (LMS) Based FIR Adaptive Filters (p. 1-2)	Filter with LMS techniques
Recursive Least Squares (RLS) Based FIR Adaptive Filters (p. 1-3)	Filter with RLS techniques
Affine Projection (AP) FIR Adaptive Filters (p. 1-3)	Filter with affine projection
FIR Adaptive Filters in the Frequency Domain (FD) (p. 1-4)	Filter in the frequency domain
Lattice Based (L) FIR Adaptive Filters (p. 1-4)	Filter with lattice filters

Least Mean Squares (LMS) Based FIR Adaptive Filters

`adaptfilt.adjlms`	FIR adaptive filter that uses adjoint LMS algorithm
`adaptfilt.blms`	FIR adaptive filter that uses BLMS
`adaptfilt.blmsfft`	FIR adaptive filter that uses FFT-based BLMS
`adaptfilt.dlms`	FIR adaptive filter that uses delayed LMS
`adaptfilt.filtxlms`	FIR adaptive filter that uses filtered-x LMS
`adaptfilt.lms`	FIR adaptive filter that uses LMS
`adaptfilt.nlms`	FIR adaptive filter that uses NLMS
`adaptfilt.sd`	FIR adaptive filter that uses sign-data algorithm
`adaptfilt.se`	FIR adaptive filter that uses sign-error algorithm

`adaptfilt.ss`	FIR adaptive filter that uses sign-sign algorithm
`adaptfilt.swftf`	FIR adaptive filter that uses sliding window fast transversal LMS

Recursive Least Squares (RLS) Based FIR Adaptive Filters

`adaptfilt.ftf`	Fast transversal LMS adaptive filter
`adaptfilt.hrls`	FIR adaptive filter that uses householder (RLS)
`adaptfilt.hswrls`	FIR adaptive filter that uses householder sliding window RLS
`adaptfilt.qrdrls`	FIR adaptive filter that uses QR-decomposition-based RLS
`adaptfilt.rls`	FIR adaptive filter that uses direct form RLS
`adaptfilt.swrls`	FIR adaptive filter that uses window recursive least squares (RLS)

Affine Projection (AP) FIR Adaptive Filters

`adaptfilt.ap`	FIR adaptive filter that uses direct matrix inversion
`adaptfilt.apru`	FIR adaptive filter that uses recursive matrix updating
`adaptfilt.bap`	FIR adaptive filter that uses block affine projection

FIR Adaptive Filters in the Frequency Domain (FD)

`adaptfilt.fdaf`	FIR adaptive filter that uses frequency-domain with bin step size normalization
`adaptfilt.pbfdaf`	FIR adaptive filter that uses PBFDAF with bin step size normalization
`adaptfilt.pbufdaf`	FIR adaptive filter that uses PBUFDAF with bin step size normalization
`adaptfilt.tdafdct`	Adaptive filter that uses discrete cosine transform
`adaptfilt.tdafdft`	Adaptive filter that uses discrete Fourier transform
`adaptfilt.ufdaf`	FIR adaptive filter that uses unconstrained frequency-domain with quantized step size normalization

Lattice Based (L) FIR Adaptive Filters

`adaptfilt.gal`	FIR adaptive filter that uses gradient lattice
`adaptfilt.lsl`	Adaptive filter that uses LSL
`adaptfilt.qrdlsl`	Adaptive filter that uses QR-decomposition-based LSL

Discrete-Time Filter Constructors

`dfilt.allpass`	Allpass filter
`dfilt.calattice`	Coupled-allpass, lattice filter
`dfilt.calatticepc`	Coupled-allpass, power-complementary lattice filter
`dfilt.cascade`	Cascade of discrete-time filters
`dfilt.cascadeallpass`	Cascade of allpass discrete-time filters
`dfilt.cascadewdfallpass`	Cascade allpass WDF filters to construct allpass WDF
`dfilt.df1`	Discrete-time, direct-form I filter
`dfilt.df1sos`	Discrete-time, SOS direct-form I filter
`dfilt.df1t`	Discrete-time, direct-form I transposed filter
`dfilt.df1tsos`	Discrete-time, SOS direct-form I transposed filter
`dfilt.df2`	Discrete-time, direct-form II filter
`dfilt.df2sos`	Discrete-time, SOS, direct-form II filter
`dfilt.df2t`	Discrete-time, direct-form II transposed filter
`dfilt.df2tsos`	Discrete-time, SOS 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.farrowfd`	Fractional Delay Farrow filter
`dfilt.farrowlinearfd`	Farrow Linear Fractional Delay 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 with maximum phase
`dfilt.latticemamin`	Discrete-time, lattice, moving-average filter with minimum phase
`dfilt.parallel`	Discrete-time, parallel structure filter
`dfilt.scalar`	Discrete-time, scalar filter
`dfilt.wdfallpass`	Wave digital allpass filter

Filter Specification Objects (fdesign) — Response Types

`fdesign.arbmag`	Arbitrary response magnitude filter specification object
`fdesign.arbmagnphase`	Arbitrary response magnitude and phase filter specification object
`fdesign.bandpass`	Bandpass filter specification object
`fdesign.bandstop`	Bandstop filter specification object
`fdesign.ciccomp`	CIC compensator filter specification object
`fdesign.decimator`	Decimator filter specification object
`fdesign.differentiator`	Differentiator filter specification object
`fdesign.halfband`	Halfband filter specification object
`fdesign.highpass`	Highpass filter specification object
`fdesign.hilbert`	Hilbert filter specification object
`fdesign.interpolator`	Interpolator filter specification
`fdesign.isinclp`	Inverse-sinc filter specification
`fdesign.lowpass`	Lowpass filter specification
`fdesign.notch`	Notch filter specification
`fdesign.nyquist`	Nyquist filter specification
`fdesign.octave`	Octave filter specification
`fdesign.peak`	Peak filter specification
`fdesign.rsrc`	Rational-factor sample-rate converter specification

Filter Specification Objects (fdesign) — Design Methods

`cheby1`	Chebyshev Type I filter using specification object
`cheby2`	Chebyshev Type II filter using specification object
`designmethods`	Methods available for designing filter from specification object
`ellip`	Elliptic filter using specification object
`equiripple`	Equiripple single-rate or multirate FIR filter from specification object
`fdesign.polysrc`	Construct polynomial sample-rate converter (POLYSRC) filter designer
`ifir`	Interpolated FIR filter from filter specification
`kaiserwin`	Kaiser window filter from specification object
`multistage`	Multistage filter from specification object
`window`	FIR filter using windowed impulse response

Multirate Filter Constructors

`mfilt.cascade`	Cascade filter objects
`mfilt.cicdecim`	Fixed-point CIC decimator
`mfilt.cicinterp`	Fixed-point CIC interpolator
`mfilt.farrowsrc`	Sample rate converter with arbitrary conversion factor
`mfilt.fftfirinterp`	Overlap-add FIR polyphase interpolator
`mfilt.firdecim`	Direct-form FIR polyphase decimator
`mfilt.firfracdecim`	Direct-form FIR polyphase fractional decimator
`mfilt.firfracinterp`	Direct-form FIR polyphase fractional interpolator
`mfilt.firinterp`	FIR filter-based interpolator
`mfilt.firsrc`	Direct-form FIR polyphase sample rate converter
`mfilt.firtdecim`	Direct-form transposed FIR filter
`mfilt.holdinterp`	FIR hold interpolator
`mfilt.iirdecim`	IIR decimator
`mfilt.iirinterp`	IIR interpolator
`mfilt.iirwdfdecim`	IIR wave digital filter decimator
`mfilt.iirwdfinterp`	IIR wave digital filter interpolator
`mfilt.linearinterp`	Linear interpolator

GUI-Based Filter Design Methods

`fdatool`	Open Filter Design and Analysis Tool
`filterbuilder`	GUI-based filter design

Filter Analysis Methods

`autoscale`	Automatic dynamic range scaling
`block`	Generate block from multirate filter
`coeffs`	Coefficients for filters
`cost`	Cost of using discrete-time or multirate filter
`cumsec`	Vector of SOS filters for cumulative sections
`denormalize`	Undo filter coefficient and gain changes caused by `normalize`
`designmethods`	Methods available for designing filter from specification object
`designopts`	Valid input arguments and values for specification object and method
`disp`	Filter properties and values
`double`	Cast fixed-point filter to use double-precision arithmetic
`euclidfactors`	Euclid factors for multirate filter
`fftcoeffs`	Frequency-domain coefficients
`filter`	Filter data with filter object
`filtstates.cic`	Store CIC filter states
`firtype`	Type of linear phase FIR filter
`freqrespest`	Estimate fixed-point filter frequency response through filtering
`freqrespopts`	`freqrespest` parameters and values
`freqsamp`	Real or complex frequency-sampled FIR filter from specification object
`freqz`	Frequency response of filter
`grpdelay`	Filter group delay

`help`	Help for design method with filter specification
`impz`	Filter impulse response
`isfir`	Determine whether filter is FIR
`islinphase`	Determine whether filter is linear phase
`ismaxphase`	Determine whether filter is maximum phase
`isminphase`	Determine whether filter is minimum phase
`isreal`	Determine whether filter uses real coefficients
`isstable`	Determine whether filter is stable
`limitcycle`	Response of single-rate, fixed-point IIR filter
`maxstep`	Maximum step size for adaptive filter convergence
`measure`	Measure filter magnitude response
`msepred`	Predicted mean-squared error for adaptive filter
`msesim`	Measured mean-squared error for adaptive filter
`noisepsd`	Power spectral density of filter output
`noisepsdopts`	Options for running filter output noise PSD
`norm`	P-norm of filter
`normalize`	Normalize filter numerator or feed-forward coefficients
`normalizefreq`	Switch filter specification between normalized frequency and absolute frequency

`nstates`	Number of filter states
`order`	Order of fixed-point filter
`phasedelay`	Phase delay of filter
`phasez`	Unwrapped phase response for filter
`polyphase`	Polyphase decomposition of multirate filter
`qreport`	Most recent fixed-point filtering operation report
`realizemdl`	Simulink® subsystem block for filter
`reffilter`	Reference filter for fixed-point or single-precision filter
`reorder`	Rearrange sections in SOS filter
`reset`	Reset filter properties to initial conditions
`scale`	Scale sections of SOS filter
`scalecheck`	Check scaling of SOS filter
`set2int`	Configure filter for integer filtering
`setspecs`	Specifications for filter specification object
`specifyall`	Fixed-point scaling modes in direct-form FIR filter
`stepz`	Step response for filter
`validstructures`	Structures for specification object with design method
`zerophase`	Zero-phase response for filter
`zplane`	Zero-pole plot for filter

To see the full listing of analysis methods that apply to the `adaptfilt`, `dfilt`, or `mfilt` objects, enter `help adaptfilt`, `help dfilt`, or `help mfilt` at the MATLAB® prompt.

Fixed-Point Filter Construction and Properties

`cell2sos`	Convert cell array to SOS matrix
`get`	Properties of quantized filter
`isreal`	Test if filter coefficients are real
`reset`	Reset properties of quantized filter to initial values
`scale`	Scale sections of SOS filters
`scalecheck`	Check scaling of SOS filter
`scaleopts`	Scaling options for second-order section scaling
`set`	Properties of quantized filter
`sos`	Convert quantized filter to SOS form, order, and scale
`sos2cell`	Convert SOS matrix to cell array

Quantized Filter Analysis Functions

`freqz`	Frequency response of filter
`impz`	Filter impulse response
`isallpass`	Determine whether filter is allpass
`isfir`	Determine whether filter is FIR
`islinphase`	Determine whether filter is linear phase
`ismaxphase`	Determine whether filter is maximum phase
`isminphase`	Determine whether filter is minimum phase
`isreal`	Determine whether filter uses real coefficients
`issos`	Determine whether filter is SOS form
`isstable`	Determine whether filter is stable
`noisepsd`	Power spectral density of filter output
`noisepsdopts`	Options for running filter output noise PSD
`zplane`	Zero-pole plot for filter

SOS Conversion Functions

`cell2sos`	Convert a cell array to a second-order sections matrix
`sos`	Convert a quantized filter to second-order sections form, order, and scale
`sos2cell`	Convert a second-order sections matrix to a cell array

Filter Design Functions

`farrow`	Farrow filter
`fdatool`	Open Filter Design and Analysis Tool
`filterbuilder`	GUI-based filter design
`fircband`	Constrained-band equiripple FIR filter
`firceqrip`	Constrained, equiripple FIR filter
`fireqint`	Equiripple FIR interpolators
`firgr`	Parks-McClellan FIR filter
`firhalfband`	Halfband FIR filter
`firlpnorm`	Least P-norm optimal FIR filter
`firminphase`	Minimum-phase FIR spectral factor
`firnyquist`	Lowpass Nyquist (Lth-band) FIR filter
`ifir`	Interpolated FIR filter from filter specification
`iircomb`	IIR comb notch or peak filter
`iirgrpdelay`	Optimal IIR filter with prescribed group-delay
`iirlpnorm`	Least P-norm optimal IIR filter
`iirlpnormc`	Constrained least Pth-norm optimal IIR filter
`iirnotch`	Second-order IIR notch filter
`iirpeak`	Second-order IIR peak or resonator filter

Filter Conversion Functions

`ca2tf`	Convert coupled allpass filter to transfer function form
`cl2tf`	Convert coupled allpass lattice to transfer function form
`convert`	Convert filter structure of discrete-time or multirate filter
`firlp2hp`	Convert FIR lowpass filter to Type I FIR highpass filter
`firlp2lp`	Convert FIR Type I lowpass to FIR Type 1 lowpass with inverse bandwidth
`iirlp2bp`	Transform IIR lowpass filter to IIR bandpass filter
`iirlp2bs`	Transform IIR lowpass filter to IIR bandstop filter
`iirlp2hp`	Transform lowpass IIR filter to highpass filter
`iirlp2lp`	Transform lowpass IIR filter to different lowpass filter
`iirpowcomp`	Power complementary IIR filter
`set2int`	Configure filter for integer filtering
`tf2ca`	Transfer function to coupled allpass
`tf2cl`	Transfer function to coupled allpass lattice

2

Functions — Alphabetical List

Purpose

Adaptive filter

Syntax

```
ha = adaptfilt.algorithm('input1',input2,...)
```

Description

`ha = adaptfilt.algorithm('input1',input2,...)` returns the adaptive filter object `ha` that uses the adaptive filtering technique specified by *`algorithm`*. When you construct an adaptive filter object, include an *`algorithm`* specifier to implement a specific adaptive filter. Note that you do not enclose the algorithm option in single quotation marks as you do for most strings. To construct an adaptive filter object you must supply an *`algorithm`* string — there is no default algorithm, although every constructor creates a default adaptive filter when you do not provide input arguments such as `input1` or `input2` in the calling syntax.

Algorithms

For adaptive filter (`adaptfilt`) objects, the *`algorithm`* string determines which adaptive filter algorithm your `adaptfilt` object implements. Each available algorithm entry appears in one of the tables along with a brief description of the algorithm. Click on the algorithm in the first column to get more information about the associated adaptive filter technique.

- LMS based adaptive filters
- RLS based adaptive filters
- Affine projection adaptive filters
- Adaptive filters in the frequency domain
- Lattice based adaptive filters

Least Mean Squares (LMS) Based FIR Adaptive Filters

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
`adaptfilt.adjlms`	Use the Adjoint LMS FIR adaptive filter algorithm
`adaptfilt.blms`	Use the Block LMS FIR adaptive filter algorithm
`adaptfilt.blmsfft`	Use the FFT-based Block LMS FIR adaptive filter algorithm
`adaptfilt.dlms`	Use the delayed LMS FIR adaptive filter algorithm
`adaptfilt.filtxlms`	Use the filtered-x LMS FIR adaptive filter algorithm
`adaptfilt.lms`	Use the LMS FIR adaptive filter algorithm
`adaptfilt.nlms`	Use the normalized LMS FIR adaptive filter algorithm
`adaptfilt.sd`	Use the sign-data LMS FIR adaptive filter algorithm
`adaptfilt.se`	Use the sign-error LMS FIR adaptive filter algorithm
`adaptfilt.ss`	Use the sign-sign LMS FIR adaptive filter algorithm

For further information about an adapting algorithm, refer to the reference page for the algorithm.

Recursive Least Squares (RLS) Based FIR Adaptive Filters

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
`adaptfilt.ftf`	Use the fast transversal least squares adaptation algorithm
`adaptfilt.qrdrls`	Use the QR-decomposition RLS adaptation algorithm
`adaptfilt.hrls`	Use the householder RLS adaptation algorithm
`adaptfilt.hswrls`	Use the householder SWRLS adaptation algorithm
`adaptfilt.rls`	Use the recursive-least squares (RLS) adaptation algorithm
`adaptfilt.swrls`	Use the sliding window (SW) RLS adaptation algorithm
`adaptfilt.swftf`	Use the sliding window FTF adaptation algorithm

For more complete information about an adapting algorithm, refer to the reference page for the algorithm.

Affine Projection (AP) FIR Adaptive Filters

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
`adaptfilt.ap`	Use the affine projection algorithm that uses direct matrix inversion
`adaptfilt.apru`	Use the affine projection algorithm that uses recursive matrix updating
`adaptfilt.bap`	Use the block affine projection adaptation algorithm

To find more information about an adapting algorithm, refer to the reference page for the algorithm.

FIR Adaptive Filters in the Frequency Domain (FD)

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
`adaptfilt.fdaf`	Use the frequency domain adaptation algorithm
`adaptfilt.pbfdaf`	Use the partition block version of the FDAF algorithm
`adaptfilt.pbufdaf`	Use the partition block unconstrained version of the FDAF algorithm
`adaptfilt.tdafdct`	Use the transform domain adaptation algorithm using DCT
`adaptfilt.tdafdft`	Use the transform domain adaptation algorithm using DFT
`adaptfilt.ufdaf`	Use the unconstrained FDAF algorithm for adaptation

For more information about an adapting algorithm, refer to the reference page for the algorithm.

Lattice Based (L) FIR Adaptive Filters

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
`adaptfilt.gal`	Use the gradient adaptive lattice filter adaptation algorithm
`adaptfilt.lsl`	Use the least squares lattice adaptation algorithm
`adaptfilt.qrdlsl`	Use the QR decomposition least squares lattice adaptation algorithm

For more information about an adapting algorithm, refer to the reference page for the algorithm.

Properties for All Adaptive Filter Objects

Each reference page for an algorithm and `adaptfilt.algorithm` object specifies which properties apply to the adapting algorithm and how to use them.

Methods for Adaptive Filter Objects

As is true with all objects, methods enable you to perform various operations on `adaptfilt` objects. To use the methods, you apply them to the object handle that you assigned when you constructed the `adaptfilt` object.

Most of the analysis methods that apply to `dfilt` objects also work with `adaptfilt` objects. Methods like `freqz` rely on the filter coefficients in the `adaptfilt` object. Since the coefficients change each time the filter adapts to data, you should view the results of using a method as an analysis of the filter at a moment in time for the object. Use caution when you apply an analysis method to your adaptive filter objects — always check that your result approached your expectation.

In particular, the Filter Visualization Tool (FVTool) supports all of the `adaptfilt` objects. Analyzing and viewing your `adaptfilt` objects is straightforward — use the `fvtool` method with the name of your object

```
fvtool(objectname)
```

to launch FVTool and work with your object.

Some methods share their names with functions in Signal Processing Toolbox™ software, or even functions in this toolbox. Functions that share names with methods behave in a similar way. Using the same name for more than one function or method is called *overloading* and is common in many toolboxes.

Method	Description
adaptfilt/coefficients	Return the instantaneous adaptive filter coefficients
adaptfilt/filter	Apply an adaptfilt object to your signal
adaptfilt/freqz	Plot the instantaneous adaptive filter frequency response
adaptfilt/grpdelay	Plot the instantaneous adaptive filter group delay
adaptfilt/impz	Plot the instantaneous adaptive filter impulse response.
adaptfilt/info	Return the adaptive filter information.
adaptfilt/isfir	Test whether an adaptive filter is an finite impulse response (FIR) filters.
adaptfilt/islinphase	Test whether an adaptive filter is linear phase
adaptfilt/ismaxphase	Test whether an adaptive filter is maximum phase
adaptfilt/isminphase	Test whether an adaptive filter is minimum phase
adaptfilt/isreal	True whether an adaptive filter has real coefficients
adaptfilt/isstable	Test whether an adaptive filter is stable
adaptfilt/maxstep	Return the maximum step size for an adaptive filter
adaptfilt/msepred	Return the predicted mean square error
adaptfilt/msesim	Return the measured mean square error via simulation.

Method	Description
adaptfilt/phasez	Plot the instantaneous adaptive filter phase response
adaptfilt/reset	Reset an adaptive filter to initial conditions
adaptfilt/stepz	Plot the instantaneous adaptive filter step response
adaptfilt/tf	Return the instantaneous adaptive filter transfer function
adaptfilt/zerophase	Plot the instantaneous adaptive filter zerophase response
adaptfilt/zpk	Return a matrix containing the instantaneous adaptive filter zero, pole, and gain values
adaptfilt/zplane	Plot the instantaneous adaptive filter in the Z-plane

Working with Adaptive Filter Objects

The next sections cover viewing and changing the properties of `adaptfilt` objects. Generally, modifying the properties is the same for `adaptfilt`, `dfilt`, and `mfilt` objects and most of the same methods apply to all.

Viewing Object Properties

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

```
get(ha,'property')
```

To see all properties for an object, use

```
get(ha)
```

Changing Object Properties

To set specific properties, use

```
set(ha,'property1',value1,'property2',value2,...)
```

You must use single quotation marks around the property name so MATLAB treats them as strings.

Copying an Object

To create a copy of an object, use `copy`.

```
ha2 = copy(ha)
```

Note Using the syntax `ha2 = ha` copies only the object handle and does not create a new object — `ha` and `ha2` are not independent. When you change the characteristics of `ha2`, those of `ha` change as well.

Using Filter States

Two properties control your adaptive 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.
- `PersistentMemory` — resets the filter before filtering. The default value is `false` which causes the properties that are modified by the filter, such as `coefficients` and `states`, to be reset to the value you specified when you constructed the object, before you use the object to filter data. Setting `PersistentMemory` to `true` allows the object to retain its current properties between filtering operations, rather than resetting the filter to its property values at construction.

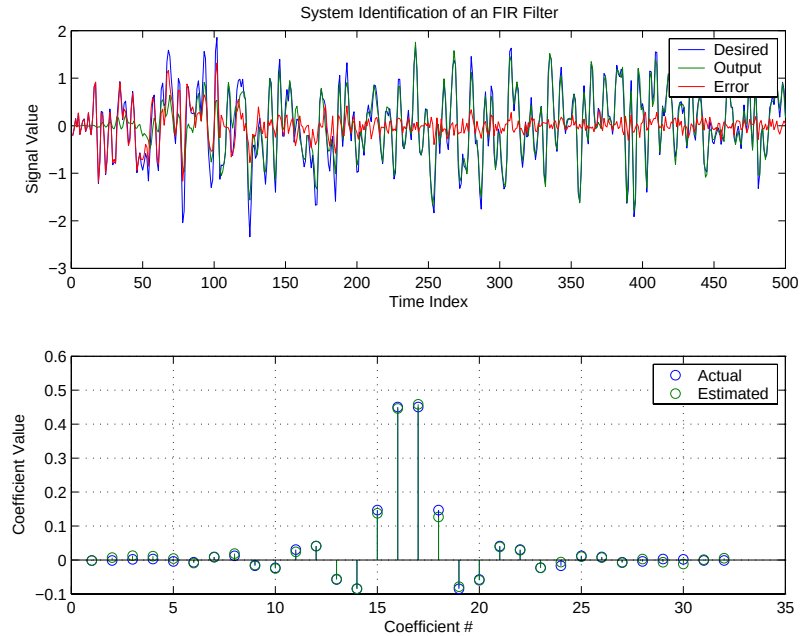
Examples

Construct an LMS adaptive filter object and use it to identify an unknown system. For this example, use 500 iteration of the adapting process to determine the unknown filter coefficients. Using the LMS

algorithm represents one of the most straightforward technique for adaptive filters.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 0.008;            % LMS step size.
ha = adaptfilt.lms(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

Glancing at the figure shows you the coefficients after adapting closely match the desired unknown FIR filter.



See Also

`dfilt`, `filter`, `mfilt`

adaptfilt.adjlms

Purpose

FIR adaptive filter that uses adjoint LMS algorithm

Syntax

```
ha = adaptfilt.adjlms(l,step,leakage,pathcoeffs,pathest,...
errstates,pstates,coeffs,states)
```

Description

`ha = adaptfilt.adjlms(l,step,leakage,pathcoeffs,pathest,... errstates,pstates,coeffs,states)` constructs object `ha`, an FIR adjoint LMS adaptive filter. `l` is the adaptive filter length (the number of coefficients or taps) and must be a positive integer. `l` defaults to 10 when you omit the argument. `step` is the adjoint LMS step size. It must be a nonnegative scalar. When you omit the `step` argument, `step` defaults to 0.1.

`leakage` is the adjoint LMS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, you implement a leaky version of the `adjlms` algorithm to determine the filter coefficients. `leakage` defaults to 1 specifying no leakage in the algorithm.

`pathcoeffs` is the secondary path filter model. This vector should contain the coefficient values of the secondary path from the output actuator to the error sensor.

`pathest` is the estimate of the secondary path filter model. `pathest` defaults to the values in `pathcoeffs`.

`errstates` is a vector of error states of the adaptive filter. It must have a length equal to the filter order of the secondary path model estimate. `errstates` defaults to a vector of zeros of appropriate length. `pstates` contains the secondary path FIR filter states. It must be a vector of length equal to the filter order of the secondary path model. `pstates` defaults to a vector of zeros of appropriate length. The initial filter coefficients for the secondary path filter compose vector `coeffs`. It must be a length `l` vector. `coeffs` defaults to a length `l` vector of zeros. `states` is a vector containing the initial filter states. It must be a vector of length `l+ne-1`, where `ne` is the length of `errstates`. When you omit `states`, it defaults to an appropriate length vector of zeros.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object created. This table lists the properties for the adjoint LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Specifies the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Length l vector with zeros for all elements	Adjoint LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need `l` entries in `coefficients`. Updated filter coefficients are returned in `coefficients` when you use `s` as an output argument.
`ErrorStates`	[0,...,0]	A vector of the error states for your adaptive filter, with length equal to the order of your secondary path filter.
`FilterLength`	10	The number of coefficients in your adaptive filter.
`Leakage`	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
`SecondaryPathCoeffs`	No default	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.
`SecondaryPathEstimate`	`pathcoeffs` values	An estimate of the secondary path filter model.

Property	Default Value	Description
`SecondaryPathStates`	Length of the secondary path filter. All elements are zeros.	The states of the secondary path filter, the unknown system
`States`	l+ne+1, where ne is `length(errstates)`	Contains the initial conditions for your adaptive filter and returns the states of the FIR filter after adaptation. If omitted, it defaults to a zero vector of length equal to l+ne+1. When you use `adaptfilt.adjlms` in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.
`Stepsize`	0.1	Sets the adjoint LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.

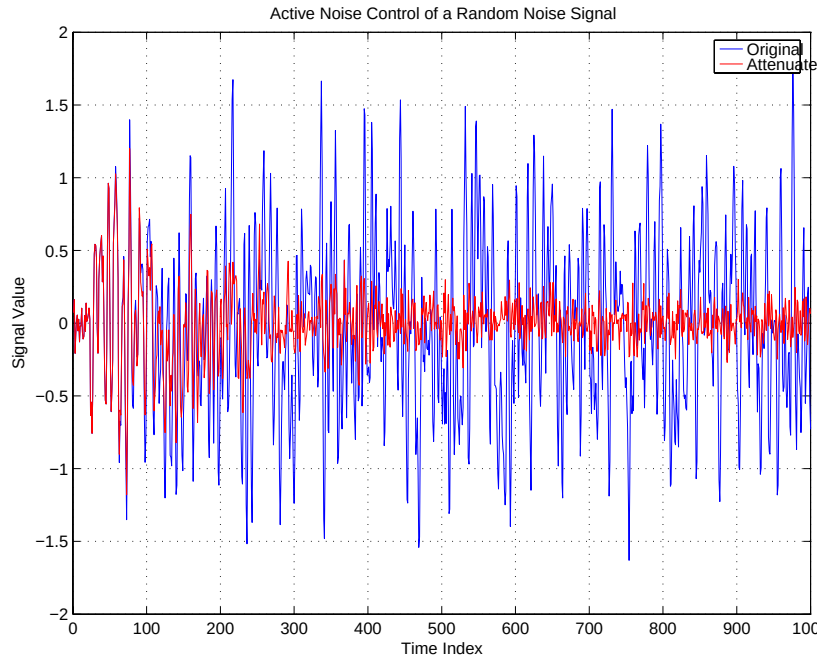
Example

Demonstrate active noise control of a random noise signal that runs for 1000 samples.

```
x  = randn(1,1000);       % Noise source
g  = fir1(47,0.4);        % FIR primary path system model
```

```
n  = 0.1*randn(1,1000);  % Observation noise signal
d  = filter(g,1,x)+n;    % Signal to be canceled (desired)
b  = fir1(31,0.5);       % FIR secondary path system model
mu = 0.008;              % Adjoint LMS step size
ha = adaptfilt.adjlms(32,mu,1,b);
[y,e] = filter(ha,x,d);
plot(1:1000,d,'b',1:1000,e,'r');
title('Active Noise Control of a Random Noise Signal');
legend('Original','Attenuated');
xlabel('Time Index'); ylabel('Signal Value');  grid on;
```

Reviewing the figure shows that the adaptive filter attenuates the original noise signal as you expect.



See Also

`adaptfilt.dlms`, `adaptfilt.filtxlms`

References

Wan, Eric., “Adjoint LMS: An Alternative to Filtered-X LMS and Multiple Error LMS,” Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP), pp. 1841-1845, 1997

Purpose

FIR adaptive filter that uses direct matrix inversion

Syntax

```
ha = adaptfilt.ap(l,step,projectord,offset,coeffs,states,...
errstates,epsstates)
```

Description

`ha = adaptfilt.ap(l,step,projectord,offset,coeffs,states,... errstates,epsstates)` constructs an affine projection FIR adaptive filter `ha` using direct matrix inversion.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ap`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Affine projection step size. This is a scalar that should be a value between zero and one. Setting `step` equal to one provides the fastest convergence during adaptation. `step` defaults to 1.
`projectord`	Projection order of the affine projection algorithm. `projectord` defines the size of the input signal covariance matrix and defaults to two.
`offset`	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. `offset` should be positive. `offset` defaults to one.

Input Argument	Description
`coeffs`	Vector containing the initial filter coefficients. It must be a length `l` vector, the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`states`	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l + projectord - 2`).
`errstates`	Vector of the adaptive filter error states. `errstates` defaults to a zero vector with length equal to (`projectord - 1`).
`epsstates`	Vector of the epsilon values of the adaptive filter. `epsstates` defaults to a vector of zeros with (`projectord - 1`) elements.

Properties

Since your `adaptfilt.ap` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.ap` objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Name	Range	Description
`ProjectionOrder`	1 to as large as needed.	Projection order of the affine projection algorithm. `ProjectionOrder` defines the size of the input signal covariance matrix and defaults to two.
`OffsetCov`	Matrix of values	Contains the offset covariance matrix
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector, the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`ErrorStates`	Vector of elements	Vector of the adaptive filter error states. `errstates` defaults to a zero vector with length equal to (`projectord` - 1).
`EpsilonStates`	Vector of elements	Vector of the epsilon values of the adaptive filter. `epsstates` defaults to a vector of zeros with (`projectord` - 1) elements.

Name	Range	Description
`StepSize`	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true`.

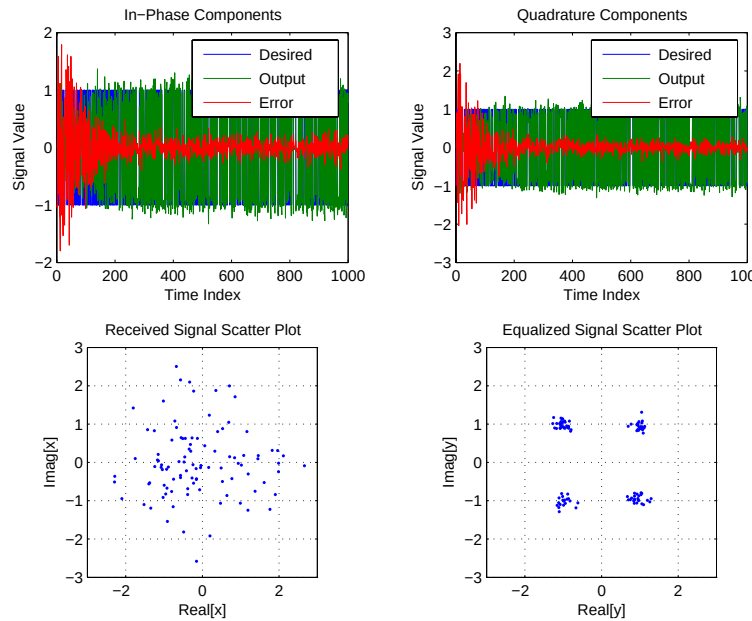
Example

Quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. Run the adaptation for 1000 iterations.

```
D  = 16;                    % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];  % Numerator coefficients of channel
a  = [1 -0.7];              % Denominator coefficients of channel
ntr= 1000;                  % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D));% Baseband
                                                   % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;       % Received signal
x  = r(1+D:ntr+D);    % Input signal (received signal)
d  = s(1:ntr);        % Desired signal (delayed QPSK signal)
mu = 0.1;             % Step size
po = 4;               % Projection order
offset = 0.05;        % Offset for covariance matrix
ha = adaptfilt.ap(32,mu,po,offset);
[y,e] = filter(ha,x,d);
```

```
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

The four plots shown reveal the QPSK process at work.



See Also

`msesim`

References

[1] Ozeki, K. and Umeda, T., “An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties,” Electronics and Communications in Japan, vol.67-A, no. 5, pp. 19-27, May 1984

[2] Maruyama, Y., “A Fast Method of Projection Algorithm,” Proc. 1990 IEICE Spring Conf., B-744

Purpose

FIR adaptive filter that uses recursive matrix updating

Syntax

```
ha = adaptfilt.apru(l,step,projectord,offset,coeffs,states,
...errstates,epsstates)
```

Description

`ha = adaptfilt.apru(l,step,projectord,offset,coeffs,states, ...errstates,epsstates)` constructs an affine projection FIR adaptive filter `ha` using recursive matrix updating.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.apru`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps). It must be a positive integer. `l` defaults to 10.
`step`	Affine projection step size. This is a scalar that should be a value between zero and one. Setting `step` equal to one provides the fastest convergence during adaptation. `step` defaults to 1.
`projectord`	Projection order of the affine projection algorithm. `projectord` defines the size of the input signal covariance matrix and defaults to two.
`offset`	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. `offset` should be positive. `offset` defaults to one.
`coeffs`	Vector containing the initial filter coefficients. It must be a length `l` vector, the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.

Input Argument	Description
`states`	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`1 + projectord - 2`).
`errstates`	Vector of the adaptive filter error states. `errstates` defaults to a zero vector with length equal to (`projectord - 1`).
`epsstates`	Vector of the epsilon values of the adaptive filter. `epsstates` defaults to a vector of zeros with (`projectord - 1`) elements.

Properties

Since your `adaptfilt.apru` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.apru` objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ProjectionOrder`	1 to as large as needed.	Projection order of the affine projection algorithm. `ProjectionOrder` defines the size of the input signal covariance matrix and defaults to two.

Name	Range	Description
`OffsetCov`	Matrix of values	Contains the offset covariance matrix
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector, the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`ErrorStates`	Vector of elements	Vector of the adaptive filter error states. `errstates` defaults to a zero vector with length equal to (`projectord` - 1).
`EpsilonStates`	Vector of elements	Vector of the epsilon values of the adaptive filter. `epsstates` defaults to a vector of zeros with (`projectord` - 1) elements.

Name	Range	Description
`StepSize`	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true`.

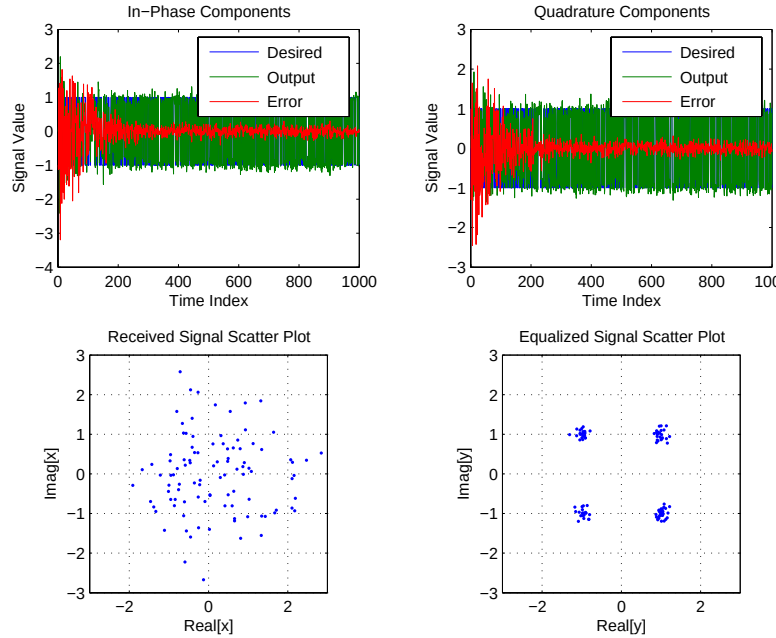
Example

Demonstrate quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. This example runs the adaptation process for 1000 iterations.

```
D  = 16;                        % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
                                                    % QPSK sig
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));  % Noise signal
r  = filter(b,a,s)+n;                          % Received signal
x  = r(1+D:ntr+D);             % Input signal (received signal)
d  = s(1:ntr);                  % Desired signal (delayed QPSK signal)
mu = 0.1;                      % Step size
po = 4;                        % Projection order
del = 0.05;                    % Offset
ha = adaptfilt.apru(32,mu,po,offset);
```

```
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

In the following component and scatter plots, you see the results of QPSK equalization.



See Also

`adaptfilt`, `adaptfilt.ap`, `adaptfilt.bap`

References

[1] Ozeki. K., T. Omeda, “An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties,” Electronics and Communications in Japan, vol. 67-A, no. 5, pp. 19-27, May 1984

[2] Maruyama, Y, “A Fast Method of Projection Algorithm,” Proceedings 1990 IEICE Spring Conference, B-744

Purpose

FIR adaptive filter that uses block affine projection

Syntax

```
ha = adaptfilt.bap(l,step,projectord,offset,coeffs,states)
```

Description

`ha = adaptfilt.bap(l,step,projectord,offset,coeffs,states)` constructs a block affine projection FIR adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.bap`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Affine projection step size. This is a scalar that should be a value between zero and one. Setting `step` equal to one provides the fastest convergence during adaptation. `step` defaults to 1.
`projectord`	Projection order of the affine projection algorithm. `projectord` defines the size of the input signal covariance matrix and defaults to two.
`offset`	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. `offset` should be positive. `offset` defaults to one.

Input Argument	Description
`coeffs`	Vector containing the initial filter coefficients. It must be a length `l` vector, the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`states`	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l + projectord - 2`).

Properties

Since your `adaptfilt.bap` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.bap` objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ProjectionOrder`	1 to as large as needed.	Projection order of the affine projection algorithm. `ProjectionOrder` defines the size of the input signal covariance matrix and defaults to two.
`OffsetCov`	Matrix of values	Contains the offset covariance matrix

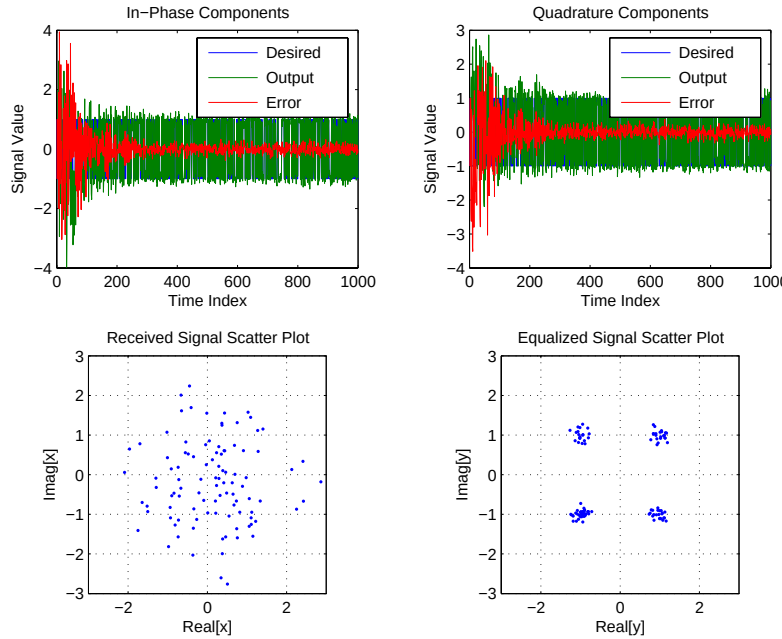
Name	Range	Description
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector, the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l + projectord - 2`).
`StepSize`	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true`.

Example

Show an example of quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D  = 16;                              % Number of samples of delay
```

```
b  = exp(j*pi/4)*[-0.7 1];           % Numerator coefficients of
                                     % channel
a  = [1 -0.7];                       % Denominator coefficients
                                     % of channel
ntr= 1000;                           % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D));  % Baseband
                                     % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));        % Noise signal
r  = filter(b,a,s)+n;                % Received signal
x  = r(1+D:ntr+D);                   % Input signal (received signal)
d  = s(1:ntr);                       % Desired signal (delayed
                                     % QPSK signal)
mu = 0.5;                            % Step size
po = 4;                              % Projection order
offset = 1.0;                       % Offset for covariance matrix
ha = adaptfilt.bap(32,mu,po,offset);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



Using the block affine projection object in QPSK results in the plots shown here.

See Also

`adaptfilt`, `adaptfilt.ap`, `adaptfilt.apru`

References

[1] Ozeki, K. and T. Omeda, “An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties,” Electronics and Communications in Japan, vol. 67-A, no. 5, pp. 19-27, May 1984

[2] Montazeri, M. and Duhamel, P, “A Set of Algorithms Linking NLMS and Block RLS Algorithms,” IEEE Transactions Signal Processing, vol. 43, no. 2, pp, 444-453, February 1995

adaptfilt.blms

Purpose

FIR adaptive filter that uses BLMS

Syntax

```
ha = adaptfilt.blms(l,step,leakage,blocklen,coeffs,states)
```

Description

`ha = adaptfilt.blms(l,step,leakage,blocklen,coeffs,states)` constructs an FIR block LMS adaptive filter `ha`, where `l` is the adaptive filter length (the number of coefficients or taps) and must be a positive integer. `l` defaults to 10.

`step` is the block LMS step size. You must set step to a nonnegative scalar. You can use function `maxstep` to determine a reasonable range of step size values for the signals being processed. When unspecified, `step` defaults to 0.

`leakage` is the block LMS leakage factor. It must be a scalar between 0 and 1. If you set leakage to be less than one, you implement the leaky block LMS algorithm. `leakage` defaults to 1 specifying no leakage in the adapting algorithm.

`blocklen` is the block length used. It must be a positive integer and the signal vectors `d` and `x` should be divisible by `blocklen`. Larger block lengths result in faster per-sample execution times but with poor adaptation characteristics. When you choose `blocklen` such that `blocklen + length(coeffs)` is a power of 2, use `adaptfilt.blmsfft`. `blocklen` defaults to `l`.

`coeffs` is a vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector of zeros.

`states` contains a vector of your initial filter states. It must be a length `l` vector and defaults to a length `l` vector of zeros when you do not include it in your calling function.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object created. This table lists the properties for the adjoint LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `l`
`Leakage`		Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
`BlockLength`	Vector of length `l`	Size of the blocks of data processed in each iteration

Property	Default Value	Description
`StepSize`	0.1	Sets the block LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution. Use `maxstep` to determine the maximum usable step size.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.

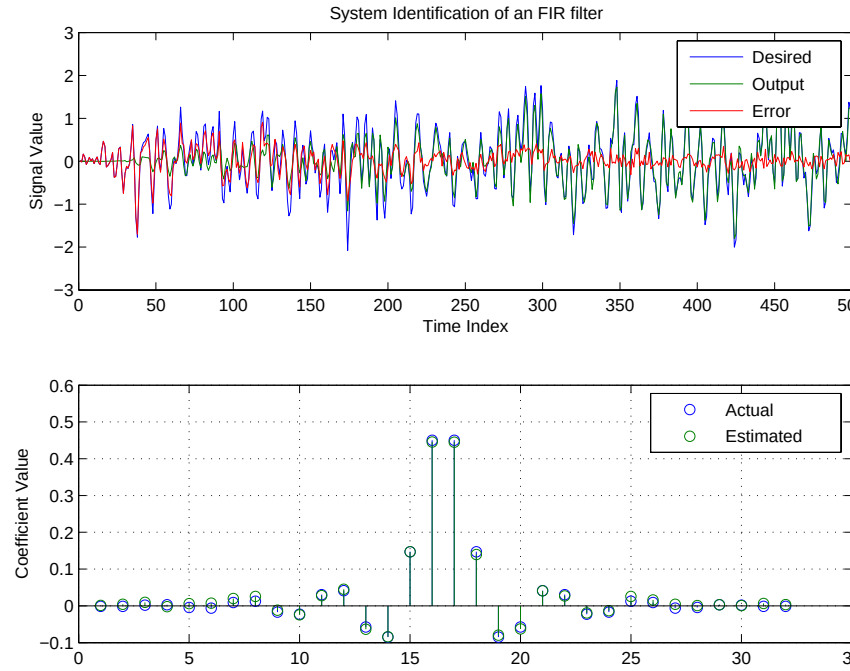
Example

Use an adaptive filter to identify an unknown 32nd-order FIR filter. In this example 500 input samples result in 500 iterations of the adaptation process. You see in the plot that follows the example code that the adaptive filter has determined the coefficients of the unknown system under test.

```
x  = randn(1,500);        % Input to the filter
b  = fir1(31,0.5);        % FIR system to be identified
no  = 0.1*randn(1,500);   % Observation noise signal
d  = filter(b,1,x)+no;    % Desired signal
mu = 0.008;               % Block LMS step size
n  = 5;                   % Block length
ha = adaptfilt.blms(32,mu,1,n);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
```

```
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

Based on looking at the figures here, the adaptive filter correctly identified the unknown system after 500 iterations, or fewer. In the lower plot, you see the comparison between the actual filter coefficients and those determined by the adaptation process.



See Also

`adaptfilt.blmsfft`, `adaptfilt.fdaf`, `adaptfilt.lms`

References

Shynk, J.J.,"Frequency-Domain and Multirate Adaptive Filtering," IEEE® Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

Purpose

FIR adaptive filter that uses FFT-based BLMS

Syntax

```
ha = adaptfilt.blmsfft(l,step,leakage,blocklen,coeffs,
states)
```

Description

`ha = adaptfilt.blmsfft(l,step,leakage,blocklen,coeffs, states)` constructs an FIR block LMS adaptive filter object `ha` where `l` is the adaptive filter length (the number of coefficients or taps) and must be a positive integer. `l` defaults to 10. `step` is the block LMS step size. It must be a nonnegative scalar. The function `maxstep` may be helpful to determine a reasonable range of step size values for the signals you are processing. `step` defaults to 0.

`leakage` is the block LMS leakage factor. It must also be a scalar between 0 and 1. When `leakage` is less than one, the `adaptfilt.blmsfft` implements a leaky block LMS algorithm. `leakage` defaults to 1 (no leakage). `blocklen` is the block length used. It must be a positive integer such that

```
blocklen + length(coeffs)
```

is a power of two; otherwise, an `adaptfilt.blms` algorithm is used for adapting. Larger block lengths result in faster execution times, with poor adaptation characteristics as the cost of the speed gained. `blocklen` defaults to `l`. Enter your initial filter coefficients in `coeffs`, a vector of length `l`. When omitted, `coeffs` defaults to a length `l` vector of all zeros. `states` contains a vector of initial filter states; it must be a length `l` vector. `states` defaults to a length `l` vector of all zeros when you omit the `states` argument in the calling syntax.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for the block LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coefficients` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements of length `l`	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `l`
`Leakage`	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
`BlockLength`	Vector of length `l`	Size of the blocks of data processed in each iteration

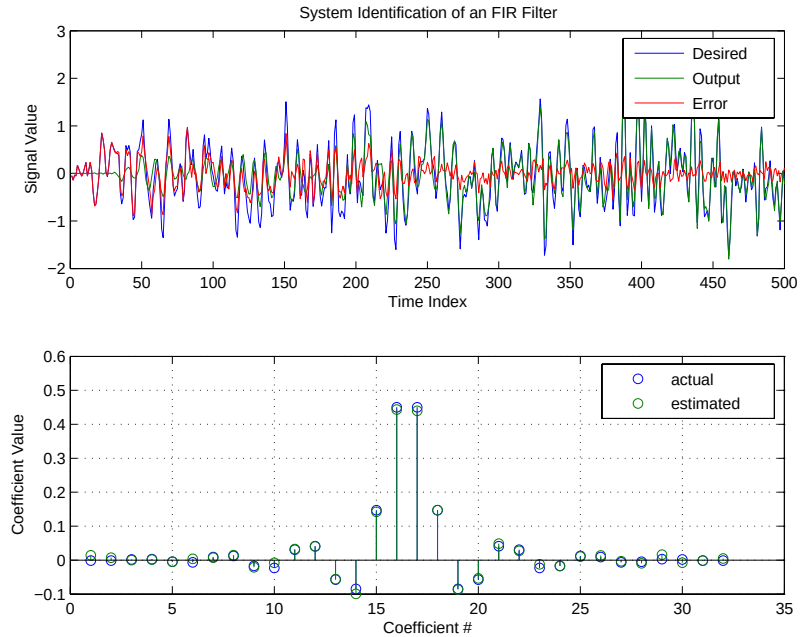
Property	Default Value	Description
`StepSize`	0.1	Sets the block LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution. Use `maxstep` to determine the maximum usable step size.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.

Example

Identify an unknown FIR filter with 32 coefficients using 512 iterations of the adapting algorithm.

```
x  = randn(1,512);       % Input to the filter
b  = fir1(31,0.5);       % FIR system to be identified
no  = 0.1*randn(1,512);  % Observation noise signal
d  = filter(b,1,x)+no;   % Desired signal
mu = 0.008;              % Step size
n  = 16;                 % Block length
ha = adaptfilt.blmsfft(32,mu,1,n);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d(1:500);y(1:500);e(1:500)]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
```

```
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('actual','estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```



As a result of running the adaptation process, filter object `ha` now matches the unknown system FIR filter `b`, based on comparing the filter coefficients derived during adaptation.

See Also

`adaptfilt.blms`, `adaptfilt.fdaf`, `adaptfilt.lms`, `filter`

References

Shynk, J.J., “Frequency-Domain and Multirate Adaptive Filtering,” IEEE® Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

Purpose

FIR adaptive filter that uses delayed LMS

Syntax

```
ha = adaptfilt.dlms(l,step,leakage,delay,errstates,coeffs,
...states)
```

Description

`ha = adaptfilt.dlms(l,step,leakage,delay,errstates,coeffs, ...states)` constructs an FIR delayed LMS adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.dlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	LMS step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.
`leakage`	Your LMS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.lms` implements a leaky LMS algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`delay`	Update delay given in time samples. This scalar should be a positive integer — negative delays do not work. `delay` defaults to 1.
`errstates`	Vector of the error states of your adaptive filter. It must have a length equal to the update delay (`delay`) in samples. `errstates` defaults to an appropriate length vector of zeros.

Input Argument	Description
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l`-1 vector. `states` defaults to a length l-1 vector of zeros.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for the block LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input. LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need `l` entries in `coeffs`.
`Delay`	1	Specifies the update delay for the adaptive algorithm.

Property	Default Value	Description
`ErrorStates`	Vector of zeros with the number of elements equal to `delay`	A vector comprising the error states for the adaptive filter.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
`Leakage`	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.

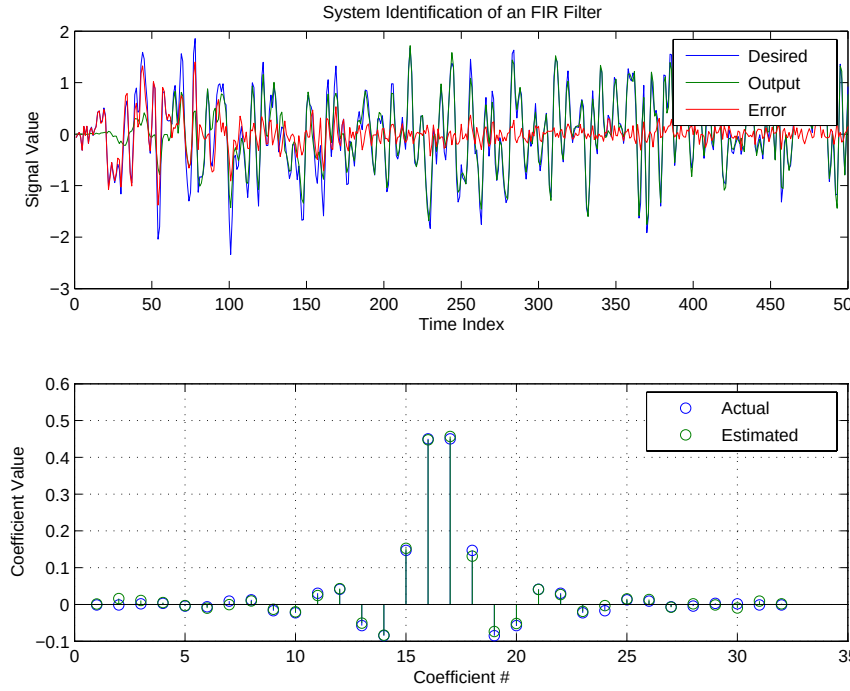
Property	Default Value	Description
`StepSize`	0.1	Sets the LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).

Example

System identification of a 32-coefficient FIR filter. Refer to the figure that follows to see the results of the adapting filter process.

```
x  = randn(1,500);      % Input to the filter
b  = fir1(31,0.5);      % FIR system to be identified
n  = 0.1*randn(1,500);  % Observation noise signal
d  = filter(b,1,x)+n;   % Desired signal
mu = 0.008;             % LMS step size.
delay = 1;              % Update delay
ha = adaptfilt.dlms(32,mu,1,delay);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

Using a delayed LMS adaptive filter in the process to identify an unknown filter appears to work as planned, as shown in this figure.



See Also

`adaptfilt.adjlms`, `adaptfilt.filtxlms`, `adaptfilt.lms`

References

Shynk, J.J.,"Frequency-Domain and Multirate Adaptive Filtering," IEEE® Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

Purpose

FIR adaptive filter that uses frequency-domain with bin step size normalization

Syntax

```
ha = adaptfilt.fdaf(l,step,leakage,delta,lambda,blocklen,
offset,...coeffs,states)
```

Description

`ha = adaptfilt.fdaf(l,step,leakage,delta,lambda,blocklen, offset,...coeffs,states)` constructs a frequency-domain FIR adaptive filter `ha` with bin step size normalization. If you omit all the input arguments you create a default object with l = 10 and `step` = 1.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.fdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps). l must be a positive integer; it defaults to 10 when you omit the argument.
`step`	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.
`leakage`	Leakage parameter of the adaptive filter. If this parameter is set to a value between zero and one, you implement a leaky FDAF algorithm. `leakage` defaults to 1 — no leakage provided in the algorithm.
`delta`	Initial common value of all of the FFT input signal powers. Its initial value should be positive. `delta` defaults to 1.
`lambda`	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. `lambda` should lie in the range (0,1]. `lambda` defaults to 0.9.

Input Argument	Description
`blocklen`	Block length for the coefficient updates. This must be a positive integer. For faster execution, (`blocklen` + l) should be a power of two. `blocklen` defaults to l.
`offset`	Offset for the normalization terms in the coefficient updates. Use this to avoid divide by zeros or by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to zero.
`coeffs`	Initial time-domain coefficients of the adaptive filter. `coeff` should be a length `l` vector. The adaptive filter object uses these coefficients to compute the initial frequency-domain filter coefficients via an FFT computed after zero-padding the time-domain vector by the `blocklen`.
`states`	The adaptive filter states. `states` defaults to a zero vector that has length equal to l.

Properties

Since your `adaptfilt.fdaf` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.fdaf` objects. To show you the properties that apply, this table lists and describes each property for the `adaptfilt.fdaf` filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation.

Name	Range	Description
`AvgFactor`	(0, 1]	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. Same as the input argument `lambda`.
`BlockLength`	Any integer	Block length for the coefficient updates. This must be a positive integer. For faster execution, (`blocklen` + l) should be a power of two. `blocklen` defaults to l.
`FFTCoefficients`		Stores the discrete Fourier transform of the filter coefficients in `coeffs`.
`FFTStates`		States for the FFT operation.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
`Leakage`		Leakage parameter of the adaptive filter. if this parameter is set to a value between zero and one, you implement a leaky FDAF algorithm. `leakage` defaults to 1 — no leakage provided in the algorithm.
`Offset`	Any positive real value	Offset for the normalization terms in the coefficient updates. Use this to avoid dividing by zero or by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to zero.

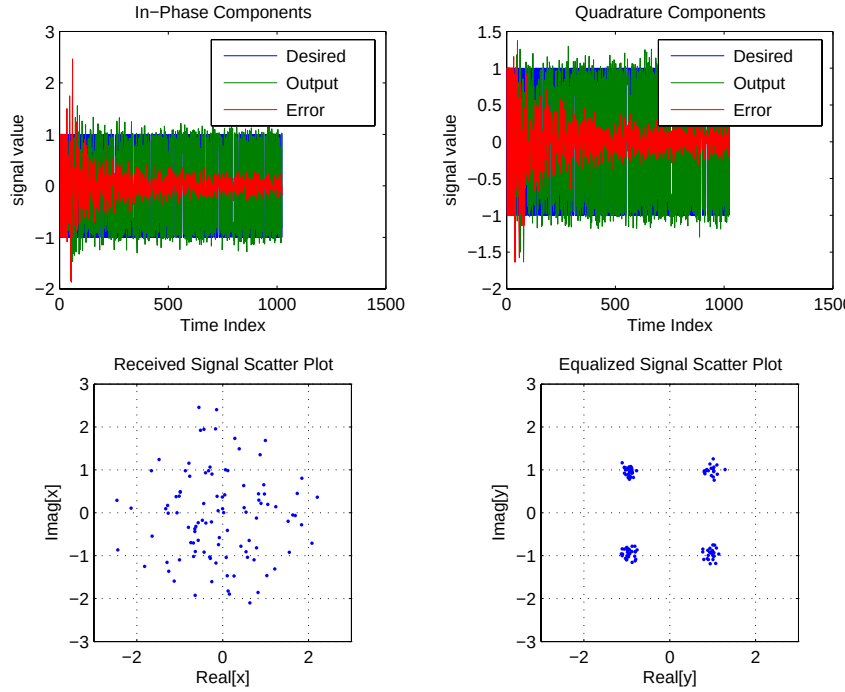
Name	Range	Description
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`Power`		A vector of 2*`l` elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.
`StepSize`	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates

Examples

Quadrature Phase Shift Keying (QPSK) adaptive equalization using 1024 iterations of a 32-coefficient FIR filter. After this example code, a figure demonstrates the equalization results.

```
D = 16;                          % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];       % Numerator coefficients of channel
a  = [1 -0.7];                    % Denominator coefficients of channel
ntr= 1024;                       % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D));  % Baseband
                                 % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));        % Noise signal
r  = filter(b,a,s)+n;             % Received signal
```

```
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                  % Desired signal (delayed QPSK
                                % signal)
del = 1;                        % Initial FFT input powers
mu  = 0.1;                      % Step size
lam = 0.9;                      % Averaging factor
ha = adaptfilt.fdaf(32,mu,1,del,lam);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



See Also

`adaptfilt.ufdaf`, `adaptfilt.pbfdaf`, `adaptfilt.blms`, `adaptfilt.blmsfft`

References

Shynk, J.J.,"Frequency-Domain and Multirate Adaptive Filtering," IEEE® Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992

Purpose

FIR adaptive filter that uses filtered-x LMS

Syntax

```
ha = adaptfilt.filtxlms(l,step,leakage,pathcoeffs,
pathest,...errstates,pstates,coeffs,states)
```

Description

`ha = adaptfilt.filtxlms(l,step,leakage,pathcoeffs, pathest,...errstates,pstates,coeffs,states)` constructs an filtered-x LMS adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.filtxlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Filtered LMS step size. it must be a nonnegative scalar. `step` defaults to 0.1.
`leakage`	is the filtered-x LMS leakage factor. it must be a scalar between 0 and 1. If it is less than one, a leaky version of `adaptfilt.filtxlms` is implemented. `leakage` defaults to 1 (no leakage).
`pathcoeffs`	is the secondary path filter model. this vector should contain the coefficient values of the secondary path from the output actuator to the error sensor.
`pathest`	is the estimate of the secondary path filter model. `pathest` defaults to the values in `pathcoeffs`.
`fstates`	is a vector of filtered input states of the adaptive filter. `fstates` defaults to a zero vector of length equal to (`l` - 1).

Input Argument	Description
`pstates`	are the secondary path FIR filter states. it must be a vector of length equal to the (`length(pathcoeffs)` - 1). `pstates` defaults to a vector of zeros of appropriate length.
`coeffs`	is a vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length l vector of zeros.
`states`	Vector of initial filter states. `states` defaults to a zero vector of length equal to the larger of (`length(pathcoeffs)` - 1) and (`length(pathest)` - 1).

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object created. This table lists the properties for the adjoint LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.

Property	Default Value	Description
`FilteredInputStates`	`l-1`	Vector of filtered input states with length equal to `l - 1`.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`States`	Vector of elements	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `(l + projectord - 2)`
`SecondaryPathCoeffs`	No default	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor
`SecondaryPathEstimate`	`pathcoeffs` values	An estimate of the secondary path filter model

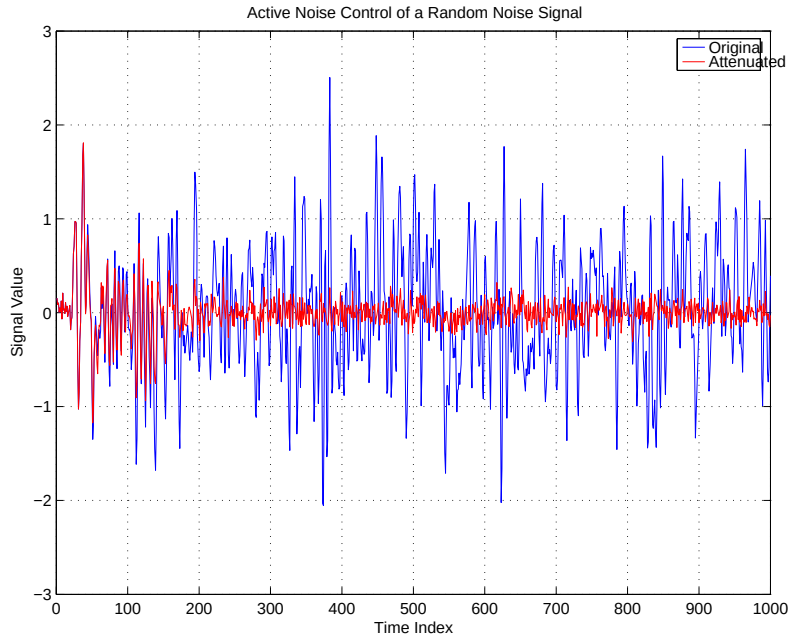
Property	Default Value	Description
`SecondaryPathStates`	Vector of size (`length(pathcoeffs)-1`) with all elements equal to zero.	The states of the secondary path FIR filter — the unknown system
`StepSize`	0.1	Sets the filtered-x algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Example

Demonstrate active noise control of a random noise signal over 1000 iterations.

As the figure that follows this code demonstrates, the filtered-x LMS filter successfully controls random noise in this context.

```
x  = randn(1,1000);      % Noise source
g  = fir1(47,0.4);       % FIR primary path system model
n  = 0.1*randn(1,1000);  % Observation noise signal
d  = filter(g,1,x)+n;    % Signal to be cancelled (desired)
b  = fir1(31,0.5);       % FIR secondary path system model
mu = 0.008;              % Filtered-X LMS step size
ha = adaptfilt.filtxlms(32,mu,1,b);
[y,e] = filter(ha,x,d);
plot(1:1000,d,'b',1:1000,e,'r');
title('Active Noise Control of a Random Noise Signal');
legend('Original','Attenuated');
xlabel('Time Index'); ylabel('Signal Value');  grid on;
```



See also

`adaptfilt.dlms`, `adaptfilt.lms`

References

Shynk J.J., “Frequency-Domain and Multirate Adaptive Filtering,” IEEE® Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

Purpose

Fast transversal LMS adaptive filter

Syntax

```
ha = adaptfilt.ftf(l,lambda,delta,gamma,gstates,coeffs,
states)
```

Description

`ha = adaptfilt.ftf(l,lambda,delta,gamma,gstates,coeffs, states)` constructs a fast transversal least squares adaptive filter object `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ftf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar that should lie in the range (1-0.5/l, 1]. `lambda` defaults to 1.
`delta`	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`gamma`	Conversion factor. `gamma` defaults to one specifying soft-constrained initialization.
`gstates`	States of the Kalman gain updates. `gstates` defaults to a zero vector of length `l`.
`coeffs`	Length `l` vector of initial filter coefficients. coeffs defaults to a length `l` vector of zeros.
`states`	Vector of initial filter States. `states` defaults to a zero vector of length (l-1).

Properties

Since your `adaptfilt.ftf` filter is an object, it has properties that define its operating behavior. Note that many of the properties are also input arguments for creating `adaptfilt.ftf` objects. To show you the properties that apply, this table lists and describes each property for the fast transversal least squares filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`BkwdPrediction`		Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`ConversionFactor`		Conversion factor. Called `gamma` when it is an input argument, it defaults to the matrix [1 -1] that specifies soft-constrained initialization.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`		RLS forgetting factor. This is a scalar that should lie in the range (1-0.5/`l`, 1]. `lambda` defaults to 1.

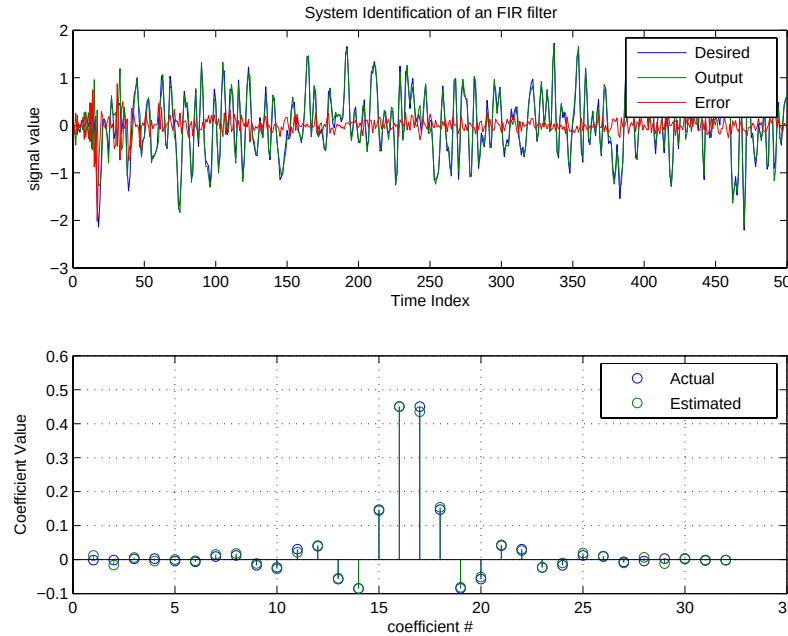
Name	Range	Description
`FwdPrediction`		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
`InitFactor`		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`KalmanGain`		Empty when you construct the object, this gets populated after you run the filter.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).

Examples

System Identification of a 32-coefficient FIR filter by running the identification process for 500 iterations.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
N  = 31;               % Adaptive filter order
lam = 0.99;            % RLS forgetting factor
del = 0.1;             % Soft-constrained
                       %    initialization factor
ha = adaptfilt.ftf(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('coefficient #'); ylabel('Coefficient Value');
grid on;
```

For this example of identifying an unknown system, the figure shows that the adaptation process identifies the filter coefficients for the unknown FIR filter within the first 150 iterations.



See Also

`adaptfilt.swftf`, `adaptfilt.rls`, `adaptfilt.lsl`

References

D.T.M. Slock and Kailath, T., “Numerically Stable Fast Transversal Filters for Recursive Least Squares Adaptive Filtering,” IEEE® Trans. Signal Processing, vol. 38, no. 1, pp. 92-114.

Purpose

FIR adaptive filter that uses gradient lattice

Syntax

```
ha = adaptfilt.gal(l,step,leakage,offset,rstep,delta,
lambda,...rcoeffs,coeffs,states)
```

Description

`ha = adaptfilt.gal(l,step,leakage,offset,rstep,delta, lambda,...rcoeffs,coeffs,states)` constructs a gradient adaptive lattice FIR filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.gal`.

Input Argument	Description
`l`	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the reflection coefficients plus one. `l` defaults to 10.
`step`	Joint process step size of the adaptive filter. This scalar should be a value between zero and one. `step` defaults to 0.
`leakage`	Leakage factor of the adaptive filter. It must be a scalar between 0 and 1. Setting leakage less than one implements a leaky algorithm to estimate both the reflection and the joint process coefficients. `leakage` defaults to 1 (no leakage).
`offset`	Specifies an optional offset for the denominator of the step size normalization term. It must be a scalar greater or equal to zero. A non-zero `offset` is useful to avoid divide-by-near-zero conditions when the input signal amplitude becomes very small. `offset` defaults to 1.

Input Argument	Description
`rstep`	Reflection process step size of the adaptive filter. This scalar should be a value between zero and one. `rstep` defaults to `step`.
`delta`	Initial common value of the forward and backward prediction error powers. It should be a positive value. 0.1 is the default value for `delta`.
`lambda`	Specifies the averaging factor used to compute the exponentially windowed forward and backward prediction error powers for the coefficient updates. `lambda` should lie in the range (0, 1]. `lambda` defaults to the value (1 - `step`).
`rcoeffs`	Vector of initial reflection coefficients. It should be a length (l-1) vector. `rcoeffs` defaults to a zero vector of length (`l`-1).
`coeffs`	Vector of initial joint process filter coefficients. It must be a length `l` vector. `coeffs` defaults to a length l vector of zeros.
`states`	Vector of the backward prediction error states of the adaptive filter. `states` defaults to a zero vector of length (`l`-1).

Properties

Since your `adaptfilt.gal` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.gal` objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation

Name	Range	Description
`AvgFactor`		Specifies the averaging factor used to compute the exponentially-windowed forward and backward prediction error powers for the coefficient updates. Same as the input argument `lambda`.
`BkwdPredErrorPower`		Returns the minimum mean-squared prediction error. Refer to [2] in the bibliography for details about linear prediction
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`FwdPredErrorPower`		Returns the minimum mean-squared prediction error in the forward direction. Refer to [2] in the bibliography for details about linear prediction.

Name	Range	Description
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. If this parameter is set to a value between zero and one, you implement a leaky GAL algorithm. `leakage` defaults to 1 — no leakage provided in the algorithm.
`Offset`		Offset for the normalization terms in the coefficient updates. Use this to avoid dividing by zero or by very small numbers when input signal amplitude becomes very small. `offset` defaults to one.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`ReflectionCoeffs`		Coefficients determined for the reflection portion of the filter during adaptation.

Name	Range	Description
`ReflectionCoeffsStep`		Size of the steps used to determine the reflection coefficients.
`States`	Vector of elements	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`StepSize`	0 to 1	Specifies the step size taken between filter coefficient updates

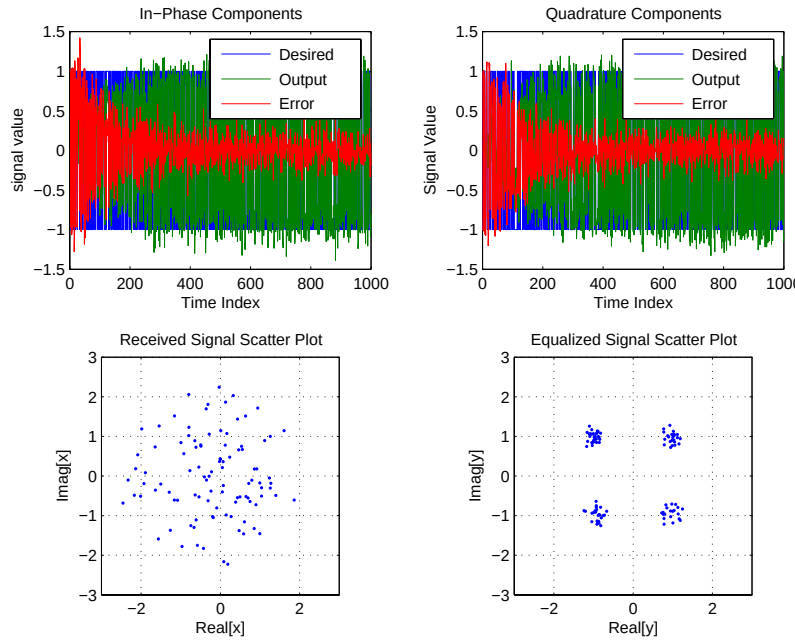
Examples

Perform a Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter over 1000 iterations.

```
D  = 16;                    % Number of delay samples
b  = exp(j*pi/4)*[-0.7 1];  % Numerator coefficients
a  = [1 -0.7];              % Denominator coefficients
ntr= 1000;                  % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
                            % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));   % Noise signal
r  = filter(b,a,s)+n;       % Received signal
x  = r(1+D:ntr+D);          % Input signal (received signal)
d  = s(1:ntr);              % Desired signal (delayed QPSK signal)
L = 32;                     % filter length
mu = 0.007;                 % Step size
ha = adaptfilt.gal(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
```

```
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');
axis('square');
xlabel('Real[x]'); ylabel('Imag[x]');
grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');
axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square');
xlabel('Real[y]'); ylabel('Imag[y]');
grid on;
```

To see the results, look at this figure.



See Also

`adaptfilt.qrdlsl`, `adaptfilt.lsl`, `adaptfilt.tdafdft`

References

Griffiths, L.J. “A Continuously Adaptive Filter Implemented as a Lattice Structure,” Proc. IEEE® Int. Conf. on Acoustics, Speech, and Signal Processing, Hartford, CT, pp. 683-686, 1977

Haykin, S.,*Adaptive Filter Theory*, 3rd Ed., Upper Saddle River, NJ, Prentice Hall, 1996

Purpose

FIR adaptive filter that uses householder (RLS)

Syntax

```
ha = adaptfilt.hrls(l,lambda,sqrtinvcov,coeffs,states)
```

Description

`ha = adaptfilt.hrls(l,lambda,sqrtinvcov,coeffs,states)` constructs an FIR householder RLS adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.hrls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1 meaning the adaptation process retains infinite memory.
`sqrtinvcov`	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
`coeffs`	Vector of initial filter coefficients. It must be a length `l` vector. `coeffs` defaults to being a length `l` vector of zeros.
`states`	Vector of initial filter states. It must be a length `l`-1 vector. `states` defaults to a length `l`-1 vector of zeros.

Properties

Since your `adaptfilt.hrls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.hrls` objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`	Scalar	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. Same as input argument `lambda`. It defaults to 1 meaning the adaptation process retains infinite memory.
`KalmanGain`	Vector of size (1,1)	Empty when you construct the object, this gets populated after you run the filter.

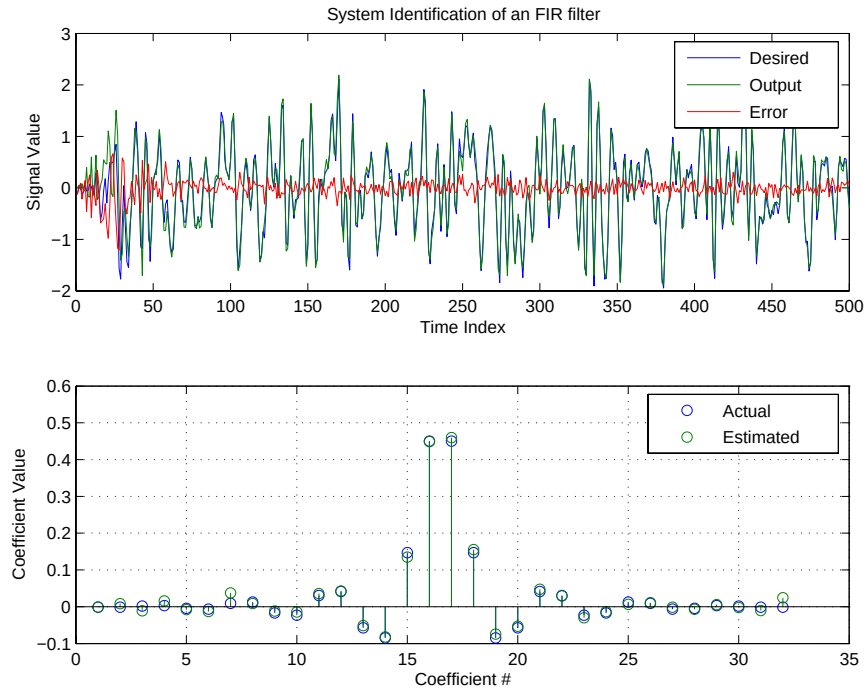
Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. Defaults to false.
SqrtInvCov	Matrix of doubles	Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 - 1).

Examples

Use 500 iterations of an adaptive filter object to identify a 32-coefficient FIR filter system. Both the example code and the resulting figure show the successful filter identification through adaptive filter processing.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
G0 = sqrt(10)*eye(32); % Initial sqrt correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
ha = adaptfilt.hrls(32,lam,G0);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
```

```
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```



See Also

`adaptfilt.rls`, `adaptfilt.qrdrls`, `adaptfilt.hswrls`

Purpose

FIR adaptive filter that uses householder sliding window RLS

Syntax

```
ha = adaptfilt.hswrls(l,lambda,sqrtinvcov,swblocklen,
dstates,coeffs,states)
```

Description

`ha = adaptfilt.hswrls(l,lambda,sqrtinvcov,swblocklen, dstates,coeffs,states)` constructs an FIR householder sliding window recursive-least-square adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.hswrls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	Recursive least square (RLS) forgetting factor. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1 meaning the adaptation process retains infinite memory.
`sqrtinvcov`	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
`swblocklen`	Block length of the sliding window. This integer must be at least as large as the filter length. `swblocklen` defaults to 16.
`dstates`	Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`swblocklen` - 1).

Input Argument	Description
`coeffs`	Vector of initial filter coefficients. It must be a length `l` vector. `coeffs` defaults to being a length `l` vector of zeros.
`states`	Vector of initial filter states. It must be a length `(l + swblocklen - 2)` vector. `states` defaults to a length `(l + swblocklen -2)` vector of zeros.

Properties

Since your `adaptfilt.hswrls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.hswrls` objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`DesiredSignalStates`	Vector	Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`swblocklen` - 1).

Name	Range	Description
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`	Scalar	Root-least-square (RLS) forgetting factor. This is a scalar and should lie in the range (0, 1]. Same as input argument `lambda`. It defaults to 1 meaning the adaptation process retains infinite memory.
`KalmanGain`	(`l`,1) vector	Empty when you construct the object, this gets populated after you run the filter.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. Defaults to `false`.
`SqrtInvCov`	`l`-by-`l` Matrix	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.

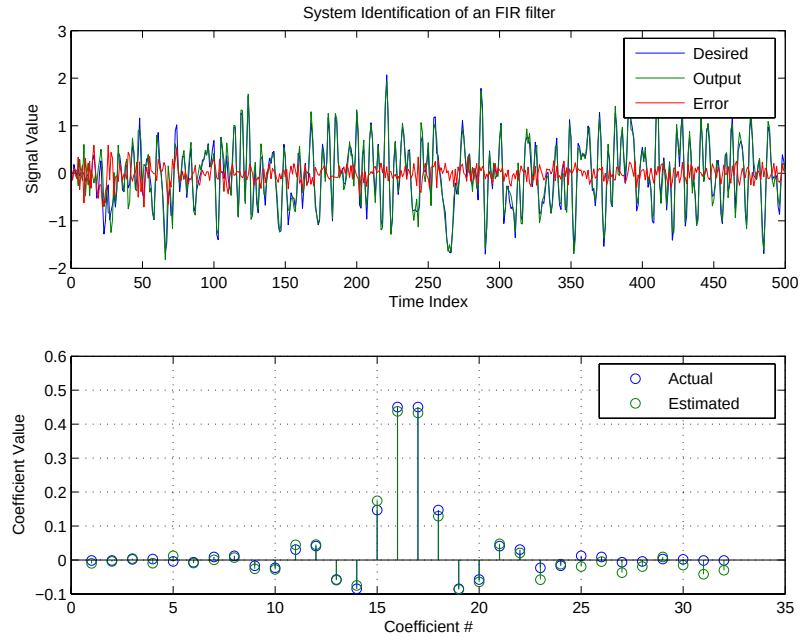
Name	Range	Description
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`SwBlockLength`	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. `swblocklen` defaults to 16.

Examples

System Identification of a 32-coefficient FIR filter.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
G0 = sqrt(10)*eye(32); % Initial sqrt correlation
                       %   matrix inverse
lam = 0.99;            % RLS forgetting factor
N  = 64;               % block length
ha = adaptfilt.hswrls(32,lam,G0,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

In the pair of plots shown in the figure you see the comparison of the desired and actual output for the adapting filter and the coefficients of both filters, the unknown and the adapted.



See Also

`adaptfilt.rls`, `adaptfilt.qrdrls`, `adaptfilt.hrls`

Purpose

FIR adaptive filter that uses LMS

Syntax

```
ha = adaptfilt.lms(l,step,leakage,coeffs,states)
```

Description

`ha = adaptfilt.lms(l,step,leakage,coeffs,states)` constructs an FIR LMS adaptive filter object `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.lms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	LMS step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.1.
`leakage`	Your LMS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.lms` implements a leaky LMS algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l`-1 vector. `states` defaults to a length l-1 vector of zeros.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object created. This table lists the properties for the `adaptfilt.lms` object, their default values, and a brief description of the property.

Property	Range	Property Description
`Algorithm`	None	Reports the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to a length `l` vector of zeros when you do not provide the vector as an input argument.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Leakage`	0 to 1	LMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. `leakage` defaults to 1 (no leakage).

Property	Range	Property Description
`PersistentMemory`	`false` or `true`	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` - 1).
`StepSize`	0 to 1	LMS step size. It must be a scalar between zero and one. Setting this step size value to one provides the fastest convergence. `step` defaults to 0.1.

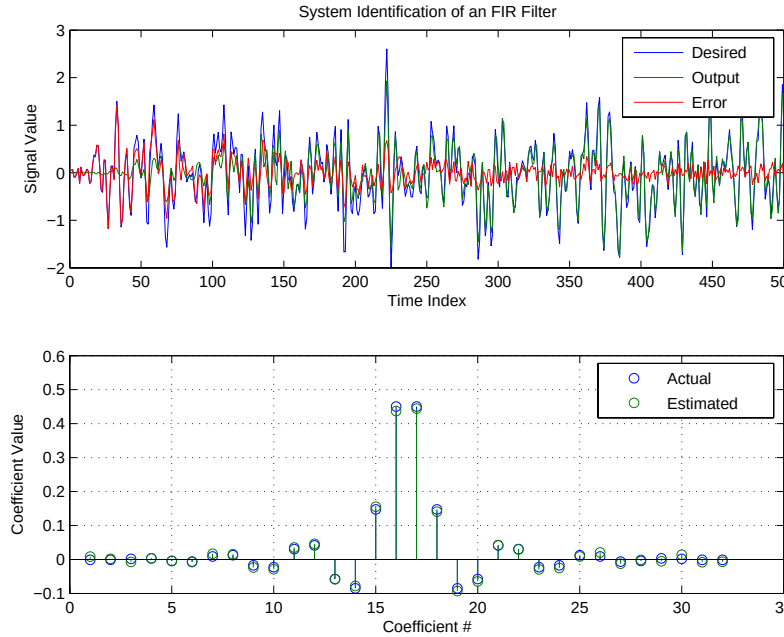
Example

Use 500 iterations of an adapting filter system to identify and unknown 32nd-order FIR filter.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 0.008;            % LMS step size.
ha = adaptfilt.lms(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
```

```
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

Using LMS filters in an adaptive filter architecture is a time honored means for identifying an unknown filter. By running the example code provided you can demonstrate one process to identify an unknown FIR filter.



See Also

`adaptfilt.blms`, `adaptfilt.blmsfft`, `adaptfilt.dlms`, `adaptfilt.nlms`, `adaptfilt.tdafdft`, `adaptfilt.sd`, `adaptfilt.se`, `adaptfilt.ss`

References

Shynk J.J., “Frequency-Domain and Multirate Adaptive Filtering,” IEEE® Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

Purpose

Adaptive filter that uses LSL

Syntax

```
ha = adaptfilt.lsl(l,lambda,delta,coeffs,states)
```

Description

`ha = adaptfilt.lsl(l,lambda,delta,coeffs,states)` constructs a least squares lattice adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.lsl`.

Input Argument	Description
`l`	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the prediction coefficients plus one. L defaults to 10.
`lambda`	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1. `lambda` = 1 denotes infinite memory while adapting to find the new filter.
`delta`	Soft-constrained initialization factor in the least squares lattice algorithm. It should be positive. `delta` defaults to 1.
`coeffs`	Vector of initial joint process filter coefficients. It must be a length `l` vector. `coeffs` defaults to a length `l` vector of all zeros.
`states`	Vector of the backward prediction error states of the adaptive filter. `states` defaults to a length `l` vector of all zeros, specifying soft-constrained initialization for the algorithm.

Properties

Since your `adaptfilt.lsl` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are

also input arguments for creating `adaptfilt.lsl` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation.
`BkwdPrediction`		Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
`ForgettingFactor`		Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting `forgetting factor = 1` denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.

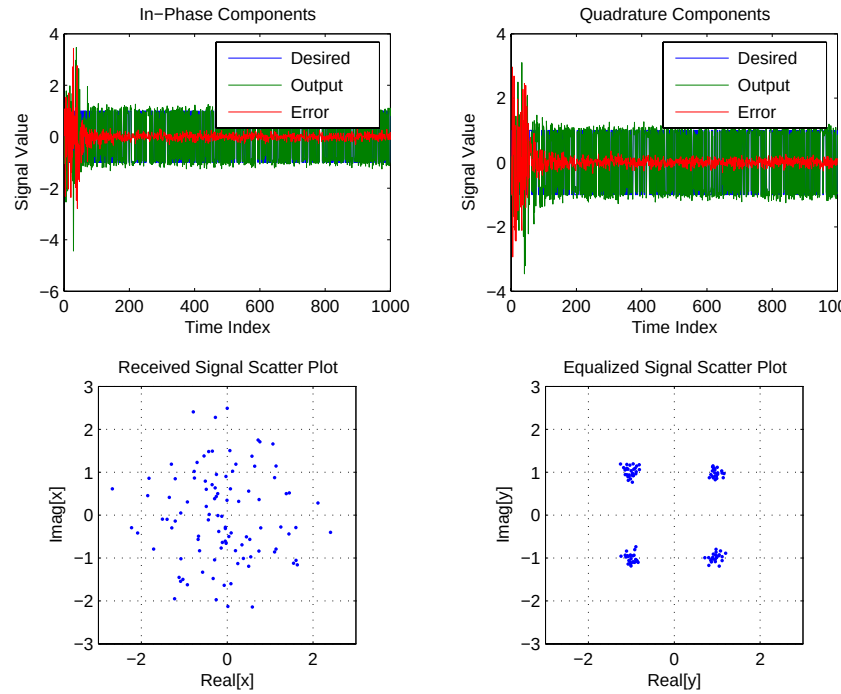
Name	Range	Description
`FwdPrediction`		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
`InitFactor`		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `l`.

Examples

Demonstrate Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter running for 1000 iterations. After you review the example code, the figure shows the results of running the example to use QPSK adaptive equalization with a 32nd-order FIR filter. The error between the in-phase and quadrature components, as shown by the errors plotted in the upper plots, falls to near zero. Also, the equalized signal shows the clear quadrature nature.

```
D  = 16;                      % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];    % Numerator coefficients of channel
a  = [1 -0.7];                % Denominator coefficients of channel
ntr= 1000;                    % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D));% Baseband
                                                   % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;         % Received signal
x  = r(1+D:ntr+D);            % Input signal (received signal)
d  = s(1:ntr);                     % Desired signal (delayed QPSK
                                   % signal)
lam = 0.995;                       % Forgetting factor
del = 1;                             % Soft-constrained initialization
factor
ha = adaptfilt.lsl(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');
axis('square');
xlabel('Real[x]'); ylabel('Imag[x]');
```

```
grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');
axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square');
xlabel('Real[y]'); ylabel('Imag[y]');
grid on;
```



See Also

`adaptfilt.qrdlsl`, `adaptfilt.gal`, `adaptfilt.ftf`, `adaptfilt.rls`

References

Haykin, S., *Adaptive Filter Theory*, 2nd Edition, Prentice Hall, N.J., 1991

Purpose

FIR adaptive filter that uses NLMS

Syntax

```
ha = adaptfilt.nlms(l,step,leakage,offset,coeffs,states)
```

Description

`ha = adaptfilt.nlms(l,step,leakage,offset,coeffs,states)` constructs a normalized least-mean squares (NLMS) FIR adaptive filter object named `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.nlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	NLMS step size. It must be a scalar between 0 and 2. Setting this step size value to one provides the fastest convergence. `step` defaults to 1.
`leakage`	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. `leakage` defaults to 1 (no leakage).
`offset`	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors. Use this to avoid dividing by zero (or by very small numbers) when the square of the input data norm becomes very small (when the input signal amplitude becomes very small). When you omit it, `offset` defaults to zero.

Input Argument	Description
`coeffs`	Vector composed of your initial filter coefficients. Enter a length `l` vector. `coeffs` defaults to a vector of zeros with length equal to the filter order.
`states`	Your initial adaptive filter states appear in the `states` vector. It must be a vector of length `l-1`. `states` defaults to a length `l-1` vector with zeros for all of the elements.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for normalized LMS objects, their default values, and a brief description of the property.

Property	Range	Property Description
`Algorithm`	None	Reports the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps

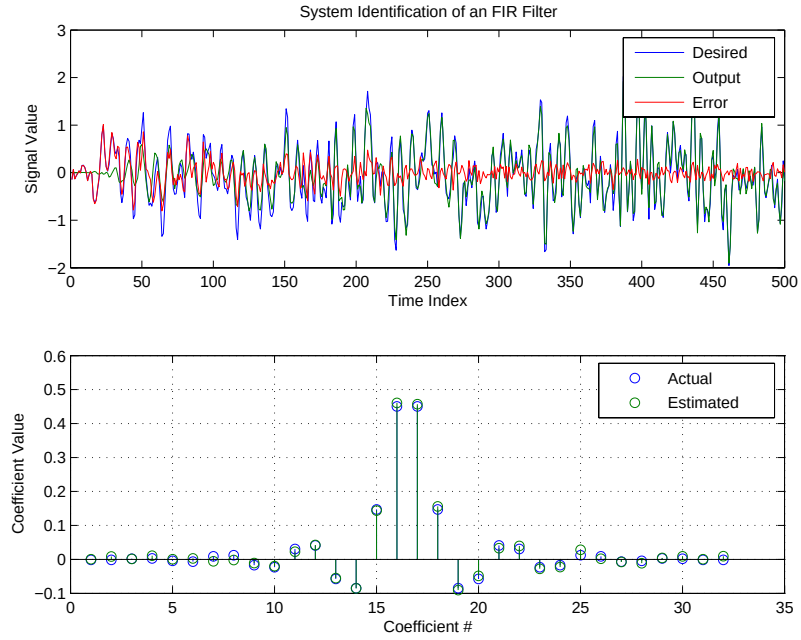
Property	Range	Property Description
`Leakage`	0 to 1	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. `leakage` defaults to 1 (no leakage).
`Offset`	0 or greater	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors. Use this to avoid dividing by zero (or by very small numbers) when the square of the input data norm becomes very small (when the input signal amplitude becomes very small). When you omit it, `offset` defaults to zero.
`PersistentMemory`	`false` or `true`	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.

Property	Range	Property Description
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l - 1).
StepSize	0 to 1	NLMS step size. It must be a scalar between zero and one. Setting this step size value to one provides the fastest convergence. step defaults to one.

Example

To help you compare this algorithm's performance to other LMS-based algorithms, such as BLMS or LMS, this example demonstrates the NLMS adaptive filter in use to identify the coefficients of an unknown FIR filter of order equal to 32 — an example used in other adaptive filter examples.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 1;                % NLMS step size
offset = 50;           % NLMS offset
ha = adaptfilt.nlms(32,mu,1,offset);
[y,e] = filter(ha,x,d);
```



As you see from the figure, the `nlms` variant again closely matches the actual filter coefficients in the unknown FIR filter.

See Also

`adaptfilt.ap`, `adaptfilt.apru`, `adaptfilt.lms`, `adaptfilt.rls`, `adaptfilt.swrls`

Purpose

FIR adaptive filter that uses PBFDAF with bin step size normalization

Syntax

```
ha = adaptfilt.pbfdaf(l,step,leakage,delta,lambda,blocklen,
offset,coeffs,states)
```

Description

`ha = adaptfilt.pbfdaf(l,step,leakage,delta,lambda,blocklen, offset,coeffs,states)` constructs a partitioned block frequency-domain FIR adaptive filter `ha` that uses bin step size normalization during adaptation.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.pbfdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. L defaults to 10.
`step`	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.
`leakage`	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. `leakage` defaults to 1— no leakage.
`delta`	Initial common value of all of the FFT input signal powers. Its initial value should be positive. delta defaults to 1.
`lambda`	Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. `lambda` should lie in the range (0,1]. `lambda` defaults to 0.9.

Input Argument	Description
`blocklen`	Block length for the coefficient updates. This must be a positive integer such that (`l/blocklen`) is also an integer. For faster execution, `blocklen` should be a power of two. `blocklen` defaults to two.
`offset`	Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. `offset` defaults to zero.
`coeffs`	Initial time-domain coefficients of the adaptive filter. It should be a vector of length `l`. The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
`states`	Specifies the filter initial conditions. `states` defaults to a zero vector of length `l`.

Properties

Since your `adaptfilt.pbfdaf` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.pbfdaf` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation.

Name	Range	Description
`AvgFactor`		Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. `AvgFactor` should lie in the range (0,1]. `AvgFactor` defaults to 0.9. Called `lambda` as an input argument.
`BlockLength`		Block length for the coefficient updates. This must be a positive integer such that (`l/blocklen`) is also an integer. For faster execution, `blocklen` should be a power of two. `blocklen` defaults to two.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
`FFTCoefficients`		Stores the discrete Fourier transform of the filter coefficients in `coeffs`.
`FFTStates`		States for the FFT operation.
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. `leakage` defaults to 1 — no leakage.

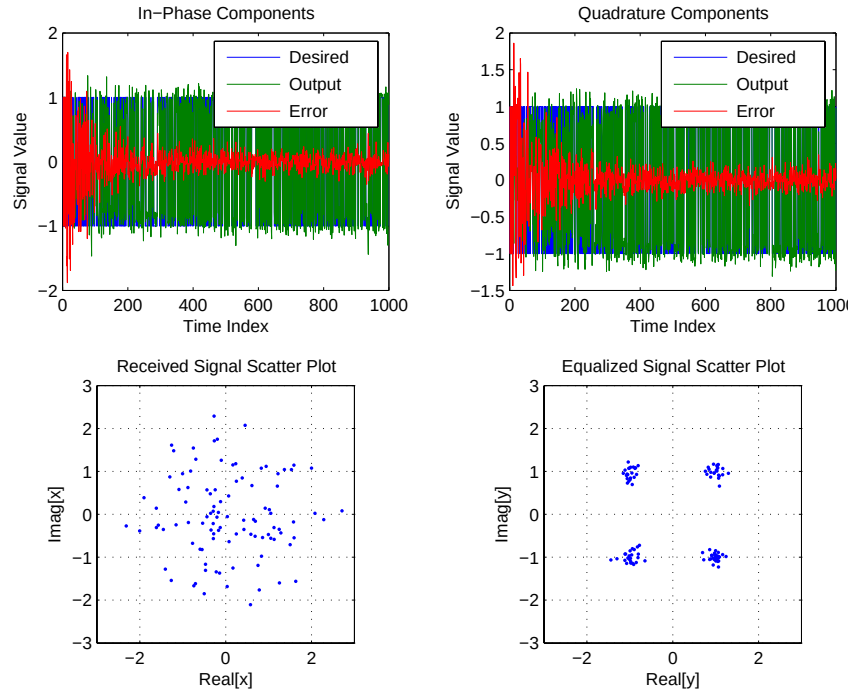
Name	Range	Description
`Offset`		Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. `offset` defaults to zero.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`Power`		A vector of 2*`l` elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.
`StepSize`	0 to 1	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.

Examples

An example of Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D  = 16;                        % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr = 1000;                     % Number of iterations
s  = sign(randn(1,ntr+D))+j*sign(randn(1,ntr+D)); % Baseband
                                                  % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));     % Noise signal
      r  = filter(b,a,s)+n;     % Received signal
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                   % Desired signal (delayed QPSK signal)
del = 1;                         % Initial FFT input powers
mu  = 0.1;                       % Step size
lam = 0.9;                       % Averaging factor
N   = 8;                         % Block size
ha = adaptfilt.pbfdaf(32,mu,1,del,lam,N);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');
axis('square');
xlabel('Real[x]'); ylabel('Imag[x]');
grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');
axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square');
xlabel('Real[y]'); ylabel('Imag[y]');
grid on;
```

In the figure shown, the four subplots provide the details of the results of the QPSK process used in the equalization for this example.



See Also

`adaptfilt.fdaf`, `adaptfilt.pbufdaf`, `adaptfilt.blmsfft`

References

So, J.S. and K.K. Pang, “Multidelay Block Frequency Domain Adaptive Filter,” IEEE® Trans. Acoustics, Speech, and Signal Processing, vol. 38, no. 2, pp. 373-376, February 1990

Paez Borrallo, J.M.and M.G. Otero, “On The Implementation of a Partitioned Block Frequency Domain Adaptive Filter (PBFDAF) For Long Acoustic Echo Cancellation,” Signal Processing, vol. 27, no. 3, pp. 301-315, June 1992

adaptfilt.pbufdaf

Purpose

FIR adaptive filter that uses PBUFDAF with bin step size normalization

Syntax

```
ha = adaptfilt.pbufdaf(l,step,leakage,delta,lambda,
blocklen,...offset,coeffs,states)
```

Description

`ha = adaptfilt.pbufdaf(l,step,leakage,delta,lambda, blocklen,...offset,coeffs,states)` constructs a partitioned block unconstrained frequency-domain FIR adaptive filter `ha` with bin step size normalization.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.pbufdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. L defaults to 10.
`step`	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.
`leakage`	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. `leakage` defaults to 1 — no leakage.
`delta`	Initial common value of all of the FFT input signal powers. Its initial value should be positive. delta defaults to 1.
`lambda`	Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. `lambda` should lie in the range (0,1]. `lambda` defaults to 0.9.

Input Argument	Description
`blocklen`	Block length for the coefficient updates. This must be a positive integer such that `(l/blocklen)` is also an integer. For faster execution, `blocklen` should be a power of two. `blocklen` defaults to two.
`offset`	Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. `offset` defaults to zero.
`coeffs`	Initial time-domain coefficients of the adaptive filter. It should be a vector of length `l`. The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
`states`	Specifies the filter initial conditions. `states` defaults to a zero vector of length `l`.

Properties

Since your `adaptfilt.pbufdaf` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.pbufdaf` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation

Name	Range	Description
`AvgFactor`		Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. `AvgFactor` should lie in the range (0,1]. `AvgFactor` defaults to 0.9. Called `lambda` as an input argument.
`BlockLength`		Block length for the coefficient updates. This must be a positive integer such that (`l/blocklen`) is also an integer. For faster execution, `blocklen` should be a power of two. `blocklen` defaults to two.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`FFTCoefficients`		Stores the discrete Fourier transform of the filter coefficients in `coeffs`.
`FFTStates`		States for the FFT operation.
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. `leakage` defaults to 1 — no leakage.

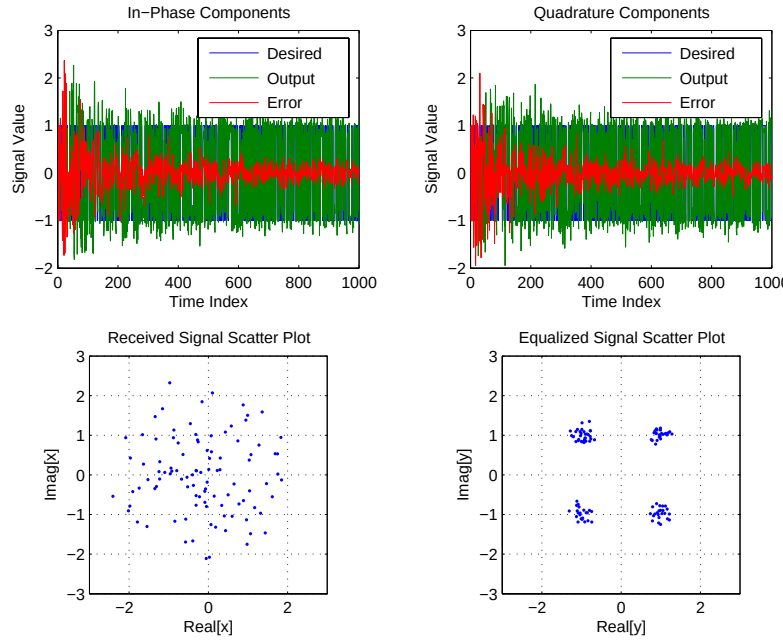
Name	Range	Description
`Offset`		Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small.`voffset` defaults to zero.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`Power`	2*`l` element vector	A vector of 2*`l` elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.
`StepSize`	0 to 1	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.

Examples

Demonstrating Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. To perform the equalization, this example runs for 1000 iterations.

```
D  = 16;                        % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
s  = sign(randn(1,ntr+D))+j*sign(randn(1,ntr+D)); % Baseband QPSK
                                                  % signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D)); % Noise signal
r  = filter(b,a,s)+n;           % Received signal
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                  % Desired signal (delayed QPSK signal)
del = 1;                        % Initial FFT input powers
mu  = 0.1;                      % Step size
lam = 0.9;                      % Averaging factor
N   = 8;                        % Block size
ha = adaptfilt.pbufdaf(32,mu,1,del,lam,N);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

You can compare this algorithm to another, such as the `pbfdaf` version. Use the same example of QPSK adaptation. The following figure shows the results.



See Also

`adaptfilt.ufdaf`, `adaptfilt.pbfdaf`, `adaptfilt.blmsfft`

References

So, J.S. and K.K. Pang, “Multidelay Block Frequency Domain Adaptive Filter,” IEEE® Trans. Acoustics, Speech, and Signal Processing, vol. 38, no. 2, pp. 373-376, February 1990

Paez Borrallo, J.M. and M.G. Otero, “On The Implementation of a Partitioned Block Frequency Domain Adaptive Filter (PBFDAF) for Long Acoustic Echo Cancellation,” Signal Processing, vol. 27, no. 3, pp. 301-315, June 1992

adaptfilt.qrdlsl

Purpose

Adaptive filter that uses QR-decomposition-based LSL

Syntax

```
ha = adaptfilt.qrdlsl(l,lambda,delta,coeffs,states)
```

Description

`ha = adaptfilt.qrdlsl(l,lambda,delta,coeffs,states)` returns a QR-decomposition-based least squares lattice adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.qrdlsl`.

Input Argument	Description
`l`	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the prediction coefficients plus one. L defaults to 10.
`lambda`	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1. `lambda` = 1 denotes infinite memory while adapting to find the new filter.
`delta`	Soft-constrained initialization factor in the least squares lattice algorithm. It should be positive. `delta` defaults to 1.
`coeffs`	Vector of initial joint process filter coefficients. It must be a length `l` vector. `coeffs` defaults to a length `l` vector of all zeros.
`states`	Vector of the angle normalized backward prediction error states of the adaptive filter

Properties

Since your `adaptfilt.qrdlsl` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.qrdlsl` objects. To

show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`BkwdPrediction`		Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`		Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting `forgetting factor = 1` denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.

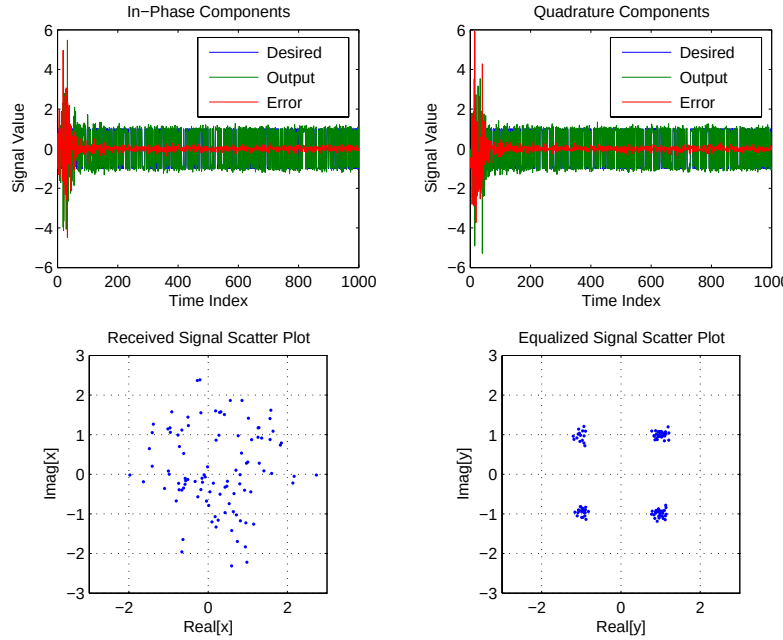
Name	Range	Description
`FwdPrediction`		Returns the predicted samples generated during adaptation in the forward direction. Refer to [2] in the bibliography for details about linear prediction.
`InitFactor`		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `l -1`

Examples

Implement Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter. To see the results of

the equalization process in this example, look at the figure that follows the example code.

```
D  = 16;                        % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
s  = sign(randn(1,ntr+D))+j*sign(randn(1,ntr+D)); % Baseband
QPSK                                % signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));   % Noise signal
r  = filter(b,a,s)+n;           % Received signal
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                  % Desired signal (delayed QPSK signal)
lam = 0.995;                   % Forgetting factor
del = 1;                        % Soft-constrained initialization
factor
ha = adaptfilt.qrdlsl(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



See Also

`adaptfilt.qrdrls`, `adaptfilt.gal`, `adaptfilt.ftf`, `adaptfilt.lsl`

References

Haykin, S.,*Adaptive Filter Theory*, 2nd Edition, Prentice Hall, N.J., 1991

Purpose

FIR adaptive filter that uses QR-decomposition-based RLS

Syntax

```
ha = adaptfilt.qrdrls(l,lambda,sqrtcov,coeffs,states)
```

Description

`ha = adaptfilt.qrdrls(l,lambda,sqrtcov,coeffs,states)` constructs an FIR QR-decomposition-based recursive-least squares (RLS) adaptive filter object `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.qrdrls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer.`l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar and should lie within the range (0, 1]. `lambda` defaults to 1.
`sqrtcov`	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
`coeffs`	Vector of initial filter coefficients. It must be a length `l` vector. `coeffs` defaults to length `l` vector whose elements are zeros.
`states`	Vector of initial filter states. It must be a length `l`-1 vector. `states` defaults to a length `l`-1 vector of zeros.

Properties

Since your `adaptfilt.qrdrls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.qrdrls` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of length `l`	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting `forgetting factor = 1` denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.

Name	Range	Description
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`SqrtCov`	Square matrix with each dimension equal to the filter length `l`	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
`States`	Vector of elements	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).

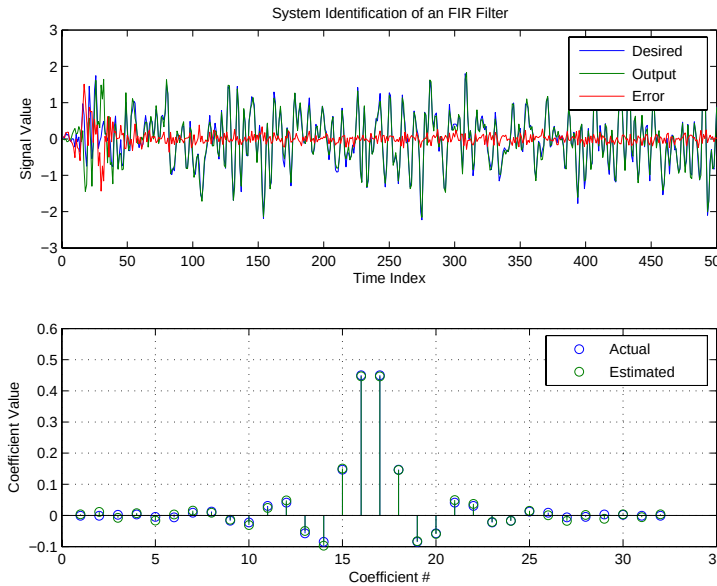
Examples

System Identification of a 32-coefficient FIR filter (500 iterations).

```
x  = randn(1,500);       % Input to the filter
b  = fir1(31,0.5);       % FIR system to be identified
n  = 0.1*randn(1,500);   % Observation noise signal
d  = filter(b,1,x)+n;    % Desired signal
G0 = sqrt(.1)*eye(32);   % Initial sqrt correlation matrix
```

```
lam = 0.99;              % RLS forgetting factor
ha = adaptfilt.qrdrls(32,lam,GO);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

Using this variant of the RLS algorithm successfully identifies the unknown FIR filter, as shown here.



See Also

`adaptfilt.rls`, `adaptfilt.hrls`, `adaptfilt.hswrls`, `adaptfilt.swrls`

Purpose

FIR adaptive filter that uses direct form RLS

Syntax

```
ha = adaptfilt.rls(l,lambda,invcov,coeffs,states)
```

Description

`ha = adaptfilt.rls(l,lambda,invcov,coeffs,states)` constructs an FIR direct form RLS adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.rls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1.
`invcov`	Inverse of the input signal covariance matrix. For best performance, you should initialize this matrix to be a positive definite matrix.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l`-1 vector. `states` defaults to a length l-1 vector of zeros.

Properties

Since your `adaptfilt.rls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.rls` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation.
`Coefficients`	Vector containing `l` elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps. Remember that filter length is filter order + 1.
`ForgettingFactor`	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting `forgetting factor = 1` denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.
`InvCov`	Matrix of size `l`-by-`l`	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
`KalmanGain`	Vector of size (`l`,1)	Empty when you construct the object, this gets populated after you run the filter.

Name	Range	Description
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. Defaults to `false`.
`States`	Double array	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).

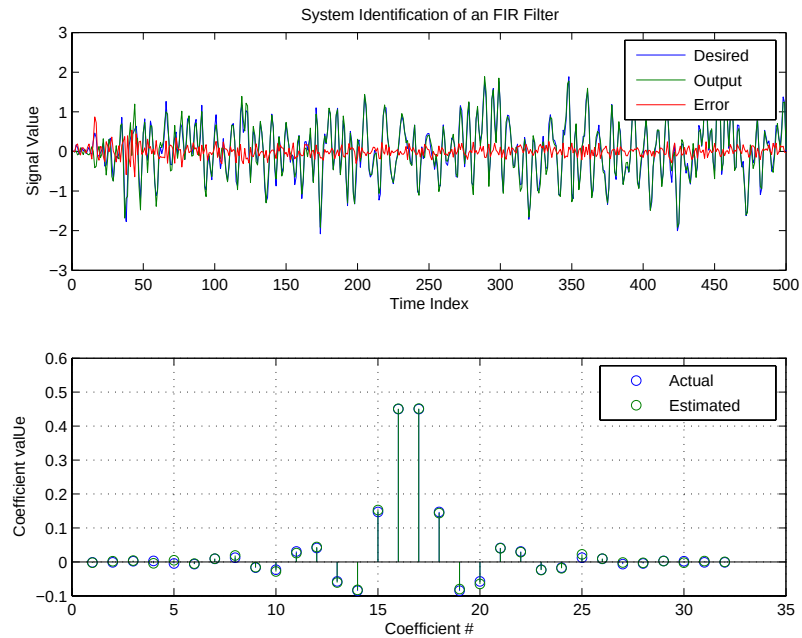
Examples

System Identification of a 32-coefficient FIR filter over 500 adaptation iterations.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
P0 = 10*eye(32); % Initial sqrt correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
ha = adaptfilt.rls(32,lam,P0);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient valUe');
```

```
grid on;
```

In this example of adaptive filtering using the RLS algorithm to update the filter coefficients for each iteration, the figure shown reveals the fidelity of the derived filter after adaptation.



See Also

`adaptfilt.hrls`,`adaptfilt.hswrls`, `adaptfilt.qrdrls`

Purpose

FIR adaptive filter that uses sign-data algorithm

Syntax

```
ha = adaptfilt.sd(l,step,leakage,coeffs,states)
```

Description

`ha = adaptfilt.sd(l,step,leakage,coeffs,states)` constructs an FIR sign-data adaptive filter object `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.sd`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	SD step size. It must be a nonnegative scalar. `step` defaults to 0.1
`leakage`	Your SD leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.sd` implements a leaky SD algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l`-1 vector. `states` defaults to a length l-1 vector of zeros.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for

sign-data objects, their default values, and a brief description of the property.

Property	Default Value	Description
`Al gorithm`	Sign-data	Defines the adaptive filter algorithm the object uses during adaptation.
`Coefficients`	`zeros(1,l)`	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need `l` entries in `coefficients`.
`FilterLength`	10	Reports the length of the filter, the number of coefficients or taps.
`Leakage`	0	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. Defaults to 0.

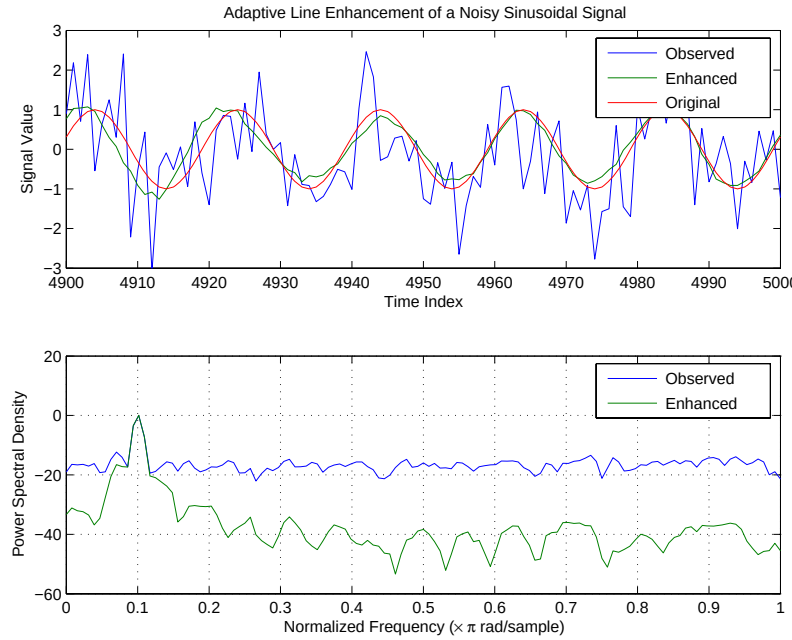
Property	Default Value	Description
`PersistentMemory`	`false` or `true`	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.
`States`	`zeros(1-1,1)`	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` - 1).
`StepSize`	0.1	Sets the SD algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Example

Adaptive line enhancement using a 32-coefficient FIR filter to perform the enhancement. This example runs for 5000 iterations, as you see in property `iter`.

```
d  = 1;                              % Number of samples of delay
ntr= 5000;                           % Number of iterations
v  = sin(2*pi*0.05*[1:ntr+d]);       % Sinusoidal signal
n  = randn(1,ntr+d);                 % Noise signal
```

```
x  = v(1:ntr)+n(1:ntr);              % Input signal (delayed desired
                                     % signal)
d  = v(1+d:ntr+d)+n(1+d:ntr+d);      % Desired signal
mu = 0.0001;                         % Sign-data step size.
ha = adaptfilt.sd(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:ntr,[d;y;v(1+d:ntr+d)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of a Noisy Sinusoidal Signal');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
[pxx,om] = pwelch(x(ntr-1000:ntr));
pyy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),pyy/max(pyy)]));
axis([0 1 -60 20]);
legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```



Each of the variants — sign-data, sign-error, and sign-sign — uses the same example. You can compare the results by viewing the figure shown for each adaptive filter method — `adaptfilt.sd`, `adaptfilt.se`, and `adaptfilt.ss`.

See Also

`adaptfilt.lms`, `adaptfilt.se`, `adaptfilt.ss`

References

Moschner, J.L., “Adaptive Filter with Clipped Input Data,” Ph.D. thesis, Stanford Univ., Stanford, CA, June 1970.

Hayes, M., *Statistical Digital Signal Processing and Modeling,* New York Wiley, 1996.

Purpose

FIR adaptive filter that uses sign-error algorithm

Syntax

```
ha = adaptfilt.se(l,step,leakage,coeffs,states)
```

Description

`ha = adaptfilt.se(l,step,leakage,coeffs,states)` constructs an FIR sign-error adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.se`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	SE step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.1
`leakage`	Your SE leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.se` implements a leaky SE algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l`-1 vector. `states` defaults to a length l-1 vector of zeros.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for the sign-error SD object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	Sign-error	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	`zeros(1,l)`	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting.
`FilterLength`	10	Reports the length of the filter, the number of coefficients or taps
`Leakage`	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. Defaults to one if omitted.

Property	Default Value	Description
`PersistentMemory`	`false` or `true`	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.
`States`	`zeros(l-1,1)`	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `(l -1)`.
`StepSize`	0.1	Sets the SE algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Use `inspect(ha)` to view or change the object properties graphically using the MATLAB Property Inspector.

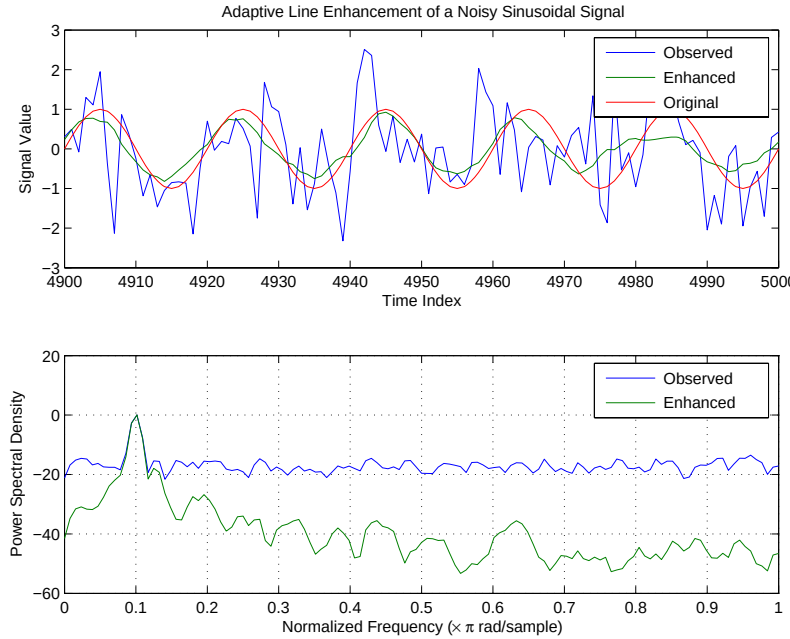
Examples

Adaptive line enhancement using a 32-coefficient FIR filter running over 5000 iterations.

```
d = 1;                          % Number of samples of delay
ntr= 5000;                      % Number of iterations
v = sin(2*pi*0.05*[1:ntr+d]);    % Sinusoidal signal
```

```
n = randn(1,ntr+d);          % Noise signal
x = v(1:ntr)+n(1:ntr);       % Input signal --
                             %  (delayed desired signal)
d = v(1+d:ntr+d)+n(1+d:ntr+d);  % Desired signal
mu = 0.0001;                    % Sign-error step size
ha = adaptfilt.se(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:ntr,[d;y;v(1+d:ntr+d)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of Noisy Sinusoid');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
[pxx,om] = pwelch(x(ntr-1000:ntr));
pyy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),pyy/max(pyy)]));
axis([0 1 -60 20]);
legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

Compare the figure shown here to the ones for `adaptfilt.sd` and `adaptfilt.ss` to see how the variants perform on the same example.



See Also

`adaptfilt.sd`, `adaptfilt.ss`, `adaptfilt.lms`

References

Gersho, A, “Adaptive Filtering With Binary Reinforcement,” IEEE® Trans. Information Theory, vol. IT-30, pp. 191-199, March 1984.

Hayes, M, *Statistical Digital Signal Processing and Modeling*, New York, Wiley, 1996.

Purpose

FIR adaptive filter that uses sign-sign algorithm

Syntax

```
ha = adaptfilt.ss(l,step,leakage,coeffs,states)
```

Description

`ha = adaptfilt.ss(l,step,leakage,coeffs,states)` constructs an FIR sign-error adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ss`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	SS step size. It must be a nonnegative scalar. `step` defaults to 0.1.
`leakage`	Your SS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.lms` implements a leaky SS algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l` -1 vector. `states` defaults to a length l-1 vector of zeros.

`adaptfilt.ss` can be called for a block of data, when `x` and `d` are vectors, or in "sample by sample mode" using a For-loop with the method filter:

```
for n = 1:length(x)
```

```
ha = adaptfilt.ss(25,0.9);
[y(n),e(n)] = filter(ha,(x(n),d(n),s));
% The property values of ha may be modified here.end
```

Properties

In the syntax for creating the `adaptfilt` object, most of the input options are properties of the object you create. This table lists the properties for sign-sign objects, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	Sign-sign	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	`zeros(1,l)`	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting.
`FilterLength`	10	Reports the length of the filter, the number of coefficients or taps

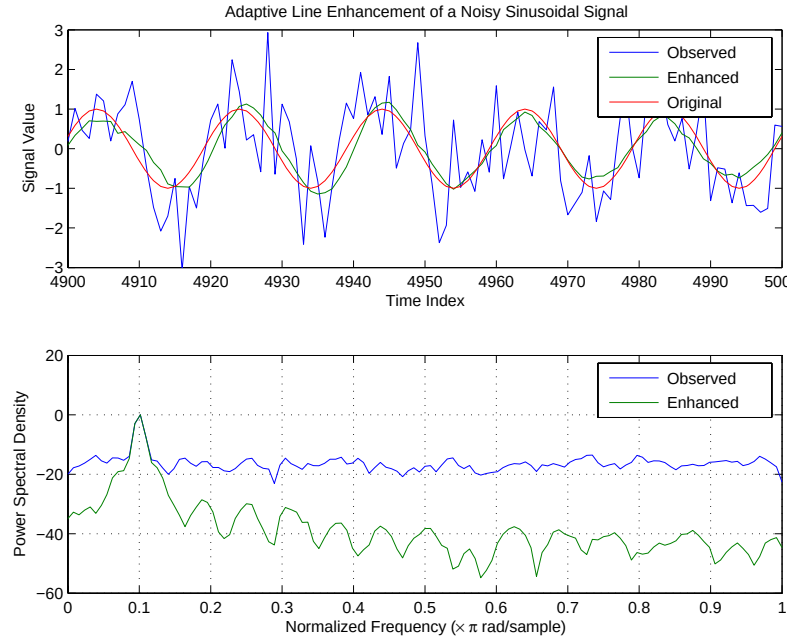
Property	Default Value	Description
`Leakage`	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. 1 is the default value.
`PersistentMemory`	`false` or `true`	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.
`States`	`zeros(l-1,1)`	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (l-1).
`StepSize`	0.1	Sets the SE algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Examples

Demonstrating adaptive line enhancement using a 32-coefficient FIR filter provides a good introduction to the sign-sign algorithm.

```
d = 1;                          % number of samples of delay
ntr= 5000;                      % number of iterations
v = sin(2*pi*0.05*[1:ntr+d]);   % sinusoidal signal
n = randn(1,ntr+d);             % noise signal
x = v(1:ntr)+n(1:ntr);          % Delayed input signal
d = v(1+d:ntr+d)+n(1+d:ntr+d);  % desired signal
mu = 0.0001;                    % sign-sign step size
ha = adaptfilt.ss(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:ntr,[d;y;v(1+d:ntr+d)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of a Noisy Sinusoid');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
[pxx,om] = pwelch(x(ntr-1000:ntr));
pyy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),pyy/max(pyy)]));
axis([0 1 -60 20]);
legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

This example is the same as the ones used for the sign-data and sign-error examples. Comparing the figures shown for each of the others lets you assess the performance of each for the same task.



See Also

`adaptfilt.se`, `adaptfilt.sd`, `adaptfilt.lms`

References

Lucky, R.W, “Techniques For Adaptive Equalization of Digital Communication Systems,” Bell Systems Technical Journal, vol. 45, pp. 255-286, Feb. 1966

Hayes, M., *Statistical Digital Signal Processing and Modeling*, New York, Wiley, 1996.

Purpose

FIR adaptive filter that uses sliding window fast transversal LMS

Syntax

```
ha = adaptfilt.swftf(l,delta,blocklen,gamma,gstates,
dstates,...coeffs,states)
```

Description

`ha = adaptfilt.swftf(l,delta,blocklen,gamma,gstates, dstates,...coeffs,states)` constructs a sliding window fast transversal least squares adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.swftf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`delta`	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to maintain stability. `delta` defaults to 1.
`blocklen`	Block length of the sliding window. This must be an integer at least as large as the filter length `l`, which is the default value.
`gamma`	Conversion factor. `gamma` defaults to the matrix [1 -1] that specifies soft-constrained initialization.
`gstates`	States of the Kalman gain updates. `gstates` defaults to a zero vector of length (`l` + `blocklen` - 1).
`dstates`	Desired signal states of the adaptive filter. `dstates` defaults to a zero vector of length equal to (`blocklen` - 1). For a default object, `dstates` is (`l`-1).

Input Argument	Description
`coeffs`	Vector of initial filter coefficients. It must be a length `l` vector. `coeffs` defaults to length `l` vector of all zeros.
`states`	Vector of initial filter states. `states` defaults to a zero vector of length equal to (`l` + `blocklen` - 2).

Properties

Since your `adaptfilt.swftf` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.swftf` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`BkwdPredictions`		Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.
`BlockLength`		Block length of the sliding window. This must be an integer at least as large as the filter length `l`, which is the default value.
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.

Name	Range	Description
`ConversionFactor`		Conversion factor. Called `gamma` when it is an input argument, it defaults to the matrix [1 -1] that specifies soft-constrained initialization.
`DesiredSignal States`		Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`blocklen` - 1).
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`FwdPrediction`		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
`InitFactor`		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`KalmanGain`		Empty when you construct the object, this gets populated after you run the filter.
`KalmanGainStates`		Contains the states of the Kalman gains for the adaptive algorithm. Initialized to a vector of double data type entries.

Name	Range	Description
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`1 + projectord - 2`).

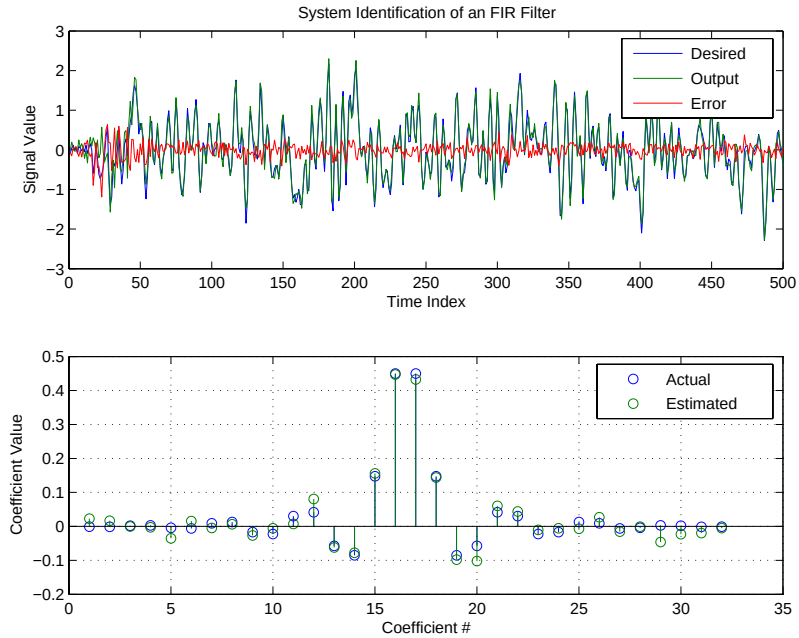
Examples

Over 500 iterations, perform a system identification of a 32-coefficient FIR filter.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
L  = 32;               % Adaptive filter length
del = 0.1;             % Soft-constrained
                       %  initialization factor
N  = 64;               % block length
ha = adaptfilt.swftf(L,del,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
```

```
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

Review the figure for the results of the example. When you evaluate the example you should get the same results, within the differences in the random noise signal you use.



See Also

`adaptfilt.ftf`, `adaptfilt.swrls`, `adaptfilt.ap`, `adaptfilt.apru`

References

Slock, D.T.M., and T. Kailath, “A Modular Prewindowing Framework for Covariance FTF RLS Algorithms,” Signal Processing, vol. 28, pp. 47-61, 1992

Slock, D.T.M., and T. Kailath, “A Modular Multichannel Multi-Experiment Fast Transversal Filter RLS Algorithm,” Signal Processing, vol. 28, pp. 25-45, 1992

Purpose

FIR adaptive filter that uses window recursive least squares (RLS)

Syntax

```
ha = adaptfilt.swrls(l,lambda,invcov,swblocklen,
dstates,...coeffs,states)
```

Description

`ha = adaptfilt.swrls(l,lambda,invcov,swblocklen, dstates,...coeffs,states)` constructs an FIR sliding window RLS adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.swrls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps). It must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar and should lie within the range (0, 1]. `lambda` defaults to 1.
`invcov`	Inverse of the input signal covariance matrix. You should initialize `invcov` to a positive definite matrix.
`swblocklen`	Block length of the sliding window. This integer must be at least as large as the filter length. `swblocklen` defaults to 16.
`dstates`	Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`swblocklen` - 1).
`coeffs`	Vector of initial filter coefficients. It must be a length `l` vector. `coeffs` defaults to length `l` vector of all zeros.
`states`	Vector of initial filter states. `states` defaults to a zero vector of length equal to (`l` + `swblocklen` - 2).

Properties

Since your `adaptfilt.swrls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.swrls` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Any vector of `l` elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`DesiredSignalStates`	Vector	Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`swblocklen` - 1).
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Name	Range	Description
`ForgettingFactor`	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting `forgetting factor = 1` denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.
`InvCov`	Matrix	Square matrix with each dimension equal to the filter length `l`.
`KalmanGain`	Vector with dimensions (`l`,1)	Empty when you construct the object, this gets populated after you run the filter.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. Defaults to `false`.

Name	Range	Description
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + swblocklen - 2)
SwBlockLength	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.

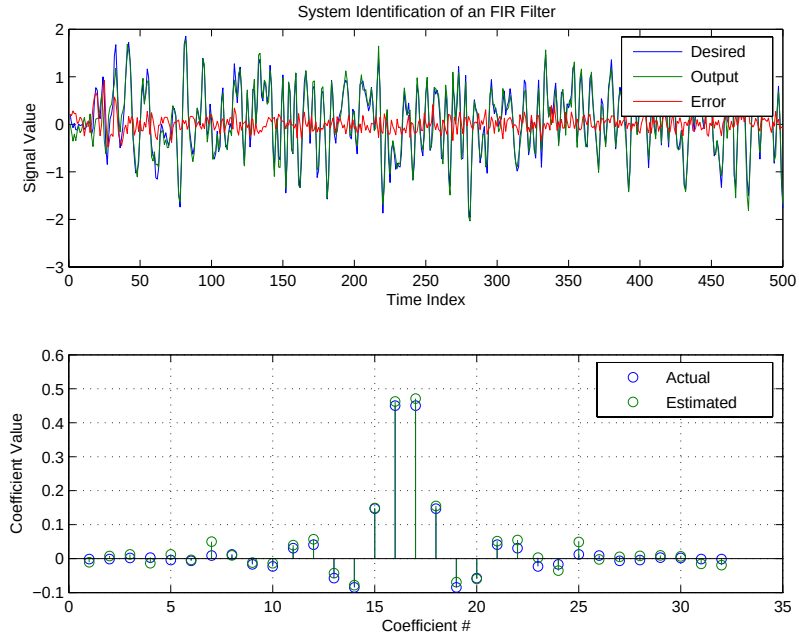
Examples

System Identification of a 32-coefficient FIR filter. Use 500 iterations to adapt to the unknown filter. After the example code, you see a figure that plots the results of the running the code.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
P0 = 10*eye(32);       % Initial correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
N  = 64;               % Block length
ha = adaptfilt.swrls(32,lam,P0,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

In the figure you see clearly that the adaptive filter process successfully identified the coefficients of the unknown FIR filter. You knew it

had to or many things that you take for granted, such as modems on computers, would not work.



See Also

`adaptfilt.rls`, `adaptfilt.qrdrls`, `adaptfilt.hswrls`

Purpose

Adaptive filter that uses discrete cosine transform

Syntax

```
ha = adaptfilt.tdafdct(l,step,leakage,offset,delta,lambda,
coeffs,states)
```

Description

`ha = adaptfilt.tdafdct(l,step,leakage,offset,delta,lambda, coeffs,states)` constructs a transform-domain adaptive filter ha object that uses the discrete cosine transform to perform filter adaptation.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.tdafdct`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Adaptive filter step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.
`leakage`	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDCT algorithm. `leakage` defaults to 1 — no leakage.
`offset`	Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zero or by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to zero.

Input Argument	Description
`delta`	Initial common value of all of the transform domain powers. Its initial value should be positive. `delta` defaults to 5.
`lambda`	Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. `lambda` should lie between zero and one. For default filter objects, `lambda` equals (`1 - step`).
`coeffs`	Initial time domain coefficients of the adaptive filter. Set it to be a length `l` vector. `coeffs` defaults to a zero vector of length `l`.
`states`	Initial conditions of the adaptive filter. `states` defaults to a zero vector with length equal to (`l - 1`).

Properties

Since your `adaptfilt.tdafdct` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.tdafdct` objects. To show you the properties that apply, this table lists and describes each property for the transform domain filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation.

Name	Range	Description
`AvgFactor`		Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. `AvgFactor` should lie between zero and one. For default filter objects, `AvgFactor` equals (`1 - step`). `lambda` is the input argument that represent `AvgFactor`.
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. `leakage` defaults to 1 — no leakage.

Name	Range	Description
`Offset`		Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to zero.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`Power`	2*`l` element vector	A vector of 2*`l` elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.

Name	Range	Description
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + projectord - 2).
StepSize	0 to 1	Step size. It must be a nonnegative scalar, greater than zero and less than or equal to 1. You can use maxstep to determine a reasonable range of step size values for the signals being processed. step defaults to 0.

For checking the values of properties for an adaptive filter object, use get(ha) or enter the object name, without a trailing semicolon, at the MATLAB prompt.

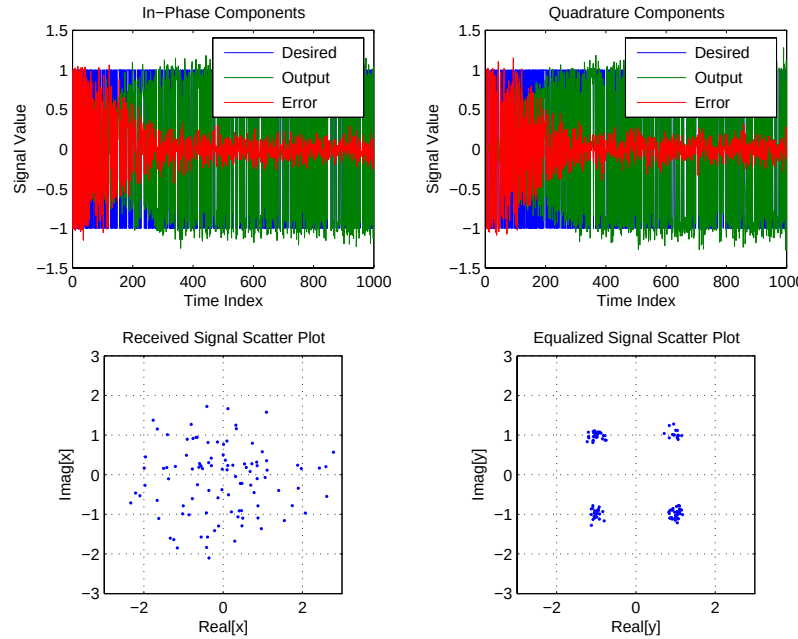
Examples

Using 1000 iterations, perform a Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D  = 16;                        % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D));% Baseband
                                                   % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D)); % Noise signal
r  = filter(b,a,s)+n;           % Received signal
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                  % Desired signal (delayed QPSK signal)
L  = 32;                        % filter length
mu = 0.01;                      % Step size
ha = adaptfilt.tdafdct(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
```

```
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

Compare the plots shown in this figure to those in the other time domain filter variations. The comparison should help you select and understand how the variants differ.



See Also

`adaptfilt.tdafdft`, `adaptfilt.fdaf`, `adaptfilt.blms`

References

Haykin, S.,*Adaptive Filter Theory*, 3rd Edition, Prentice Hall, N.J., 1996.

adaptfilt.tdafdft

Purpose

Adaptive filter that uses discrete Fourier transform

Syntax

```
ha = adaptfilt.tdafdft(l,step,leakage,offset,
delta,lambda,...coeffs,states)
```

Description

`ha = adaptfilt.tdafdft(l,step,leakage,offset, delta,lambda,...coeffs,states)` constructs a transform-domain adaptive filter object `ha` using a discrete Fourier transform.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.tdafdft`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Adaptive filter step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.
`leakage`	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. `leakage` defaults to 1 — no leakage.
`offset`	Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to zero.

Input Argument	Description
`delta`	Initial common value of all of the transform domain powers. Its initial value should be positive. `delta` defaults to 5.
`lambda`	Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. `lambda` should lie between zero and one. For default filter objects, LAMBDA equals (`1 - step`).
`coeffs`	Initial time domain coefficients of the adaptive filter. Set it to be a length `l` vector. `coeffs` defaults to a zero vector of length `l`.
`states`	Initial conditions of the adaptive filter. `states` defaults to a zero vector with length equal to (`l - 1`).

Properties

Since your `adaptfilt.tdafdft` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.tdafdft` objects. To show you the properties that apply, this table lists and describes each property for the transform domain filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation

Name	Range	Description
`AvgFactor`		Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. `AvgFactor` should lie between zero and one. For default filter objects, `AvgFactor` equals (`1 - step`). `lambda` is the input argument that represent `AvgFactor`.
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. `leakage` defaults to 1 — no leakage.

Name	Range	Description
`Offset`		Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to zero.
`PersistentMemory`	`false` or `true`	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`Power`	2*`l` element vector	A vector of 2*`l` elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`StepSize`	0 to 1	Step size. It must be a nonnegative scalar, greater than zero and less than or equal to 1. `step` defaults to 0.

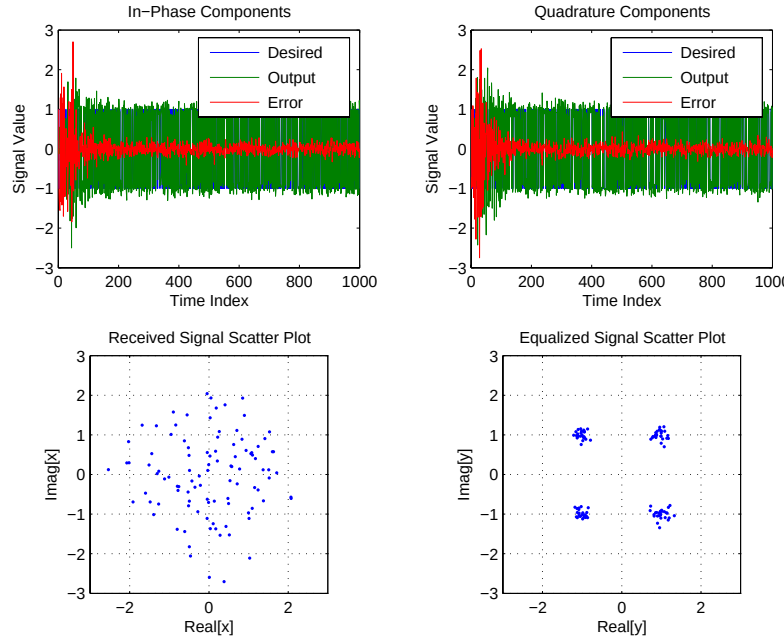
Examples

Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter (1000 iterations).

```
D  = 16;                        % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D));% Baseband
                                                   % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;           % Received signal
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                  % Desired signal (delayed QPSK signal)
L  = 32;                        % filter length
mu = 0.01;                      % Step size
ha = adaptfilt.tdafdft(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

All of the time domain adaptive filter reference pages use this QPSK example. By comparing the results for each variation you get an idea of the differences in the way each one performs.

This figure demonstrates the results of running the example code shown.



See Also

`adaptfilt.tdafdct`, `adaptfilt.fdaf`, `adaptfilt.blms`

References

Haykin, S.,*Adaptive Filter Theory*, 3rd Edition, Prentice Hall, N.J., 1996

Purpose

FIR adaptive filter that uses unconstrained frequency-domain with quantized step size normalization

Syntax

```
ha = adaptfilt.ufdaf(l,step,leakage,delta,lambda,blocklen,
offset,coeffs,states)
```

Description

```
ha = adaptfilt.ufdaf(l,step,leakage,delta,lambda,blocklen,
offset,coeffs,states)
```

constructs an unconstrained frequency-domain FIR adaptive filter `ha` with quantized step size normalization.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ufdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Adaptive filter step size. It must be a nonnegative scalar. `step` defaults to 0.
`leakage`	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the UFDAF algorithm. `leakage` defaults to 1 — no leakage.
`delta`	Initial common value of all of the FFT input signal powers. the initial value of delta should be positive, and it defaults to 1.

Input Argument	Description
`lambda`	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. `lambda` should lie in the range (0,1]. For default UFDAF filter objects, `lambda` defaults to 0.9.
`blocklen`	Block length for the coefficient updates. This must be a positive integer. For faster execution, (`blocklen` `l`) should be a power of two. `blocklen` defaults to `l`.
`offset`	Offset for the normalization terms in the coefficient updates. This can help you avoid divide by zero conditions, or divide by very small numbers conditions, when any of the FFT input signal powers become very small. Default value is zero.
`coeffs`	Initial time-domain coefficients of the adaptive filter. It should be a length `l` vector. The filter object uses these coefficients to compute the initial frequency-domain filter coefficients via an FFT computed after zero-padding the time-domain vector by `blocklen`.
`states`	Adaptive filter states. `states` defaults to a zero vector with length equal to `l`.

Properties

Since your `adaptfilt.ufdaf` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.ufdaf` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`AvgFactor`		Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. `AvgFactor` should lie in the range (0,1]. For default UFDAF filter objects, `AvgFactor` defaults to 0.9. Note that AvgFactor and lambda are the same thing — lambda is an input argument and AvgFactor a property of the object.
`BlockLength`		Block length for the coefficient updates. This must be a positive integer. For faster execution, (`blocklen + 1`) should be a power of two. `blocklen` defaults to `1`.
`FFTCoefficients`		Stores the discrete Fourier transform of the filter coefficients in `coeffs`.
`FFTStates`		States for the FFT operation.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Name	Range	Description
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the UFDAF algorithm. `leakage` defaults to 1 — no leakage.
`Offset`		Offset for the normalization terms in the coefficient updates. This can help you avoid divide by zero conditions, or divide by very small numbers conditions, when any of the FFT input signal powers become very small. Default value is zero.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.

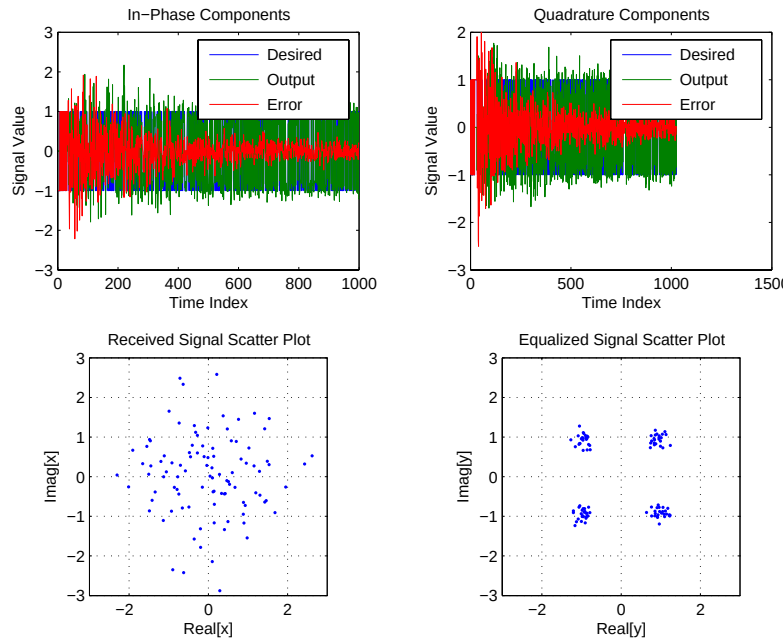
Name	Range	Description
`Power`	2*`l` element vector	A vector of 2*`l` elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.
`StepSize`	0 to 1	Adaptive filter step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.

Examples

Show an example of Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter. For fidelity, use 1024 iterations. The figure that follows the code provides the information you need to assess the performance of the equalization process.

```
D  = 16;                      % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];    % Numerator coefficients of channel
a  = [1 -0.7];                % Denominator coefficients of channel
ntr= 1024;                    % Number of iterations
s = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
                                                   % QPSK signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));  % Noise signal
r  = filter(b,a,s)+n;         % Received signal
x  = r(1+D:ntr+D);            % Input signal (received signal)
d  = s(1:ntr);                % Desired signal (delayed QPSK signal)
del = 1;                      % Initial FFT input powers
mu  = 0.1;                    % Step size
lam = 0.9;                    % Averaging factor
ha = adaptfilt.ufdaf(32,mu,1,del,lam);
[y,e] = filter(ha,x,d);
```

```
subplot(2,2,1);
plot(1:1000,real([d(1:1000);y(1:1000);e(1:1000)]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



See Also

`adaptfilt.fdaf`, `adaptfilt.pbufdaf`, `adaptfilt.blms`, `adaptfilt.blmsfft`

References

Shynk, J.J.,"Frequency-domain and Multirate Adaptive Filtering," IEEE® Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992

Purpose

Allpass filter for complex bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a complex bandpass to complex bandpass frequency transformation. This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations W_{t1} and W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle. This is very attractive for adaptive systems.

Examples

Design the allpass filter changing the complex bandpass filter with the band edges originally at W_{o1}=`0.2` and W_{o2}=`0.4` to the new band edges of W_{t1}=`0.3` and W_{t2}=`0.6` precisely defined:

```
Wo = [0.2, 0.4];
Wt = [0.3, 0.6];
[AllpassNum, AllpassDen] = allpassbpc2bpc(Wo, Wt);
```

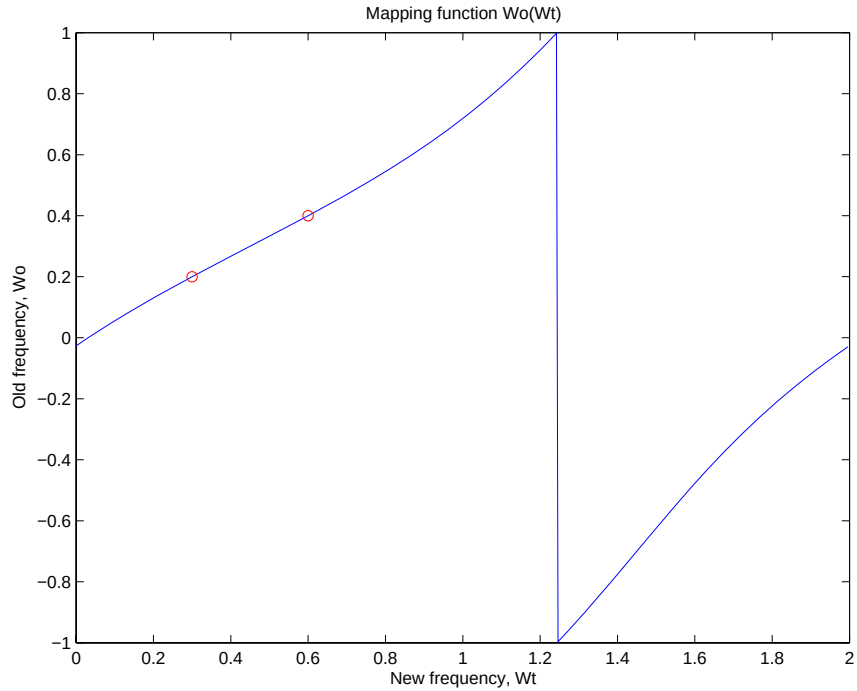
Calculate the frequency response of the mapping filter in the full range:

```
[ha, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi,angle(ha)/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

To demonstrate, the following figure shows the mapping function between old and new frequencies.



Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirbpc2bpc`, `zpkbpc2bpc`

allpasslp2bp

Purpose

Allpass filter for lowpass to bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real lowpass to real bandpass frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "DC mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and repositioned at two distinct desired frequencies.

Examples

Design the allpass filter changing the lowpass filter with cutoff frequency at W_o=0.5 to the real bandpass filter with cutoff frequencies at W_{t1}=0.25 and W_{t2}=0.375:

```
Wo = 0.5;
Wt = [0.25, 0.375];
[AllpassNum, AllpassDen] = allpasslp2bp(Wo, Wt);
```

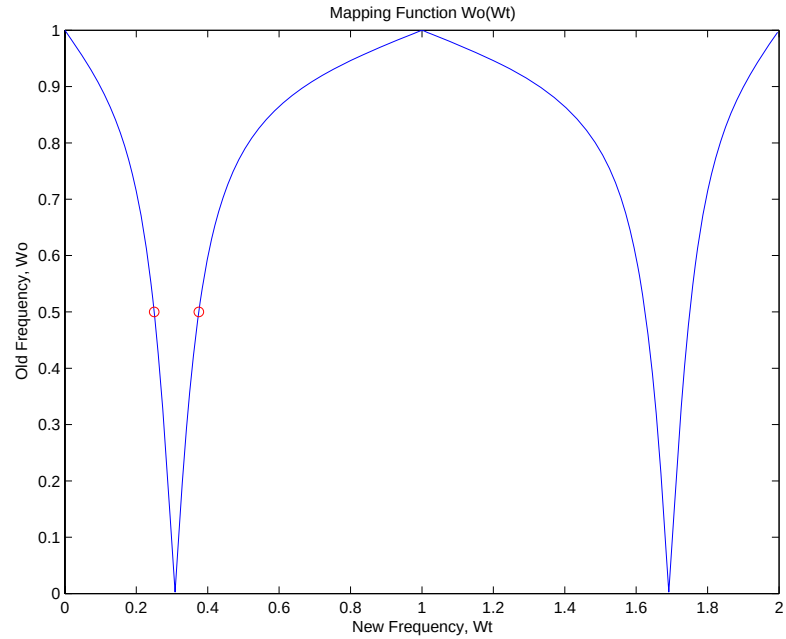
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

Shown in the figure, with the x-axis as the new frequency, you see the mapping filter for the example.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2bp`, `zpklp2bp`

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE® Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose

Allpass filter for lowpass to complex bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to complex bandpass frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=`0.5` into a complex bandpass filter with band edges of W_{t1}=`0.2` and W_{t2}=`0.4` precisely defined:

```
Wo = 0.5;
Wt = [0.2,0.4];
[AllpassNum, AllpassDen] = allpasslp2bpc(Wo, Wt);
```

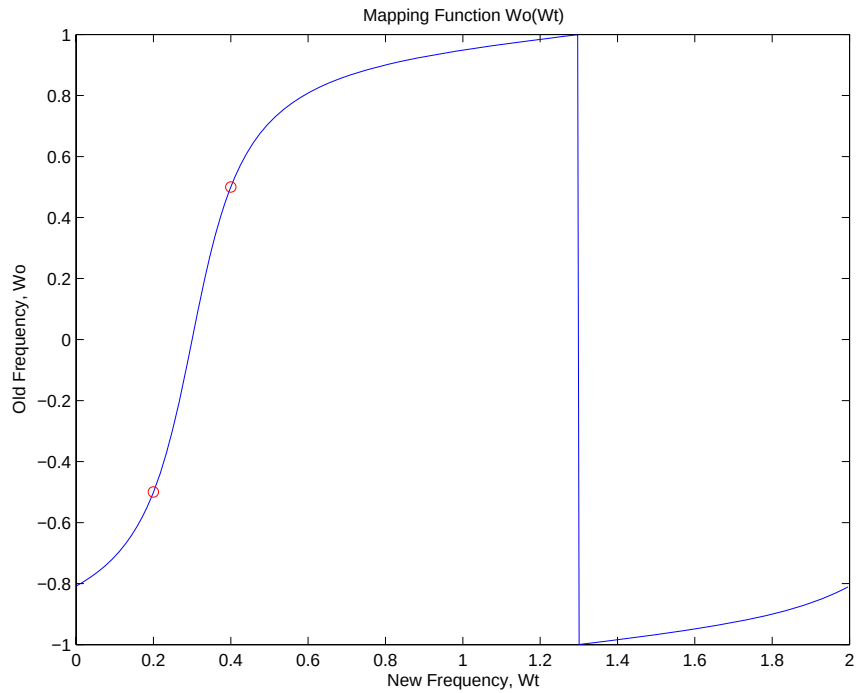
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo.*[-1,1], 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

The figure shown here details the mapping filter provided by the function.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`iirlp2bpc`, `zpklp2bpc`

allpasslp2bs

Purpose

Allpass filter for lowpass to bandstop transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real lowpass to real bandstop frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Examples

Design the allpass filter changing the lowpass filter with cutoff frequency at W_o=0.5 to the real bandstop filter with cutoff frequencies at W_{t1}=0.25 and W_{t2}=0.375:

```
Wo = 0.5;
Wt = [0.25, 0.375];
[AllpassNum, AllpassDen] = allpasslp2bs(Wo, Wt);
```

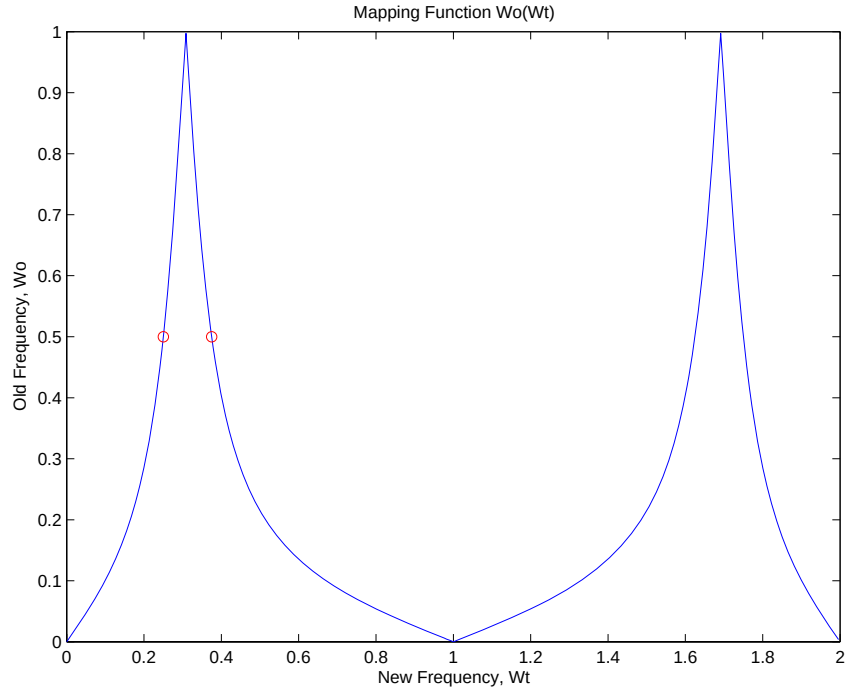
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

In the figure, you find the mapping filter function as determined by the example. Note the response is normalized to π, as mentioned earlier.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2bs`, `zpklp2bs`

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE® Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose

Allpass filter for lowpass to complex bandstop transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to complex bandstop frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandstop filters for band attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=`0.5` into a complex bandstop filter with band edges of W_{t1}=`0.2` and W_{t2}=`0.4` precisely defined:

```
Wo = 0.5;
```

```
Wt = [0.2,0.4];
[AllpassNum, AllpassDen] = allpasslp2bsc(Wo, Wt);
```

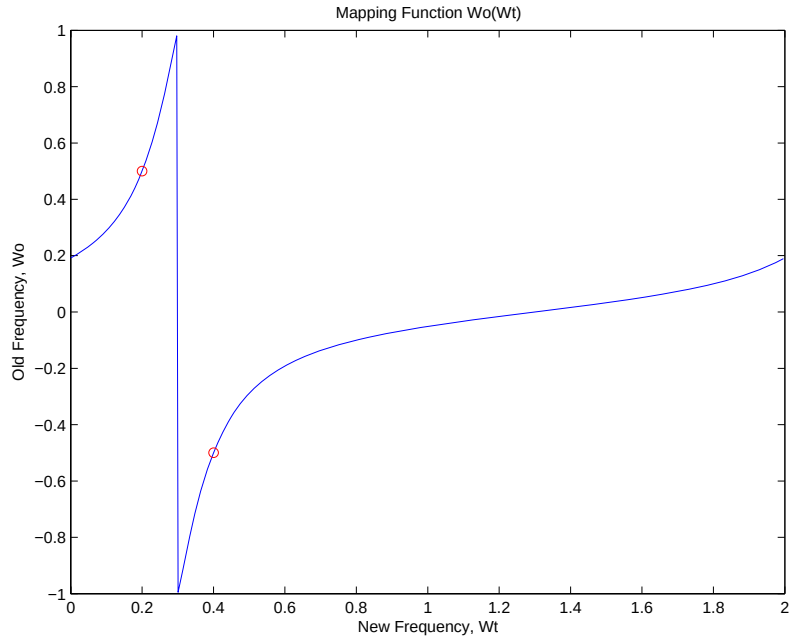
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo.*[1,-1], 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

We plot the resulting allpass mapping function response in this figure.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`iirlp2bsc`, `zpklp2bsc`

allpasslp2hp

Purpose

Allpass filter for lowpass to highpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real highpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency, W_o, at the required target frequency location, W_t, at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by using the lowpass to highpass transformation.

Examples

Design the allpass filter changing the lowpass filter to the highpass filter with its cutoff frequency moved from W_o=0.5 to W_t=0.25:

```
Wo = 0.5;
Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2hp(Wo, Wt);
```

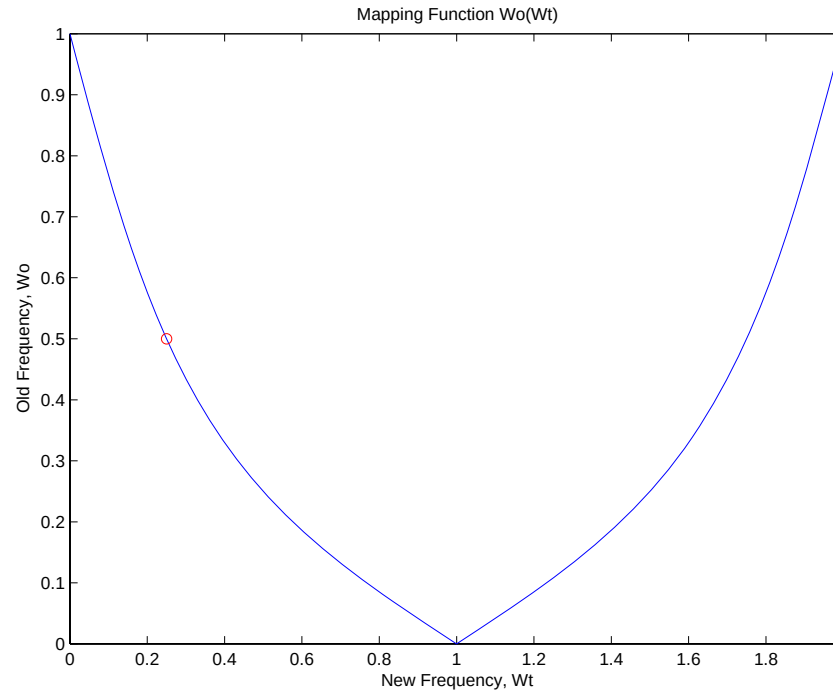
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

For transforming your lowpass filter to an highpass variation, the mapping function shown in this figure does the job.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2hp`, `zpklp2hp`

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

Purpose

Allpass filter for lowpass to lowpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real lowpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency W_o, at the required target frequency location, W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to lowpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband and so on.

Lowpass to lowpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by applying the lowpass to lowpass transformation.

Examples

Design the allpass filter changing the lowpass filter cutoff frequency originally at W_o=0.5 to W_t=0.25:

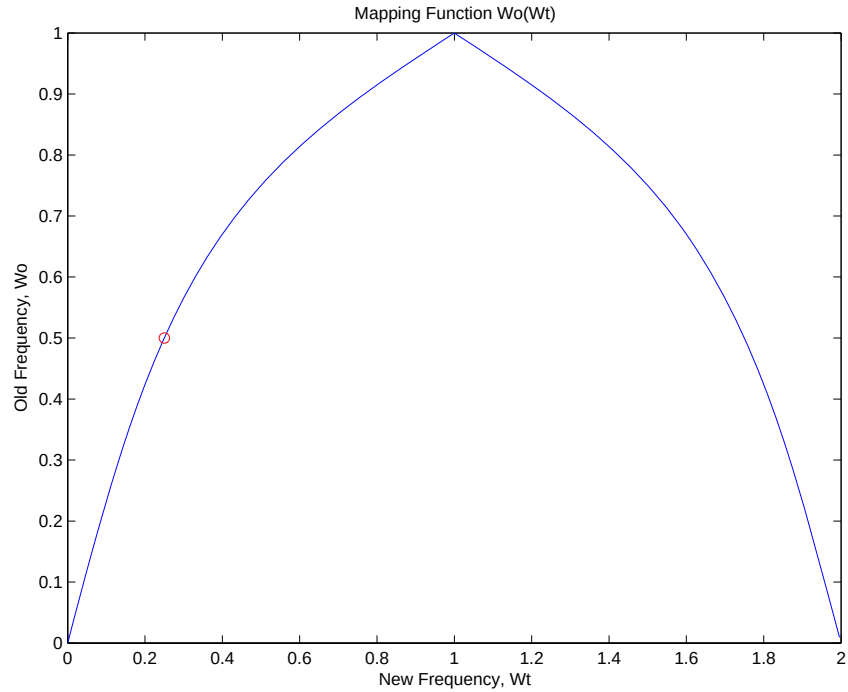
```
Wo = 0.5;
Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2lp(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```



As shown in the figure, `allpasslp2lp` generates a mapping function that converts your prototype lowpass filter to a target lowpass filter with different passband specifications.

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2lp`, `zpklp2lp`

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE® Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

allpasslp2mb

Purpose

Allpass filter for lowpass to M-band transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `M`th-order allpass mapping filter for performing a real lowpass to real multipassband frequency transformation. Parameter `M` is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},...,W_{tM}$. By default the DC feature is kept at its original location.

`[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the “Nyquist Mobility.” In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without redesigning them. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=0.5 into a real multiband filter with band edges of W_t=[1:2:9]/10 precisely defined:

```
Wo = 0.5;
Wt = [1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mb(Wo, Wt);
```

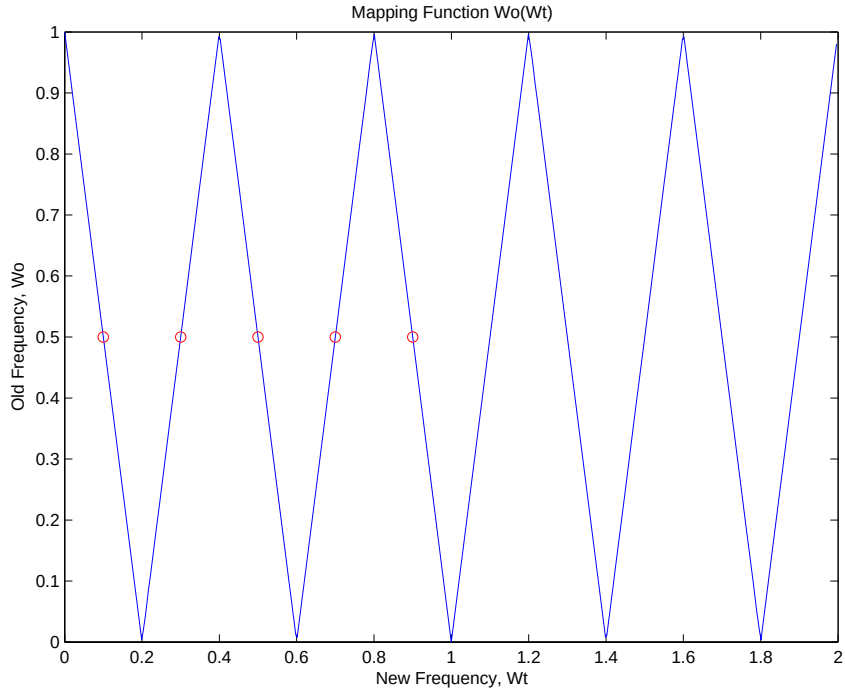
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

As the figure shows, the mapping function, or mapping filter, creates more than one band from your prototype.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2mb`, `zpklp2mb`

References

Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems,"*MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

allpasslp2mbc

Purpose

Allpass filter for lowpass to complex M-band transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `M`th-order allpass mapping filter for performing a real lowpass to complex multipassband frequency transformation. Parameter `M` is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need to design them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=`0.5` into a complex multiband filter with band edges of W_t=`[-3+1:2:9]/10` precisely defined:

```
Wo = 0.5;
Wt = [-3+1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mbc(Wo, Wt);
```

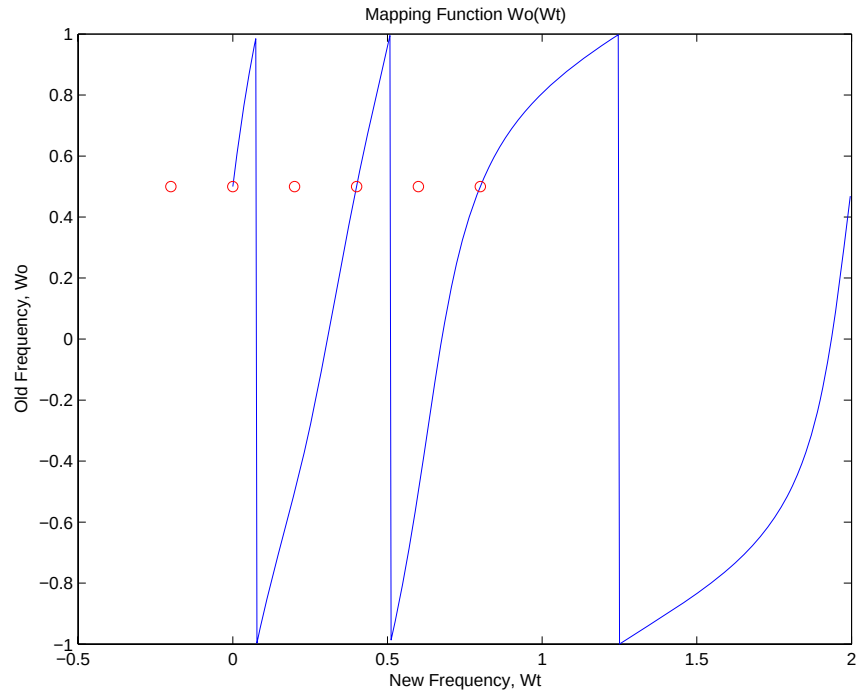
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

In this example, the resulting mapping function converts real filters to multiband complex filters.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`iirlp2mbc`, `zpklp2mbc`

Purpose

Allpass filter for lowpass to complex N-point transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter, where `N` is the allpass filter order, for performing a real lowpass to complex multipoint frequency transformation. Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of the, original filter located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter moving four features of an original complex filter given in W_o to the new independent frequency locations W_t. Please

note that the transformation creates `N` replicas of an original filter around the unit circle, where `N` is the order of the allpass mapping filter:

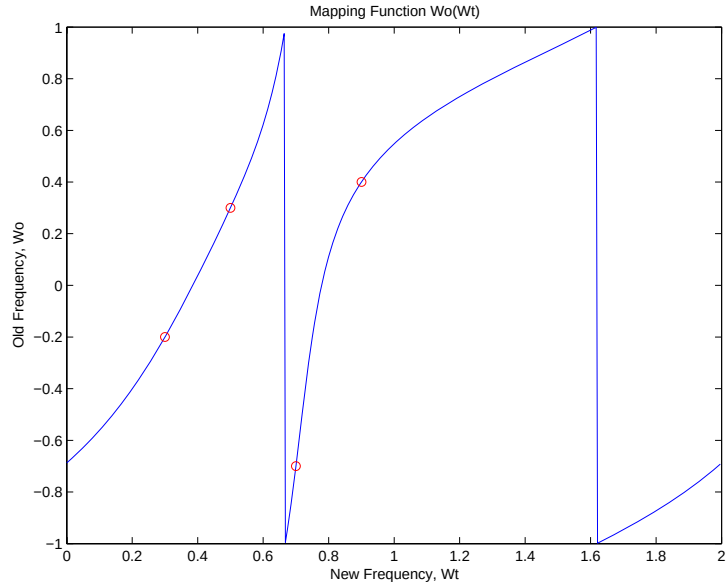
```
Wo = [-0.2, 0.3, -0.7, 0.4];
Wt = [0.3, 0.5, 0.7, 0.9];
[AllpassNum, AllpassDen] = allpasslp2xc(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```



As shown, the mapping function copies four features of interest in your prototype to multiple, independent locations in your target filter.

Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2xc`, `zpklp2xc`

allpasslp2xn

Purpose

Allpass filter for lowpass to N-point transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter, where `N` is the allpass filter order, for performing a real lowpass to real multipoint frequency transformation. Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$. By default the DC feature is kept at its original location.

`[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need of designing them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'/'stop'`) of passband/stopband at DC, `'pass'` being the default
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2xn`, `zpklp2xn`

References

Cain, G.D., A. Krukowski and I. Kale, “High Order Transformations for Flexible IIR Filter Design,” *VII European Signal Processing Conference (EUSIPCO’94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, “Custom designed high-order frequency transformations for IIR filters,” *38th Midwest Symposium on Circuits and Systems (MWSCAS’95)*, Rio de Janeiro, Brazil, August 1995.

allpassrateup

Purpose

Allpass filter for integer upsample transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassrateup(N)
```

Description

`[AllpassNum,AllpassDen] = allpassrateup(N)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter for performing the rateup frequency transformation, which creates `N` equal replicas of the prototype filter frequency response.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

Design the allpass filter creating the effect of upsampling the digital filter four times:

```
N = 4;
```

Choose any feature from an original filter, say at W_o=0.2:

```
Wo = 0.2;
Wt = Wo/N + 2*[0:N-1]/N;
[AllpassNum, AllpassDen] = allpassrateup(N);
```

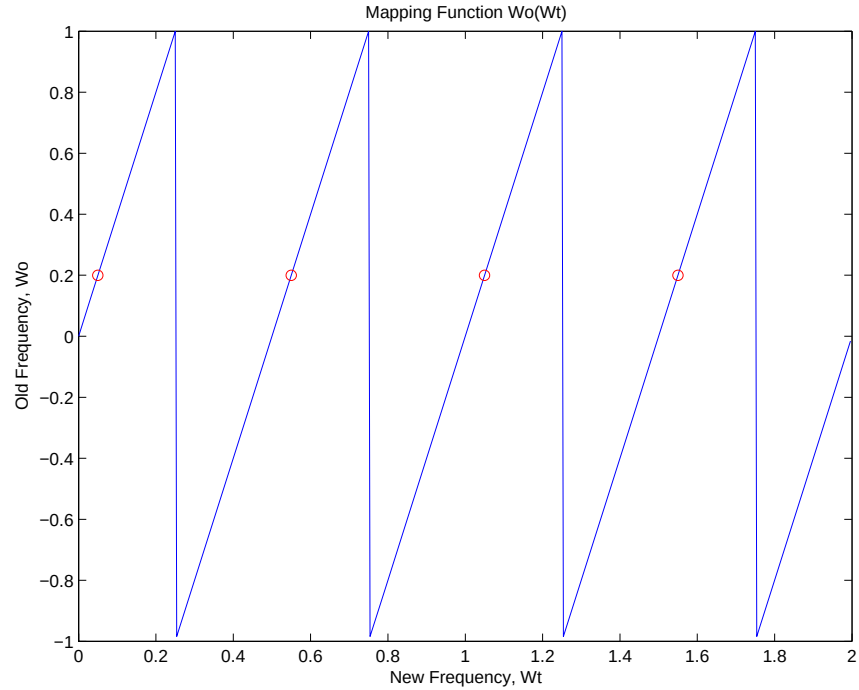
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

While this creates the effect of upsampling your prototype filter, compare the results to cicinterp for another approach to upsampling.



Arguments

Variable	Description
N	Frequency replication ratio (upsampling ratio)
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`iirrateup`, `zpkrateup`

allpassshift

Purpose

Allpass filter for real shift transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real frequency shift transformation. This transformation places one selected feature of an original filter, located at frequency W_o, at the required target frequency location, W_t. This transformation implements the "DC mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be moved to a different frequency by applying a shift transformation. In such a way you can avoid designing the filter from the beginning.

Examples

Design the allpass filter precisely shifting one feature of the lowpass filter originally at W_o=0.5 to the new frequencies of W_t=0.25:

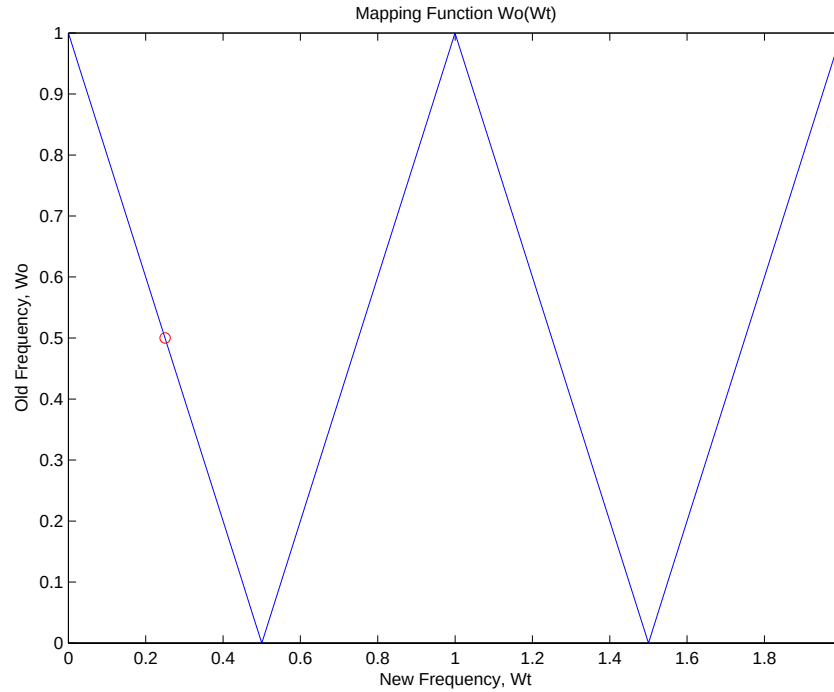
```
Wo = 0.5;
Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshift(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter

Variable	Description
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirshift`, `zpkshift`

Purpose

Allpass filter for complex shift transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)
[AllpassNum,AllpassDen] = allpassshiftc(0,0.5)
[AllpassNum,AllpassDen] = allpassshiftc(0,-0.5)
```

Description

`[AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)` returns the numerator, `AllpassNum`, and denominator, `AllpassDen`, vectors of the allpass mapping filter for performing a complex frequency shift of the frequency response of the digital filter by an arbitrary amount.

`[AllpassNum,AllpassDen] = allpassshiftc(0,0.5)` calculates the allpass filter for doing the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[AllpassNum,AllpassDen] = allpassshiftc(0,-0.5)` calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design the allpass filter precisely rotating the whole filter by the amount defined by the location of the selected feature from an original filter, W_o=0.5, and its required position in the target filter, W_t=0.25:

```
Wo = 0.5;
Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshiftc(Wo, Wt);
```

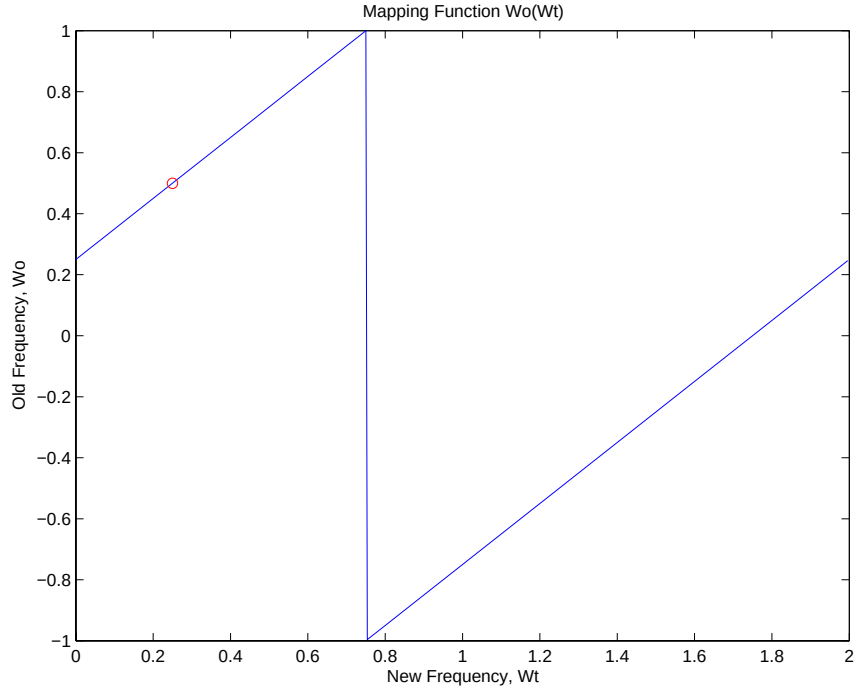
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

The figure shows you that the transformation by the mapping filter does exactly what you intend.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirshiftc`, `zpkshiftc`

References

Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On Digital Differentiators, Hilbert Transformers, and Half-band Low-pass Filters," *IEEE® Transactions on Education*, vol. 32, pp. 314-318, August 1989.

Purpose

Automatic dynamic range scaling

Syntax

```
autoscale(hd,x)
hnew = autoscale(hd,x)
```

Description

`autoscale(hd,x)` provides dynamic range scaling for each node of the filter `hd`. This method runs signal `x` through `hd` in floating-point to simulate filtering. `autoscale` uses the maximum and minimum data obtained from that simulation at each filter node to set fraction lengths to cover the simulation full range and maximize the precision. Word lengths are not changed during autoscaling.

`hnew = autoscale(hd,x)` If you request an output, `autoscale` returns a new filter with the scaled fraction lengths. The original filter is not changed.

For introductory demonstrations of the automatic scale process, refer to the following demos in the toolbox:

- Fixed-Point Scaling of an Elliptic IIR Filter
- Floating-Point to Fixed-Point Conversion of IIR Filters
- Floating-Point to Fixed-Point Conversion of IIR Filters

Examples

To demonstrate the autoscaling capability using a set of input data, this example uses a bandpass IIR lattice filter with random input data. To run this example in MATLAB, you must enable logging for `fi` objects—

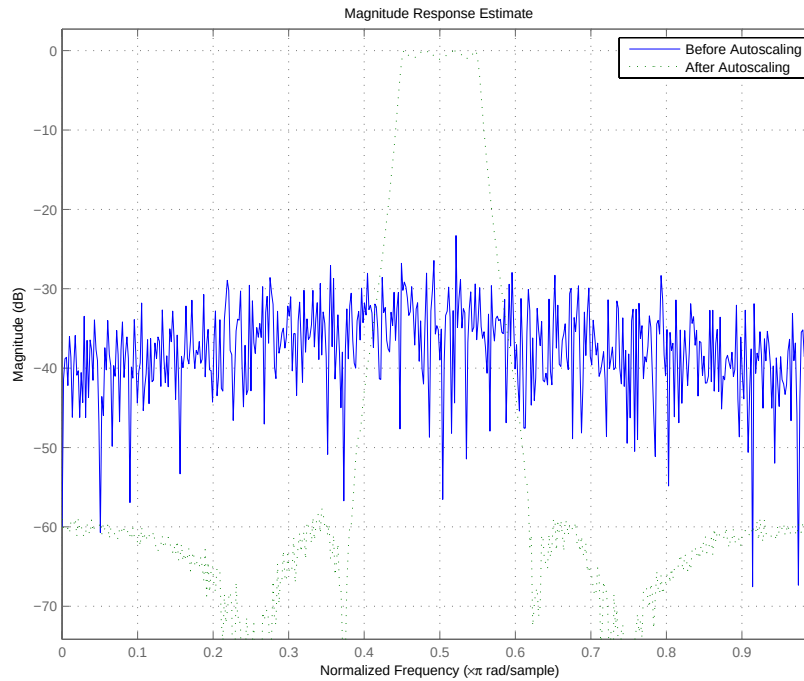
```
fipref('LoggingMode','on')
```

After you have enabled logging, this example uses a lattice ARMA filter to demonstrate automatic scaling with `autoscale`..

```
hd = design(fdesign.bandpass,'ellip');
hd = convert(hd,'latticearma');
hd.arithmetic = 'fixed';
rand('state', 4)
x = rand(100,10); % Training input data.
```

```
hd(2) = autoscale(hd,x);
hfvt = fvtool(hd,'Analysis','magestimate',...
'Showreference','off');
legend(hfvt,'Before Autoscaling', 'After Autoscaling')
```

After you run `autoscale`, the resulting plot uses `FVTool` with before and after curves.



See Also

`qreport`

block

Purpose

Generate block from multirate filter

Syntax

```
block(hm)
block(hm,'propertyname1',propertyvalue1,'propertyname2',
   propertyvalue2,...)
```

Description

`block(hm)` generates a Signal Processing Blockset block equivalent to `hm`.

`block(hm,'propertyname1',propertyvalue1,'propertyname2',propertyvalue2,..` generates a Signal Processing Blockset block using the options specified in the property name/property value pairs. The valid properties and their values are

Property Name	Description and Values
`Destination`	Determines which Simulink® model gets the block. Enter `current`, `new`, or specify the name of an existing subsystem with *`subsystemname`*. Specifying `new` opens a new model and adds the block. `Current` adds the block to your current Simulink model. `Current` is the default setting. If you provide the name of a current subsystem in *`subsystemname`*, block adds the new block to your specified subsystem.
`Blockname`	Specifies the name of the generated block. The name appears below the block in the model. When you do not specify a block name, the default is `filter`.

Property Name	Description and Values
`OverwriteBlock`	Tells `block` whether to overwrite an existing block of the same name, or create a new block. `Off` is the default setting—`block` does not overwrite existing blocks with matching names. Switching from `off` to `on` directs `block` to overwrite existing blocks.
`MapStates`	Specifies whether to apply the current filter states to the new block. 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 block. Choosing `on` preserves the current filter states in the block.

Using block to Realize Fixed-Point Multirate Filters

When the source filter `hm` is fixed-point, such as an FIR decimator with fixed-point arithmetic, `block` maps the fixed-point properties for `hm` to the new block according to a set of rules:

- The input word and fraction lengths for the block are derived from the block input signal. The realization process ignores the input word and input fraction lengths that are part of the source filter object, choosing to inherit the settings from the input data. You see a warning message in MATLAB that points this out.
- Rounding modes that the block does not support — `fix`, `ceil`, and `convergent` — convert to `nearest` in the filter block. Supported rounding modes do not change. MATLAB warns you about this change.

Other fixed-point properties map directly to settings for word and fraction length in the realized block.

Examples

Two examples of using `block` demonstrate the syntax capabilities. Both examples start from an `mfilt` object with interpolation factor of three.

In the first example, use `block` with the default syntax, letting the function determine the block name and configuration.

```
l = 3; % Interpolation factor
hm = mfilt.firdecim(l);
```

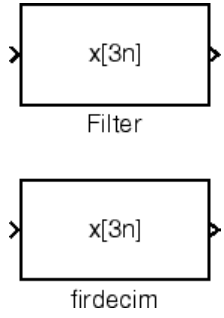
Now use the default syntax to create a block.

```
block(hm);
```

In this second example, define the block name to meet your needs by using the property name/property value pair input arguments.

```
block(hm, 'blockname', 'firdecim');
```

The figure below shows the blocks in a Simulink model. When you try these examples, you see that the second block writes over the first block location. You can avoid this by moving the first block before you generate the second, always naming your block with the `blockname` property, or setting the `Destination` property to `new` which puts the filter block in a new Simulink model.



See Also

Refer to “Realizing Filters as Simulink Subsystem Blocks” in FDATool, and `realizemdl`

Purpose

Butterworth IIR filter design using specification object

Syntax

```
hd = design(d,'butter')
hd = design(d,'butter',designoption,value...)
```

Description

`hd = design(d,'butter')` designs a Butterworth IIR digital filter using the specifications supplied in the object `d`.

`hd = design(d,'butter',designoption,value...)` returns a Butterworth IIR filter where you specify a design option and value.

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 `butter`, refer to the command line help system. For example, to get specific information about using `butter` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'butter')
```

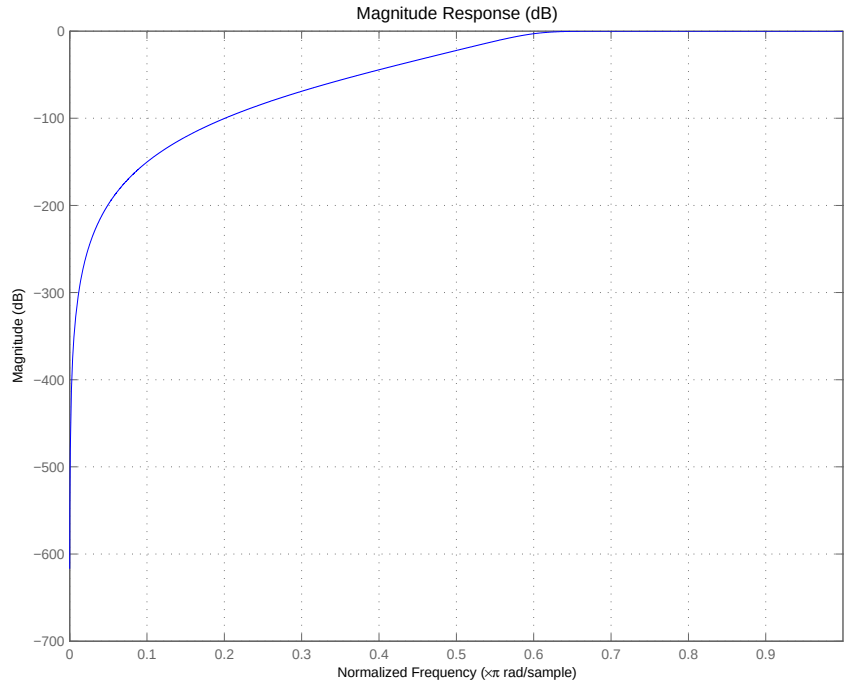
Examples

The first example constructs a default lowpass filter specification object and uses it to design a Butterworth filter.

```
d = fdesign.lowpass;
designopts(d,'butter')

ans =

    FilterStructure: 'df2sos'
       MatchExactly: 'stopband'
hd = design(d,'butter','matchexactly','stopband');
```

Example 2 constructs a highpass filter specification object with order (`n`) and cutoff frequency (`fc`) specifications, and then designs a Butterworth filter from the object.

```
d = fdesign.highpass('n,fc',8,.6);
design(d,'butter');
```



See Also

`cheby1`, `cheby2`, `ellip`

Purpose

Convert coupled allpass filter to transfer function form

Syntax

```
[b,a]=ca2tf(d1,d2)
[b,a]=ca2tf(d1,d2,beta)
[b,a,bp]=ca2tf(d1,d2)
[b,a,bp]=ca2tf(d1,d2,beta)
```

Description

`[b,a]=ca2tf(d1,d2)` returns the vector of coefficients `b` and the vector of coefficients `a` corresponding to the numerator and the denominator of the transfer function

$$H(z) = B(z)/A(z) = \frac{1}{2}[H1(z) + H2(z)]$$

`d1` and `d2` are real vectors corresponding to the denominators of the allpass filters `H1(z)` and `H2(z)`.

`[b,a]=ca2tf(d1,d2,beta)` where `d1`, `d2` and `beta` are complex, returns the vector of coefficients `b` and the vector of coefficients `a` corresponding to the numerator and the denominator of the transfer function

$$H(z) = B(z)/A(z) = \frac{1}{2}[-(\bar{\beta}) \bullet H1(z) + \beta \bullet H2(z)]$$

`[b,a,bp]=ca2tf(d1,d2)`, where `d1` and `d2` are real, returns the vector `bp` of real coefficients corresponding to the numerator of the power complementary filter G(z)

$$G(z) = Bp(z)/A(z) = \frac{1}{2}[H1(z) - H2(z)]$$

`[b,a,bp]=ca2tf(d1,d2,beta)`, where `d1`, `d2` and `beta` are complex, returns the vector of coefficients `bp` of real or complex coefficients that correspond to the numerator of the power complementary filter G(z)

$$\mathcal{G}(z) = Bp(z)/A(z) = \frac{1}{2j}[-(\bar{\beta}) \bullet H1(z) + \beta \bullet H2(z)]$$

Examples

Create a filter, convert the filter to coupled allpass form, and convert the result back to the original structure (create the power complementary filter as well).

```
[b,a]=cheby1(10,.5,.4);
[d1,d2,beta]=tf2ca(b,a);      % tf2ca returns the %
                              denominators of the %
                              allpasses.
[num,den,numpc]=ca2tf(d1,     % Reconstruct the original
d2,beta);                     % filter plus the power %
                              complementary one.
[h,w,s]=freqz(num,den);
hpc = freqz(numpc,den);
s.plot = 'mag';
s.yunits = 'sq';
freqzplot([h hpc],w,s);       % Plot the mag response of
                              the % original filter and
                              the % power complementary
                              one.
```

See Also

`cl2tf`, `iirpowcomp`, `tf2ca`, `tf2cl`

Purpose

Chebyshev Type I filter using specification object

Syntax

```
hd = design(d,'cheby1')
hd = design(d,'cheby1',designoption,value,designoption,
value,...)
```

Description

`hd = design(d,'cheby1')` designs a Chebyshev I IIR digital filter using the specifications supplied in the object `d`.

`hd = design(d,'cheby1',designoption,value,designoption, value,...)` returns a Chebyshev I IIR 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 `cheby1`, refer to the command line help system. For example, to get specific information about using `cheby1` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'cheby1')
```

Examples

These examples use filter specification objects to construct Chebyshev type I filters. In the first example, you use the `matchexactly` option to ensure the performance of the filter in the passband.

```
d = fdesign.lowpass
designopts(d,'cheby1')
ans =

    FilterStructure: 'df2sos'
       MatchExactly: 'passband'

hd = design(d,'cheby1','matchexactly','passband')
```

```
d =

               Response: 'Lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.45
                  Fstop: 0.55
                  Apass: 1
                  Astop: 60

hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [5x6 double]
             ScaleValues: [6x1 double]
        PersistentMemory: false
```

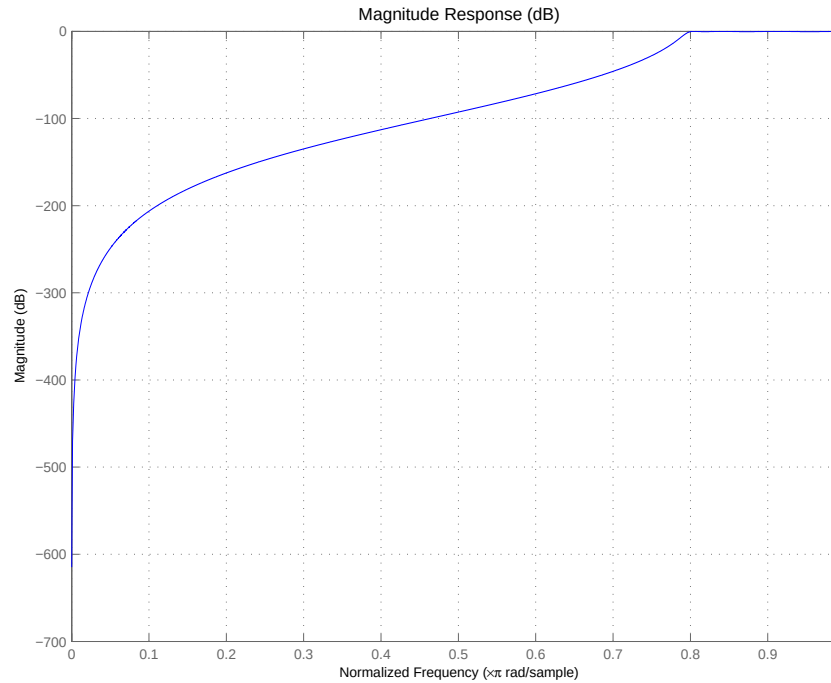
`cheby1` also design highpass filters, among others. Specify the filter order, passband edge frequency. and the passband ripple to get the filter exactly as required.

```
d = fdesign.highpass('n,fp,ap',7,20,.4,50)
hd = design(d,'cheby1')

d =

               Response: 'Highpass'
          Specification: 'N,Fp,Ap'
            Description: {3x1 cell}
    NormalizedFrequency: false
                     Fs: 50
            FilterOrder: 7
                  Fpass: 20
                  Apass: 0.4
```

```
hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'double'
                sosMatrix: [4x6 double]
             ScaleValues: [5x1 double]
        PersistentMemory: false
```

Use `fvtool` to view the resulting filter.

```
fvtool(hd)
```



By design, `cheby1` returns filters that use second-order sections. For many applications, and for most fixed-point applications, SOS filters are particularly well-suited.

See Also

`butter`, `cheby2`, `ellip`

Purpose

Chebyshev Type II filter using specification object

Syntax

```
hd = design(d,'cheby2')
hd = design(d,'cheby2',designoption,value,designoption,
value,...)
```

Description

`hd = design(d,'cheby2')` designs a Chebyshev II IIR digital filter using the specifications supplied in the object `d`.

`hd = design(d,'cheby2',designoption,value,designoption, value,...)` returns a Chebyshev II IIR 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 `cheby1`, refer to the command line help system. For example, to get specific information about using `cheby2` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'cheby2')
```

Examples

These examples use filter specification objects to construct Chebyshev type I filters. In the first example, you use the `matchexactly` option to ensure the performance of the filter in the passband.

```
d = fdesign.lowpass;
hd = design(d,'cheby2','matchexactly','passband')


hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'double'
```

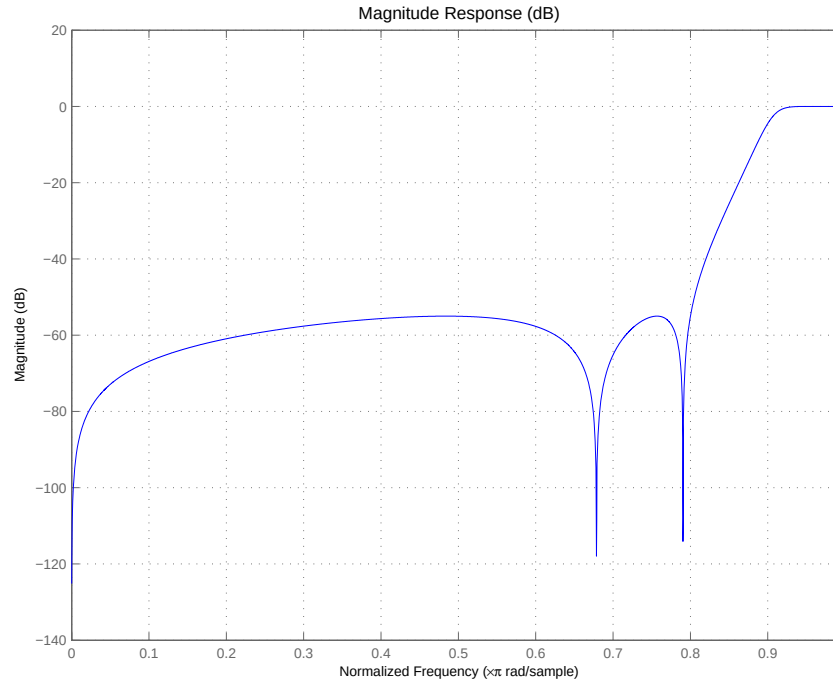
```
                sosMatrix: [5x6 double]
              ScaleValues: [6x1 double]
         PersistentMemory: false
```

`cheby2` also design highpass, bandpass, and bandstop filters. Here is a highpass filter where you specify the filter order, the stopband edge frequency. and the stopband attenuation to get the filter exactly as required.

```
d = fdesign.highpass('n,fst,ast',5,20,55,50)

d =
                Response: 'Highpass'
           Specification: 'N,Fst,Ast'
             Description: {3x1 cell}
    NormalizedFrequency: false
                     Fs: 50
             FilterOrder: 5
                   Fstop: 20
                   Astop: 55

hd=design(d,'cheby2')

hd =
 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
       sosMatrix: [3x6 double]
     ScaleValues: [4x1 double]
PersistentMemory: false
```

The Filter Visualization Tool shows the highpass filter meets the specifications.

```
fvtool(hd)
```



By design, `cheby2` returns filters that use second-order sections. For many applications, and for most fixed-point applications, SOS filters are particularly well-suited for use.

See Also

`butter`, `cheby1`, `ellip`

Purpose

Convert coupled allpass lattice to transfer function form

Syntax

```
[b,a] = cl2tf(k1,k2)
[b,a] = cl2tf(k1,k2,beta)
[b,a,bp] = cl2tf(k1,k2)
[b,a,bp] = cl2tf(k1,k2,beta)
```

Description

`[b,a] = cl2tf(k1,k2)` returns the numerator and denominator vectors of coefficients `b` and `a` corresponding to the transfer function

$$H(z) = B(z)/A(z) = \frac{1}{2}[H1(z) + H2(z)]$$

where *H1(z)* and *H2(z)* are the transfer functions of the allpass filters determined by `k1` and `k2`, and `k1` and `k2` are real vectors of reflection coefficients corresponding to allpass lattice structures.

`[b,a] = cl2tf(k1,k2,beta)` where `k1`, `k2` and `beta` are complex, returns the numerator and denominator vectors of coefficients b and a corresponding to the transfer function

$$H(z) = B(z)/A(z) = \frac{1}{2}[-(\bar{\beta}) \bullet H1(z) + \beta \bullet H2(z)]$$

`[b,a,bp] = cl2tf(k1,k2)` where `k1` and `k2` are real, returns the vector `bp` of real coefficients corresponding to the numerator of the power complementary filter *G(z)*

$$G(z) = Bp(z)/A(z) = \frac{1}{2}[H1(z) - H2(z)]$$

`[b,a,bp] = cl2tf(k1,k2,beta)` where `k1`, `k2` and `beta` are complex, returns the vector of coefficients `bp` of possibly complex coefficients corresponding to the numerator of the power complementary filter *G(z)*

$$\mathfrak{I}(z) = Bp(z)/A(z) = \frac{1}{2j}[-(\bar{\beta}) \bullet H1(z) + \beta \bullet H2(z)]$$

Examples

```
[b,a]=cheby1(10,.5,.4);
 %TF2CL returns the reflection coeffs
[k1,k2,beta]=tf2cl(b,a);
% Reconstruct the original filter
% plus the power complementary one.
[num,den,numpc]=cl2tf(k1,k2,beta);
[h,w,s1]=freqz(num,den);
hpc = freqz(numpc,den);
s.plot = 'mag';
s.yunits = 'sq';
% Plot the mag response of the original
%  filter and the power
% complementary one.
freqzplot([h hpc],w,s1);
```

See Also

tf2cl, tf2ca, ca2tf, tf2latc, latc2tf, iirpowcomp

Purpose

Coefficients for filters

Syntax

```
s = coeffs(ha)
s = coeffs(hd)
s = coeffs(hm)
```

Description

The next sections describe common `coeffs` operation with adaptive, discrete-time, and multirate filters.

Adaptive Filters

`s = coeffs(ha)` returns a structure `s` containing the coefficients of adaptive filter `ha`. These are the instantaneous filter coefficients available at the time you use the function.

Discrete-Time Filters

`s = coeffs(hd)` returns a structure `s` that contains the coefficients of discrete-time filter `hd`.

Multirate Filters

`s = coeffs(hm)` returns `s`, a structure containing the coefficients of discrete-time filter `hm`. CIC-based filters do not have coefficients and this function does not work with constructors like `mfilt.cicdecim`.

Examples

`coefficients` works the same way for all filters. This example uses a multirate filter `hm` to demonstrate the function.

```
hm=mfilt.firdecim(3)

hm =

         FilterStructure: 'Direct-Form FIR Polyphase Decimator'
               Numerator: [1x72 double]
        DecimationFactor: 3
        PersistentMemory: false
                  States: [69x1 double]

s=coeffs(hm)
```

```
s =

    [1x72 double]

s.Numerator

ans =

  Columns 1 through 8

         0   -0.0000   -0.0001         0    0.0002    0.0003         0   -0.0005

  Columns 9 through 16

   -0.0007         0    0.0011    0.0014         0   -0.0022   -0.0028         0

  Columns 17 through 24

    0.0040    0.0048         0   -0.0068   -0.0080         0    0.0111    0.0129

  Columns 25 through 32

         0   -0.0177   -0.0207         0    0.0287    0.0342         0   -0.0513

  Columns 33 through 40

   -0.0659         0    0.1363    0.2749    0.3333    0.2749    0.1363         0

  Columns 41 through 48

   -0.0659   -0.0513         0    0.0342    0.0287         0   -0.0207   -0.0177

  Columns 49 through 56

         0    0.0129    0.0111         0   -0.0080   -0.0068         0    0.0048
```

```
Columns 57 through 64

  0.0040         0   -0.0028   -0.0022         0    0.0014    0.0011         0

Columns 65 through 72

 -0.0007   -0.0005         0    0.0003    0.0002         0   -0.0001   -0.0000
```

See Also

adaptfilt, freqz, grpdelay, impz, info, phasez, stepz, zerophase, zplane

Purpose

Read Xilinx COE file

Syntax

```
hd = coeread(filename)
```

Description

`hd = coeread(filename)` extracts the Distributed Arithmetic FIR filter coefficients defined in the XILINX CORE Generator .COE file specified by `filename`. It returns a `dfilt` object, the fixed-point filter `hd`. If you do not provide the file type extension `.coe` with the `filename`, the function assumes the `.coe` extension.

See Also

`coewrite`, `dfilt`, `dfilt.dffir`

Purpose

Write Xilinx COE file

Syntax

```
coewrite(hd)
coewrite(hd,radix)
coewrite(...,filename)
```

Description

`coewrite(hd)` writes a XILINX Distributed Arithmetic FIR filter coefficient .COE file which can be loaded into the XILINX CORE Generator. The coefficients are extracted from the fixed-point `dfilt` object `hd`. Your fixed-point filter must be a direct form FIR structure `dfilt` object with one section and whose `Arithmetic` property is set to `fixed`. You cannot export single-precision, double-precision, or floating-point filters as `.coe` files, nor multiple-section filters. To enable you to provide a name for the file, `coewrite` displays a dialog box where you fill in the file name. If you do not specify the name of the output file, the default file name is `untitled.coe`.

`coewrite(hd,radix)` indicates the radix (number base) used to specify the FIR filter coefficients. Valid `radix` values are `2` for binary, `10` for decimal, and `16` for hexadecimal (default).

`coewrite(...,filename)` writes a XILINX.COE file to `filename`. If you omit the file extension, `coewrite` adds the `.coe` extension to the name of the file.

Examples

`coewrite` generates an ASCII text file that contains the filter coefficients in a format the XILINX CORE Generator can read and load. In this example, you create a 30th-order fixed-point filter and generate the `.coe` file that include the filter coefficients as well as associated information about the filter.

```
b = firceqrip(30,0.4,[0.05 0.03]);
hq = dfilt.dffir(b);
set(hq,'arithmetic','fixed');
coewrite(hq,10,'mycoefile');
```

When you look at `mycoefile.coe`, you see the following:

```
;
; XILINX CORE Generator(tm) Distributed Arithmetic
; FIR filter coefficient (.COE) File
; Generated by MATLAB(tm) and Filter Design Toolbox.
;
; Generated on: 4-Dec-2003 13:47:15
;
Radix = 10;
Coefficient_Width = 16;
CoefData =   -41,
 -851,
 -366,
  308,
  651,
   22,
 -873,
 -658,
  749,
 1504,
   21,
-2367,
-2012,
 3014,
 9900,
....
```

`coewrite` puts the filter coefficients in column-major order and reports the radix, the coefficient width, and the coefficients. These represent the minimum set of data needed in a `.coe` file.

See Also

`coeread`, `dfilt`, `dfilt.dffir`

Purpose

Convert filter structure of discrete-time or multirate filter

Syntax

```
hq = convert(hq,newstruct)
hm = convert(hm,newstruct)
```

Description

Discrete-Time Filters

`hq = convert(hq,newstruct)` returns a quantized filter whose structure has been transformed to the filter structure specified by string `newstruct`. You can enter any one of the following quantized filter structures:

- `'antisymmetricfir'`: Antisymmetric finite impulse response (FIR).
- `'df1'`: Direct form I.
- `'df1t'`: Direct form I transposed.
- `'df2'`: Direct form II.
- `'df2t'`: Direct form II transposed. Default filter structure.
- `'dffir'`: FIR.
- `'dffirt'`: Direct form FIR transposed.
- `'latcallpass'`: Lattice allpass.
- `'latticeca'`: Lattice coupled-allpass.
- `'latticecapc'`: Lattice coupled-allpass power-complementary.
- `'latticear'`: Lattice autoregressive (AR).
- `'latticema'`: Lattice moving average (MA) minimum phase.
- `'latcmax'`: Lattice moving average (MA) maximum phase.
- `'latticearma'`: Lattice ARMA.
- `'statespace'`: Single-input/single-output state-space.
- `'symmetricfir'`: Symmetric FIR. Even and odd forms.

All filters can be converted to the following structures:

- `df1`
- `df1t`
- `df2`
- `df2t`
- `statespace`
- `latticearma`

For the following filter classes, you can specify other conversions as well:

- Minimum phase FIR filters can be converted to `latticema`
- Maximum phase FIR filters can be converted to `latcmax`
- Allpass filters can be converted to `latcallpass`

`convert` generates an error when you specify a conversion that is not possible.

Multirate Filters

`hm = convert(hm,newstruct)` returns a multirate filter whose structure has been transformed to the filter structure specified by string `newstruct`. You can enter any one of the following multirate filter structures, defined by the strings shown, for `newstruct`:

Cascaded Integrator-Comb Structure

- `cicdecim` — CIC-based decimator
- `cicinterp` — CIC-based interpolator

FIR Structures

- `firdecim` — FIR decimator
- `firtdecim` — transposed FIR decimator
- `firfracdecim` — FIR fractional decimator

- `firinterp` — FIR interpolator
- `firfracinterp` — FIR fractional interpolator
- `firsrc` — FIR sample rate change filter
- `firholdinterp` — FIR interpolator that uses hold interpolation between input samples
- `firlinearinterp` — FIR interpolator that uses linear interpolation between input samples
- `fftfirinterp` — FFT-based FIR interpolator

You cannot convert between the FIR and CIC structures.

Examples

```
[b,a]=ellip(5,3,40,.7);
hq = dfilt.df2t(b,a);
hq2 = convert(hq,'df1')
hq2 =

         FilterStructure: 'Direct-Form I'
              Arithmetic: 'double'
               Numerator: [0.1980 0.7886 1.4236 1.4236 0.7886 0.1980]
             Denominator: [1 1.4339 1.8021 0.6139 0.2047 -0.2342]
        PersistentMemory: false
                  States: Numerator:  [5x1 double]
                          Denominator:[5x1 double]
```

For an example of changing the structure of a multirate filter, try the following conversion from a CIC interpolator to a CIC interpolator with zero latency.

```
hm = mfilt.cicinterp(2,2,3,8,8)
hm =

         FilterStructure: 'Cascaded Integrator-Comb Interpolator'
              Arithmetic: 'int'
       DifferentialDelay: 2
        NumberOfSections: 3
```

```
     InterpolationFactor: 2
               RoundMode: 'floor'
          PersistentMemory: false
                    States: Integrator: [3x1 States]
                                Comb: [3x1 States]

           InputWordLength: 8

      SectionWordLengthMode: 'MinWordLengths'

           OutputWordLength: 8

hm2=convert(hm,'cicinterp')

hm2 =

           FilterStructure: 'Cascaded Integrator-Comb Interpolator'
                Arithmetic: 'fixed'
        DifferentialDelay: 2
         NumberOfSections: 3
      InterpolationFactor: 2
         PersistentMemory: false

           InputWordLength: 8
           InputFracLength: 15

           FilterInternals: 'MinWordLengths'
          OutputWordLength: 8
```

See Also

`mfilt`

`dfilt` in Signal Processing Toolbox™ documentation

Purpose

Cost of using discrete-time or multirate filter

Syntax

```
c = cost(hd)
c = cost(hm)
```

Description

`c = cost(hd)` and `c = cost(hm)` return a cost estimate `c` for the filter `hd` or `hm`. The returned cost estimate contains the following fields.

Estimated Value	Property	Description
`Number of Multiplications`	`nmult`	Number of multiplications during the filter run. `nmult` ignores multiplications by -1, 0, and 1 in the total multiple.
`Number of Additions`	`nadd`	Number of additions during the filter run.
`Number of States`	`nstates`	Number of states the filter uses.
`MultPerInputSample`	`multperinputsample`	Number of multiplication operations performed for each input sample
`AddPerInputSample`	`addperinputsample`	Number of addition operations performed for each input sample

Examples

These examples show you the `cost` method applied to `dfilt` and `mfilt` objects.

```
hd = design(fdesign.lowpass);
c = cost(hd)
c =
```

```
Number of Multipliers : 43
Number of Adders      : 42
Number of States      : 42
MultPerInputSample    : 43
AddPerInputSample     : 42
hd

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x43 double]
    PersistentMemory: false
```

When you are using a multirate filter object, `cost` works the same way.

```
d = fdesign.decimator(4,'cic');
hm = design(d,'multisection')

hm =

  FilterStructure: 'Cascaded Integrator-Comb Decimator'
       Arithmetic: 'fixed'
DifferentialDelay: 1
 NumberOfSections: 2
 DecimationFactor: 4
 PersistentMemory: false

 InputWordLength: 16
 InputFracLength: 15

 FilterInternals: 'FullPrecision'

c=cost(hm)

c =
```

```
Number of Multipliers : 0
Number of Adders      : 4
Number of States      : 4
MultPerInputSample    : 0
AddPerInputSample     : 2.5
```

See Also

qreport

Purpose

Vector of SOS filters for cumulative sections

Syntax

```
h = cumsec(hd)
h = cumsec(hd,indices)
h = cumsec(hd,indices,secondary)
cumsec(hd,...)
```

Description

`h = cumsec(hd)` returns a vector `h` of SOS filter objects with the cumulative sections. Each element in `h` is a filter with the structure of the original filter. The first element is the first filter section of `hd`. The second element of `h` is a filter that represents the combination of the first and second sections of `hd`. The third element of `h` is a filter which combines sections 1, 2, and 3 of `hd`. this pattern continues until the final element of `h` contains all the sections of `hd` and should be identical to `hd`.

`h = cumsec(hd,indices)` returns a vector `h` of SOS filter objects whose indices into the original filter are in the vector `indices`. Now you can specify the filter sections `cumsec` uses to compute the cumulative responses.

`h = cumsec(hd,indices,secondary)` when `secondary` is true, `cumsec` uses the secondary scaling points in the sections to determine where the sections should be split. This option applies only when `hd` is a `df2sos` and `df1tsos` filter. For these second-order section structures, the secondary scaling points refer to the scaling locations between the recursive and the nonrecursive parts of the section (the "middle" of the section). Argument `secondary` accepts either `true` or `false`. By default, `secondary` is `false`.

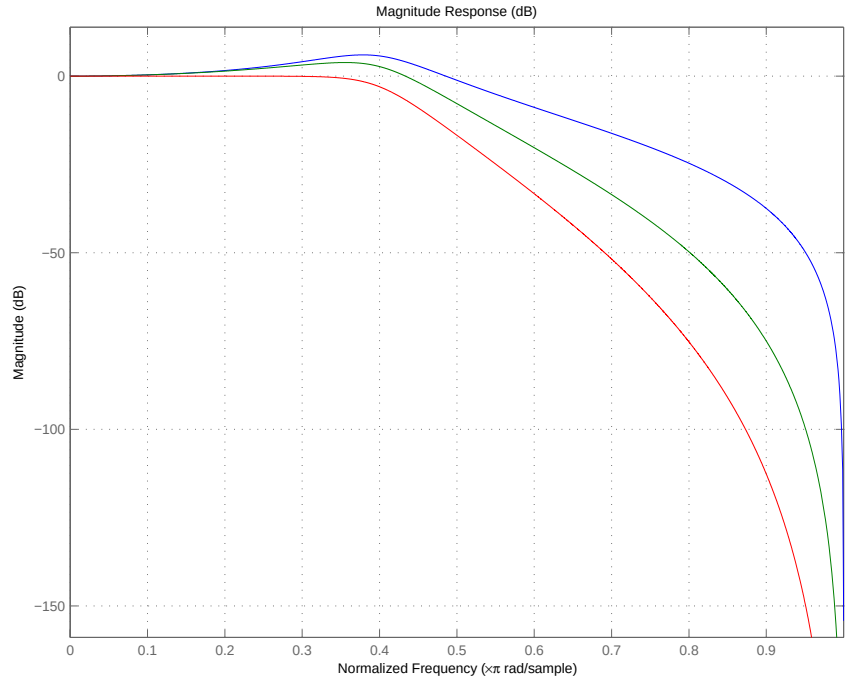
`cumsec(hd,...)` without an output arguments uses FVTool to plot the magnitude response of the cumulative sections.

Examples

To demonstrate how `cumsec` works, this example plots the relative responses of the sections of a sixth-order filter SOS filter with three sections. Each curve adds one more section to form the filter response.

```
hs = fdesign.lowpass('n,fc',6,.4);
hd = butter(hs);
```

```
h = cumsec(hd);
hfvt = fvtool(h);
legend(hfvt,'First Section','First Two Sections','Overall
Filter');
```



See Also

`scale`, `scalecheck`

Purpose

Undo filter coefficient and gain changes caused by `normalize`

Syntax

```
denormalize(hq)
```

Description

`denormalize(hq)` reverses the coefficient changes you make when you use `normalize` with `hq`. The filter coefficients do not change if you call `denormalize(hq)` before you use `normalize(hq)`. Calling `denormalize` more than once on a filter does not change the coefficients after the first `denormalize` call.

Examples

Make a quantized filter `hq` and normalize the filter coefficients. After normalizing the coefficients, restore them to their original values by reversing the effects of the `normalize` function.

```
d=fdesign.highpass('n,fc',14,0.45)

d =

               Response: 'Highpass'
          Specification: 'N,Fc'
            Description: {'Filter Order';'Cutoff Frequency'}
    NormalizedFrequency: true
            FilterOrder: 14
                Fcutoff: 0.45

hd = butter(d)

hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [7x6 double]
             ScaleValues: [8x1 double]
        PersistentMemory: false

hd.arithmetic='fixed'
```

```
hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'fixed'
               sosMatrix: [7x6 double]
             ScaleValues: [8x1 double]
        PersistentMemory: false

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

    StageInputWordLength: 16
     StageInputAutoScale: true

   StageOutputWordLength: 16
    StageOutputAutoScale: true

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

         StateWordLength: 16
         StateFracLength: 15

             ProductMode: 'FullPrecision'

               AccumMode: 'KeepMSB'
         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
hq=hd;
g=normalize(hq)'
```

```
g =

     2     2     2     2     2     2     2

hq.SosMatrix

ans =

    0.5000   -1.0000    0.5000    1.0000   -0.2817    0.8008
    0.5000   -1.0000    0.5000    1.0000   -0.2359    0.5081
    0.5000   -1.0000    0.5000    1.0000   -0.2051    0.3110
    0.5000   -1.0000    0.5000    1.0000   -0.1842    0.1776
    0.5000   -1.0000    0.5000    1.0000   -0.1704    0.0892
    0.5000   -1.0000    0.5000    1.0000   -0.1619    0.0350
    0.5000   -1.0000    0.5000    1.0000   -0.1579    0.0093

denormalize(hq)
hq.SosMatrix

ans =

    1.0000   -2.0000    1.0000    1.0000   -0.2817    0.8008
    1.0000   -2.0000    1.0000    1.0000   -0.2359    0.5081
    1.0000   -2.0000    1.0000    1.0000   -0.2051    0.3110
    1.0000   -2.0000    1.0000    1.0000   -0.1842    0.1776
    1.0000   -2.0000    1.0000    1.0000   -0.1704    0.0892
    1.0000   -2.0000    1.0000    1.0000   -0.1619    0.0350
    1.0000   -2.0000    1.0000    1.0000   -0.1579    0.0093
```

See Also

normalize

design

Purpose

Apply design method to specification object

Syntax

```
h = design(d)
h = design(d,designmethod)
h = design(d,designmethod,specname,specvalue,...)
```

Description

`h = design(d)` uses specifications object `d` to generate a filter `h`. When you do not provide a design method as an input argument, `design` chooses the design method to use by following these rules in the order listed.

1. Use `equiripple` if it applies to the object `d`.
2. When `equiripple` does not apply to `d`, use another FIR design method, such as `firls`.
3. If FIR design methods do not apply to `d`, use `ellip`.
4. When `ellip` does not apply to `d`, use another IIR design method, such as `butter` or `cheby2`, that applies to the object `d`.

More rules apply.

- `design` uses an FIR filter design method before using an IIR design method.
- `fdesign.nyquist` specifications objects use the `kaiserwin` design method as the first design choice, rather than `equiripple`, because `kaiserwin` produces better filters than `equiripple`.
- For decimators, interpolators, and rational sample rate changers that use `fdesign.nyquist` objects, the default design method is `kaiserwin`. Otherwise, those objects use the `equiripple` design method by default.

For more guidance about using `design` to design filters, refer to *Filter Design Toolbox™ Getting Started Guide* . There you find examples that use `design` to design filters and use methods in the toolbox to

analyze them. Alternatively, you can type the following at the MATLAB command prompt to obtain more information:

```
help design
```

`h = design(d,designmethod)` lets you specify a valid design method to design the filter as an input string. Note that the filter returned by `design` changes depending on the design method you choose. For more information about the filter that a design method returns, refer to the help for the design method.

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

```
designmethods(d)
```

for the specifications object `d`.

Valid entries depend on `d`. This is the complete set of design methods. The methods that apply to a specific specifications object usually represent a subset of this list.

- `butter`
- `cheby1`
- `cheby2`
- `ciccomp`
- `ellip`
- `equiripple`
- `firls`
- `ifir`
- `iirhilbert`
- `iirlinphase`
- `isinclp`

- kaiserwin
- lagrange
- multistage
- window

To help you design filters more quickly, the input argument designmethod accepts a variety of special keywords that force design to behave in different ways. The following table presents the keywords you can use for designmethod 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.
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. Refer to Examples to see this in use.

Keywords are not case sensitive and must be enclosed in single quotation marks like any string input.

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)
```

at the MATLAB prompt.

`h = design(d,designmethod,specname,specvalue,...)` with this syntax you can specify not only the design method but also values for the filter specifications in the method. Provide the specifications in the order of the name of the specification, such as the `FilterOrder`, followed by the value to assign to the specification. Enter as many `specname/specvalue` pairs as you need to define your filter. Any specification you do not define uses the default specification value. To use the `specname/specvalue` syntax, you must provide the design method to use in `designmethod`.

Examples

To demonstrate some of the design options, these examples use a few different input arguments and output arguments. For the first example, use `design` to return the default filter based on the default design method `equiripple`.

```
d = fdesign.lowpass(.2,.22);
hd = design(d) % Uses the default equiripple method.

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x202 double]
    PersistentMemory: false
```

In this example, use the **`allfir`** keyword with `design` to return an FIR filter for each valid design method for the specifications in specifications object `d`.

```
designmethods(d)


Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage

hallfir=design(d,'allfir')

hallfir =

 dfilt.basefilter: 1-by-4
```

`hallfir` contains filters designed using the `ellip`, `equiripple`, `ifir`, and `multistage` design methods, in the order shown by `designmethods(d)`. The first filter in `hallfir` comes from the `ellip` design method; the second from the `equiripple` method; the third from using `ifir` to design the filter; and the fourth from using `multistage`.

To see an individual filter, use an index with the filter object. For example, to see the second filter in `hallfir`, enter `hallfir(2)`

```
hallfir(2)

ans =

     FilterStructure: Cascade
             Stage(1): Direct-Form FIR
             Stage(2): Direct-Form FIR
    PersistentMemory: false
```

Here is the multistage filter `hallfir(4)`

```
hallfir(4)

ans =

 FilterStructure: Cascade
      Stage(1): Direct-Form FIR Polyphase Decimator
      Stage(2): Direct-Form FIR Polyphase Decimator
      Stage(3): Direct-Form FIR Polyphase Decimator
      Stage(4): Direct-Form FIR Polyphase Interpolator
      Stage(5): Direct-Form FIR Polyphase Interpolator
      Stage(6): Direct-Form FIR Polyphase Interpolator
PersistentMemory: false
```

This final example uses `equiripple` to design an FIR filter with the density factor set to 20 by using the `specname/specvalue` syntax.

```
[hd,res,err] = design(d,'equiripple','densityfactor',20);
hd

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x202 double]
    PersistentMemory: false
res

res =

    0.9903

err

err =

          order: 201
```

```
          fgrid: [2060x1 double]
              H: [2060x1 double]
          error: [2060x1 double]
            des: [2060x1 double]
             wt: [2060x1 double]
          iextr: [102x1 double]
          fextr: [102x1 double]
     iterations: 12
          evals: 12905
      edgeCheck: [4x1 double]
     returnCode: 0
```

`res` and `err` are optional output arguments that `design` returns when you specify the density factor with the `equiripple` design method.

See Also

`designmethods`, `butter`, `cheby1`, `cheby2`, `ellip`, `equiripple`, `firls`, `fdesign.halfband`, `kaiserwin`, `fdesign.nyquist`, `fdesign.rsrc`

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')
```

Description

`m = designmethods(d)` returns a list of the design methods available for the filter specification object `d` with its `Specification`. When you change the `Specification` for a filter specification object, the methods available to design filters from the object change.

Here are all the design methods and the filters they produce.

Design Method	Filter Result
`butter`	IIR
`cheby1`	IIR
`cheby2`	IIR
`ellip`	IIR
`equiripple`	FIR
`firls`	FIR
`freqsamp`	Frequency-sampled FIR
`ifir`	Interpolated FIR
`iirlinphase`	IIR filter with linear phase
`iirlpnorm`	IIR filter from an arbitrary magnitude specifications object. Compare to `iirls`.
`iirls`	IIR filter from an arbitrary magnitude and phase specifications object. Compare to `iirlpnorm`.
`kaiserwin`	FIR with Kaiser window
`lagrange`	Multirate filter with fractional delay

Design Method	Filter Result
multistage	Multistage filter that cascades multiple filters
window	FIR with windowed impulse response

`m = designmethods(d,'default')` returns the default design method for the filter specification object `d` and its current `Specification`.

`m = designmethods(d,type)` returns either the FIR or IIR design methods that apply to d, as specified by the `type` string, either `fir` or `iir`. By default, `designmethods` returns all the valid design methods when you omit the `type` string.

`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.

Examples

Construct a lowpass filter specification object and determine the design methods available to design a filter from the object.

```
d=fdesign.lowpass('n,fc',10,12000,48000)

d =

            Response: 'Lowpass'
       Specification: 'N,Fc'
         Description: {'Filter Order';'Cutoff Frequency'}
 NormalizedFrequency: false
                  Fs: 48000
         FilterOrder: 10
             Fcutoff: 12000

designmethods(d)


Design Methods for class fdesign.lowpass (N,Fc):
```

```
window

hd=window(d)

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x11 double]
    PersistentMemory: false
```

Now change the `Specification` string for `d` to `'fp,fst,ap,ast'` and determine the design methods that apply to your modified specifications object.

```
set(d,'specification','fp,fst,ap,ast');
d

d =

                Response: 'Lowpass'
           Specification: 'Fp,Fst,Ap,Ast'
             Description: {4x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
                  Fpass: 10800
                  Fstop: 13200
                  Apass: 1
                  Astop: 60

m2 = designmethods(d)
m3 = designmethods(d, 'iir')
m4 = designmethods(d, 'iir', 'full')

m2 =

    'butter'
```

```
    'cheby1'
    'cheby2'
    'ellip'
    'equiripple'
    'ifir'
    'kaiserwin'
    'multistage'


m3 =

    'butter'
    'cheby1'
    'cheby2'
    'ellip'


m4 =

    'Butterworth'
    'Chebyshev Type I'
    'Chebyshev Type II'
    'Elliptic'
```

Now you can get specific help on a particular design method for the specifications object. This example returns the help for the first design method for the `m2` set of methods — `butter`.

```
help(d,m2{1})
```

This is the same as `help(d,'butter')`.

See Also

`butter`, `cheby1`, `cheby2`, `designopts`, `ellip`, `equiripple`, `kaiserwin`, `multistage`

Purpose

Valid input arguments and values for specification object and method

Syntax

```
options = designopts(d,'designmethod')
```

Description

`options = designopts(d,'designmethod')` returns the structure `options` with the default design parameters used by the design method `designmethod`, specific to the response you defined for `d`. Replace `designmethod` with one of the strings returned by `designmethods`.

Use `help(d,`*`designmethod`*`)` to get a description of the design parameters. For example, to see the help for designing a highpass Chebyshev II filter from a specifications object `d`, enter

```
help(d,'cheby2')
```

at the prompt. MATLAB responds with help for Chebyshev II filter designs that use the specification `Fst`,`Fp`,`Ast`,`Ap`.

```
DESIGN Design a Chebyshev Type II iir filter.
HD = DESIGN(D, 'cheby2') designs a Chebyshev Type II
filter specified by the FDESIGN object H.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a
filter with the structure STRUCTURE.  STRUCTURE is
'df2sos' by default and can be any of the following.

'df1sos'
'df2sos'
'df1tsos'
'df2tsos'

HD = DESIGN(..., 'MatchExactly', MATCH) designs a
Chebyshev Type II filter and matches the frequency and
magnitude specification for the band MATCH exactly.
The other band will exceed the specification.
MATCH can be 'stopband' or 'passband' and is 'passband'
by default.
```

Examples

Design a minimum order, lowpass Butterworth filter. Use `designmethods` to determine the appropriate input arguments. Start by creating a lowpass filter specification object `d`.

```
d = fdesign.lowpass;
```

Because you want information about the input arguments for designing a filter using a design method, use `designmethods(d)` to get the list of valid methods.

```
designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

Pick one method and determine the design options for that method.

```
options = designopts(d,'butter')

options =

    FilterStructure: 'df2sos'
       MatchExactly: 'stopband'
```

In this example, the filter structure is Direct-Form II with second-order sections, and the design seeks to match the desired stopband performance exactly. As you see by reading the help, `FilterStructure` and `MatchExactly` are input arguments for designing the Butterworth filter.

Get help for designing a filter from `d` using the `butter` design method to see the arguments.

```
help(d,'butter')

 DESIGN Design a Butterworth IIR filter.
 HD = DESIGN(D, 'butter') designs a Butterworth filter specified by the
 FDESIGN object H.

 HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the
structure STRUCTURE.  STRUCTURE is 'df2sos' by default and can be any of
the following.

    'df1sos'
    'df2sos'
    'df1tsos'
    'df2tsos'

 HD = DESIGN(..., 'MatchExactly', MATCH) designs a Butterworth filter
 and matches the frequency and magnitude specification for the band
 MATCH exactly.  The other band will exceed the specification.  MATCH
 can be 'stopband' or 'passband' and is 'stopband' by default.
```

See Also

`design`, `designmethods`, `fdesign`

Purpose

Discrete-time filter

Syntax

```
hd = dfilt.structure(input1,...)
hd = [dfilt.structure(input1,...), dfilt.structure(input1,
   ...),...]
hd = design(d,'designmethod')
```

Description

`hd = dfilt.structure(input1,...)` returns a discrete-time filter, `hd`, of type *`structure`*. Each structure takes one or more inputs. When 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.structure` specify the type of filter structure. Available types of structures for `dfilt` are shown below.

dfilt.structure	Description
`dfilt.allpass`	Allpass filter
`dfilt.cascadeallpass`	Cascade of allpass filter sections
`dfilt.cascadewdfallpass`	Cascade of allpass wave digital filters
`dfilt.delay`	Delay
`dfilt.df1`	Direct-form I
`dfilt.df1sos`	Direct-form I, second-order sections
`dfilt.df1t`	Direct-form I transposed
`dfilt.df1tsos`	Direct-form I transposed, second-order sections

dfilt.structure	Description
dfilt.df2	Direct-form II
dfilt.df2sos	Direct-form II, second-order sections
dfilt.df2t	Direct-form II transposed
dfilt.df2tsos	Direct-form II transposed, second-order sections
dfilt.dffir	Direct-form FIR
dfilt.dffirt	Direct-form FIR transposed
dfilt.dfsymfir	Direct-form symmetric FIR
dfilt.dfasymfir	Direct-form antisymmetric FIR
dfilt.farrowfd	Generic fractional delay Farrow filter
dfilt.farrowlinearfd	Linear fractional delay Farrow filter
dfilt.fftfir	Overlap-add FIR
dfilt.latticeallpass	Lattice allpass
dfilt.latticear	Lattice autoregressive (AR)
dfilt.latticearma	Lattice autoregressive moving- average (ARMA)
dfilt.latticemamax	Lattice moving-average (MA) for maximum phase
dfilt.latticemamin	Lattice moving-average (MA) for minimum phase
dfilt.calattice	Coupled, allpass lattice
dfilt.calatticepc	Coupled, allpass lattice with power complementary output
dfilt.statespace	State-space
dfilt.scalar	Scalar gain object
dfilt.wdfallpass	Allpass wave digital filter object
dfilt.cascade	Filters arranged in series
dfilt.parallel	Filters arranged in parallel

For more information on each structure, refer to its reference page.

`hd = design(d,'designmethod')` returns the `dfilt` object `hd` resulting from the filter specification object `d` and the design method you specify in *`designmethod`*. When you omit the `designmethod` argument, `design` uses the default design method to construct a filter from the object `d`.

With this syntax, you design filters by

1. Specifying the filter specifications, such as the response shape (perhaps highpass) and details (passband edges and attenuation).
2. Selecting a method (such as `equiripple`) to design the filter.
3. Applying the method to the specifications object with `design(d,'designmethod)`.

Using the specification-based technique can be more effective than the coefficient-based filter design techniques.

Design Methods for design Syntax

When you use the `hd = design(d,'designmethod')` syntax, you have a range of design methods available depending on `d`, the filter specification object. The table below lists all of the design methods in the toolbox.

Design Method String	Filter Design Result
`butter`	Butterworth IIR
`cheby1`	Chebyshev Type I IIR
`cheby2`	Chebyshev Type II IIR
`ellip`	Elliptic IIR
`equiripple`	Equiripple with the same ripple in the pass and stopbands
`firls`	Least-squares FIR

Design Method String	Filter Design Result
freqsamp	Frequency-Sampled FIR
ifir	Interpolated FIR
iirlpnorm	Least Pth norm IIR
iirls	Least-Squares IIR
kaiserwin	Kaiser-windowed FIR
lagrange	Fractional delay filter
multistage	Multistage FIR
window	Windowed FIR

As specifications object d changes, the methods that apply for designing filters from d change. For instance, if d is a lowpass filter, these are the applicable methods:

```
% Create an object to design a lowpass filter.
d=fdesign.lowpass

d =

               Response: 'Lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.45
                  Fstop: 0.55
                  Apass: 1
                  Astop: 60

designmethods(d) % What design methods apply to object d?

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

When d is a bandstop filter, the design methods change.

```
% Create default bandstop specifications
d=fdesign.bandstop
object.

d =

               Response: 'Bandstop'
          Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
            Description: {7x1 cell}
    NormalizedFrequency: true
                  Fpass1: 0.35
                  Fstop1: 0.45
                  Fstop2: 0.55
                  Fpass2: 0.65
                  Apass1: 1
                   Astop: 60
                  Apass2: 1

designmethods(d) % Show design methods that apply to d.

Design Methods for class fdesign.bandstop
(Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2):

butter
cheby1
cheby2
```

```
ellip
equiripple
kaiserwin
```

The`ifir` and `multistage` design methods do not apply to this bandstop specifications object `d`.

Analysis 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 described here in this way.

Note If your variable `hd` 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 `zplane` is used without outputs.

Some of the methods listed here have the same name as functions in Signal Processing Toolbox™ or Filter Design Toolbox™ software. They behave similarly.

Method	Description
`addstage`	Adds a stage to a `cascade` or `parallel` object, where a stage is a separate, modular filter. Refer to `dfilt.cascade` and `dfilt.parallel`.

Method	Description
`block`	(Available only with Signal Processing Blockset) `block(hd)` creates a Signal Processing Blockset 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. Default value is `'Current'`. `'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 `Blockame`. Default is `'off'`. `'MapStates'` specifies initial conditions in the block (`'on'`). Default is `'off'`. Refer to "Using Filter States" in Signal Processing Toolbox documentation.
`cascade`	Returns the series combination of two `dfilt` objects. Refer to `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

Method	Description
`fcfwrite`	Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. If Filter Design Toolbox software 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`
`filter`	Performs filtering using the `dfilt` object
`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 `dfilt` information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length.

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

Method	Description
`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).
`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. Refer to `dfilt.parallel`.
`phasez`	Plots the phase response in `fvtool`

Method	Description
`realizemdl`	(Available only with Simulink) `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 or create a new model. Valid values are `'Current'` and `'New'`. `'OverwriteBlock'` specifies whether to overwrite an existing block that was created by `realizemdl` or create a new block. Valid values are `'on'` and `'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 block is `'off'`. `'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.

Method	Description
`removestage`	Removes a stage from a cascade or parallel `dfilt`. Refer to `dfilt.cascade` and `dfilt.parallel`.
`setstage`	Overwrites a stage of a cascade or parallel `dfilt`. Refer to `dfilt.cascade` and `dfilt.parallel`.
`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)`.

Method	Description
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.
tf	Converts the dfilt to a transfer function
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

Viewing Properties

As with any object, 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)
```

Note If you have Filter Design Toolbox software, dfilt objects include an arithmetic property. You can change the internal arithmetic of the filter from double- precision to single-precision using: hd.arithmetic = 'single'.

If you have both Filter Design Toolbox software and \&tm_fixedpointtoolbox; software Toolbox, you can change the arithmetic property to fixed-point using: hd.arithmetic = 'fixed'

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. Use single quotation marks around the `value` argument when the value is a string, such as `specifyall` or `fixed`.

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, independent 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 stage 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 `filtstates` 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 from a previous filtering operation as the initial conditions of the next filtering operation. The `true` setting also displays information about the filter `states`.

Note If you set the `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)

hd =
         FilterStructure: 'Direct-Form I'
               Numerator: [1x9 double]
             Denominator: [1x9 double]
        PersistentMemory: false

isstable(hd)
ans =
     1
```

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')

ans =
Columns 1 through 6
```

```
    0.0001  0.0009  0.0030  0.0060  0.0076  0.0060
  Columns 7 through 9
    0.0030  0.0009  0.0001
```

Create an array containing two `dfilt` objects, apply a method and verify that the method acts on both objects, and use a method to test whether the objects are FIR objects.

```
b = fir1(5,.5);
hd = dfilt.dffir(b);          % Create an FIR object
[b,a] = butter(5,.5);
hd(2) = dfilt.df2t(b,a);    % Create DF2T object and place
                           % in the second column of hd.

[h,w] = freqz(hd);
size(h)        % Verify that resulting h is
ans =        % 2 columns.
        8192           2
size(w)        % Verify that resulting w is
ans =        % 1 column.
        8192           1

test_fir = isfir(hd)
test_fir =
     1     0        % hd(1) is FIR and hd(2) is not.
```

Refer to the reference pages for each structure for more examples.

See Also

`dfilt`, `design`, `fdesign`, `realizemdl`, `sos`, `stepz`

`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`, `zplane` in Signal Processing Toolbox documentation

dfilt.allpass

Purpose

Allpass filter

Syntax

```
hd = dfilt.allpass(c)
```

Description

`hd = dfilt.allpass(c)` constructs an allpass filter with the minimum number of multipliers from the elements in vector `c`. To be valid, `c` must contain one, two, three, or four real elements. The number of elements in `c` determines the order of the filter. For example, `c` with two elements creates a second-order filter and `c` with four elements creates a fourth-order filter.

The transfer function for the allpass filter is defined by

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

given the coefficients in `c`.

To construct a cascade of allpass filter objects, use `dfilt.cascadeallpass`. For more information about creating cascades of allpass filters, refer to `dfilt.cascadeallpass`.

Properties

The following table provides a list of all the properties associated with an allpass `dfilt` object.

Property Name	Brief Description
`AllpassCoefficients`	Contains the coefficients for the allpass filter object
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.

Property Name	Brief Description
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to `filtstates` in Signal Processing Toolbox™ documentation or in the Help system.

Examples

This example constructs and displays the information about a second-order allpass filter that uses the minimum number of multipliers.

```
c = [1.5, 0.7];
% Create a second-order dfilt object.
hd = dfilt.allpass(c)
hd =

         FilterStructure: 'Minimum-Multiplier Allpass'
    AllpassCoefficients: [1.5 0.7]
       PersistentMemory: false
                 States: [0;0;0;0]

info(hd) % Gets information about the filter.
Discrete-Time IIR Filter (real)
-------------------------------
Filter Structure       : Minimum-Multiplier Allpass
Number of Multipliers  : 2
Stable                 : Yes
```

```
Linear Phase           : No

Implementation Cost
Number of Multipliers : 2
Number of Adders      : 4
Number of States      : 4
MultPerInputSample    : 2
AddPerInputSample     : 4
```

See Also

dfilt, dfilt.cascadeallpass, dfilt.cascadewdfallpass, dfilt.latticeallpass, mfilt.iirdecim, mfilt.iirinterp

Purpose

Coupled-allpass, lattice filter

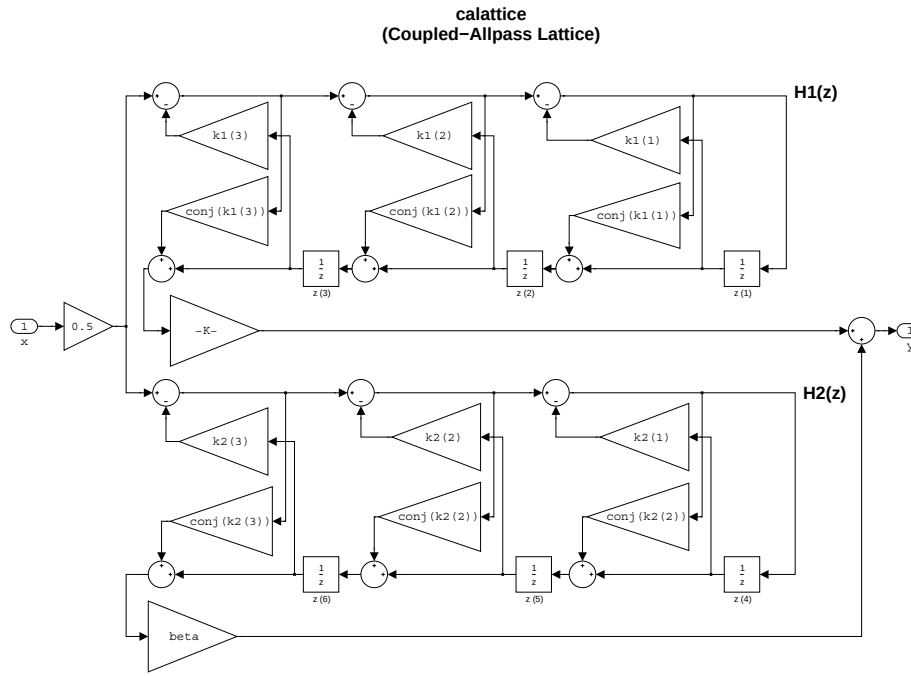
Syntax

```
hd = dfilt.calattice(k1,k2,beta)
hd = dfilt.calattice
```

Description

`hd = dfilt.calattice(k1,k2,beta)` returns a discrete-time, coupled-allpass, lattice filter object `hd`, which is two allpass, lattice filter structures coupled together. The lattice coefficients for each structure are vectors `k1` and `k2`. Input argument `beta` is shown in the diagram below.

`hd = dfilt.calattice` returns a default, discrete-time coupled-allpass, lattice filter object, `hd`. The default values are `k1 = k2 = []`, which is the default value for `dfilt.latticeallpass`, and `beta = 1`. This filter passes the input through to the output unchanged.



Example

Specify a third-order lattice coupled-allpass filter structure for a `dfilt` filter, `hd` with the following code.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i]
k2 = 0.7502 - 0.1218i
beta = 0.1385 + 0.9904i
hd = dfilt.calattice(k1,k2,beta)

k1 =

   0.9511 + 0.3088i
   0.7511 + 0.1158i

k2 =

   0.7502 - 0.1218i

beta =

   0.1385 + 0.9904i

 hd =

         FilterStructure: 'Coupled-Allpass Lattice'
                 Allpass1: [2x1 double]
                 Allpass2: 0.7502- 0.1218i
                     Beta: 0.1385+ 0.9904i
        PersistentMemory: false
                  States: [3x1 double]
```

The `Allpass1` and `Allpass2` properties store vectors of coefficients.

```
hd.Allpass1

ans =

   0.9511 + 0.3088i
   0.7511 + 0.1158i
```

See Also

`dfilt.calatticepc`

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticearma`, `dfilt.latticemamax`, `dfilt.latticemamin` in Signal Processing Toolbox™ documentation

dfilt.calatticepc

Purpose

Coupled-allpass, power-complementary lattice filter

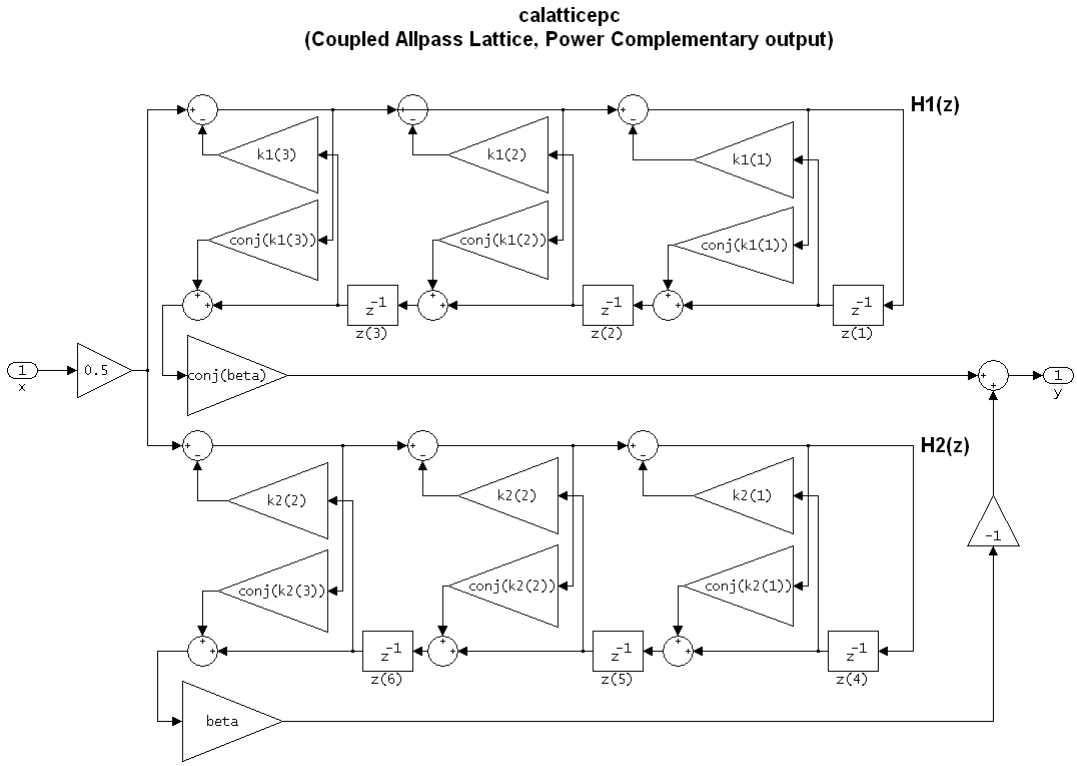
Syntax

```
hd = dfilt.calatticepc(k1,k2)
hd = dfilt.calatticepc
```

Description

`hd = dfilt.calatticepc(k1,k2)` returns a discrete-time, coupled-allpass, lattice filter object `hd`, with power-complementary output. This object is two allpass lattice filter structures coupled together to produce complementary output. The lattice coefficients for each structure are vectors, `k1` and `k2`, respectively. `beta` is shown in the following diagram.

`hd = dfilt.calatticepc` returns a default, discrete-time, coupled-allpass, lattice filter object `hd`, with power-complementary output. The default values are `k1 = k2 = []`, which is the default value for the `dfilt.latticeallpass`. The default for `beta = 1`. This filter passes the input through to the output unchanged.



Example

Specify a third-order lattice coupled-allpass power complementary filter structure for a filter `hd` with the following code. You see from the returned properties that `Allpass1` and `Allpass2` contain vectors of coefficients for the constituent filters.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i]
k2 = 0.7502 - 0.1218i
beta = 0.1385 + 0.9904i
hd = dfilt.calatticepc(k1,k2,beta)
k1 =

   0.9511 + 0.3088i
```

```
   0.7511 + 0.1158i


k2 =

   0.7502 - 0.1218i


beta =

   0.1385 + 0.9904i


hd =

          FilterStructure: 'Coupled-Allpass Lattice, Power
Complementary Output'
                 Allpass1: [2x1 double]
                 Allpass2: 0.7502- 0.1218i
                     Beta: 0.1385+ 0.9904i
         PersistentMemory: false
                   States: [3x1 double]
```

To see the coefficients for `Allpass1`, check the property values.

```
get(hd,'Allpass1')

ans =

   0.9511 + 0.3088i
   0.7511 + 0.1158i
```

See Also

`dfilt.calattice`

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticearma`, `dfilt.latticemamax`, `dfilt.latticemamin` in Signal Processing Toolbox™ documentation

Purpose

Cascade of discrete-time filters

Syntax

Refer to `dfilt.cascade` in Signal Processing Toolbox™ documentation for more information.

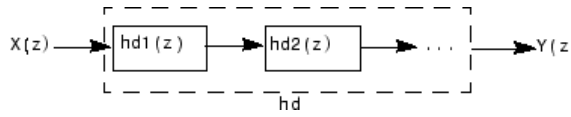
Description

`hd = dfilt.cascade(filterobject1,filterobject2,...)` returns a discrete-time filter object `hd` of type `cascade`, which is a serial interconnection of two or more filter objects `filterobject1`, `filterobject2`, and so on. `dfilt.cascade` accepts any combination of `dfilt` objects (discrete time filters) to cascade, as well as Farrow filter objects.

You can use the standard notation to cascade one or more filters:

```
cascade(hd1,hd2,...)
```

where `hd1`, `hd2`, and so on can be mixed types, such as `dfilt` objects and `mfilt` objects.



`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the cascade must be the same arithmetic format — `double`, `single`, or `fixed`. `hd`, the filter object returned, inherits the format of the cascaded filters.

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 with passband 0.4-0.6
hcas =
         Filterstructure: Cascade
              Section(1): Direct Form II Transposed
```

```
                Section(2): Direct Form II Transposed
          PersistentMemory: false
```

To view the details of one filter section, use

```
hcas.section(1)
 ans =
           FilterStructure: 'Direct Form II Transposed'
               Arithmetic: 'double'
                Numerator: [1x9 double]
              Denominator: [1x9 double]
        PersistentMemory: false
                  States: [8x1 double]
```

See Also

`dfilt`, `dfilt.parallel`, `dfilt.scalar`

Purpose

Cascade of allpass discrete-time filters

Syntax

```
hd = dfilt.cascadeallpass(c1,c2,...)
```

Description

`hd = dfilt.cascadeallpass(c1,c2,...)` constructs a cascade of allpass filters, each of which uses the minimum number of multipliers, given the filter coefficients provided in `c1`, `c2`, and so on.

Each vector `c` represents one section in the cascade filter. `c` vectors must contain one, two, three, or four elements as the filter coefficients for each section. As a result of the design algorithm, each section is a `dfilt.allpass` structure whose coefficients are given in the matching `c` vector, such as the `c1` vector contains the coefficients for the first stage.

States for each section are shared between sections.

Vectors `c` do not have to be the same length. You can combine various length vectors in the input arguments. For example, you can cascade fourth-order sections with second-order sections, or first-order sections.

For more information about the vectors `ci` and about the transfer function of each section, refer to `dfilt.allpass`.

Generally, you do not construct these allpass cascade filters directly. Instead, they result from the design process for an IIR filter. Refer to the first example in Examples for more about using `dfilt.cascadeallpass` to design an IIR filter.

Properties

In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
`AllpassCoefficients`	Contains the coefficients for the allpass filter object

Property Name	Brief Description
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to `filtstates` in Signal Processing Toolbox™ documentation or in the Help system.

Examples

Two examples show how `dfilt.cascadeallpass` works in very different applications — designing a halfband IIR filter and constructing an allpass cascade of `dfilt` objects.

First, design the IIR halfband filter using cascaded allpass filters. Each branch of the parallel cascade construction is a cascadeallpass filter object.

```
tw = 100;  % Transition width of filter to be designed, 100 Hz.
ast = 80;  % Stopband attenuation of filter to be designed, 80dB.
fs = 2000; % Sampling frequency of signal to be filtered.
% Store halfband design specs in the specifications object d.
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

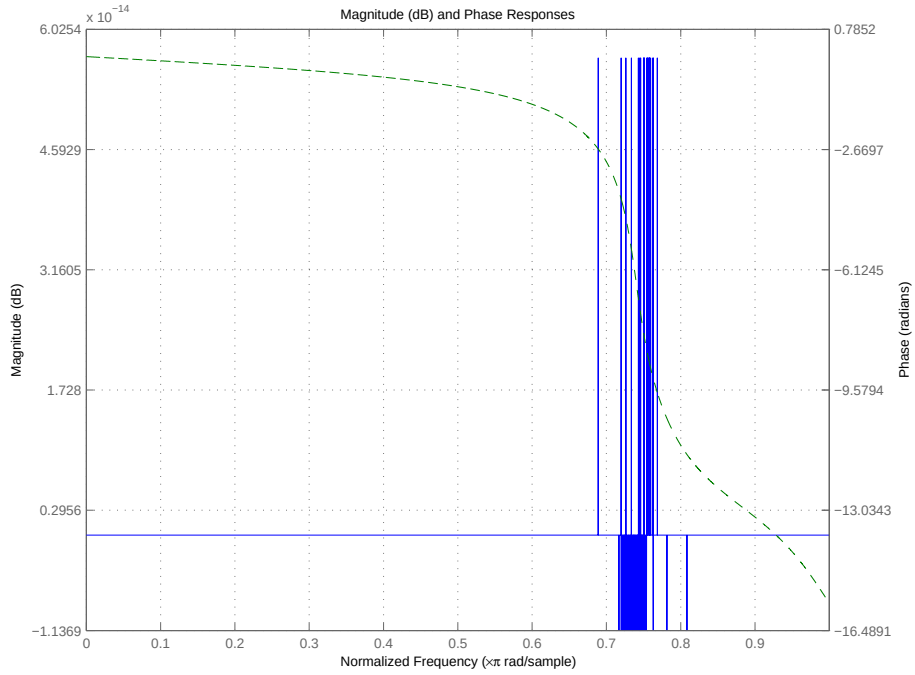
Now perform the actual filter design. `hd` contains two `dfilt.cascadeallpass` objects.

```
hd = design(d,'ellip','filterstructure','cascadeallpass');
% Get summary information about one dfilt.cascadeallpass stage.
hd.Stage(2).Stage(1)
ans =

        FilterStructure: 'Cascade Minimum-Multiplier Allpass'
    AllpassCoefficients: Section1: [0 0.0602973909571244]
                         Section2: [0 0.412590720361056]
                         Section3: [0 0.772715653742923]
       PersistentMemory: false
                 States: [0;0;0;0;0;0;0;0]
    NumSamplesProcessed: 0

hd

hd =

     FilterStructure: Cascade
             Stage(1): Scalar
             Stage(2): Parallel
                       Stage(1): Cascade Minimum-Multiplier Allpass
                       Stage(2): Cascade
                             Stage(1): Delay
                             Stage(2): Cascade Minimum-Multiplier Allpass
    PersistentMemory: false
```

This second example constructs a `dfilt.cascadeallpass` filter object directly given allpass coefficients for the input vectors.

```
section1 = 0.8;
section2 = [1.2,0.7];
section3 = [1.3,0.9];
hd = dfilt.cascadeallpass(section1,section2,section3);
info(hd)    % Get information about the filter.
fvtool(hd)  % Visualize the filter.
```

`hd` looks like this, showing both the magnitude and phase responses in FVTool. Note the units for the magnitude response on the left *y*-axis. Clearly this is an allpass filter.



See Also

`dfilt`, `dfilt.allpass`, `dfilt.cascadewdfallpass`, `mfilt.iirdecim`, `mfilt.iirinterp`

Purpose

Cascade allpass WDF filters to construct allpass WDF

Syntax

```
hd = dfilt.cascadewdfallpass(c1,c2,...)
```

Description

`hd = dfilt.cascadewdfallpass(c1,c2,...)` constructs a cascade of allpass wave digital filters given the allpass coefficients in the vectors `c1`, `c2`, and so on.

Each `c` vector contains the coefficients for one section of the cascaded filter. `C` vectors must have one, two, or four elements (coefficients). Three element vectors are not supported.

When the `c` vector has four elements, the first and third elements of the vector must be 0. Each section of the cascade is an allpass wave digital filter, from `dfilt.wdfallpass`, with the coefficients given by the corresponding `c` vector. That is, the first section has coefficients from vector `c1`, the second section coefficients come from `c2`, and on until all of the `c` vectors are used.

You can mix the lengths of the `c` vectors. They do not need to be the same length. For example, you can cascade several fourth-order sections (`length(c) = 4`) with first or second-order sections.

Wave digital filters are usually used to create other filters. This toolbox uses them to implement halfband filters, which the first example in Examples demonstrates. They are most often building blocks for filters.

Generally, you do not construct these WDF allpass cascade filters directly. Instead, they result from the design process for an IIR filter. Refer to the first example in Examples for more about using `dfilt.cascadewdfallpass` to design an IIR filter.

For more information about the `c` vectors and the transfer function for the allpass filters, refer to `dfilt.wdfallpass`.

Properties

In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
`AllpassCoefficients`	Contains the coefficients for the allpass wave digital filter object
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to `filtstates` in Signal Processing Toolbox™ documentation or in the Help system.

Examples

To demonstrate two approaches to using `dfilt.cascadewdfallpass` to design a filter, these examples show both direct construction and construction as part of another filter.

The first design shown creates an IIR halfband filter that uses lattice wave digital filters. Each branch of the parallel connection in the lattice is an allpass cascade wave digital filter.

```
tw = 100;  % Transition width of filter, 100 Hz.
ast = 80;  % Stopband attenuation of filter, 80 dB.
fs = 2000; % Sampling frequency of signal to filter.
% Store halfband specs.
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

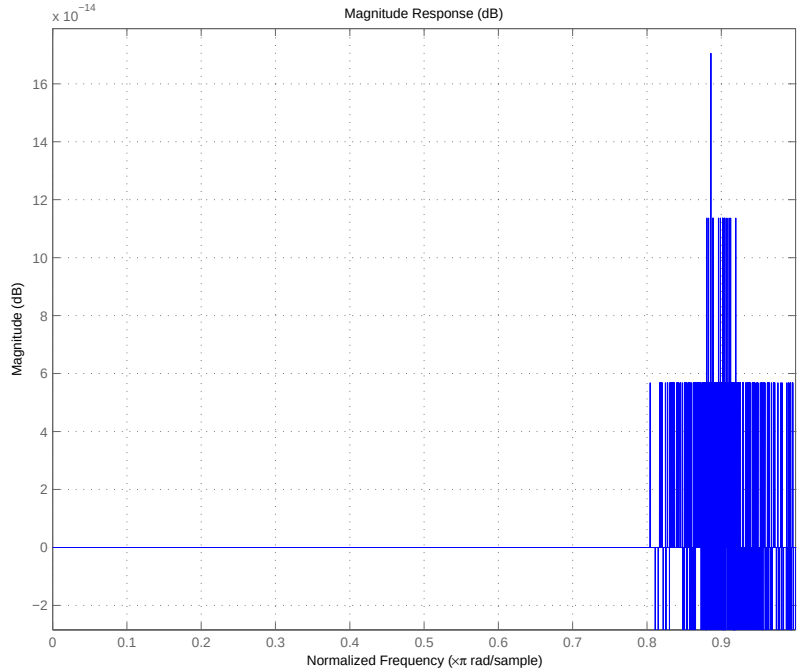
Now perform the actual halfband design process. `hd` contains two `dfilt.cascadewdfallpass` filters.

```
hd = design(f,'ellip','filterstructure','cascadewdfallpass');
hd.stage(2).stage(1) % Summary info on dfilt.cascadewdfallpass.
realizemdl(hd.stage(2).stage(1)) % Requires Simulink to realize model.
```

This example demonstrates direct construction of a `dfilt.cascadewdfallpass` filter with allpass coefficients.

```
section1 = 0.8;
section2 = [1.5,0.7];
section3 = [1.8,0.9];
hd = dfilt.cascadewdfallpass(section1,section2,section3);
info(hd)   % Show information about the filter.
fvtool(hd) % Visualize the filter.
```

Using FVTool lets you view the filter response.



See Also

`dfilt`, `dfilt.wdfallpass`

Purpose

Discrete-time, direct-form I filter

Syntax

Refer to `dfilt.df1` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.df1` returns a default discrete-time, direct-form I filter object that uses double-precision arithmetic. By default, the numerator and denominator coefficients b and a are set to 1. With these coefficients the filter passes the input to the output without changes.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

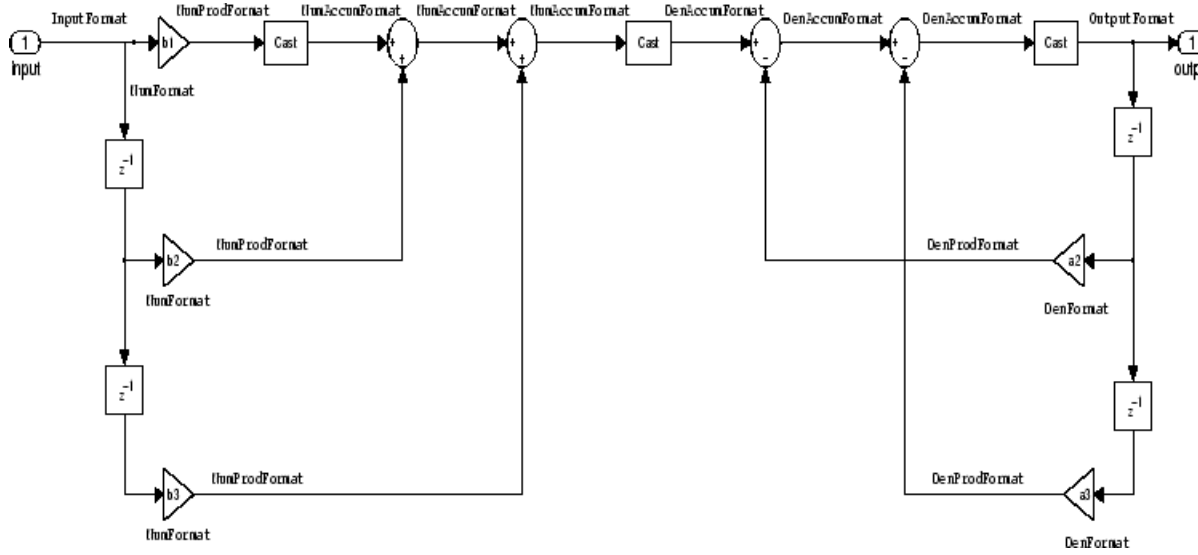
```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

Note `a(1)`, the leading denominator coefficient, cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I filter implemented by `dfilt.df1`. To help you see how the filter processes the coefficients, input, output, and states of the filter, as well as numerical operations, the figure includes the locations of the arithmetic and data type format elements within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFormat	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFormat	CoeffWordLength	DenFracLength	CoeffAutoScale , SignedDenominator
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
InputFormat	InputWordLength	InputFracLength	None
NumAccumFormat	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFormat	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
OutputFormat	OutputWordLength	OutputFracLength	OutputMode

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFormat, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFormat refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with df1 implementations of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use `get(hd)` where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length the filter algorithm uses to interpret the results of product operations involving denominator coefficients. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
`DenFracLength`	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`Denominator`	Stores the denominator coefficients for the IIR filter.
`DenProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set `ProductMode` to `SpecifyPrecision`.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.

Property Name	Brief Description
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`NumAccumFracLength`	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set `AccumMode` to `SpecifyPrecision`.
`Numerator`	Holds the numerator coefficient values for the filter.
`NumFracLength`	Sets the fraction length used to interpret the value of numerator coefficients.
`NumProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set `ProductMode` to `SpecifyPrecision`.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.
`OutputWordLength`	Determines the word length used for the output data.

Property Name	Brief Description
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to `filtstates` in Signal Processing Toolbox documentation or in the Help system.

Examples

Specify a second-order direct-form I structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1(b,a)
hd =

          FilterStructure: 'Direct-Form I'
               Arithmetic: 'double'
                Numerator: [0.3000 0.6000 0.3000]
              Denominator: [1 0 0.2000]
         PersistentMemory: false
                   States: Numerator:  [2x1 double]
                           Denominator:[2x1 double]
```

Now convert `hd` to a fixed-point filter:

```
set(hd,'arithmetic','fixed')
hd

hd =
```

```
 FilterStructure: 'Direct-Form I'
      Arithmetic: 'fixed'
       Numerator: [0.3000 0.6000 0.3000]
     Denominator: [1 0 0.2000]
PersistentMemory: false
          States: Numerator:  [2x1 fi]
                  Denominator:[2x1 fi]


 CoeffWordLength: 16
  CoeffAutoScale: true
          Signed: true

 InputWordLength: 16
 InputFracLength: 15

OutputWordLength: 16
OutputFracLength: 15

     ProductMode: 'FullPrecision'

       AccumMode: 'KeepMSB'
 AccumWordLength: 40
   CastBeforeSum: true

       RoundMode: 'convergent'
    OverflowMode: 'wrap'
```

See Also

`dfilt`, `dfilt.df1t`, `dfilt.df2`, `dfilt.df2t`

Purpose

Discrete-time, SOS direct-form I filter

Syntax

Refer to `dfilt.df1sos` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.df1sos(s)` returns a discrete-time, second-order section, direct-form I filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.df1sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I filter 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, and so on.

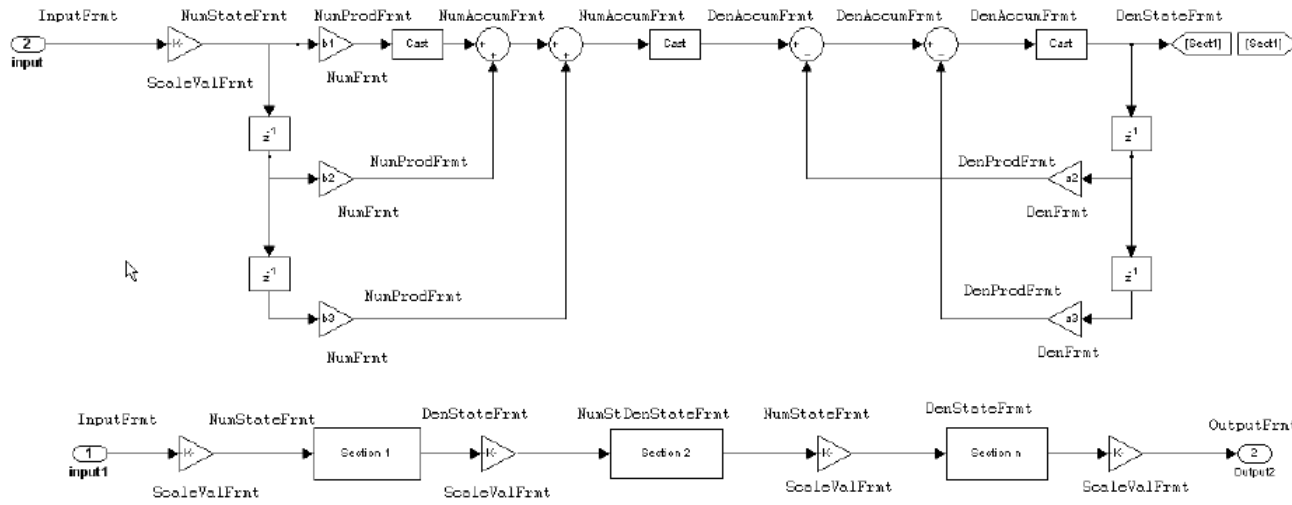
`hd = dfilt.df1sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. When you do not specify `g`, all gains default to one.

`hd = dfilt.df1sos` returns a default, discrete-time, second-order section, direct-form I filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I filter implemented in second-order sections by `dfilt.df1sos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the

table) store the word length and the fraction length in bits. Similarly consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, `Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
DenStateFrmt	`DenStateWordLength`	`DenStateFracLength`	`CastBeforeSum`, `States`
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
NumStateFrmt	`NumStateWordLength`	`NumStateFracLength`	`States`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ScaleValueFrmt	`CoeffWordLength`	`ScaleValueFracLength`	`CoeffAutoScale`, `ScaleValues`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The

label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with SOS implementation of direct-form I `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties”.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.

Property Name	Brief Description
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.

Property Name	Brief Description
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
DenStateFracLength	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
DenStateWordLength	Specifies the word length used to represent the states associated with denominator coefficients in the filter.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumStateFracLength	Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.

Property Name	Brief Description
`NumWordFracLength`	Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length applied for the output data.

Property Name	Brief Description
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.
ScaleValues	Scaling for the filter objects in SOS filters.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a fixed-point, second-order section, direct-form I dfilt object with the following code:

```
b=[0.3 0.6 0.3];
a=[1 0 0.2];
```

```
hd=dfilt.df1sos(b,a)

hd =

   FilterStructure: 'Direct-Form I, Second-Order Sections'
         Arithmetic: 'double'
          sosMatrix: [0.3000 0.6000 0.3000 1 0 0.2000]
       ScaleValues: [2x1 double]
 PersistentMemory: false
           States: Numerator:  [2x1 double]
                   Denominator:[2x1 double]

hd.arithmetic='fixed'

hd =

  FilterStructure: 'Direct-Form I, Second-Order Sections'
      ScaleValues: [2x1 double]
       Arithmetic: 'fixed'
        sosMatrix: [0.3000 0.6000 0.3000 1 0 0.2000]
 PersistentMemory: false
           States: Numerator:  [2x1 fi]
                   Denominator:[2x1 fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

      NumStateWordLength: 16
      NumStateFracLength: 15
```

```
DenStateWordLength: 16
DenStateFracLength: 15

       ProductMode: 'FullPrecision'

         AccumMode: 'KeepMSB'
   AccumWordLength: 40
     CastBeforeSum: true

         RoundMode: 'convergent'
      OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.df2tsos

Purpose

Discrete-time, direct-form I transposed filter

Syntax

Refer to `dfilt.df1t` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.df1t(b,a)` returns a discrete-time, direct-form I transposed filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

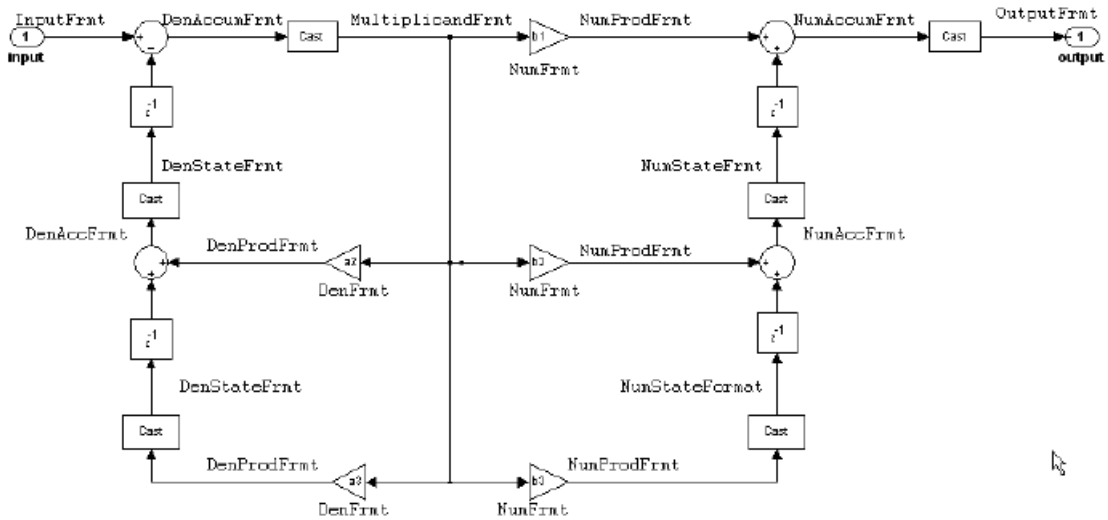
For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.df1t` returns a default, discrete-time, direct-form I transposed filter object `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. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the transposed direct-form I filter implemented by `dfilt.df1t`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale, , Signed, Denominator
DenProdFrmt	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
DenStateFrmt	DenStateWordLength	DenStateFracLength	CastBeforeSum, States
InputFrmt	InputWordLength	InputFracLength	None
Multiplicandfrmt	Multiplicand-WordLength	Multiplicand-FracLength	CastBeforeSum
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFrmt	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
NumStateFrmt	NumStateWord-Length	NumStateFrac-Length	States
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From

reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with df1t implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.

Property Name	Brief Description
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
`DenFracLength`	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`Denominator`	Holds the denominator coefficients for the filter.

Property Name	Brief Description
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
DenStateFracLength	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
MultiplicandFracLength	Sets the fraction length for values (multiplicands) used in multiply operations in the filter.
MultiplicandWordLength	Sets the word length applied to the values input to a multiply operation (the multiplicands).
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.

Property Name	Brief Description
`Numerator`	Holds the numerator coefficient values for the filter.
`NumFracLength`	Sets the fraction length used to interpret the value of numerator coefficients.
`NumProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set `ProductMode` to `SpecifyPrecision`.
`NumStateFracLength`	For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.

Property Name	Brief Description
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.

Property Name	Brief Description
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`StateAutoScale`	Setting autoscaling for filter states to `true` reduces the possibility of overflows occurring during fixed-point operations. Set to `false`, `StateAutoScale` lets the filter select the fraction length to limit the overflow potential.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
`StateWordLength`	Sets the word length used to represent the filter states.

Examples

Specify a second-order direct-form I transposed filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1t(b,a)

hd =

          FilterStructure: 'Direct-Form I Transposed'
               Arithmetic: 'double'
                Numerator: [0.3000 0.6000 0.3000]
              Denominator: [1 0 0.2000]
        PersistentMemory: false
                  States: Numerator:  [2x1 double]
                          Denominator:[2x1 double]
```

Now convert the filter to single-precision filtering arithmetic.

```
set(hd,'arithmetic','single')
hd
hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'fixed'
               Numerator: [0.3000 0.6000 0.3000]
             Denominator: [1 0 0.2000]
        PersistentMemory: false
                  States: Numerator:  [2x1 fi]
                          Denominator:[2x1 fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

  MultiplicandWordLength: 16
  MultiplicandFracLength: 15

        StateWordLength: 16
         StateAutoScale: true

            ProductMode: 'FullPrecision'

              AccumMode: 'KeepMSB'
        AccumWordLength: 40
          CastBeforeSum: true

              RoundMode: 'convergent'
           OverflowMode: 'wrap'
```

See Also dfilt, dfilt.df1, dfilt.df2, dfilt.df2t

dfilt.df1tsos

Purpose

Discrete-time, SOS direct-form I transposed filter

Syntax

Refer to `dfilt.df1tsos` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.df1tsos(s)` returns a discrete-time, second-order section, direct-form I, transposed filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.df1tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I, transposed filter 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.

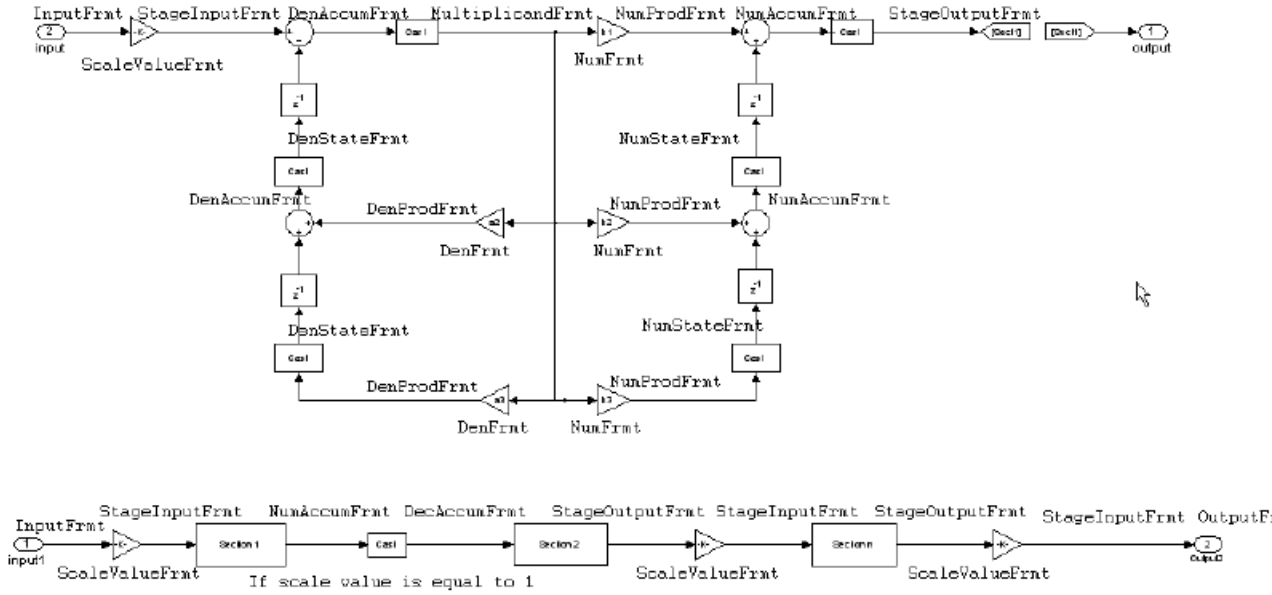
`hd = dfilt.df1tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df1tsos` returns a default, discrete-time, second-order section, direct-form I, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I transposed filter implemented using second-order sections by `dfilt.df1tsos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated

with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, `Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
DenStateFrmt	`DenStateWordLength`	`DenStateFracLength`	`CastBeforeSum`, `States`
InputFrmt	`InputWordLength`	`InputFracLength`	None
MultiplicandFrmt	`Multiplicand-WordLength`	`Multiplicand-FracLength`	`CastBeforeSum`
NumAccumFrmt	`AccumWordLength`	`NumAccum-FracLength`	`AccumMode`, `CastBeforeSum`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
NumStateFrmt	`NumStateWordLength`	`NumStateFracLength`	`States`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ScaleValueFrmt	`CoeffWordLength`	`ScaleValue-FracLength`	`CoeffAutoScale`, `ScaleValues`
StageInputfrmt	`StageInput-WordLength`	`StageInput-FracLength`	`StageInput-AutoScale`
StageOutputFrmt	`StageOutput-WordLength`	`StageOutput-FracLength`	`StageOutput-AutoScale`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with SOS implementation of transposed direct-form I `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.

Property Name	Brief Description
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
`DenFracLength`	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.

Property Name	Brief Description
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
DenStateFracLength	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
MultiplicandFracLength	Sets the fraction length for values (multiplicands) used in multiply operations in the filter.
MultiplicandWordLength	Sets the word length applied to the values input to a multiply operation (the multiplicands)
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.

Property Name	Brief Description
`Numerator`	Holds the numerator coefficient values for the filter.
`NumProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set `ProductMode` to `SpecifyPrecision`.
`NumStateFracLength`	For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
`NumStateWordLength`	For IIR filters, this defines the word length applied to the numerator states of the filter. Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.

Property Name	Brief Description
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.

Property Name	Brief Description
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.

Property Name	Brief Description
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
`ScaleValueFracLength`	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable `AutoScaleMode` by setting it to `false`.
`ScaleValues`	Scaling for the filter objects in SOS filters.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`SosMatrix`	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/sectiondatatype]. A `[15x6 double]` SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type `double` to represent the coefficients.
`StageInputAutoScale`	Tells the filter whether to set the stage input data format to minimize the occurrence of overflow conditions.
`StageInputFracLength`	Lets you set the fraction length for stage inputs in SOS filters, if you set `StageInputAutoScale` to `false`.
`StageInputWordLength`	Lets you set the word length for stage inputs in SOS filters, if you set `StageInputAutoScale` to `false`.

Property Name	Brief Description
StageOutputAutoScale	Tells the filter whether to set the stage output data format to minimize the occurrence of overflow conditions.
StageOutputFracLength	Lets you set the fraction length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.
StageOutputWordLength	Lets you set the word length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

With the following code, this example specifies a second-order section, direct-form I transposed dfilt object for a filter. Then convert the filter to fixed-point operation.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
```

```
hd = dfilt.df1t(b,a)

hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'double'
               Numerator: [0.3000 0.6000 0.3000]
             Denominator: [1 0 0.2000]
        PersistentMemory: false
                  States: Numerator:  [2x1 double]
                          Denominator:[2x1 double]


set(hd,'arithmetic','fixed')
hd

hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'fixed'
               Numerator: [0.3000 0.6000 0.3000]
             Denominator: [1 0 0.2000]
        PersistentMemory: false
                  States: Numerator:  [2x1 fi]
                          Denominator:[2x1 fi]


         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'
```

```
MultiplicandWordLength: 16
MultiplicandFracLength: 15

       StateWordLength: 16
        StateAutoScale: true

           ProductMode: 'FullPrecision'

             AccumMode: 'KeepMSB'
       AccumWordLength: 40
         CastBeforeSum: true

             RoundMode: 'convergent'
          OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.df1sos, dfilt.df2sos, dfilt.df2tsos

dfilt.df2

Purpose

Discrete-time, direct-form II filter

Syntax

Refer to `dfilt.df2` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.df2(b,a)` returns a discrete-time, direct-form II filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

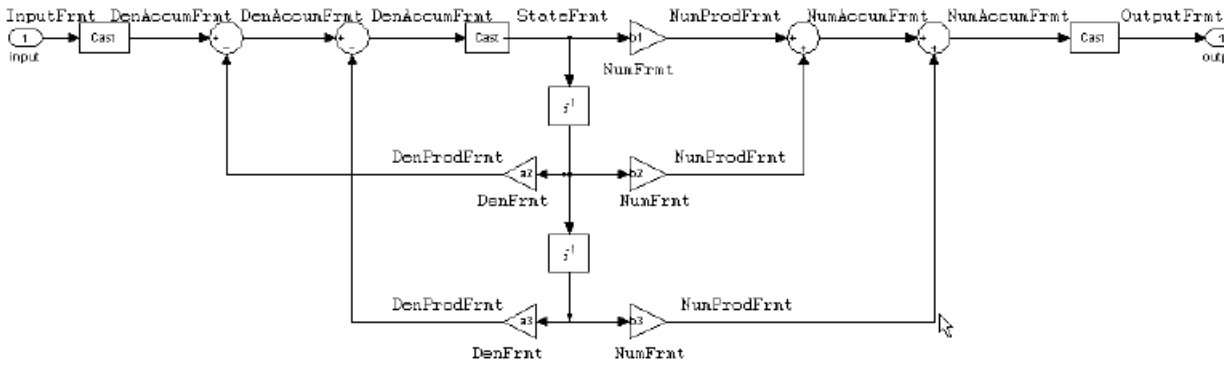
For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.df2` returns a default, discrete-time, direct-form II filter object `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. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form II filter implemented by `dfilt.df2`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters“frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFrmt	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
InputFrmt	InputWordLength	InputFracLength	None
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFrmt	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode
StateFrmt	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with the df2 implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties”.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.

Property Name	Brief Description
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Holds the denominator coefficients for IIR filters.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

Property Name	Brief Description
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.

Property Name	Brief Description
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a second-order direct-form II filter structure for a dfilt object, hd, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2(b,a)
hd =
             FilterStructure: 'Direct Form II'
                   Numerator: [0.3000 0.6000 0.3000]
                 Denominator: [1 0 0.2000]
    NumberOfSamplesProcessed: 0
                 ResetStates: 'on'
                      States: [2x1 double]
```

To convert the filter to fixed-point arithmetic, change the value of the Arithmetic property

```
set(hd,'arithmetic','fixed')
```

to specify the fixed-point option.

See Also

dfilt, dfilt.df1, dfilt.df1t, dfilt.df2t

dfilt.df2sos

Purpose

Discrete-time, SOS, direct-form II filter

Syntax

Refer to `dfilt.df2sos` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`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.

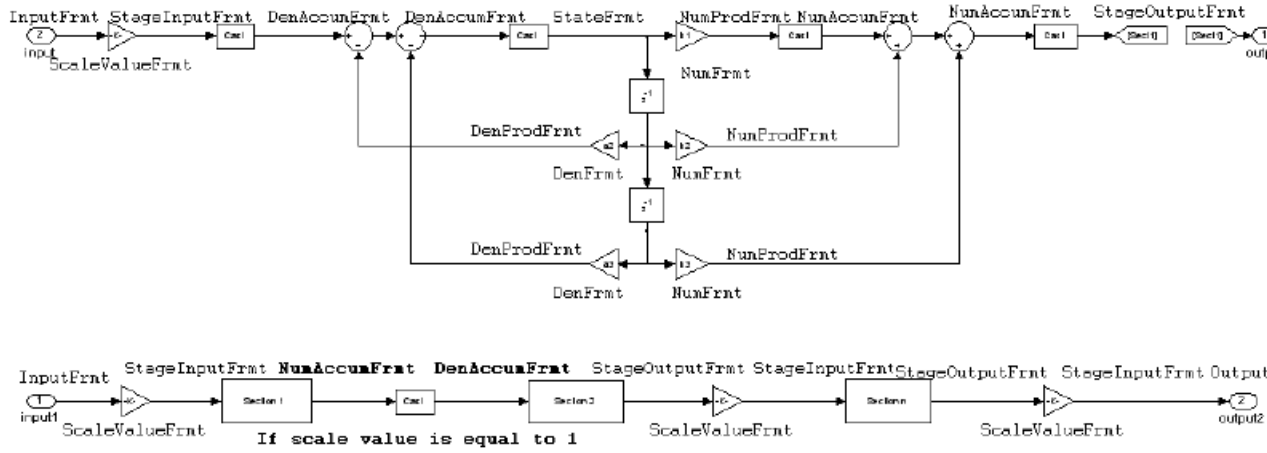
`hd = dfilt.df2sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df2sos` returns a default, discrete-time, second-order section, direct-form II filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The figure below shows the signal flow for the direct-form II filter implemented with second-order sections by `dfilt.df2sos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The frmt properties `InputWordLength` and `InputFracLength` (as shown

in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, `Signed`, `sosMatrix`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`, `sosMatrix`
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `sosMatrix`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ScaleValueFrmat	`CoeffWordLength`	`ScaleValue-FracLength`	`CoeffAutoScale`, `ScaleValues`
StageInputFormt	`StageInput-WordLength`	`StageInput-FracLength`	`StageInput-AutoScale`
StageOutputFrmt	`StageOutput-WordLength`	`StageOutput-FracLength`	`StageOutput-AutoScale`
StateFrmt	`StateWordLength`	`StateFracLength`	`CastBeforeSum`, `States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with second-order section implementation of direct-form II `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties”.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.

Property Name	Brief Description
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.

Property Name	Brief Description
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.

Property Name	Brief Description
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
`ScaleValueFracLength`	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable `AutoScaleMode` by setting it to `false`.
`ScaleValues`	Scaling for the filter objects in SOS filters.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`SosMatrix`	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A `[15x6 double]` SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type `double` to represent the coefficients.
`StageInputAutoScale`	Tells the filter whether to set the stage input data format to minimize the occurrence of overflow conditions.
`StageInputFracLength`	Lets you set the fraction length for stage inputs in SOS filters, if you set `StageInputAutoScale` to `false`.
`StageInputWordLength`	Lets you set the word length for stage inputs in SOS filters, if you set `StageInputAutoScale` to `false`.
`StageOutputAutoScale`	Tells the filter whether to set the stage output data format to minimize the occurrence of overflow conditions.

Property Name	Brief Description
StageOutputFracLength	Lets you set the fraction length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.
StageOutputWordLength	Lets you set the word length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a second-order section, direct-form II `dfilt` object for a Butterworth filter converted to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);
[s,g] = zp2sos(z,p,k);
hd = dfilt.df2sos(s,g)

hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [15x6 double]
            ScaleValues: [16x1 double]
        PersistentMemory: false
                  States: [2x15 double]
```

With the SOS filter constructed, now change the filter operation to single-precision filtering, and then to fixed-point filtering.

```
set(hd,'arithmetic','single')
hd

hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'single'
               sosMatrix: [15x6 double]
            ScaleValues: [16x1 double]
        PersistentMemory: false
                  States: [2x15 single]

hd.arithmetic='fixed'

hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'fixed'
               sosMatrix: [15x6 double]
            ScaleValues: [16x1 double]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

    StageInputWordLength: 16
     StageInputAutoScale: true

   StageOutputWordLength: 16
```

```
StageOutputAutoScale: true

    OutputWordLength: 16
          OutputMode: 'AvoidOverflow'

     StateWordLength: 16
     StateFracLength: 15

         ProductMode: 'FullPrecision'

           AccumMode: 'KeepMSB'
     AccumWordLength: 40
       CastBeforeSum: true

           RoundMode: 'convergent'
        OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.df1sos, dfilt.df1tsos, dfilt.df2tsos

Purpose

Discrete-time, direct-form II transposed filter

Syntax

Refer to `dfilt.df2t` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.df2t(b,a)` returns a discrete-time, direct-form II transposed filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

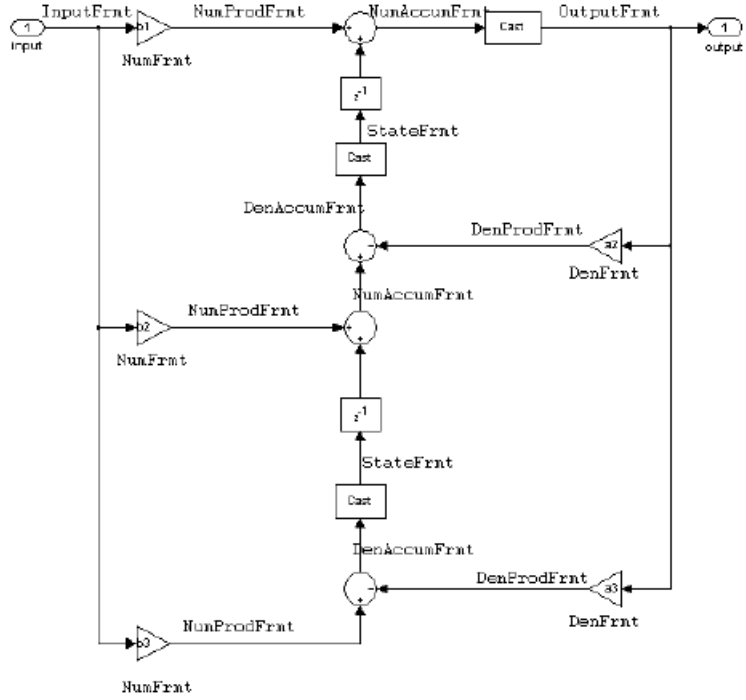
For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.df2t` returns a default, discrete-time, direct-form II transposed filter object `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. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form II transposed filter implemented by `dfilt.df2t`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt.” In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The

format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`,`Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`
StateFrmt	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the

properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with df2t implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.

Property Name	Brief Description
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
`DenFracLength`	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`Denominator`	Holds the denominator coefficients for IIR filters.

Property Name	Brief Description
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

Property Name	Brief Description
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Create a fixed-point filter by specifying a second-order direct-form II transposed filter structure for a dfilt object, and then converting the double-precision arithmetic setting to fixed-point.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2t(b,a)

hd =

         FilterStructure: 'Direct-Form II Transposed'
              Arithmetic: 'double'
               Numerator: [0.3000 0.6000 0.3000]
             Denominator: [1 0 0.2000]
        PersistentMemory: false
                  States: [2x1 double]
```

```
set(hd,'arithmetic','fixed')
hd

hd =

         FilterStructure: 'Direct-Form II Transposed'
              Arithmetic: 'fixed'
               Numerator: [0.3000 0.6000 0.3000]
             Denominator: [1 0 0.2000]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
        OutputFracLength: 15

         StateWordLength: 16
          StateAutoScale: true

             ProductMode: 'FullPrecision'

               AccumMode: 'KeepMSB'
         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.df1, dfilt.df1t, dfilt.df2

dfilt.df2tsos

Purpose

Discrete-time, SOS direct-form II transposed filter

Syntax

Refer to `dfilt.df2tsos` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients given in the matrix `s`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.df2tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II, transposed filter 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.

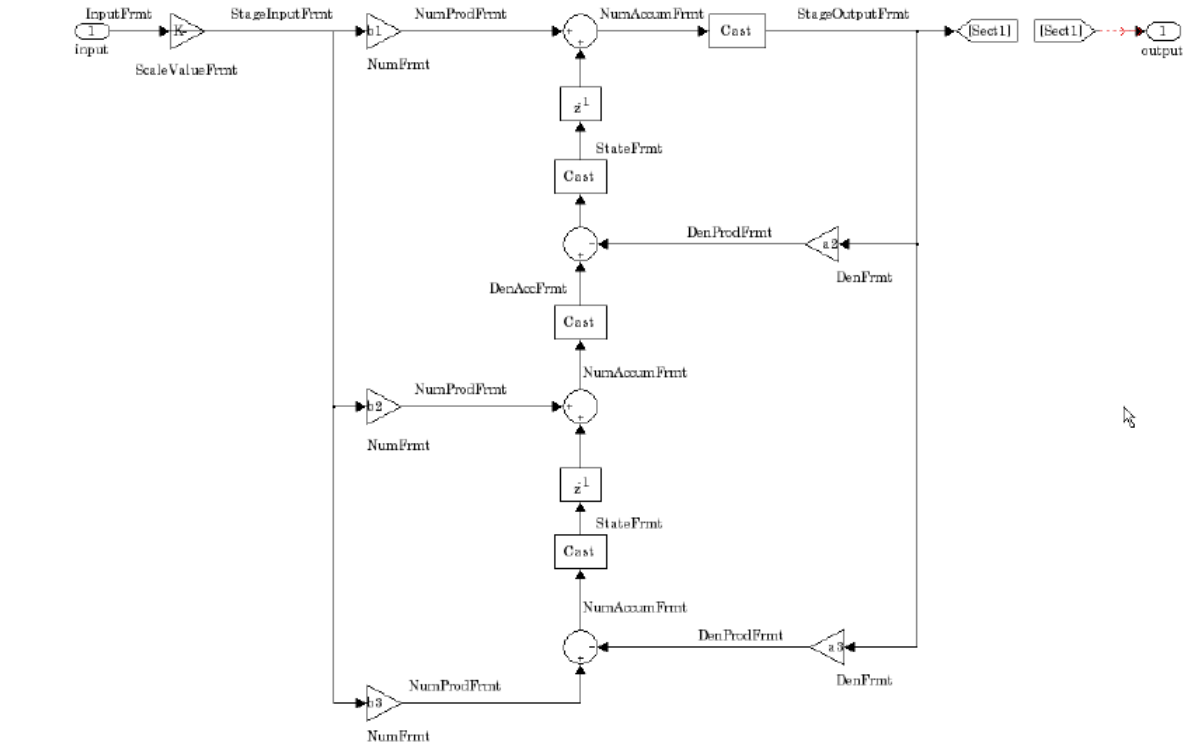
`hd = dfilt.df2tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

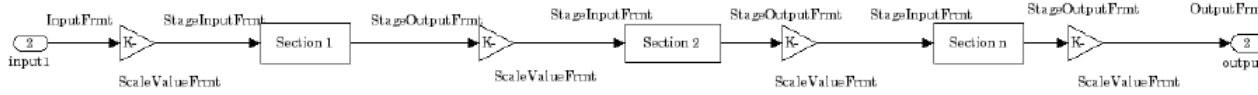
`hd = dfilt.df2tsos` returns a default, discrete-time, second-order section, direct-form II, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The figure below shows the signal flow for the second-order section transposed direct-form II filter implemented by `dfilt.dftsos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.





Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table

describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, `Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `SignedNumerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
ScaleValueFrmt	`CoeffWordLength`	`ScaleValueFracLength`	`CoeffAutoScale`, , `ScaleValues`
StageInputFrmt	`StageInput-WordLength`	`StageInput-FracLength`	`StageInput-AutoScale`
StageOutputFrmt	`StageOutput-WordLength`	`StageOutput-FracLength`	`StageOutput-AutoScale`
StateFrmt	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with second-order section implementation of transposed direct-form II `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.

Property Name	Brief Description
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`ScaleValueFracLength`	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable `AutoScaleMode` by setting it to `false`.
`ScaleValues`	Scaling for the filter objects in SOS filters.

Property Name	Brief Description
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`SosMatrix`	Holds the filter coefficients as property values — you use `set` and `get` to modify them. Displays the matrix in the format [sections x coefficients/section data type]. A `[15x6 double]` SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type `double` to represent the coefficients.
`StageInputFracLength`	Lets you set the fraction length for stage inputs in SOS filters, if you set `StageInputAutoScale` to `false`.
`StageInputWordLength`	Lets you set the word length for stage inputs in SOS filters, if you set `StageInputAutoScale` to `false`.
`StageOutputAutoScale`	Tells the filter whether to set the stage output data format to minimize the occurrence of overflow conditions.
`StageOutputFracLength`	Lets you set the fraction length for stage outputs in SOS filters, if you set `StageOutputAutoScale` to `off`.
`StageOutputWordLength`	Lets you set the word length for stage outputs in SOS filters, if you set `StageOutputAutoScale` to `false`.
`StateAutoScale`	Setting autoscaling for filter states to `true` reduces the possibility of overflows occurring during fixed-point operations. Set to `false`, `StateAutoScale` lets the filter select the fraction length to limit the overflow potential.
`StateFracLength`	When you set `StateAutoScale` to `false`, you enable the `StateFracLength` property that lets you set the fraction length applied to interpret the filter states.

Property Name	Brief Description
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Construct a second-order section Butterworth filter for fixed-point filtering. Start by specifying a Butterworth filter, and then convert the filter to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);
[s,g] = zp2sos(z,p,k);
hd = dfilt.df2tsos(s,g)

hd =

         FilterStructure: [1x48 char]
              Arithmetic: 'double'
               sosMatrix: [15x6 double]
             ScaleValues: [16x1 double]
        PersistentMemory: false
                  States: [2x15 double]
```

Now change the setting of the property `Arithmetic` to convert the filter to fixed-point operation.

```
hd.arithmetic='fixed'

hd =

         FilterStructure: [1x48 char]
              Arithmetic: 'fixed'
               sosMatrix: [15x6 double]
             ScaleValues: [16x1 double]
        PersistentMemory: false
                  States: [1x1 embedded.fi]
```

```
          CoeffWordLength: 16
           CoeffAutoScale: true
                   Signed: true

          InputWordLength: 16
          InputFracLength: 15

       StageInputWordLength: 16
       StageInputFracLength: 15

      StageOutputWordLength: 16
      StageOutputFracLength: 15

           OutputWordLength: 16
                 OutputMode: 'AvoidOverflow'

          StateWordLength: 16
           StateAutoScale: true

              ProductMode: 'FullPrecision'

                AccumMode: 'KeepMSB'
          AccumWordLength: 40
            CastBeforeSum: true

                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

See Also

`dfilt`, `dfilt.df1sos`, `dfilt.df1tsos`, `dfilt.df2sos`

Purpose

Discrete-time, direct-form antisymmetric FIR filter

Syntax

Refer to `dfilt.dfasymfir` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.dfasymfir(b)` returns a discrete-time, direct-form, antisymmetric FIR filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

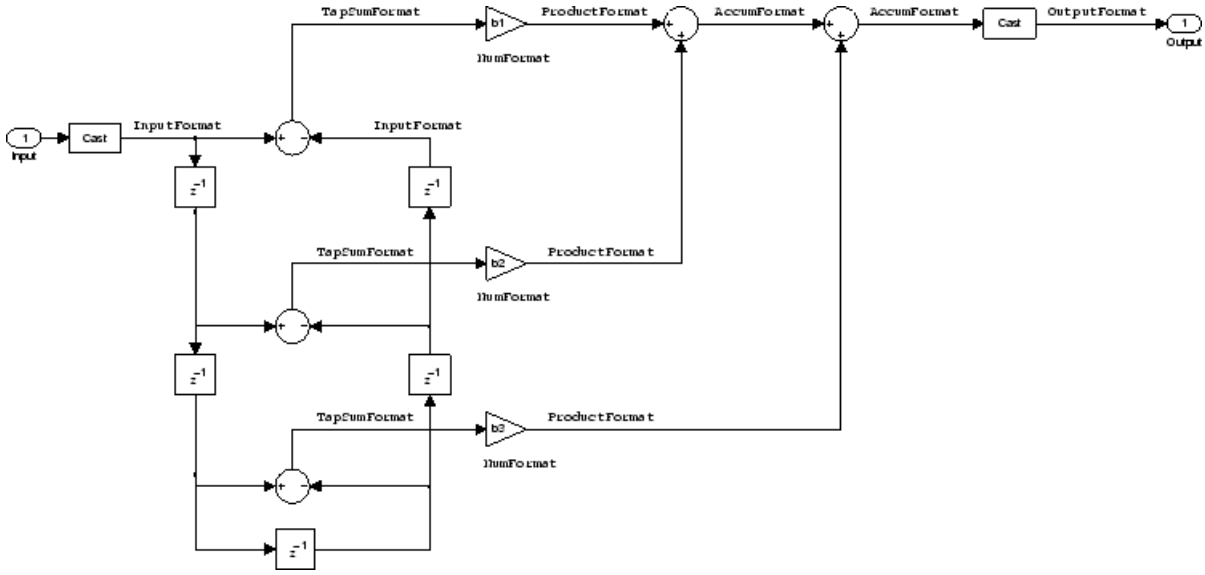
`hd = dfilt.dfasymfir` returns a default, discrete-time, direct-form, antisymmetric FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

Note Only the coefficients in the first half of vector `b` are used because `dfilt.dfasymfir` assumes the coefficients in the second half are antisymmetric to those in the first half. For example, in the figure coefficients, b(4) = -b(3), b(5) = -b(2), and b(6) = -b(1).

Fixed-Point Filter Structure

The following figure shows the signal flow for the odd-order antisymmetric FIR filter implemented by `dfilt.dfasymfir`. The even-order filter uses similar flow. To help you see how the filter processes the coefficients, input, and states of the filter, as well as

numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths

(`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	None
InputFormat	`InputWordLength`	`InputFracLength`	None
NumFormat	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, , `Signed`, `Numerator`
OutputFormat	`OutputWordLength`	`OutputFracLength`	None
ProductFormat	`ProductWordLength`	`ProductFracLength`	None
TapSumFormat	`InputWordLength`	`InputFracLength`	`InputFormat`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength` and `ProductWordLength` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with an antisymmetric FIR implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits [27]	Specifies the fraction length used to interpret data output by the accumulator.
`AccumWordLength`	Any integer number of bits[33]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data. Also controls `TapSumFracLength`.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data. Also determines `TapSumWordLength`.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilerInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.

Name	Values	Description
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`ProductFracLength`	Any positive or negative integer number of bits [27]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Any integer number of bits [33]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.

Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; software Toolbox documentation or in the online Help system.

Examples

Odd Order

Specify a fifth-order direct-form antisymmetric 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.dfasymfir(b)

hd =

     FilterStructure: 'Direct-Form Antisymmetric FIR'
          Arithmetic: 'double'
           Numerator: [-0.0080 0.0600 -0.4400 0.4400 -0.0600 0.0080]
    PersistentMemory: false

set(hd,'arithmetic','fixed')
hd =

     FilterStructure: 'Direct-Form Antisymmetric FIR'
          Arithmetic: 'fixed'
           Numerator: [-0.0080 0.0600 -0.4400 0.4400 -0.0600 0.0080]
    PersistentMemory: false
```

```
    CoeffWordLength: 16
     CoeffAutoScale: true
             Signed: true

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'FullPrecision'
```

Now look at the coefficients after converting `hd` to fixed-point format.

```
get(hd,'numerator')

ans =
  -0.0080    0.0600   -0.4400   0.4400   -0.0600    0.0080
```

Even Order

Specify a fourth-order direct-form antisymmetric FIR filter structure for `dfilt` object `hd`, with the following code:

```
b = [-0.01 0.1 0.0 -0.1 0.01];
hd = dfilt.dfasymfir(b)

hd =

 FilterStructure: 'Direct-Form Antisymmetric FIR'
      Arithmetic: 'double'
       Numerator: [-0.0100 0.1000 0 -0.1000 0.0100]
PersistentMemory: false

hd.arithmetic='fixed'

hd =

 FilterStructure: 'Direct-Form Antisymmetric FIR'
      Arithmetic: 'fixed'
       Numerator: [-0.0100 0.1000 0 -0.1000 0.0100]
PersistentMemory: false
```

```
    CoeffWordLength: 16
     CoeffAutoScale: true
             Signed: true

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'FullPrecision'

   get(hd,'numerator')

   ans =

      -0.0100    0.1000         0   -0.1000    0.0100
```

See Also

dfilt, dfilt.dffir, dfilt.dffirt, dfilt.dfsymfir

Purpose

Discrete-time, direct-form FIR filter

Syntax

Refer to `dfilt.dffir` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.dffir(b)` returns a discrete-time, direct-form finite impulse response (FIR) filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

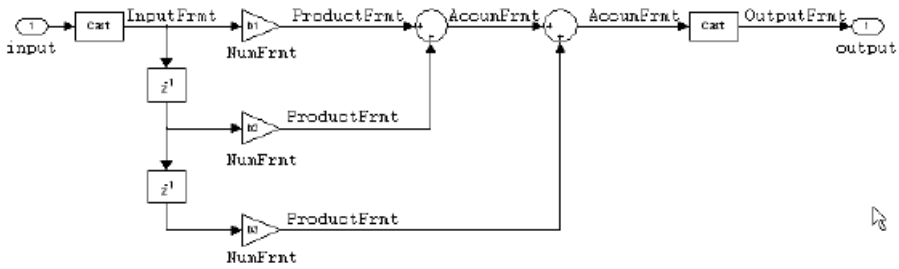
```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.dffir` returns a default, discrete-time, direct-form FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form FIR filter implemented by `dfilt.dffir`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters“frmt” (format). In this use, “frmt” indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFrmt	`AccumWordLength`	`AccumFracLength`	None

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	None
ProductFrmt	`ProductWordLength`	`ProductFracLength`	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFrmt, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFrmt refers to the properties `ProductFracLength` and `ProductWordLength` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with direct-form FIR implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties”.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits [30]	Specifies the fraction length used to interpret data output by the accumulator.
`AccumWordLength`	Any integer number of bits[34]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.

Name	Values	Description
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilerInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`ProductFracLength`	Any positive or negative integer number of bits [30]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Any integer number of bits [32]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.

Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system.

Examples

Specify a second-order direct-form FIR filter structure for a `dfilt` object `hd`, with the following code that constructs the filter in double-precision format and then converts the filter to fixed-point operation:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffir(b)

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [0.0500 0.9000 0.0500]
    PersistentMemory: false

hd.arithmetic='fixed'

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'fixed'
           Numerator: [0.0500 0.9000 0.0500]
    PersistentMemory: false
```

```
      CoeffWordLength: 16
       CoeffAutoScale: true
               Signed: true

      InputWordLength: 16
      InputFracLength: 15

      FilterInternals: 'FullPrecision'

hd.filterInternals='specifyPrecision'

hd =

      FilterStructure: 'Direct-Form FIR'
           Arithmetic: 'fixed'
            Numerator: [0.0500 0.9000 0.0500]
     PersistentMemory: false

      CoeffWordLength: 16
       CoeffAutoScale: true
               Signed: true

      InputWordLength: 16
      InputFracLength: 15

      FilterInternals: 'SpecifyPrecision'

     OutputWordLength: 34
     OutputFracLength: 30

    ProductWordLength: 32
    ProductFracLength: 30

      AccumWordLength: 34
      AccumFracLength: 30
```

```
                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.dfasymfir, dfilt.dffirt, dfilt.dfsymfir

Purpose

Discrete-time, direct-form FIR transposed filter

Syntax

Refer to `dfilt.dffirt` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.dffirt(b)` returns a discrete-time, direct-form FIR transposed filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

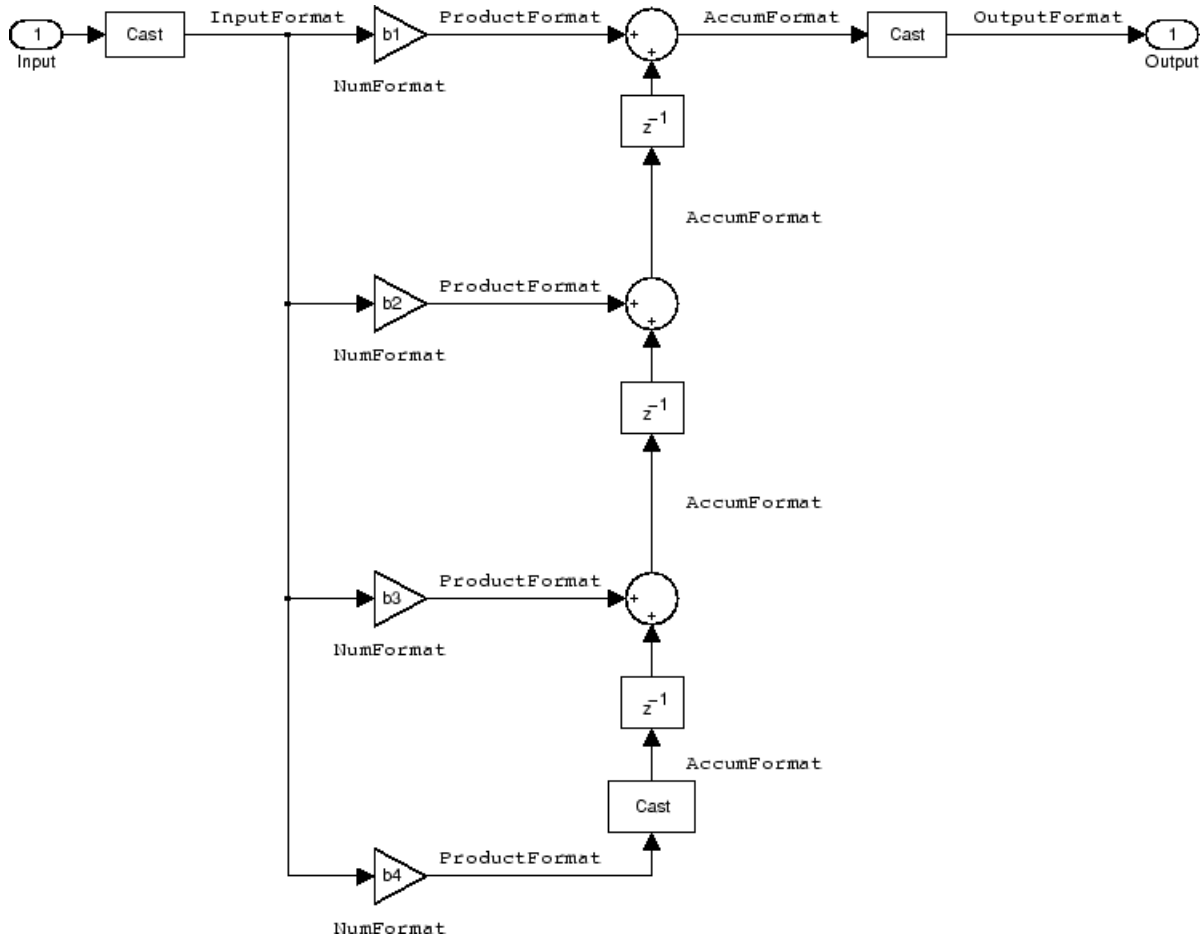
```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.dffirt` returns a default, discrete-time, direct-form FIR transposed filter object `hd`, with `b = 1`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the transposed direct-form FIR filter implemented by `dfilt.dffirt`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	None
InputFormat	`InputWordLength`	`InputFracLength`	None
NumFormat	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
OutputFormat	`OutputWordLength`	`OutputFracLength`	None
ProductFormat	`ProductWordLength`	`ProductFracLength`	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength` and `ProductWordLength` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the transposed direct-form FIR implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits [30]	Specifies the fraction length used to interpret data output by the accumulator.
`AccumWordLength`	Any integer number of bits[34]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [30]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilerInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [34]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.

Name	Values	Description
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.

Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system.

Examples

Specify a second-order direct-form FIR transposed filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffirt(b)

hd =

     FilterStructure: 'Direct-Form FIR Transposed'
          Arithmetic: 'double'
           Numerator: [0.0500 0.9000 0.0500]
    PersistentMemory: false
```

Now use the filter property `Arithmetic` to change the filter to fixed-point format.

```
set(hd,'arithmetic','fixed')
hd

hd =

     FilterStructure: 'Direct-Form FIR Transposed'
```

```
            Arithmetic: 'fixed'
             Numerator: [0.0500 0.9000 0.0500]
       PersistentMemory: false

        CoeffWordLength: 16
         CoeffAutoScale: true
                 Signed: true

        InputWordLength: 16
        InputFracLength: 15

        FilterInternals: 'FullPrecision'

hd.filterInternals='specifyPrecision'

hd =

        FilterStructure: 'Direct-Form FIR Transposed'
            Arithmetic: 'fixed'
             Numerator: [0.0500 0.9000 0.0500]
       PersistentMemory: false

        CoeffWordLength: 16
         CoeffAutoScale: true
                 Signed: true

        InputWordLength: 16
        InputFracLength: 15

        FilterInternals: 'SpecifyPrecision'

       OutputWordLength: 34
       OutputFracLength: 30

      ProductWordLength: 32
      ProductFracLength: 30
```

```
          AccumWordLength: 34
          AccumFracLength: 30

                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.dffir, dfilt.dfasymfir, dfilt.dfsymfir

Purpose

Discrete-time, direct-form symmetric FIR filter

Syntax

Refer to `dfilt.dfsymfir` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.dfsymfir(b)` returns a discrete-time, direct-form symmetric FIR filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

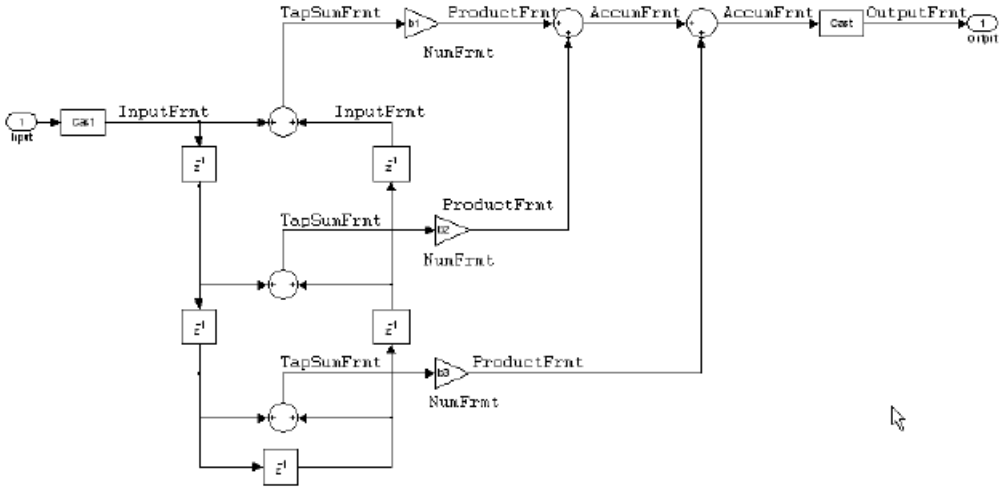
For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.dfsymfir` returns a default, discrete-time, direct-form symmetric FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

Note Only the coefficients in the first half of vector `b` are used because `dfilt.dfsymfir` assumes the coefficients in the second half are symmetric to those in the first half. In the following figure, for example, b(3) = 0, b(4) = b(2) and b(5) = b(1).

Fixed-Point Filter Structure

In the following figure you see the signal flow diagram for the symmetric FIR filter that `dfilt.dfsymfir` implements.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters“frmt” (format). In this use, “frmt” indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFrmt	`AccumWordLength`	`AccumFracLength`	None
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	None
ProductFrmt	`ProductWordLength`	`ProductFracLength`	None
TapSumFrmt	`InputWordLength`	`InputFracLength`	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFrmt, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFrmt refers to the properties `ProductFracLength` and `ProductWordLength` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the symmetric FIR implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits [27]	Specifies the fraction length used to interpret data output by the accumulator.
`AccumWordLength`	Any integer number of bits[33]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilerInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.

Name	Values	Description
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
`ProductFracLength`	Any positive or negative integer number of bits [29]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Any integer number of bits [33]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.

Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system.

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)

hd =

 FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'double'
       Numerator: [-0.0080 0.0600 0.4400 0.4400 0.0600 -0.0080]
PersistentMemory: false

set(hd,'arithmetic','fixed')
hd

hd =

 FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'fixed'
```

```
       Numerator: [-0.0080 0.0600 0.4400 0.4400 0.0600 -0.0080]
PersistentMemory: false

  CoeffWordLength: 16
  CoeffAutoScale: true
          Signed: true

 InputWordLength: 16
 InputFracLength: 15

 FilterInternals: 'FullPrecision'

hd.filterinternals='specifyPrecision'

hd =

 FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'fixed'
       Numerator: [-0.0080 0.0600 0.4400 0.4400 0.0600 -0.0080]
PersistentMemory: false

 CoeffWordLength: 16
  CoeffAutoScale: true
          Signed: true

 InputWordLength: 16
 InputFracLength: 15

FilterInternals: 'SpecifyPrecision'


OutputWordLength: 36
OutputFracLength: 31

ProductWordLength: 33
ProductFracLength: 31
```

```
AccumWordLength: 36
 AccumFracLength: 31

      RoundMode: 'convergent'
   OverflowMode: 'wrap'
```

To use `hd` for fixed-point filtering, change the value of the property `Arithmetic` to `fixed` with the following command:

```
hd.arithmetic = 'fixed'
```

Even Order

Specify a fourth-order, fixed-point, 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)

hd =

 FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'double'
       Numerator: [-0.0100 0.1000 0.8000 0.1000 -0.0100]
PersistentMemory: false

set(hd,'arithmetic','fixed')
hd

hd =

 FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'fixed'
       Numerator: [-0.0100 0.1000 0.8000 0.1000 -0.0100]
PersistentMemory: false

 CoeffWordLength: 16
  CoeffAutoScale: true
          Signed: true
```

```
     InputWordLength: 16
     InputFracLength: 15

     FilterInternals: 'FullPrecision'

    hd.filterinternals='specifyPrecision'

    hd =

     FilterStructure: 'Direct-Form Symmetric FIR'
          Arithmetic: 'fixed'
           Numerator: [-0.0100 0.1000 0.8000 0.1000 -0.0100]
    PersistentMemory: false

     CoeffWordLength: 16
      CoeffAutoScale: true
              Signed: true

     InputWordLength: 16
     InputFracLength: 15

     FilterInternals: 'SpecifyPrecision'

    OutputWordLength: 36
    OutputFracLength: 30

    ProductWordLength: 33
    ProductFracLength: 30

     AccumWordLength: 36
     AccumFracLength: 30

           RoundMode: 'convergent'
        OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.dfasymfir, dfilt.dffir, dfilt.dffirt

dfilt.farrowfd

Purpose

Fractional Delay Farrow filter

Syntax

```
Hd = DFILT.FARROWFD(D, COEFFS)
```

Description

```
Hd = DFILT.FARROWFD(D, COEFFS)
```

Constructs a discrete-time fractional delay Farrow filter with COEFFS coefficients and D delay.

Examples

Farrow filters can be designed with the dfilt.farrowfd filter designer.

```
coeffs = [-1/6 1/2 -1/3 0;1/2 -1 -1/2 1;
-1/2 1/2 1 0;1/6 0 -1/6 0];
Hd = dfilt.farrowfd(.5, coeffs)
y = filter(Hd,1:10)
```

Design a cubic fractional delay filter with the Lagrange method.

```
fdelay = .2; % Fractional delay
d = fdesign.fracdelay(fdelay,'N',3);
Hd = design(d, 'lagrange', 'FilterStructure', 'fd');
fvtool(Hd, 'Analysis', 'grpdelay')
```

For more information about fractional delay filter implementations, see the “Fractional Delay Filters Using Farrow Structures” demo, farrowdemo.

See Also

`dfilt`

Purpose

Farrow Linear Fractional Delay filter

Syntax

```
Hd = DFILT.FARROWLINEARFD(D)
```

Description

```
Hd = DFILT.FARROWLINEARFD(D)
```

Constructs a discrete-time linear fractional delay Farrow filter with the delay D.

Examples

Farrow filters can be designed with the fdesign.fracdelay filter designer.

```
Hd = dfilt.farrowlinearfd(.5)
y = filter(Hd,1:10)
```

For more information about fractional delay filter implementations, see the “Fractional Delay Filters Using Farrow Structures” demo.

See Also

`dfilt`

dfilt.latticeallpass

Purpose

Discrete-time, lattice allpass filter

Syntax

Refer to `dfilt.latticeallpass` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.latticeallpass(k)` returns a discrete-time, lattice allpass filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

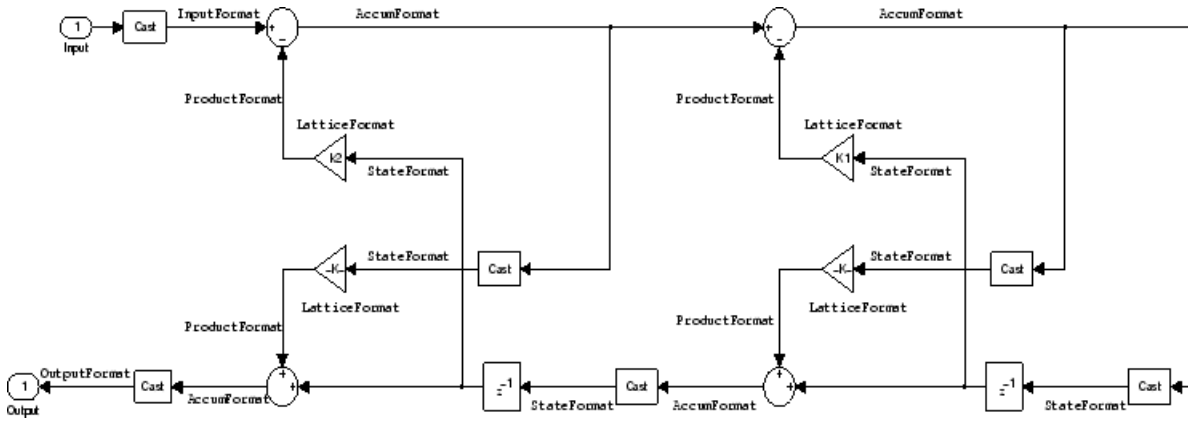
```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.latticeallpass` returns a default, discrete-time, lattice allpass filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the allpass lattice filter implemented by `dfilt.latticeallpass`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	`AccumMode`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
InputFormat	`InputWordLength`	`InputFracLength`	None
LatticeFormat	`CoeffWordLength`	`LatticeFracLength`	`CoeffAutoScale`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ProductFormat	`ProductWordLength`	`ProductFracLength`	`ProductMode`
StateFormat	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength`, `ProductWordLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the allpass lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
`AccumFracLength`	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties —`DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `LatticeFracLength` property value to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`Lattice`	Any lattice structure coefficients. No default value.
`LatticeFracLength`	Sets the fraction length applied to the lattice coefficients.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.

Property Name	Brief Description
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
`ProductFracLength`	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.

Property Name	Brief Description
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a third-order lattice allpass filter structure for a dfilt object hd, with the following code:

```
k = [.66 .7 .44];
hd=dfilt.latticeallpass(k);
```

Now convert hd to fixed-point arithmetic form.

```
hd.arithmetic='fixed'

hd =

          FilterStructure: 'Lattice Allpass'
               Arithmetic: 'fixed'
                  Lattice: [0.6600 0.7000 0.4400]
         PersistentMemory: false
                   States: [1x1 embedded.fi]

          CoeffWordLength: 16
```

```
      CoeffAutoScale: true
              Signed: true

     InputWordLength: 16
     InputFracLength: 15

    OutputWordLength: 16
          OutputMode: 'AvoidOverflow'

     StateWordLength: 16
     StateFracLength: 15

         ProductMode: 'FullPrecision'

           AccumMode: 'KeepMSB'
     AccumWordLength: 40
       CastBeforeSum: true

           RoundMode: 'convergent'
        OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.latticear, dfilt.latticearma, dfilt.latticemamax, dfilt.latticemamin

dfilt.latticear

Purpose

Discrete-time, lattice, autoregressive filter

Syntax

Refer to `dfilt.latticear` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.latticear(k)` returns a discrete-time, lattice autoregressive filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

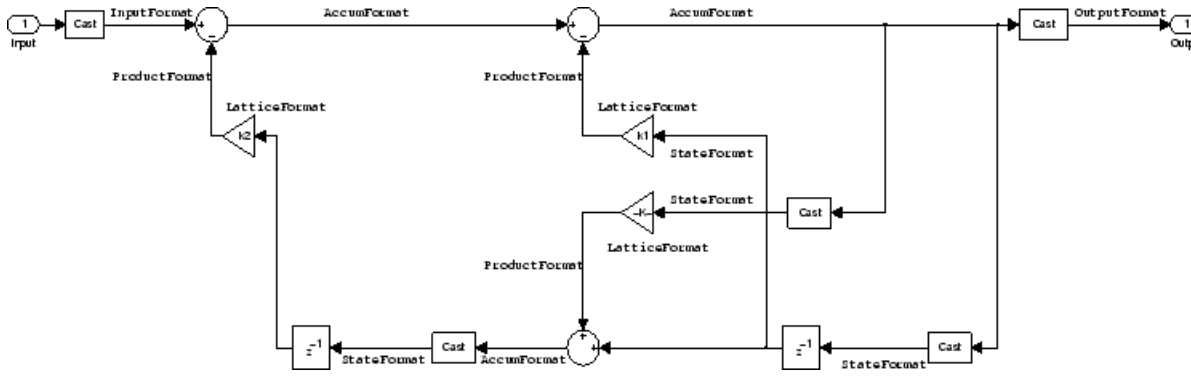
```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd = dfilt.latticear` returns a default, discrete-time, lattice autoregressive filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticear`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	`AccumMode`
InputFormat	`InputWordLength`	`InputFracLength`	None

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
LatticeFormat	CoeffWordLength	LatticeFracLength	CoeffAutoScale
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ProductFormat	ProductWordLength	ProductFracLength	ProductMode
StateFormat	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength`, `ProductWordLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the autoregressive lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties”.

Property Name	Brief Description
`AccumFracLength`	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `LatticeFracLength` to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`Lattice`	Any lattice structure coefficients.
`LatticeFracLength`	Sets the fraction length applied to the lattice coefficients.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.

Property Name	Brief Description
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`ProductFracLength`	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.

Property Name	Brief Description
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a third-order lattice autoregressive filter structure for a dfilt object, hd, with the following code that creates a fixed-point filter.

```
k = [.66 .7 .44];
hd1=dfilt.latticear(k)

hd1 =

         FilterStructure: 'Lattice Autoregressive (AR)'
              Arithmetic: 'double'
                 Lattice: [0.6600 0.7000 0.4400]
        PersistentMemory: false
                  States: [3x1 double]

hd1.arithmetic='fixed'

hd1 =

         FilterStructure: 'Lattice Autoregressive (AR)'
```

```
                Arithmetic: 'fixed'
                   Lattice: [0.6600 0.7000 0.4400]
          PersistentMemory: false
                    States: [1x1 embedded.fi]

           CoeffWordLength: 16
            CoeffAutoScale: true
                    Signed: true

           InputWordLength: 16
           InputFracLength: 15

          OutputWordLength: 16
                OutputMode: 'AvoidOverflow'

           StateWordLength: 16
           StateFracLength: 15

               ProductMode: 'FullPrecision'

                 AccumMode: 'KeepMSB'
           AccumWordLength: 40
             CastBeforeSum: true

                 RoundMode: 'convergent'
              OverflowMode: 'wrap'

specifyall(hd1)
hd1

hd1 =

         FilterStructure: 'Lattice Autoregressive (AR)'
                Arithmetic: 'fixed'
                   Lattice: [0.6600 0.7000 0.4400]
          PersistentMemory: false
                    States: [1x1 embedded.fi]
```

```
          CoeffWordLength: 16
           CoeffAutoScale: false
        LatticeFracLength: 15
                  Signed: true

          InputWordLength: 16
          InputFracLength: 15

         OutputWordLength: 16
               OutputMode: 'SpecifyPrecision'
         OutputFracLength: 12

          StateWordLength: 16
          StateFracLength: 15

              ProductMode: 'SpecifyPrecision'
        ProductWordLength: 32
        ProductFracLength: 30

                AccumMode: 'SpecifyPrecision'
          AccumWordLength: 40
          AccumFracLength: 30
            CastBeforeSum: true

                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.latticeallpass, dfilt.latticearma, dfilt.latticemamax, dfilt.latticemamin

Purpose

Discrete-time, lattice, autoregressive, moving-average filter

Syntax

Refer to `dfilt.latticearma` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.latticearma(k)` returns a discrete-time, lattice moving-average autoregressive filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

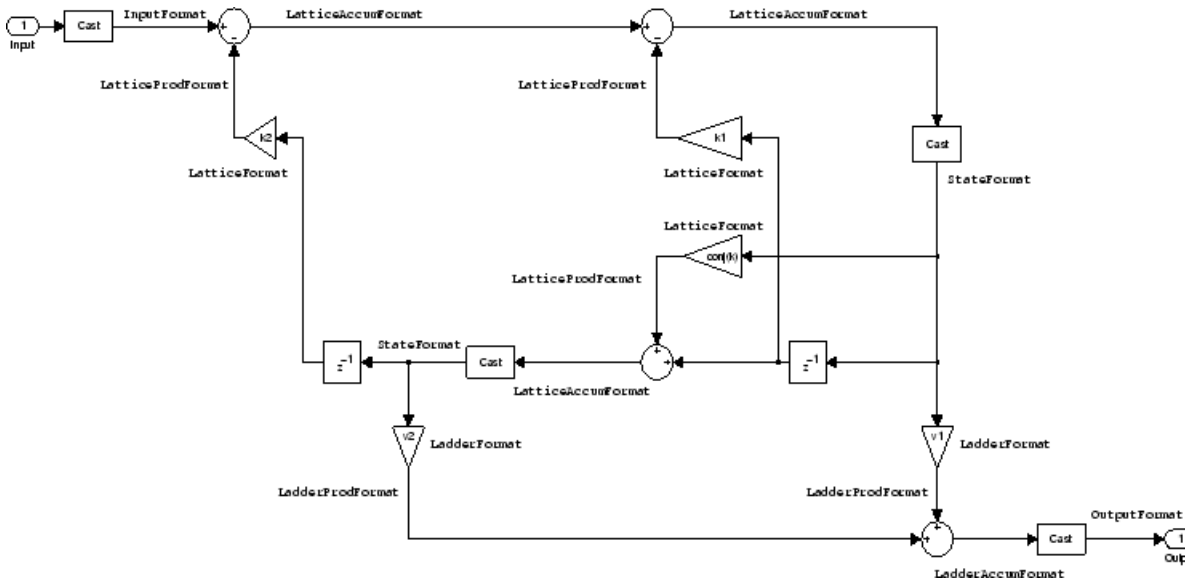
```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`hd dfilt.latticearma` returns a default, discrete-time, lattice moving-average, autoregressive filter object `hd`, with `k = [ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticearma`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
InputFormat	`InputWordLength`	`InputFracLength`	None
LadderAccumFormat	`AccumWordLength`	`LadderAccumFracLength`	`AccumMode`
LadderFormat	`CoeffWordLength`	`LadderFracLength`	`CoeffAutoScale`
LadderProdFormat	`ProductWordLength`	`LadderProdFracLength`	`ProductMode`
LatticeAccumFormat	`AccumWordLength`	`LatticeAccum-FracLength`	`AccumMode`
LatticeFormat	`CoeffWordLength`	`LatticeFracLength`	`CoeffAutoScale`
LatticeProdFormat	`ProductWordLength`	`LatticeProdFracLength`	`ProductMode`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`
StateFormat	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label LatticeProdFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that lattice coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the LatticeProdFormat refers to the properties `ProductWordLength`, `LatticeProdFracLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the autoregressive moving-average lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
`AccumFracLength`	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.

Property Name	Brief Description
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `LatticeFracLength` property to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`Ladder`	Stores the ladder coefficients for lattice ARMA (`dfilt.latticearma`) filters.

Property Name	Brief Description
LadderAccumFracLength	Sets the fraction length used to interpret the output from sum operations that include the ladder coefficients. You can change this property value after you set AccumMode to SpecifyPrecision.
LadderFracLength	Determines the precision used to represent the ladder coefficients in ARMA lattice filters.
Lattice	Stores the lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.

Property Name	Brief Description
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
`ProductFracLength`	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.

Property Name	Brief Description
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

See Also

dfilt, dfilt.latticeallpass, dfilt.latticear, dfilt.latticemamin, dfilt.latticemamin

Purpose

Discrete-time, lattice, moving-average filter with maximum phase

Syntax

Refer to `dfilt.latticemamax` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.latticemamax(k)` returns a discrete-time, lattice, moving-average filter object `hd`, with lattice coefficients `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

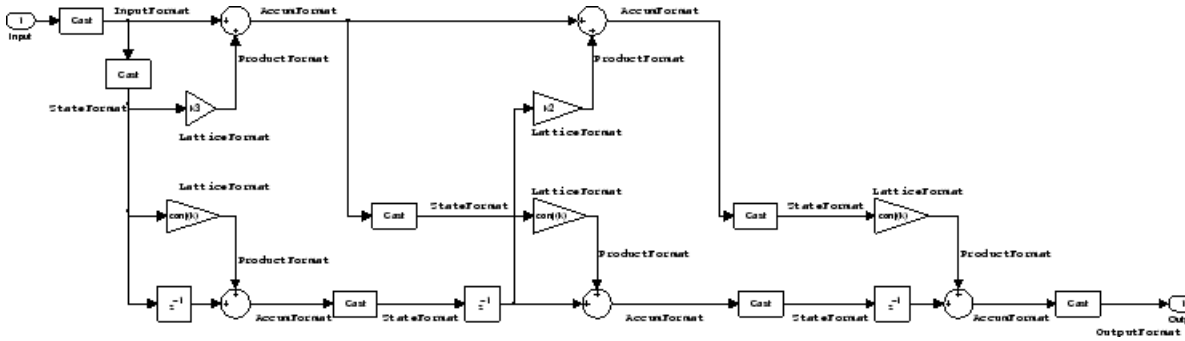
Note When the `k` coefficients define a maximum phase filter, the resulting filter in this structure is maximum phase. When your coefficients do not define a maximum phase filter, placing them in this structure does not produce a maximum phase filter.

`hd = dfilt.latticemamax` returns a default discrete-time, lattice, moving-average filter object `hd`, with `k = [ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the maximum phase implementation of a moving-average lattice filter implemented by `dfilt.latticemamax`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical

operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	`AccumMode`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
InputFormat	`InputWordLength`	`InputFracLength`	None
LatticeFormat	`CoeffWordLength`	`LatticeFracLength`	`CoeffAutoScale`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ProductFormat	`ProductWordLength`	`ProductFracLength`	`ProductMode`
StateFormat	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength`, `ProductWordLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the maximum phase, moving average lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties”.

Property Name	Brief Description
`AccumFracLength`	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `LatticeFracLength` property to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`Lattice`	Any lattice structure coefficients.
`LatticeFracLength`	Sets the fraction length applied to the lattice coefficients.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.

Property Name	Brief Description
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
`ProductFracLength`	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.

Property Name	Brief Description
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a fourth-order lattice, moving-average, maximum phase filter structure for a dfilt object, hd, with the following code:

```
k = [.66 .7 .44 .33];
hd = dfilt.latticemamax(k)
hd =
             FilterStructure: 'Lattice maximum phase'
                     Lattice: [1x4 double]
    NumberOfSamplesProcessed: 0
                 ResetStates: 'on'
                      States: [4x1 double]
```

See Also

dfilt, dfilt.latticeallpass, dfilt.latticear, dfilt.latticearma, dfilt.latticemamin

dfilt.latticemamin

Purpose

Discrete-time, lattice, moving-average filter with minimum phase

Syntax

Refer to `dfilt.latticemamin` in Signal Processing Toolbox™ documentation.

Description

`hd = dfilt.latticemamin(k)` returns a discrete-time, lattice, moving-average, minimum phase, filter object `hd`, with lattice coefficients `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

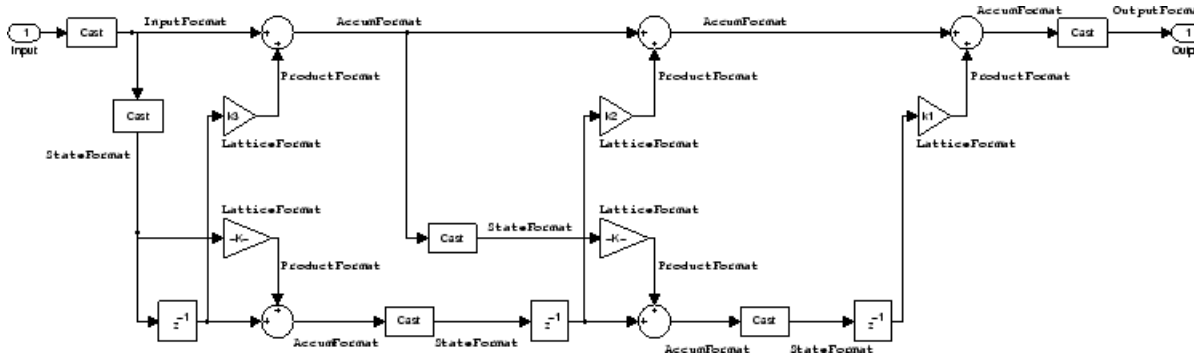
Note When the `k` coefficients define a minimum phase filter, the resulting filter in this structure is minimum phase. When your coefficients do not define a minimum phase filter, placing them in this structure does not produce a minimum phase filter.

`hd = dfilt.latticemamin` returns a default discrete-time, lattice, moving-average, minimum phase, filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the minimum phase implementation of a moving-average lattice filter implemented by `dfilt.latticemamin`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical

operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	AccumMode
InputFormat	InputWordLength	InputFracLength	None
LatticeFormat	CoeffWordLength	LatticeFracLength	CoeffAutoScale
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ProductFormat	ProductWordLength	ProductFracLength	ProductMode
StateFormat	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength`, `ProductWordLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the minimum phase, moving average lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
`AccumFracLength`	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `LatticeFracLength` property to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`Lattice`	Any lattice structure coefficients.
`LatticeFracLength`	Sets the fraction length applied to the lattice coefficients.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.

Property Name	Brief Description
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`ProductFracLength`	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.

Property Name	Brief Description
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a third-order lattice, moving-average, minimum phase, filter structure for a dfilt object, hd, with the following code:

```
k = [.66 .7 .44];
hd = dfilt.latticemamin(k)

hd =

         FilterStructure: 'Lattice Moving-Average (MA) For Minimum
Phase'
              Arithmetic: 'double'
                 Lattice: [0.6600 0.7000 0.4400]
        PersistentMemory: false
                  States: [3x1 double]

set(hd,'arithmetic','fixed')
specifyall(hd)
hd
```

```
hd =

         FilterStructure: 'Lattice Moving-Average (MA) For Minimum
Phase'
              Arithmetic: 'fixed'
                 Lattice: [0.6600 0.7000 0.4400]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: false
       LatticeFracLength: 15
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'SpecifyPrecision'
        OutputFracLength: 12

         StateWordLength: 16
         StateFracLength: 15

             ProductMode: 'SpecifyPrecision'
       ProductWordLength: 32
       ProductFracLength: 30

               AccumMode: 'SpecifyPrecision'
         AccumWordLength: 40
         AccumFracLength: 30
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

See Also

dfilt, dfilt.latticeallpass, dfilt.latticear, dfilt.latticearma, dfilt.latticemamax

Purpose

Discrete-time, parallel structure filter

Syntax

Refer to `dfilt.parallel` in Signal Processing Toolbox™ documentation.

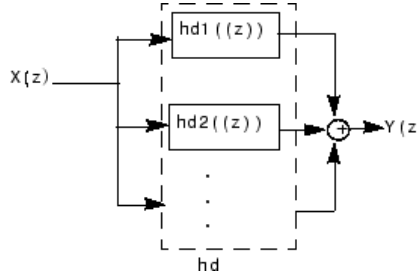
Description

`hd = dfilt.parallel(hd1,hd2,...)` returns a discrete-time filter object `hd`, which is a structure of two or more `dfilt` filter objects, `hd1`, `hd2`, and so on arranged in parallel.

You can also use the standard notation to combine filters into a parallel structure.

```
parallel(hd1,hd2,...)
```

In this syntax, `hd1`, `hd2`, and so on can be a mix of `dfilt` objects, `mfilt` objects, and `adaptfilt` objects.



`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the parallel structure must be the same arithmetic format — double, single, or fixed. `hd`, the filter returned, inherits the format of the individual filters.

See Also

`dfilt`, `dfilt.cascade`, `parallel`

`dfilt.cascade`, `dfilt.parallel` in Signal Processing Toolbox documentation

dfilt.scalar

Purpose

Discrete-time, scalar filter

Syntax

Refer to `dfilt.scalar` in Signal Processing Toolbox™ documentation.

Description

`dfilt.scalar(g)` returns a discrete-time, scalar filter object with gain `g`, where `g` is a scalar.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic".

`dfilt.scalar` returns a default, discrete-time scalar gain filter object `hd`, with gain `1`.

Properties

In this table you see the properties associated with the scalar implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties”.

Property Name	Brief Description
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `CoeffFracLength` property to specify the precision used.
`CoeffFracLength`	Set the fraction length the filter uses to interpret coefficients. `CoeffFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`Gain`	Returns the gain for the scalar filter. Scalar filters do not alter the input data except by adding gain.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.

Property Name	Brief Description
`InputWordLength`	Specifies the word length applied to interpret input data.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

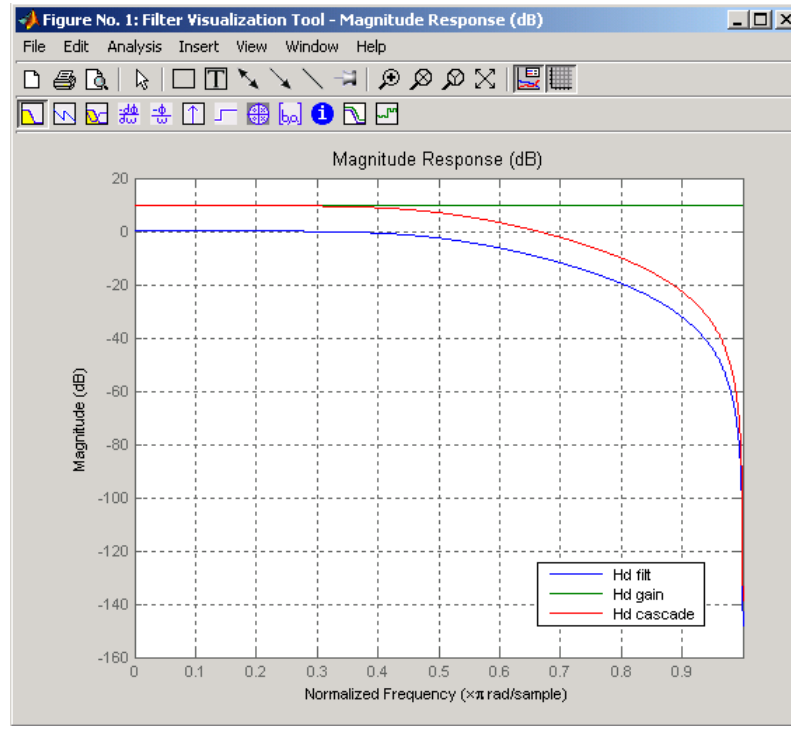
Property Name	Brief Description
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to `filtstates` in Signal Processing Toolbox documentation or in the Help system.

Example

Create a direct-form I filter object `hd_filt` and a scalar object with a gain of 3 `hd_gain` 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(hd_gain,hd_filt)
fvtool(hd_filt,hd_gain,hd)
hd_filt =
         FilterStructure: 'direct-form I'
              Arithmetic: 'double'
               Numerator: [0.3000 0.6000 0.3000]
             Denominator: [1 0 0.2000]
        PersistentMemory: false
                  States: [4x1 double]

hd_gain =
         FilterStructure: 'Scalar'
              Arithmetic: 'double'
                    Gain: 3
        PersistentMemory: false
                  States: []
```

```
hd =
          FilterStructure: Cascade
               Section(1): Scalar
               Section(2): Direct Form I
         PersistentMemory: false
```



To view the sections of the cascaded filter, use

```
hd.section(1)

ans =
          FilterStructure: 'Scalar'
               Arithmetic: 'double'
                     Gain: 3
```

```
          PersistentMemory: false
                    States: []
```

and

```
hd.section(2)

ans =
           FilterStructure: 'Direct Form I'
                Arithmetic: 'double'
                 Numerator: [0.3000 0.6000 0.3000]
               Denominator: [1 0 0.2000]
          PersistentMemory: false
                    States: [4x1 double]
```

See Also

`dfilt`, `dfilt.cascade`

Purpose

Wave digital allpass filter

Syntax

```
hd = dfilt.wdfallpass(c)
```

Description

`hd = dfilt.wdfallpass(c)` constructs an allpass wave digital filter structure given the allpass coefficients in vector `c`.

Vector `c` must have, one, two, or four elements (filter coefficients). Filters with three coefficients are not supported. When you use `c` with four coefficients, the first and third coefficients must be 0.

Given the coefficients in `c`, the transfer function for the wave digital allpass filter is defined by

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

Internally, the allpass coefficients are converted to wave digital filters for filtering. Note that `dfilt.wdfallpass` allows only stable filters. Also note that the leading coefficient in the denominator, a 1, does not need to be included in vector `c`.

Use the constructor `dfilt.cascadewdfallpass` to cascade `wdfallpass` filters.

To compare these filters to other similar filters, `dfilt.wdfallpass` and `dfilt.cascadewdfallpass` filters have the same number of multipliers as the non-wave digital filters `dfilt.allpass` and `dfilt.cascadeallpass`. However, the wave digital filters use fewer states and they may require more adders in the filter structure.

Wave digital filters are usually used to create other filters. This toolbox uses them to implement halfband filters, which the first example in Examples demonstrates. They are most often building blocks for filters.

Properties

In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
`AllpassCoefficients`	Contains the coefficients for the allpass wave digital filter object
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to `filtstates` in Signal Processing Toolbox™ documentation or in the Help system.

Filter Structure

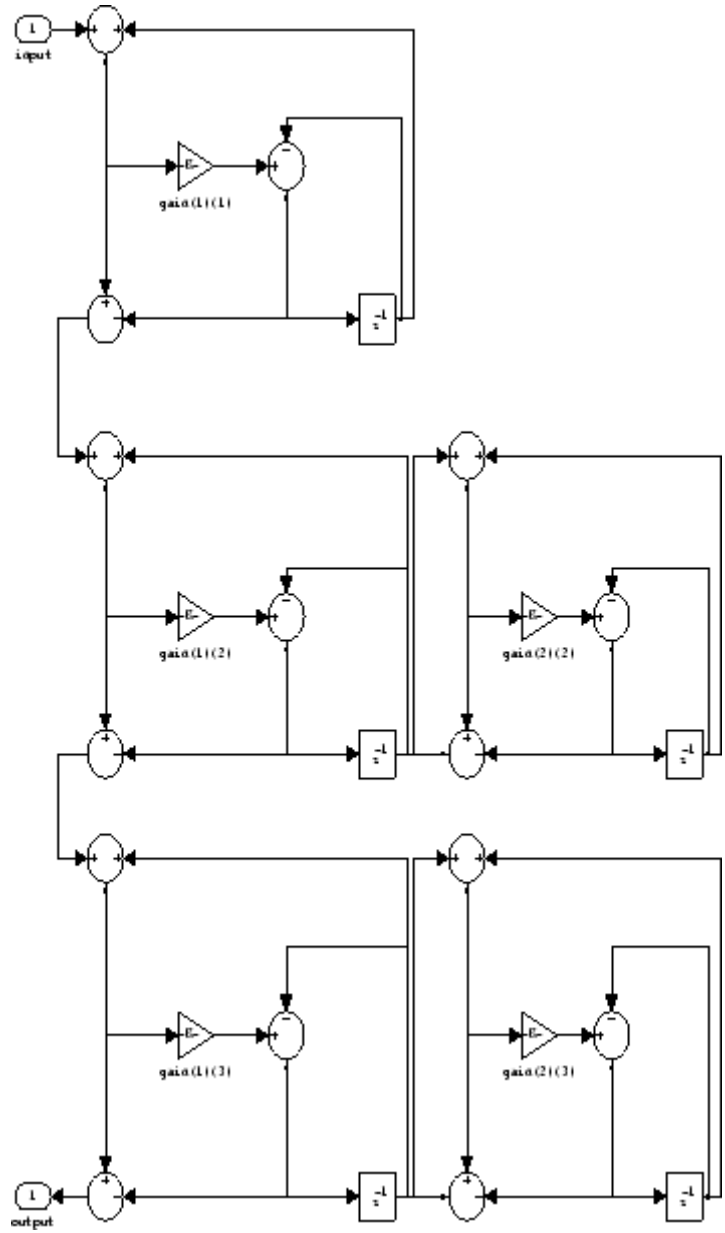
When you change the order of the wave digital filters in the cascade, the filter structure changes as well.

As shown in this example, `realizemdl` lets you see the filter structure used for your filter, if you have Simulink® installed.

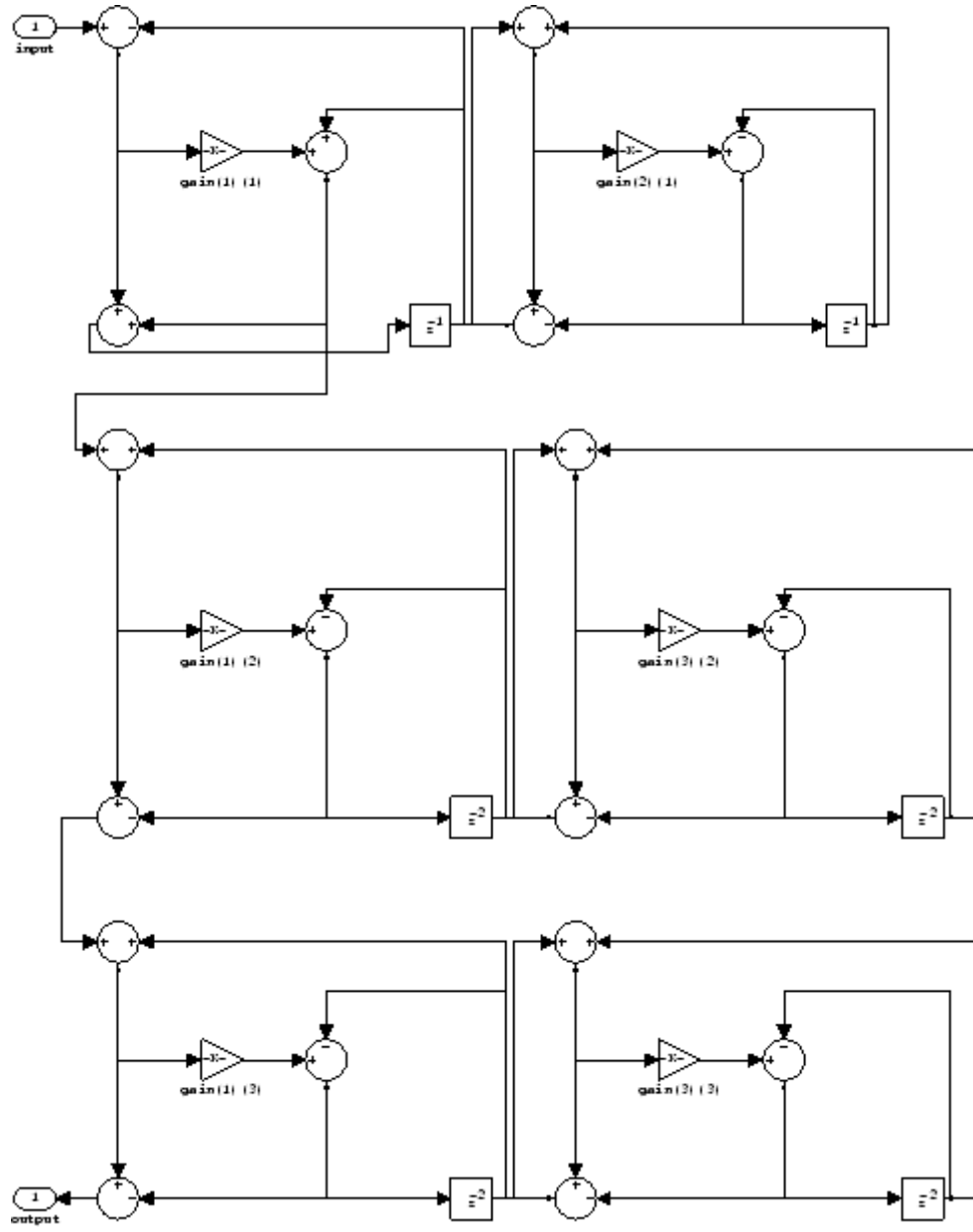
```
section11=0.8;
section12=[1.5,0.7];
section13=[1.8,0.9];
hd1=dfilt.cascadewdfallpass(section11,section12,section13);
realizemdl(hd1)

section21=[0.8,0.4];
section22=[0,1.5,0,0.7];
```

```
section23=[0,1.8,0,0.9];
hd2=dfilt.cascadewdfallpass(section21,section22,section23);
realizemdl(hd2)
```

`hd1` has this filter structure with three sections.


1
input
gain(1)(1)
gain(1)(2)
gain(2)(2)
gain(1)(3)
gain(2)(3)
1
output

The filter structure for `hd2` is somewhat different, with the different orders and interconnections between the three sections.

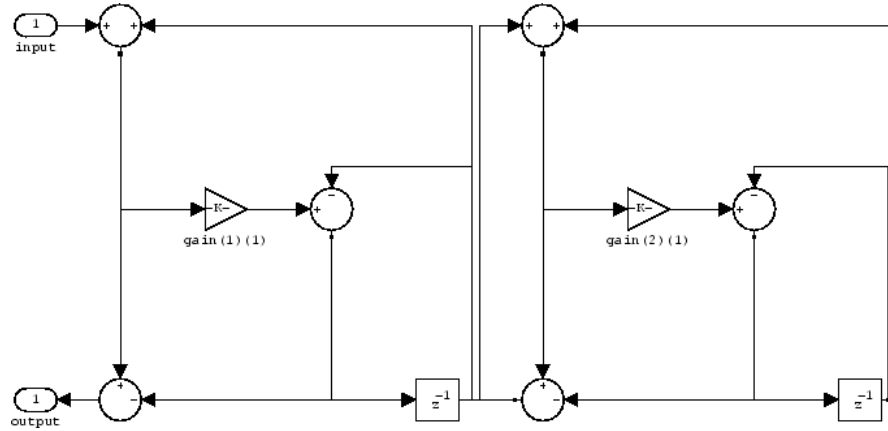

1
input
gain(1) (1)
gain(2) (1)
gain(1) (2)
gain(3) (2)
gain(1) (3)
gain(3) (3)
1
output

Examples

Construct a second-order wave digital allpass filter with two coefficients. Note that to use `realizemdl`, you must have Simulink.

```
c = [1.5,0.7];
hd = dfilt.wdfallpass(c);
info(hd)

Discrete-Time IIR Filter (real)
-------------------------------
Filter Structure       : Wave Digital Filter Allpass
Number of Multipliers  : 2
Stable                 : Yes
Linear Phase           : No

Implementation Cost
Number of Multipliers : 2
Number of Adders      : 6
Number of States      : 2
MultPerInputSample    : 2
AddPerInputSample     : 6

realizemdl(hd)
```

With Simulink installed, `realizemdl` returns this structure for `hd`.



See Also

`dfilt`, `dfilt.allpass`, `dfilt.latticeallpass`, `dfilt.cascadewdfallpass`, `dfilt.cascadeallpass`, `mfilt.iirdecim`, `mfilt.iirinterp`

Purpose

Filter properties and values

Syntax

```
disp(hd)
disp(ha)
disp(hm)
```

Description

Similar to omitting the closing semicolon from an expression on the command line, except that `disp` does not display the variable name. `disp` lists the property names and property values for any filter object, such as a `dfilt` object or `adaptfilt` object.

The following examples illustrate the default display for an adaptive filter `ha` and a multirate filter `hm`.

```
ha=adaptfilt.rls

ha =

      Algorithm: 'Direct Form FIR RLS Adaptive Filter'
    FilterLength: 10
    Coefficients: [0 0 0 0 0 0 0 0 0 0]
          States: [9x1 double]
ForgettingFactor: 1
      KalmanGain: []
         InvCov: [10x10 double]
PersistentMemory: false

disp(ha)
        Algorithm: 'Direct-Form FIR RLS Adaptive Filter'
     FilterLength: 10
     Coefficients: [0 0 0 0 0 0 0 0 0 0]
           States: [9x1 double]
 ForgettingFactor: 1
       KalmanGain: []
          InvCov: [10x10 double]
PersistentMemory: false
```

```
hm=mfilt.cicdecim(6)

hm =

 FilterStructure: 'Cascaded Integrator-Comb Decimator'
      Arithmetic: 'fixed'
DifferentialDelay: 1
 NumberOfSections: 2
 DecimationFactor: 6
 PersistentMemory: false

  InputWordLength: 16
  InputFracLength: 15

SectionWordLengthMode: 'MinWordLengths'

     OutputWordLength: 16

disp(hm)

FilterStructure: 'Cascaded Integrator-Comb
                  Decimator'
       Arithmetic: 'fixed'
DifferentialDelay: 1
 NumberOfSections: 2
 DecimationFactor: 6
 PersistentMemory: false

  InputWordLength: 16
  InputFracLength: 15

SectionWordLengthMode: 'MinWordLengths'

     OutputWordLength: 16
```

See Also

set

Purpose

Cast fixed-point filter to use double-precision arithmetic

Syntax

```
hd = double(h)
```

Description

`hd = double(h)` returns a new filter `hd` that has the same structure and coefficients as `h`, but whose arithmetic property is set to `double` to use double-precision arithmetic for filtering. `double(h)` is not the same as the `reffilter(h)` function:

- `hd`, the filter returned by `double` has the quantized coefficients of `h` represented in double-precision floating-point format
- The reference filter returned by `reffilter` has double-precision, floating-point coefficients that have not been quantized.

You might find `double(h)` useful to isolate the effects of quantizing the coefficients of a filter by using `double` to create a filter `hd` that operates in double-precision but uses the quantized filter coefficients.

Examples

Use the same filter, once with fixed-point arithmetic and once with floating-point, to compare fixed-point filtering with double-precision floating-point filtering.

```
h = dfilt.dffir(firgr(27,[0 .4 .6 1],...
[1 1 0 0]));              % Lowpass filter.
% Set h to use fixed-point arithmetic to filter.
% Quantize the coeffs.
h.arithmetic = 'fixed';
hd = double(h);           % Cast h to double-precision
                          % floating-point coefficients.
n = 0:99; x = sin(0.7*pi*n(:)); % Set up an input signal.
y = filter(h,x);                % Fixed-point output.
yd = filter(hd,x);              % Floating-point output.
norm(yd-double(y),inf)
ans =

  9.2014e-004
```

`norm` shows that the largest difference (maximum error) between the output values from the fixed versus floating filtering comparison is about 0.0009 — either good or less good depending on your application.

See Also

`reffilter`

Purpose

Elliptic filter using specification object

Syntax

```
hd = design(d,'ellip')
hd = design(d,'ellip',designoption,value,designoption,...
value,...)
```

Description

`hd = design(d,'ellip')` designs an elliptical IIR digital filter using the specifications supplied in the object `h`.

`hd = design(d,'ellip',designoption,value,designoption,...
value,...)` returns an elliptical or Cauer 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 `ellip`, refer to the command line help system. For example, to get specific information about using `ellip` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'ellip')
```

Examples

These example demonstrate using `ellip` to design filters based on filter specification objects.

Example 1

Construct the default bandpass filter specification object and design an elliptic filter.

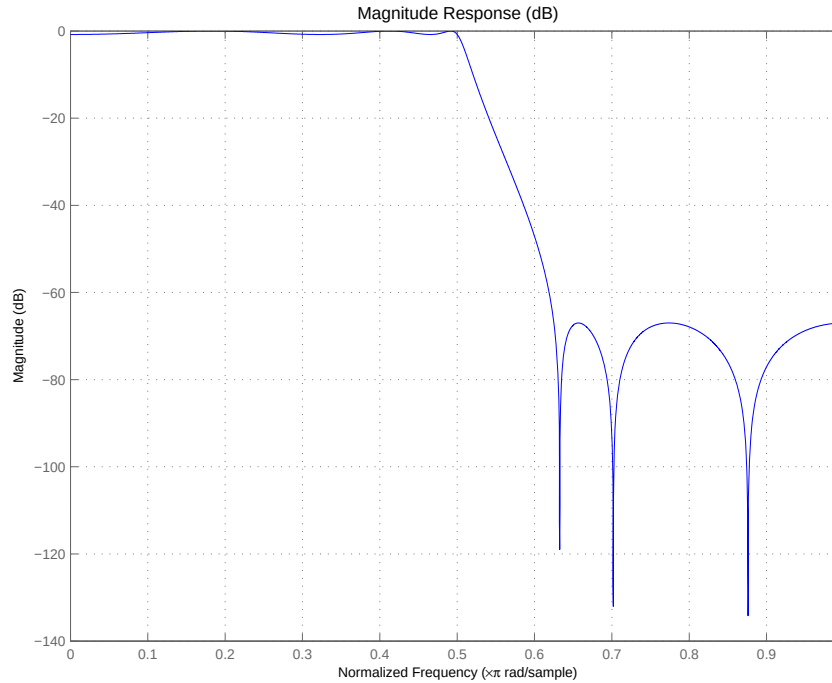
```
d = fdesign.bandpass;
designopts(d,'ellip')

ans =

    FilterStructure: 'df2sos'
```

```
       MatchExactly: 'both'

hd = design(d,'ellip','matchexactly','both');

hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
       sosMatrix: [4x6 double]
     ScaleValues: [5x1 double]
PersistentMemory: false
```

Example 2

Construct a lowpass object with order, passband-edge frequency, stopband-edge frequency, and passband ripple specifications, and then design an elliptic filter.

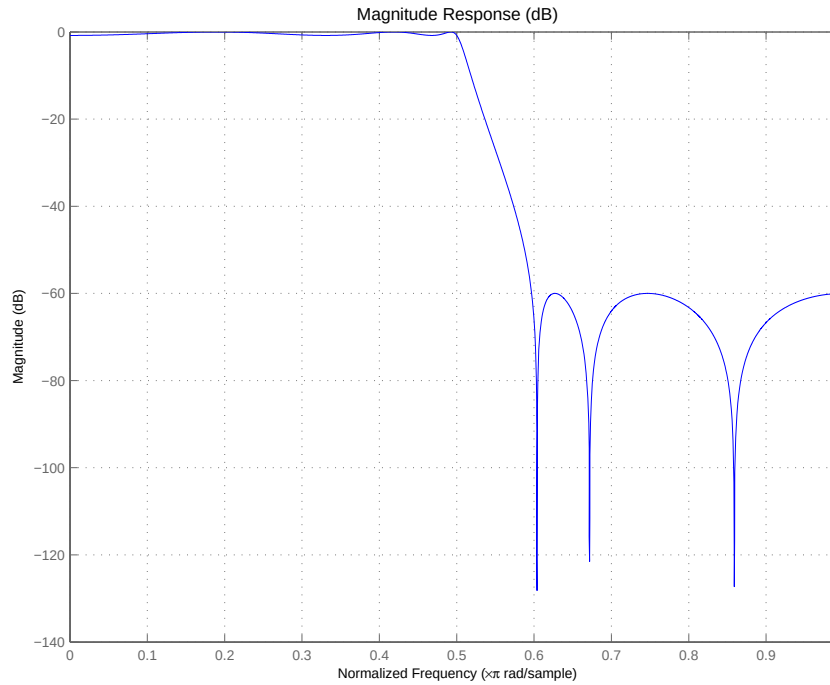
```
d = fdesign.lowpass('n,fp,fst,ap',6,20,25,.8,80);
design(d,'ellip'); % Starts FVtool to display the filter.
```



Example 3

Construct a lowpass object with filter order, passband edge frequency, passband ripple, and stopband attenuation specifications, and then design an elliptic filter.

```
d = fdesign.lowpass('n,fp,ap,ast',6,20,.8,60,80);
design(d,'ellip'); % Starts FVTool to display the filter.
```



See Also

`butter`, `cheby1`, `cheby2`

Purpose

Euclid factors for multirate filter

Syntax

```
[lo,mo] = euclidfactors(hm)
```

Description

`[lo,mo] = euclidfactors(hm)` returns integer factors `lo` and `mo` such that (`lo`*L)-(`mo`*M) = -1. L and M are relatively prime and represent the interpolation and decimation factors of the multirate filter `hm`.

`euclidfactors` works with multirate filters that have both decimation and interpolation factors, such as `mfilt.firfracdecim`, `mfilt.firfracinterp`, or `mfilt.firsrc`. You cannot return `lo` and `mo` for decimators or interpolators.

Examples

Use an FIR fractional decimator, with L = 5 and M = 7, to show what `euclidfactors` does.

```
hm=mfilt.firfracdecim(5,7)

hm =

 FilterStructure: 'Direct-Form FIR Polyphase Fractional Decimator'
       Numerator: [1x168 double]
RateChangeFactors: [5 7]
 PersistentMemory: false
           States: [62x1 double]

[lo,mo]=euclidfactors(hm)

lo =

     4

mo =

     3
```

Indeed, (`lo`*L)-(`mo`*M) = (4*5)-(3*7) = -1.

See Also polyphase, nstates

Purpose

Equiripple single-rate or multirate 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 or multirate filter using the specifications supplied in the object `d`. Equiripple filter designs minimize the maximum ripple in the passbands and stopbands.

`hd` is either a `dfilt` object (a single-rate digital filter) or an `mfilt` object (a multirate digital filter) depending on the `Specification` property of the filter specification object `d` and the specifications object type — halfband or interpolator.

When you use `equiripple` with Nyquist filter specification objects, you might encounter design cases where the filter design does not converge. Convergence errors occur mostly at large filter orders, or small transition widths, or large stopband attenuations. These specifications, alone or combined, can cause design failures. For more information, refer to `fdesign.nyquist` in the online Help system.

`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')
```

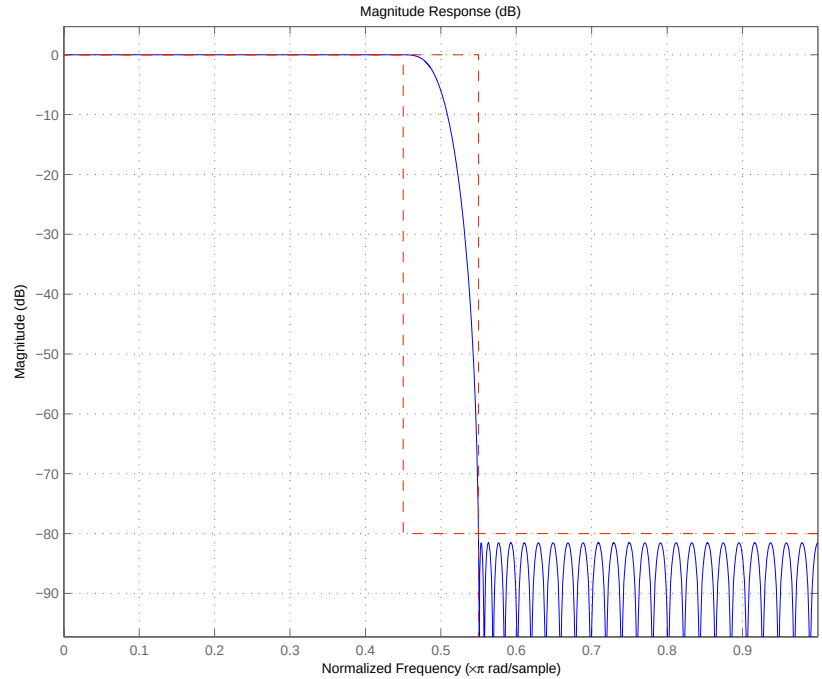
Examples

Here is an example of designing a single-rate equiripple filter from a halfband filter specification object. Notice the `help` command used to learn about the options for the specification object and method.

```
d = fdesign.halfband(tw,ast,0.1,80);
designmethods(d)

Design Methods for class fdesign.halfband (TW,Ast):

butter
ellip
iirlinphase
equiripple
kaiserwin

help(d,'equiripple')

DESIGN Design an equiripple FIR filter
HD = DESIGN(D, 'equiripple') designs an equiripple filter
specified by the FDESIGN object D.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter
with the structure STRUCTURE.  STRUCTURE is 'dffir' by default and
can be any of the following:

    'dffir'
    'dffirt'
    'dfsymfir'
    'dfasymfir'
    'fftfir'

designopts(d,'equiripple')

ans =

    FilterStructure: 'dffir'
           MinPhase: 0
```

```
        StopbandShape: 'flat'
        StopbandDecay: 0

hd = design(d,'equiripple','stopbandshape','flat');
fvtool(hd);
```

Displaying the filter in FVTool shows the equiripple nature of the filter.



`equiripple` also designs multirate filters. This example generates a halfband interpolator filter.

```
d = fdesign.interpolator(2); % Interpolation factor = 2.
hd = design(d,'equiripple');

hd
```

```
hd =

       FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
            Arithmetic: 'double'
             Numerator: [1x95 double]
    InterpolationFactor: 2
       PersistentMemory: false
```

This final example designs an equiripple filter with a direct-form structure by specifying the **`filterstructure`** argument. To set the design options for the filter, use the `designopts` method and options object `opts`.

```
d = fdesign.lowpass('fp,fst,ap,ast');
designopts(d,'equiripple')

ans =

    FilterStructure: 'dffir'
      DensityFactor: 16
            MinPhase: 0
            MinOrder: 'any'
      StopbandShape: 'flat'
      StopbandDecay: 0

opts=designopts(d,'equiripple')

opts =

    FilterStructure: 'dffir'
      DensityFactor: 16
            MinPhase: 0
            MinOrder: 'any'
      StopbandShape: 'flat'
      StopbandDecay: 0
```

```
opts.FilterStructure='dffirt'

opts =

    FilterStructure: 'dffirt'
      DensityFactor: 16
            MinPhase: 0
            MinOrder: 'any'
      StopbandShape: 'flat'
      StopbandDecay: 0

opts.MinPhase=1;

opts.DensityFactor=20;

opts =

    FilterStructure: 'dffirt'
      DensityFactor: 20
            MinPhase: 1
            MinOrder: 'any'
      StopbandShape: 'flat'
      StopbandDecay: 0

hd=design(d,'equiripple',opts)

hd =

     FilterStructure: 'Direct-Form FIR Transposed'
          Arithmetic: 'double'
           Numerator: [1x37 double]
    PersistentMemory: false
```

See Also

`fdesign.nyquist`, `firls`, `kaiserwin`

Purpose

Farrow filter

Syntax

```
hd = farrow.structure(delay,...)
```

Note The `farrow` function has been deprecated, and if possible you should use`dfilt` to create Farrow filters instead.

Description

`hd = farrow.structure(delay,...)` returns a Farrow filter `hd` that associates `delay`, the fractional delay, with a filter structure specified by `structure`.

Digital fractional delay filters are useful tools for fine-tuning the sampling instants of signals, such as implementing the required bandlimited interpolation. They can be found in the synchronization of digital modems where the delay parameter varies over time, or in wireless communications systems where the signal delay changes with location and distance from the transmitter. Farrow filters are one such fractional delay filter that allows the user to vary the delay.

More information about Farrow filters is available in References.

You can change the fractional delay input value as you filter by assigning a new value to `delay` before you filter with `hd`. Thus Farrow filters provide delay tunability when your input signals have time-varying delays.

Provide the fractional delay as a decimal part of an input sample, such as 0.2. `delay` must be positive and between 0 and 1.

`structure` accepts the following strings that describe the filter structure to use:

structure String	Description
`fd`	Generic fractional delay Farrow filter
`linearfd`	Linear fractional delay Farrow filter

In the `farrow.fd` syntax

```
hd = farrow.fd(delay,...)
```

you must specify the coefficients as input arguments. Use `fdesign.fracdelay` to generate `farrow.fd` filter design coefficients. For more information about the coefficients, refer to References.

Farrow filters support numerous functions for analyzing and simulating the filter, and for generating code from the filter. To learn about the functions you use with Farrow filters, enter

```
help farrow/functions
```

at the Command prompt to see the complete list of functions.

The functions and methods that you use most often with digital filters are

Function	Description
`cost`	Estimate the hardware implementation cost in terms of mathematical operations like add and multiply
`filter`	Execute the filter by using it to filter data
`fvtool`	Display and analyze the filter
`freqrespest`	Use filtering to estimate filter frequency response
`freqz`	Compute the instantaneous frequency response of the filter
`realizemdl`	Generate a Simulink® subsystem model of the filter as a block (Requires Simulink)

Fixed-Point Farrow Filters

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

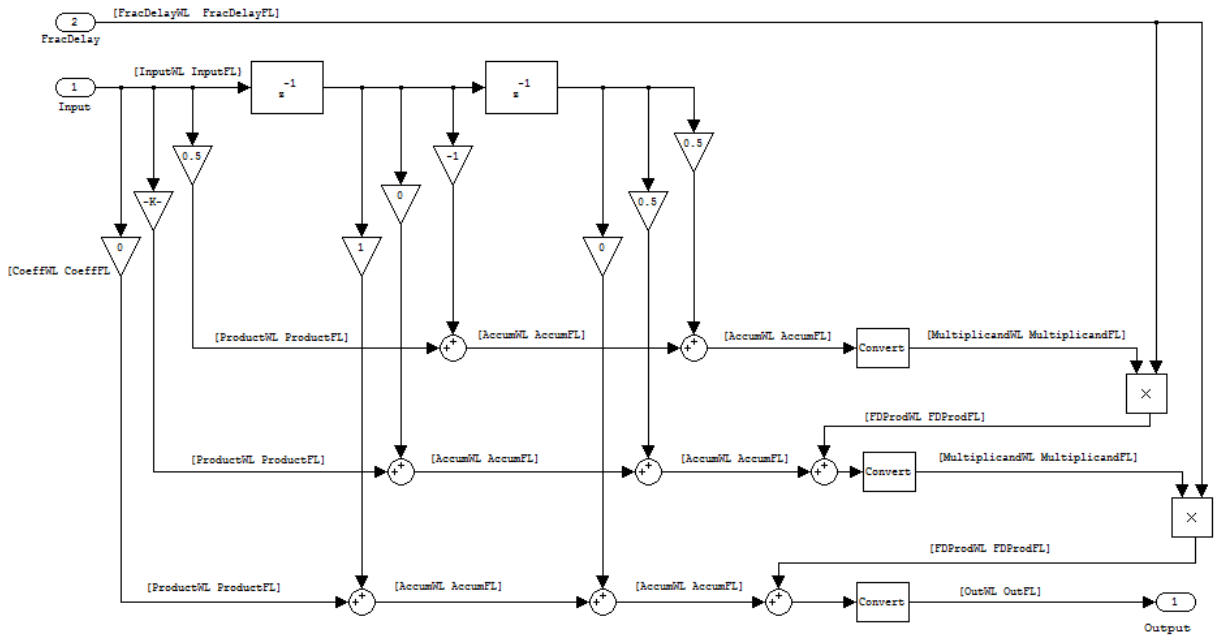
For more information about the property `Arithmetic`, refer to "Arithmetic".

Note `a(1)`, the leading denominator coefficient, cannot be 0. To be able to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

You cannot use `qreport` to log the filtering operations of a fixed-point Farrow filter.

Fixed-Point Filter Structure

The following figure shows the signal flow for the fractional delay Farrow filter implemented by `farrow.fd`. To help you see how the filter processes the coefficients, input, output, and states of the filter, as well as numerical operations, the figure includes the locations of the arithmetic and data type format elements within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the preceding signal flow diagram includes labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that correspond to it.

The labels use a common format — a descriptor followed by WL or FL. WL stands for word length and FL for fraction length. The pairing of WL and FL entries explain the data format at the labeled location in the filter.

For example, InputWL label refers to the word length and InputFL to the fraction length used to interpret data you input to the filter. The corresponding filter properties `InputWordLength` and `InputFracLength` (as shown in the following table) store the word length and the fraction

length in bits in the filter object. Or consider `CoeffFormat`, which refers to the word and fraction lengths (`CoeffWordLength`, `CoeffFracLength`) associated with representing filter coefficients.

Signal Flow Label	Corresponding Filter Property
InputWL	`InputWordLength`
InputFL	`InputFracLength`
FracDelayWL	`FDWordLength`
FracDelayFL	`FDFracLength`
CoeffWL	`CoeffWordLength`
CoeffFL	`CoeffFracLength`
ProductWL	`ProductWordLength`
ProductFL	`ProductFracLength`
AccumWL	`AccumWordLength`
AccumFL	`AccumFracLength`
MultiplicandWL	`MultiplicandWordLength`
MultiplicandFL	`MultiplicandFracLength`
FracDelayProdWL	`FDProdWordLength`
FracDelayProdFL	`FDProdFracLength`
OutputWL	`OutputWordLength`
OutputFL	`OutputFracLength`

Properties

In this table you see the properties associated with Farrow filters in fixed-point form.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties".

Property Name	Brief Description
`AccumFracLength`	Sets the fraction length used to store data in the accumulator/buffer.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `CoeffWordLength` and `CoeffFracLength` properties to specify the data format used.
`CoeffFracLength`	Specifies the fraction length to apply to filter coefficients.
`Coefficients`	Contains the coefficients for the filter.

Property Name	Brief Description
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`FDAutoScale`	Specifies whether the filter automatically chooses the proper scaling to represent the fractional delay value without overflowing. Turning this off by setting the value to `false` enables you to change the `FDWordLength` and `FDFracLength` properties to specify the data format applied.
`FDFracLength`	Specifies the fraction length to represent the fractional delay.
`FDProdFracLength`	Specifies the fraction length to represent the result of multiplying the coefficients with the fractional delay.
`FDProdWordLength`	Specifies the word length to represent result of multiplying the coefficients with the fractional delay.
`FDWordLength`	Specifies the word length to represent the fractional delay.

Property Name	Brief Description
`FilterInternals`	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`FracDelay`	Specifies the fractional delay provided by the filter, in decimal fractions of a sample.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`MultiplicandFracLength`	Specifies the fraction length to use for multiplication operation inputs. This property becomes writable (you can change the value) when you set `FilterInternals` to `SpecifyPrecision`.

Property Name	Brief Description
`MultiplicandWordLength`	Specifies the word length to use for multiplication operation inputs. This property becomes writable (you can change the value) when you set `FilterInternals` to `SpecifyPrecision`.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`ProductFracLength`	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `FilterInternals` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `FilterInternals` to `SpecifyPrecision`.

Property Name	Brief Description
`RoundMode`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

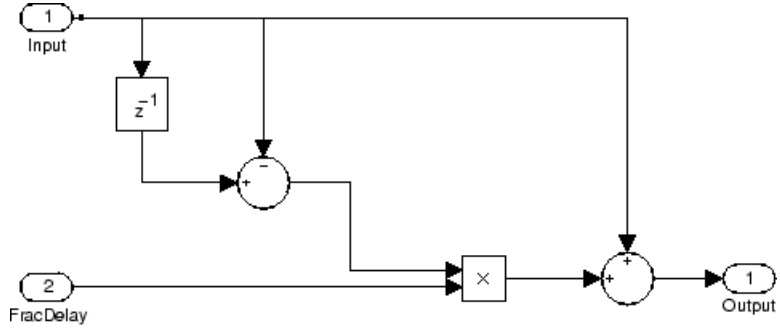
Property Name	Brief Description
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to `filtstates` in Signal Processing Toolbox™ documentation or in the Help system.

Examples

Construct a filter with linear fractional delay of 0.4 samples. Use linearfd for the structure and set delay equal to 0.4.

```
delay = 0.4;
hd = farrow.linearfd(delay);
fvtool(hd) % Analyze the filter.
```

`realizemdl` produces this model from basic Signal Processing blockset blocks.



References

Erup, L., Floyd M. Gardner, and Robert A. Harris, “Interpolation in Digital Modems-Part II: Implementation and Performance,” *IEEE® Transactions on Communications*, vol. 41, No. 6, June 1993, pp. 998-1008.

Marvasti, F., *Nonuniform Sampling—Theory and Practice*, Kluwer Academic/Plenum Publishers, New York, 2001.

See Also

`adaptfilt`, `dfilt`, `fdesign`, `mfilt`

Purpose

Write file containing filter coefficients

Syntax

```
fcfwrite(h)
fcfwrite(h,filename)
fcfwrite(...,'fmt')
```

Description

`fcfwrite(h)` writes a filter coefficient ASCII file to a directory you choose, or your current MATLAB working directory. `h` can be a single filter object or a vector of filter objects. On execution, `fcfwrite` opens the **Export Filter Coefficients to .FCF File** dialog box to let you assign a file name for the output file. You can choose the destination directory within this dialog as well.

The default file name is `untitled.fcf`. When you have Filter Design Toolbox™ software, you can use `fcfwrite(h)` to write filter coefficient files for multirate filters, adaptive filters, and discrete-time filters.

`fcfwrite(h,filename)` writes the filter coefficients and general information to a text file called `filename` in your present MATLAB working directory and opens the file in the MATLAB editor for you to review or modify.

If you do not include a file extension in `filename`, `fcfwrite` adds the extension `fcf` to `filename`.

`fcfwrite(...,'fmt')` writes the filter coefficients in the format specified by the input argument `fmt`. Valid `fmt` values are `hex` for hexadecimal, `dec` for decimal, or `bin` for binary representation of the filter coefficients.

Examples

To demonstrate `fcfwrite`, create a fixed-point IIR filter at the command line, and then write the filter coefficients to a file named `iirfilter.fcf`.

```
d=fdesign.lowpass

d =

                Response: 'Lowpass'
```

```
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.45
                  Fstop: 0.55
                  Apass: 1
                  Astop: 60

hd=butter(d)

hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
       sosMatrix: [13x6 double]
     ScaleValues: [14x1 double]
PersistentMemory: false

set(hd,'arithmetic','fixed');

fcfwrite(hd,'iirfilter.fcf');
```

Here is the output from `fcfwrite` as it appears in the MATLAB editor. Not shown here is the filename — `iirfilter.fcf` as specified and some comments at the top of the file.

```
%
%
% Coefficient Format: Decimal
%
% Discrete-Time IIR Filter (real)
% -------------------------------
% Filter Structure    : Direct-Form II, Second-Order
%                           Sections
% Number of Sections  : 13
% Stable              : Yes
% Linear Phase        : No
```

```
% Arithmetic          : fixed
% Numerator           : s16,13 -> [-4 4)
% Denominator         : s16,14 -> [-2 2)
% Scale Values        : s16,14 -> [-2 2)
% Input               : s16,15 -> [-1 1)
% Section Input       : s16,8 -> [-128 128)
% Section Output      : s16,10 -> [-32 32)
% Output              : s16,10 -> [-32 32)
% State               : s16,15 -> [-1 1)
% Numerator Prod      : s32,28 -> [-8 8)
% Denominator Prod    : s32,29 -> [-4 4)
% Numerator Accum     : s40,28 -> [-2048 2048)
% Denominator Accum   : s40,29 -> [-1024 1024)
% Round Mode          : convergent
% Overflow Mode       : wrap
% Cast Before Sum     : true


SOS matrix:
1  2  1  1  -0.22222900390625  0.88262939453125
1  2  1  1  -0.19903564453125  0.68621826171875
1  2  1  1  -0.18060302734375  0.5303955078125
1  2  1  1  -0.1658935546875   0.40570068359375
1  2  1  1  -0.154052734375    0.305419921875
1  2  1  1  -0.14453125        0.22479248046875
1  2  1  1  -0.136962890625    0.16015625
1  2  1  1  -0.13092041015625  0.10906982421875
1  2  1  1  -0.126220703125    0.06939697265625
1  2  1  1  -0.12274169921875  0.0399169921875
1  2  1  1  -0.12030029296875  0.01947021484375
1  2  1  1  -0.118896484375    0.0074462890625
1  1  0  1  -0.0592041015625   0

Scale Values:
0.41510009765625
0.371826171875
0.33746337890625
```

```
0.3099365234375
0.287841796875
0.27008056640625
0.25579833984375
0.2445068359375
0.23577880859375
0.22930908203125
0.22479248046875
0.22216796875
0.47039794921875
1
```

To write two or more filters out to one file, provide the filters as a vector to `fcfwrite`:

```
fcfwrite([hd hd1 hd2])
```

See Also

`adaptfilt`, `mfilt`

`dfilt` in Signal Processing Toolbox™ documentation

Purpose

Open Filter Design and Analysis Tool

Syntax

```
fdatool
```

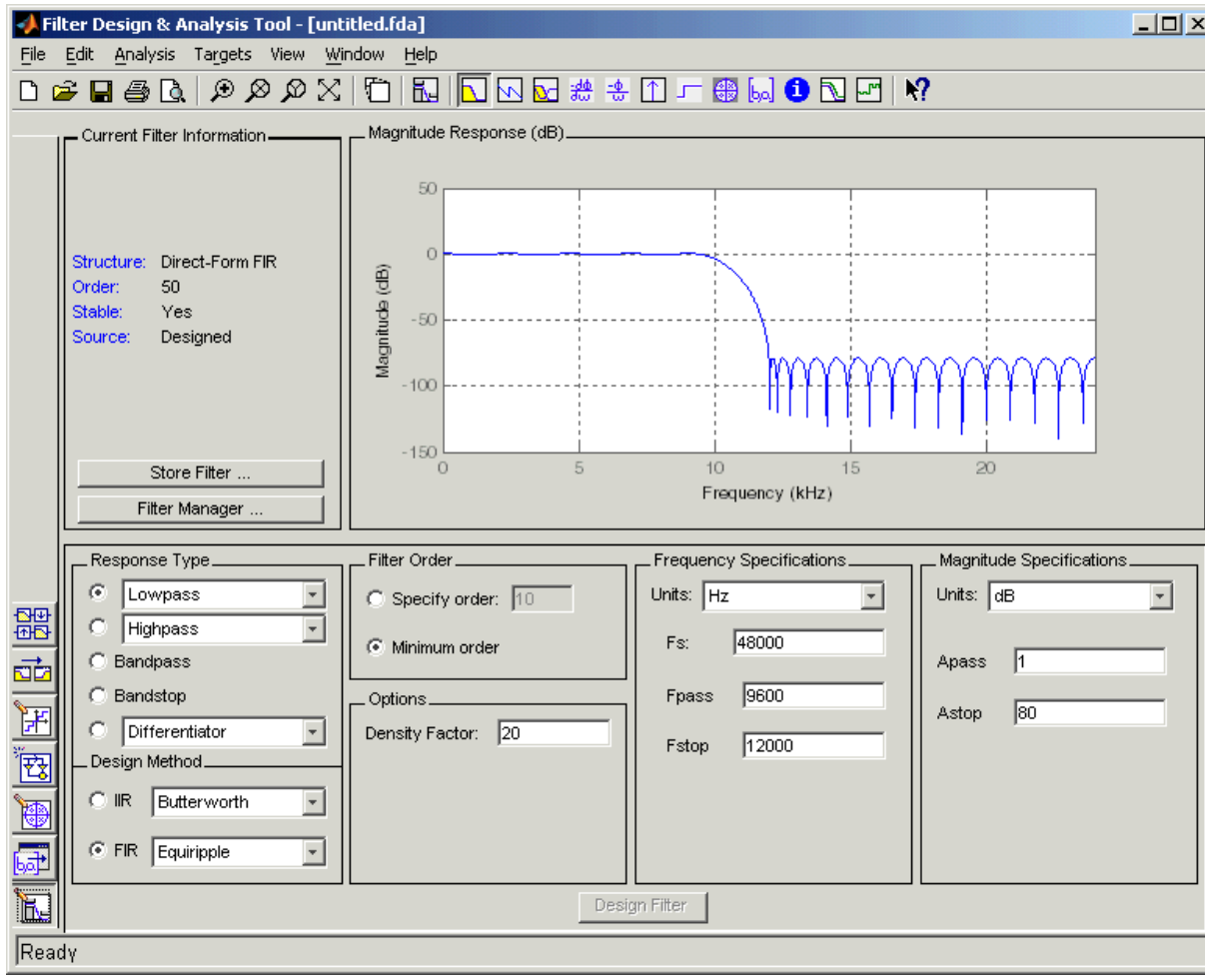
Description

`fdatool` opens the Filter Design and Analysis Tool (FDATool). Use this tool to:

- Design filters
- Quantize filters (with Filter Design Toolbox™ software installed)
- Analyze filters
- Modify existing filter designs
- Create multirate filters (with Filter Design Toolbox software installed)
- Realize Simulink® models of quantized, direct-form, FIR filters (with Filter Design Toolbox software installed)
- Import filters into FDATool
- Perform digital frequency transformations of filters (with Filter Design Toolbox software installed)

Refer to "Using FDATool with Filter Design Toolbox Software" for more information about using the analysis, design, and quantization features of FDATool. For general information about using FDATool, refer to "FDATool: A Filter Design and Analysis GUI" in Signal Processing Toolbox™ documentation.

When you open FDATool and you have Filter Design Toolbox software installed, FDATool incorporates features that are added by Filter Design Toolbox software. With Filter Design Toolbox software installed, FDATool lets you design and analyze quantized filters, as well as convert quantized filters to various filter structures, transform filters, design multirate filters, and realize models of filters.



Use the buttons on the sidebar to configure the design area to use various tools in FDATool.

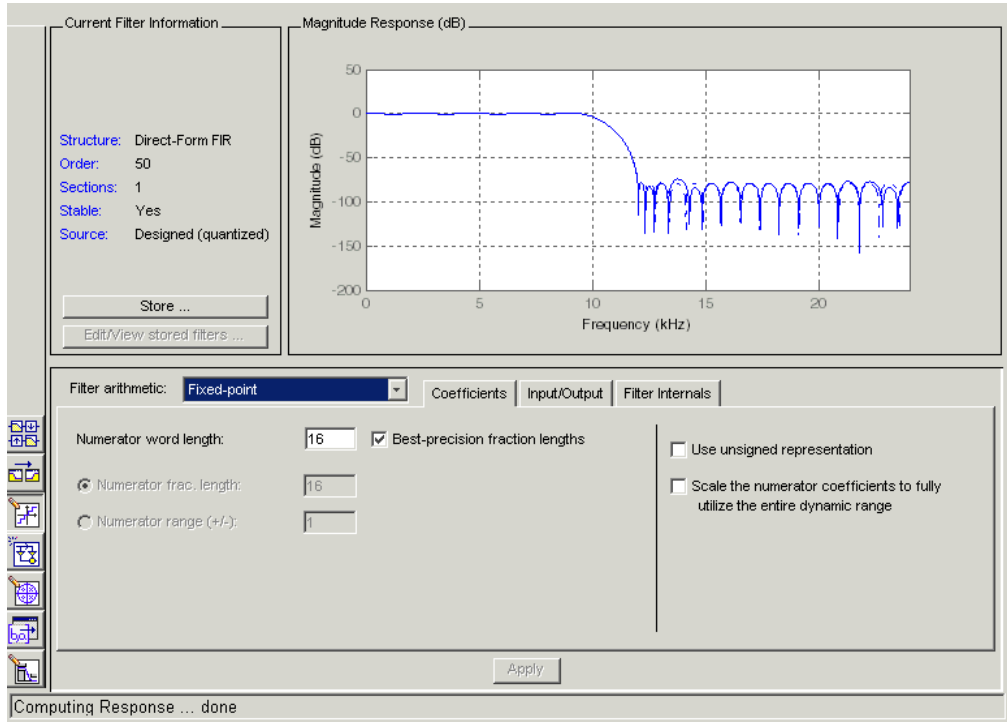
Set Quantization Parameters — provides access to the properties of the quantizers that compose a quantized filter. When you click **Set Quantization Parameters**, you see FDATool displaying the

quantization options at the bottom of the dialog box (the design area), as shown in the figure.

Transform Filter — clicking this button opens the *Frequency Transformations* pane so you can use digital frequency transformations to change the magnitude response of your filter.

Create a multirate filter — clicking this button switches FDATool to multirate filter design mode so you can design interpolators, decimators, and fractional rate change filters.

Realize Model — starting from your quantized, direct-form, FIR filter, clicking this button creates a Simulink model of your filter structure in new model window.



Other options in the menu bar let you convert the filter structure to a new structure, change the order of second-order sections in a filter, or change the scaling applied to the filter, among many possibilities.

Remarks

By incorporating many advanced filter design methods from Filter Design Toolbox software, FDATool provides more design methods than the SPTool Filter Designer.

See Also

`fdatool`, `fvtool`, `sptool` in Signal Processing Toolbox documentation

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 2-544.

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.

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 *Filter Design Toolbox™ Getting Started Guide*. 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
`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.
`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.
`decimator`	`fdesign.decimator` creates an object to specify decimators.
`differentiator`	`fdesign.differentiator` creates an object to specify differentiators.
`fracdelay`	`fdesign.fracdelay` creates an object to specify fractional delay filters.
`halfband`	`fdesign.halfband` creates an object to specify halfband filters.
`highpass`	`fdesign.highpass` creates an object to specify highpass filters.

fdesign Response String	Description
`hilbert`	`fdesign.hilbert` creates an object to specify Hilbert filters.
`interpolator`	`fdesign.interpolator` creates an object to specify interpolators.
`isinclp`	`fdesign.isinclp` creates an object to specify lowpass filters that use inverse-sinc form.
`lowpass`	`fdesign.lowpass` creates an object to specify lowpass filters.
`notch`	`fdesign.notch` creates an object to specify notch filters.
`nyquist`	`fdesign.nyquist` creates an object to specify nyquist filters.
`octave`	`fdesign.octave` creates an object to specify octave and fractional octave filters.
`parameq`	`fdesign.parameq` creates an object to specify parametric equalizer filters.
`peak`	`fdesign.peak` creates an object to specify peak filters.
`rsrc`	`fdesign.rsrc` creates an object to specify rational-factor sample-rate convertors.

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 Fstop 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.

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 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. These variables are applied to the filter design method you choose to design your filter.

For example, when you create a default lowpass filter specification object `d`, `fdesign` sets the passband frequency `Fpass`, the stopband frequency `Fstop`, the stopband attenuation `Astop`, and the passband attenuation `Apass` (ripple in the passband) for `d`:

```
d = fdesign.lowpass

d =

                Response: 'Lowpass'
           Specification: 'Fp,Fst,Ap,Ast'
             Description: {4x1 cell}
    NormalizedFrequency: true
                   Fpass: 0.45
                   Fstop: 0.55
                   Apass: 1
                   Astop: 60
```

However, lowpass design syntax accepts any one of the following `Spec` strings (among others) to define the filter response:

Spec String	Description
`Fp,Fst,Ap,Ast`	Define the filter by specifying the passband cutoff, the stopband cutoff, the ripple in the passband, and the attenuation in the stopband. This is the default string for a lowpass filter.
`N,Fc`	Set the filter order and the cutoff frequency to define the filter.
`N,Fp,Ap`	Set the filter order, passband cutoff frequency, and passband ripple.

Spec String	Description
`N,Fst,Ast`	Define the filter by setting the order, stopband frequency, and stopband attenuation.
`N,Fp,Ap,Ast`	Set the order, passband cutoff frequency, passband ripple, and stopband attenuation.
`N,Fp,Fst,Ap`	Set the filter order, passband cutoff frequency, stopband frequency, and passband ripple.

Other filter object types, such as Nyquist or highpass, accept a different set of strings for `Spec`. Refer to the Help system for details about the strings for each filter type.

One important note is that the `Spec` string you choose controls which design method works for the specifications object.

For the lowpass filter specification object `d` from earlier, you can use `butter`, `cheby1`, `cheby2`, or `ellip` (to name a few) to design a filter. However, if the `Spec` string had been `'n,fp,fst,ap'`, you could only use the `ellip` design method to design your filter.

When you implement this lowpass filter `hd` using a filter design method such as Butterworth (the `butter` design function), the constraints in `fp`, `fst`, `ap`, and `ast` (the default string and filter specification) define the response of the final minimum-order lowpass filter:

```
hd = design(d,'butter')

hd =

          FilterStructure: 'Direct-Form II, Second-Order Sections'
               Arithmetic: 'double'
                sosMatrix: [13x6 double]
              ScaleValues: [14x1 double]
         PersistentMemory: false
```

FVTool shows that `hd` is a lowpass filter that meets the design specification.

`d = fdesign.response(...,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.response(...,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 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. The following methods are available for filter specification objects, but all methods do not apply to all object types. Also, the specification string you use to define the object changes the algorithms available to design a filter. Enter `doc butter`, for example, to get more information about using the Butterworth design method with your filter specification object.

Design Function	Description
`butter`	Implement a Butterworth filter resulting in an SOS filter with direct-form II structure
`cheby1`	Implement a Chebyshev Type I filter, resulting in a direct-form II second-order filter
`cheby2`	Implement a Chebyshev Type II filter, resulting in an SOS filter with direct-form II structure

Design Function	Description
`ellip`	Implement an elliptic filter resulting in an SOS filter with direct-form II structure
`equiripple`	Implement an equiripple filter
`firls`	Implement a least-squares filter
`kaiserwin`	Implement a filter that uses a Kaiser window
lagrange	Implement a Lagrange fractional delay filter
`multistage`	Implement a multistage filter

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

These examples show a few default filter objects constructed from the MATLAB command prompt, and how to design a Butterworth filter.

Example 1

Halfband filter specification object with filter order and stopband attenuation provided as input arguments. Add the `linear magunits` option so you specify the attenuation in decimal — 0.0001.

```
n = 80;
ast = 1e-4;
fs = 48000
d=fdesign.halfband('n,ast',n,ast,fs,'linear')

d =

               Response: [1x51 char]
          Specification: 'N,Ast'
            Description: {2x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
            FilterOrder: 80
                  Astop: 80

d.description

ans =

    'Filter Order'
    'Stopband Attenuation (dB)'
```

Example 2

Interpolator filter specification object

```
d = fdesign.interpolator % Specifications object.
```

```
d =

               Response: 'Minimum-order halfband'
          Specification: 'TW,Ast'
            Description: {2x1 cell}
    InterpolationFactor: 2
    NormalizedFrequency: true
                     Fs: 'Normalized'
        TransitionWidth: 0.1000
                  Astop: 80


d.Description

ans =

    'Transition Width'
    'Stopband Attenuation (dB)'
```

Example 3

Highpass filter specification object

```
d=fdesign.highpass % Creates specifications object.

d =

               Response: 'Minimum-order highpass'
      Specification    : 'Fst,Fp,Ast,Ap'
            Description: {4x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
                  Fstop: 0.4500
                  Fpass: 0.5500
                  Astop: 60
                  Apass: 1
```

```
d.Description

ans =

    'Stopband Frequency'
    'Passband Frequency'
    'Stopband Attenuation (dB)'
    'Passband Ripple (dB)'
```

Notice the correspondence between the properties `Specification` and `Description` — in `Description` you see in words the definitions of the variables shown in `Specification`.

Example 4

Only the Kaiser window-based design method applies to default Nyquist filter objects.

Lowpass Butterworth filter specification object

Use a filter specification object to construct a lowpass Butterworth filter with default `Specification` `fp,fst,ap,ast` — the edge frequencies of the passband and stopband, the attenuation in the passband, and the attenuation in the stopband. Start by creating the specifications object `d` and providing the filter order and cutoff frequency values.

```
d = fdesign.lowpass(0.4,0.5,1,80);
d

d =

               Response: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
                  Fpass: 0.4000
                  Fstop: 0.5000
                  Apass: 1
```

```
                    Astop: 80
```

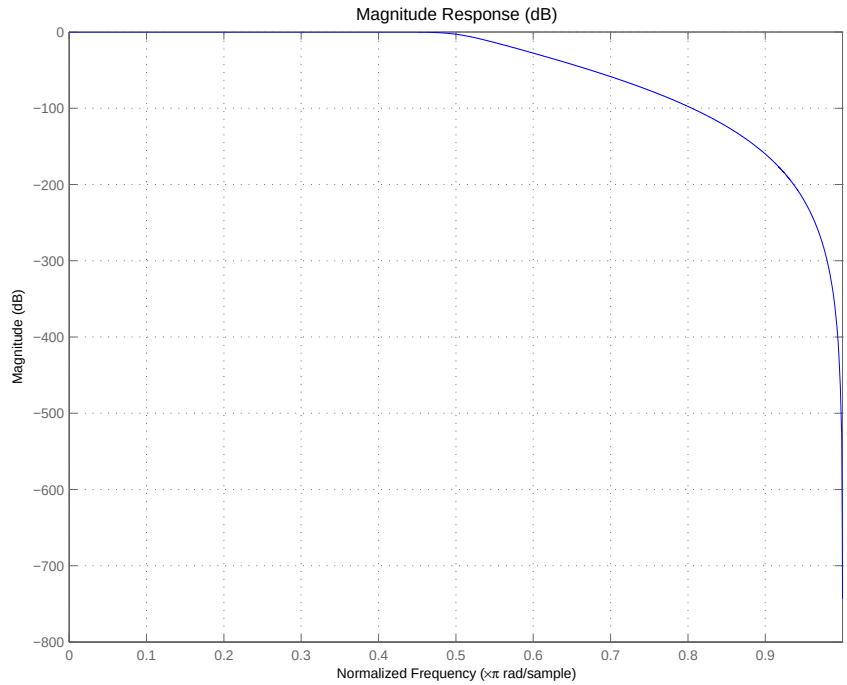
Determine which design methods apply to `d`.

```
designmethods(d)


Design Methods for class fdesign.lowpass:


butter
cheby1
cheby2
ellip
```

Now 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 had a default Nyquist filter specification object `d`

```
d = fdesign.nyquist
```

you could find out which design methods apply to `d` by entering

```
designmethods(d)

Design methods for class fdesign.nyquist:

kaiserwin
```

See Also

butter, cheby1, cheby2, designmethods, designopts, ellip, equiripple, fdatool, fdesign.bandpass, fdesign.bandstop, fdesign.decimator, fdesign.halfband, fdesign.highpass, fdesign.interpolator, fdesign.lowpass, fdesign.nyquist, fdesign.rsrc, firls, fvtool, kaiserwin, lagrange, multistage, setspecs, validstructures

fdesign.arbmag

Purpose

Arbitrary response magnitude filter specification object

Syntax

```
d = fdesign.arbmag
d = fdesign.arbmag(specification)
d = fdesign.arbmag(specification,specvalue1,specvalue2,...)
d = fdesign.arbmag(specvalue1,specvalue2,specvalue3)
d = fdesign.arbmag(...,fs)
```

Description

`d = fdesign.arbmag` constructs an arbitrary magnitude filter designer `d`.

`d = fdesign.arbmag(specification)` initializes the `Specification` property for specifications object `d` to the string in `specification`. The input argument `specification` must be one of the strings shown in the following table. Specification strings are not case sensitive.

Specification String	Description of Resulting Filter
`n,f,a`	Single band design (default). FIR and IIR (`n` is the order for both numerator and denominator).
`n,b,f,a`	Multiband design where `b` defines the number of bands.
`nb,na,f,a`	IIR single band design.
`nb,na,b,f,a`	IIR multiband design where `b` defines the number of bands

The following table describes the arguments in the `specification` strings.

Argument	Description
`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.
`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.

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

Specifying f and a

`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 example creates a filter with two passbands (`b = 4`) and shows how `f` and `a` are related. This example is for illustration only. It is not an actual filter.

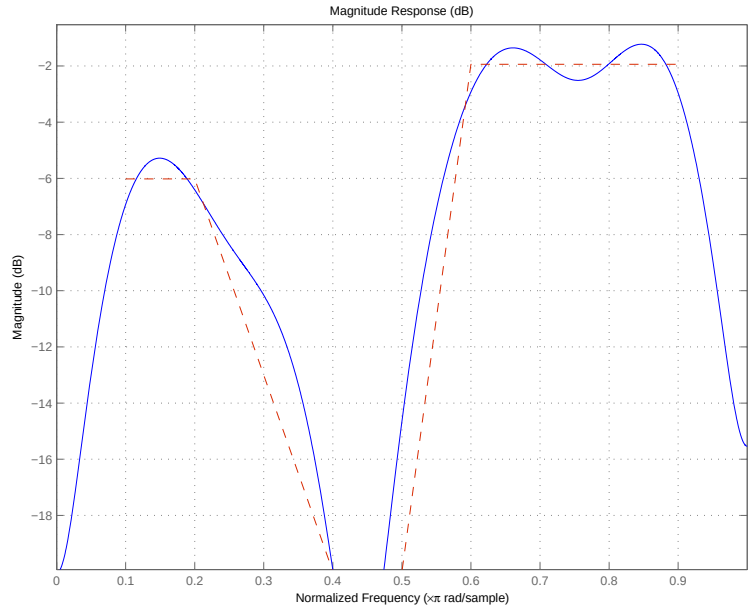
Define the frequency vector `f` as `[0 0.1 0.2 0.4 0.5 0.6 0.9 1.0]`

Define the response vector `a` as `[0 0.5 0.5 0.1 0.1 0.8 0.8 0]`

These specifications connect `f` and `a` as shown in the following table.

f (Normalized Frequency)	a (Response Desired at f)
0	0
0.1	0.5
0.2	0.5
0.4	0.1
0.5	0.1
0.6	0.8
0.9	0.8
1.0	0.0

A response with two passband—one roughly between 0.1 and 0.2 and the second between 0.6 and 0.9—results from the mapping between `f` and `a`. A filter that used `f` and `a` might look like the one shown in the following figure.



Different specification types often have different design methods available. Use `designmethods(d)` to get a list of design methods available for a given specification string and specifications object.

`d = fdesign.arbmag(specification,specvalue1,specvalue2,...)` initializes the filter specification object specifications with `specvalue1`, `specvalue2`, and so on. Use `get(d,'description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ... `spec`*n*.

`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

These three examples introduce designing filters that have arbitrary filter response shapes. In this first example, use `fdesign.arbmag` to design a single-band, arbitrary-magnitude FIR filter. The design process uses the default design method for the `n,f,a` specification, as shown in the following code:

```
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; % Optical absorption of atomic Rubidium 87 vapor.
d = fdesign.arbmag(n,f,a);
hd1 = design(d,'freqsamp');
```

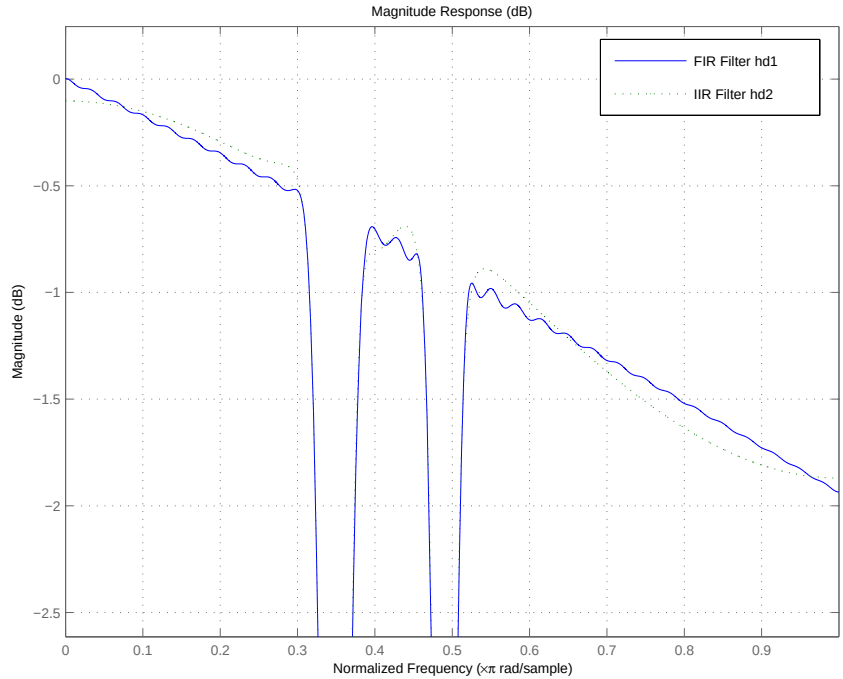
Next, design a single-band, arbitrary-magnitude IIR filter and display the magnitude response in FVTool. Use `f` and `a` from the previous example as input arguments for this case. Display the response from the previous example in FVTool as well, because the FIR and IIR filters are similar.

To demonstrate that the same specification generates both FIR and IIR filters, use the same specifications object `d`, but change the design method to `iirlpnorm`.

```
d.filterorder=10

d =

                Response: 'Arbitrary Magnitude'
           Specification: 'N,F,A'
             Description: {'Filter Order';'Frequency Vector';'
                            Amplitude Vector'}
     NormalizedFrequency: true
             FilterOrder: 10
             Frequencies: [1x100 double]
              Amplitudes: [1x100 double]
```

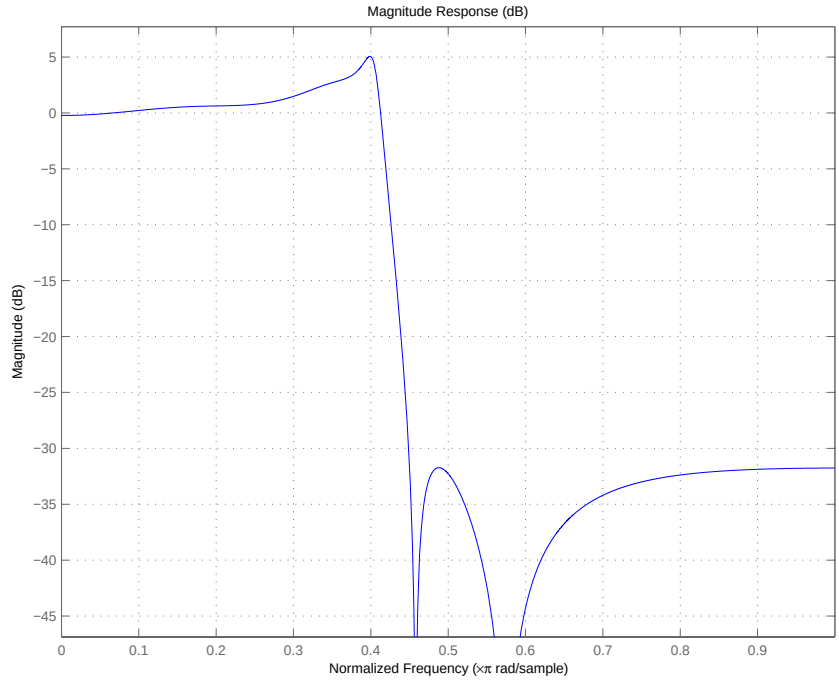
```
hd2=design(d,'iirlpnorm') % Design an IIR filter from the same object.

hd2 =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [5x6 double]
            ScaleValues: [0.85714867585342;1;1;1;1;1]
        PersistentMemory: false

fvtool(hd1,hd2)
```

FVTool returns the following plot for the filters.



For the third example, design a multiband filter for noise shaping when you are simulating the Rayleigh fading phenomenon in a wireless communications channel. This example uses the default design method for `fdesign.arbmag` specifications objects with the `nb,na,nbands` specification—`iirlpnorm`.

```
nb = 4;     % Numerator order.
na = 6;     % Denominator order.
nbands = 2; % Number of filter bands.
f1 = 0:0.01:0.4; % Frequency vector values.
a1 = 1.0 ./ (1 - (f1./0.42).^2).^0.25; % Amplitude values.
f2 = [.45 1];
a2 = [0 0];
d = fdesign.arbmag('nb,na,b,f,a',nb,na,nbands,f1,a1,f2,a2);
design(d); % Starts FVTool to display the filter response.
```



The filter response shows the characteristic shape for noise shaping—increasing gain with increasing frequency in the passband, and a narrow transition region.

See Also

`design`, `designopts`, `fdesign`, `setspecs`

fdesign.arbmagnphase

Purpose

Arbitrary response magnitude and phase filter specification object

Syntax

```
d = fdesign.arbmagnphase
d = fdesign.arbmagnphase(specification)
d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,
   ...)
d = fdesign.arbmagnphase(specvalue1,specvalue2,specvalue3)
d = fdesign.arbmagnphase(...,fs)
```

Description

`d = fdesign.arbmagnphase` constructs an arbitrary magnitude filter specification object `d`.

`d = fdesign.arbmagnphase(specification)` initializes the `Specification` property for specifications object `d` to the string in `specification`. The input argument `specification` must be one of the strings shown in the following table. Specification strings are not case sensitive.

Specification String	Description of Resulting Filter
`n,f,h`	Single band design (default). FIR and IIR (`n` is the order for both numerator and denominator).
`n,b,f,h`	FIR multiband design where `b` defines the number of bands.
`nb,na,f,h`	IIR single band design.

The following table describes the arguments in the strings.

Argument	Description
`b`	Number of bands in the multiband filter.
`f`	Frequency vector. Frequency values specified in `f` indicate locations where you provide specific filter response amplitudes. When you provide `f` you must also provide `h` which contains the response values.

Argument	Description
h	Complex frequency response values.
n	Filter order for FIR filters and the numerator and denominator orders for IIR filters (when not specified by nb and na).
nb	Numerator order for IIR filters.
na	Denominator order for IIR filter designs.

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

Specifying f and h

f and h 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 h. This example creates a filter with two passbands (b = 4) and shows how f and h are related. This example is for illustration only. It is not an actual filter.

Define the frequency vector f as `[0 0.1 0.2 0.4 0.5 0.6 0.9 1.0]`

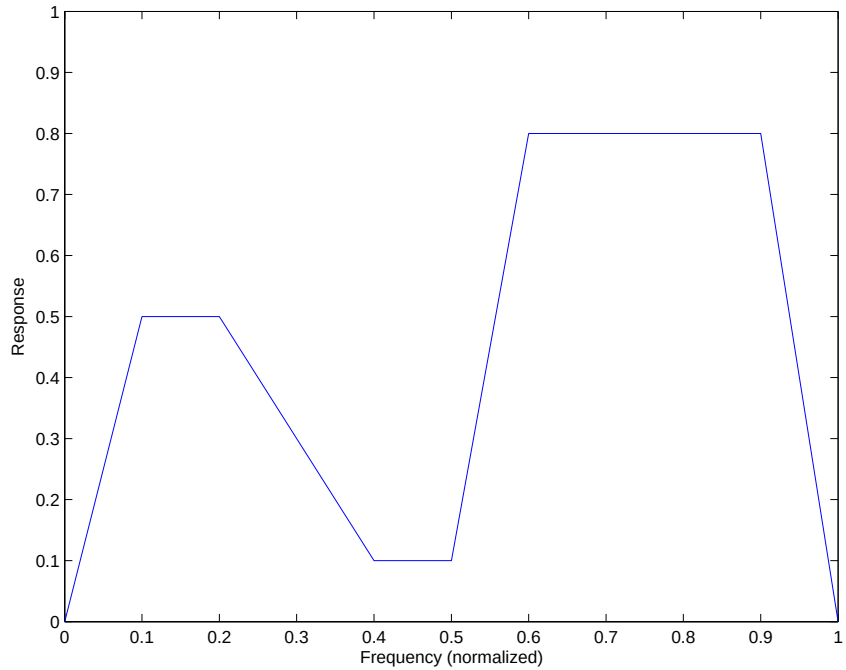
Define the response vector h as `[0 0.5 0.5 0.1 0.1 0.8 0.8 0]`

These specifications connectf and h as shown in the following table.

f (Normalized Frequency)	h (Response Desired at f)
0	0
0.1	0.5
0.2	0.5
0.4	0.1
0.5	0.1
0.6	0.8

f (Normalized Frequency)	h (Response Desired at f)
0.9	0.8
1.0	0.0

A response with two passbands—one roughly between 0.1 and 0.2 and the second between 0.6 and 0.9—results from the mapping between `f` and `h`. Plotting `f` and `h` yields the following figure that resembles a filter with two passbands.



The second example in Examples shows this plot in more detail with a complex filter response for `h`. In the example, `h` uses complex values for the response.

Different specification types often have different design methods available. Use `designmethods(d)` to get a list of design methods available for a given specification string and specifications object.

`d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,...)` initializes the filter specification object with `specvalue1`, `specvalue2`, and so on. Use `get(d,'description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ...spec*n*.

`d = fdesign.arbmagnphase(specvalue1,specvalue2,specvalue3)` uses the default specification string `n,f,h`, setting the filter order, filter frequency vector, and the complex frequency response vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

`d = fdesign.arbmagnphase(...,fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `fs`.

Examples

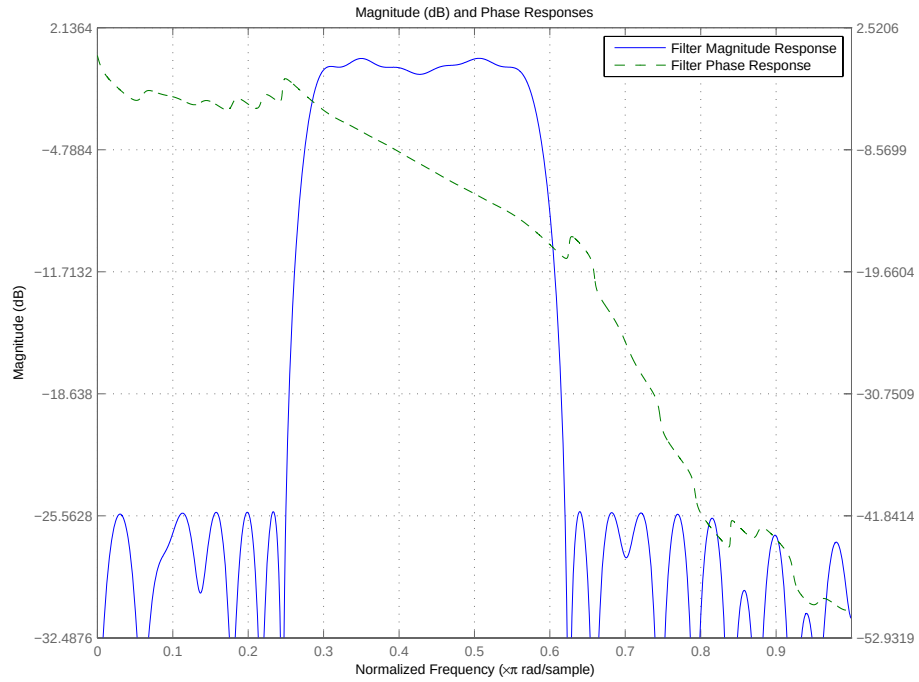
Use `fdesign.arbmagnphase` to model a complex analog filter:

```
d=fdesign.arbmagnphase('n,f,h',100); % N=100, f and h set to defaults.
design(d,'freqsamp');
```

For a more complex example, design a bandpass filter with low group delay by specifying the desired delay and using `f` and `h` to define the filter bands.

```
n = 50;     % Group delay of a linear phase filter would be 25.
gd = 12;    % Set the desired group delay for the filter.
f1=linspace(0,.25,30); % Define the first stopband frequencies.
f2=linspace(.3,.56,40);% Define the passband frequencies.
f3=linspace(.62,1,30); % Define the second stopband frequencies.
h1 = zeros(size(f1));  % Specify the filter response at the freqs in f1.
h2 = exp(-j*pi*gd*f2); % Specify the filter response at the freqs in f2.
h3 = zeros(size(f3));  % Specify the response at the freqs in f3.
d=fdesign.arbmagnphase('n,b,f,h',50,3,f1,h1,f2,h2,f3,h3);
design(d,'equiripple')
```

In the following figure, displaying the filter in FVTool shows both the magnitude response and the nearly linear phase.



See Also

`fdesign`, `design`, `designmethods`, `setspecs`

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.

Using `fdesign.bandpass` with a design method generates a `dfilt` object.

`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.

- `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,fp1,fp2,ap`

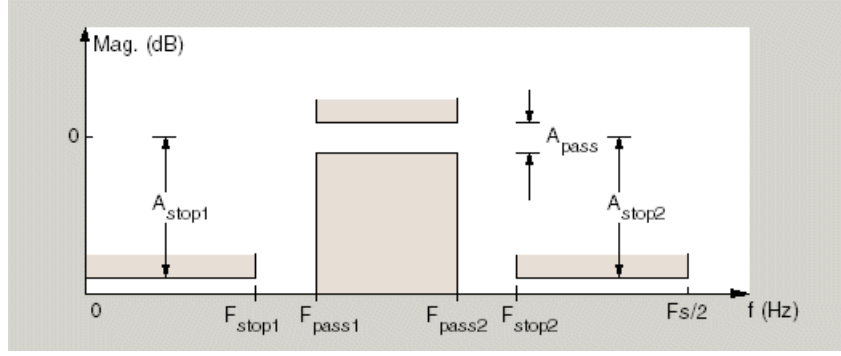
- `n,fp1,fp2,ast1,ap,ast2`
- `n,fst1,fp1,fp2,fst2`
- `n,fst1,fp1,fp2,fst2,ap`
- `n,fst1,fst2,ast`
- `nb,na,fst1,fp1,fp2,fst2`

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.
- `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 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (FIR filters)
- `fc2` — cutoff frequency for the point 3 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 method applies to an object and its specification string.

`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`, an object 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.

`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

These examples show how to construct a bandpass filter specification object. First, create a default specifications object without using input arguments.

```
d = fdesign.bandpass
d =

                Response: 'Minimum-order bandpass'
           Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
             Description: {7x1 cell}
```

```
    NormalizedFrequency: true
                Fstop1: 0.3500
                Fpass1: 0.4500
                Fpass2: 0.5500
                Fstop2: 0.6500
                Astop1: 60
                 Apass: 1
                Astop2: 60
```

Now, pass the filter specifications that correspond to the default `Specification` — `fst1,fp1,fp2,fst2,ast1,ap,ast2` — without specifying the `Specification` string. This example adds `fs` as the final input argument to specify the sampling frequency of 48 Hz.

```
d = fdesign.bandpass(10, 12, 14, 16, 80, .5, 60, 48)
d =

               Response: 'Minimum-order bandpass'
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7x1 cell}
    NormalizedFrequency: false
                     Fs: 48
                Fstop1: 10
                Fpass1: 12
                Fpass2: 14
                Fstop2: 16
                Astop1: 80
                 Apass: 0.5000
        Astop2: 60
```

Next create a specifications object by passing a specification type string 'n,fc1,fc2' — the resulting object uses default values for `n`, `fc1`, and `fc2`.

```
d = fdesign.bandpass('n,fc1,fc2')
d =

                Response: 'Bandpass with cutoff'
```

```
            Specification: 'N,Fc1,Fc2'
              Description: {3x1 cell}
        NormalizedFrequency: true
              FilterOrder: 10
                 Fcutoff1: 0.4000
                 Fcutoff2: 0.6000
```

Create the same filter, passing the specification values to the object rather than accepting the default values for `n`, `fc1`, and `fc2`. You can include the sampling frequency `fs` as the final input argument, and that you specify the cutoff frequencies in Hz since `fs` is in Hz.

```
d = fdesign.bandpass('n,fc1,fc2', 10, 9600, 14400, 48000)
d =

                 Response: 'Bandpass with cutoff'
            Specification: 'N,Fc1,Fc2'
              Description: {3x1 cell}
        NormalizedFrequency: false
                       Fs: 48000
              FilterOrder: 10
                 Fcutoff1: 9600
                 Fcutoff2: 14400
```

See Also

`fdesign`, `fdesign.bandstop`, `fdesign.highpass`, `fdesign.lowpass`

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`.

Using `fdesign.bandstop` with a design method generates a `dfilt` object.

`d = fdesign.bandstop(spec)` constructs object `d` and sets its 'Specification' 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.

- `fp1,fst1,fst2,fp2,ap1,ast,ap2` (defaultspec)
- `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,fp1,fp2,ap`
- `n,fp1,fp2,ap,ast`

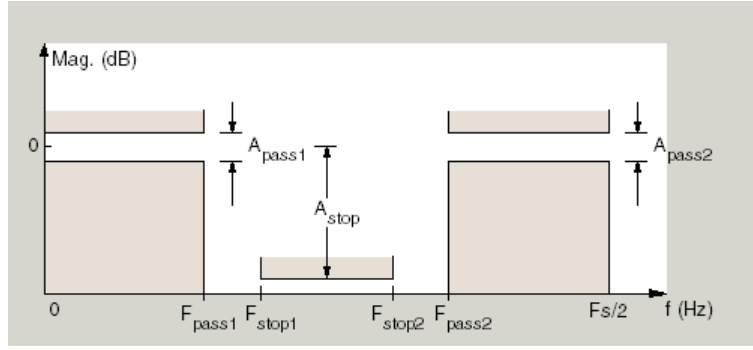
- `n,fp1,fst1,fst2,fp2`
- `n,fp1,fst1,fst2,fp2,ap`
- `n,fst1,fst2,ast`
- `nb,na,fp1,fst1,fst2,fp2`

The string entries are defined as follows:

- `ap` — amount of ripple allowed in the passband in decibels (the default units). Also called Apass.
- `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.
- `f3dB1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units.
- `f3dB2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units.
- `fp1` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass1.
- `fp2` — frequency at the end of the pass band. Specified in normalized frequency units. Also called Fpass2.
- `fst1` — frequency at the end of the first stop band. Specified in normalized frequency units. Also called Fstop1.
- `fst2` — frequency at the start of the second stop band. Specified in normalized frequency units. 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 method applies to an object and its specification string.

`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 `fpass1,fstop1,fstop2,fpass2,apass1,astop,apass2`, 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 to use. In this case, 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.

Examples

These examples show how to construct a bandpass filter specification object. First, create a default specifications object without using input arguments.

```
d = fdesign.bandstop
d =

               Response: 'Minimum-order bandstop'
            Description: {7x1 cell}
          Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
    NormalizedFrequency: true
                Fpass1: 0.3500
                Fstop1: 0.4500
                Fstop2: 0.5500
                Fpass2: 0.6500
                Apass1: 1
                 Astop: 60
                Apass2: 1
```

Now create an object by passing a specification type string 'n,fc1,fc2' — the resulting object uses default values for n, fc1, and fc2.

```
d=fdesign.bandstop('n,f3dB1,f3dB2')

d =

               Response: 'Bandstop with cutoff'
          Specification: 'N,F3dB1,F3dB2'
            Description: {3x1 cell}
    NormalizedFrequency: true
            FilterOrder: 10
               Fcutoff1: 0.4000
               Fcutoff2: 0.6000

designmethods(d)

Design Methods for class fdesign.bandstop:

butter
cheby1
cheby2
ellip
```

Create another bandstop filter, passing the specification values to the object rather than accepting the default values for n, f3db1, and fc2. You can add fs as the final input argument to specify the sampling frequency of 48 kHz.

```
d = fdesign.bandstop('n,f3db1,f3db2', 10, 9600, ...
  14400, 48000)

d =

               Response: 'Bandstop with cutoff'
          Specification: 'N,F3dB1,F3dB2'
            Description: {3x1 cell}
    NormalizedFrequency: false
```

```
                    Fs: 48000
           FilterOrder: 10
              Fcutoff1: 9600
              Fcutoff2: 14400
```

For this bandstop filter, pass the filter specifications that correspond to the default `Specification` — `fp1`,`fst1`,`fst2`,`fp2`,`ap1`,`ast`,`ap2`.

```
d = fdesign.bandstop(0.3,0.4,0.6,0.7,0.5,60,1)

d =

               Response: 'Minimum-order bandstop'
          Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
            Description: {7x1 cell}
    NormalizedFrequency: true
                  Fpass1: 0.3000
                  Fstop1: 0.4000
                  Fstop2: 0.6000
                  Fpass2: 0.7000
                  Apass1: 0.5000
                   Astop: 60
                  Apass2: 1
```

And for the final example, pass the magnitude specifications in squared units, using the `magunits` option `squared`.

```
d = fdesign.bandstop(0.4,0.5,0.6,0.7,0.98,...
0.01,0.99,'squared')
d =

               Response: 'Minimum-order bandstop'
          Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
            Description: {7x1 cell}
    NormalizedFrequency: true
                  Fpass1: 0.4000
                  Fstop1: 0.5000
                  Fstop2: 0.6000
```

```
Fpass2: 0.7000
Apass1: 0.0877
 Astop: 20
Apass2: 0.0436
```

See Also

`fdesign`, `fdesign.bandpass`, `fdesign.highpass`, `fdesign.lowpass`

fdesign.ciccomp

Purpose

CIC compensator filter specification object

Syntax

```
h = fdesign.ciccomp
h = fdesign.ciccomp(d,nsections)
h = fdesign.ciccomp(...,spec)
h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)
```

Description

`h = fdesign.ciccomp` constructs a CIC compensator specifications object `d`, applying default values for the properties Fpass, Fstop, Apass, and Astop. In this syntax, the filter has two sections and the differential delay is 1.

Using `fdesign.ciccomp` with a design method creates a `dfilt` object, a single-rate discrete-time filter.

`h = fdesign.ciccomp(d,nsections)` constructs a CIC compensator specifications object with the filter differential delay set to `d` and the number of sections in the filter set to `nsections`. By default, `d` and `nsections` are 1 and 2 if you omit them as input arguments.

`h = fdesign.ciccomp(...,spec)` constructs a CIC Compensator specifications object 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 in the list below. The strings are not case sensitive.

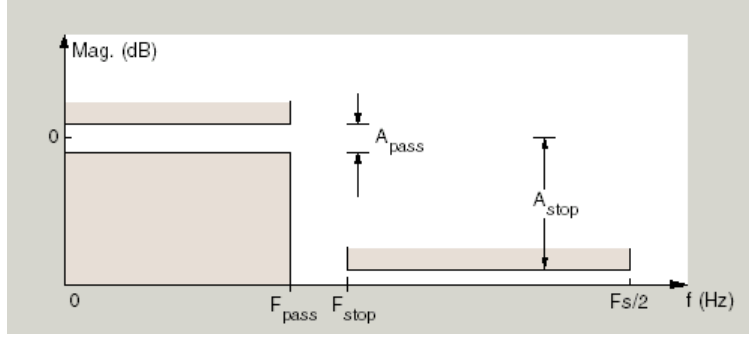
- `fp,fst,ap,ast` (default `spec`)
- `n,fc,ap,ast`
- `n,fp,ap,ast`
- `n,fp,fst`
- `n,fst,ap,ast`

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.
- `fc` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- `fp` — frequency at the end of the pass band. Specified in normalized frequency units. Also called Fpass.
- `fst` — frequency at the start of the stop band. Specified in normalized frequency units. Also called Fstop.
- `n` — filter order.

In graphic form, the filter specifications look like this:



Regions between specification values like `fp` and `fst` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a CIC compensator specifications object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string.

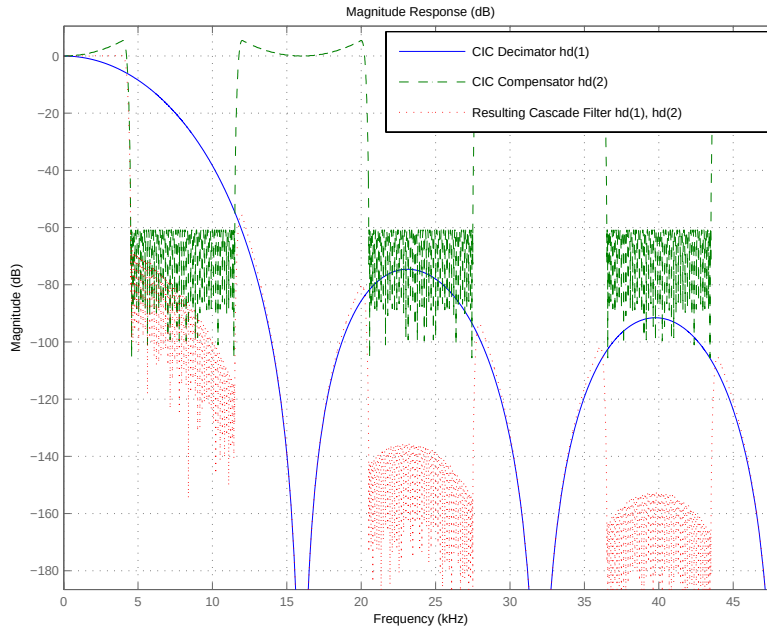
`h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)` constructs an object and sets the specifications in the order they are specified in the `spec` input when you construct the object.

Designing CIC Compensators

Typically, when they develop filters, designers want flat passbands and transition regions that are as narrow as possible. CIC filters present a (sin*x*/*x*) profile in the passband and relatively wide transitions.

To compensate for this fall off in the passband, and to try to reduce the width of the transition region, you can use a CIC compensator filter that demonstrates an (*x*/sin*x*) profile in the passband. `fdesign.ciccomp` is specifically tailored to designing CIC compensators.

Here is a plot of a CIC filter and a compensator for that filter. The example that produces these filters follows the plot.



Given a CIC filter, how do you design a compensator for that filter? CIC compensators share three defining properties with the CIC filter —

differential delay, `d`; number of sections, `numberofsections`; and the usable passband frequency, `Fpass`.

By taking the number of sections, passband, and differential delay from your CIC filter and using them in the definition of the CIC compensator, the resulting compensator filter effectively corrects for the passband droop of the CIC filter, and narrows the transition region.

As a demonstration of this concept, this example creates a CIC decimator and its compensator.

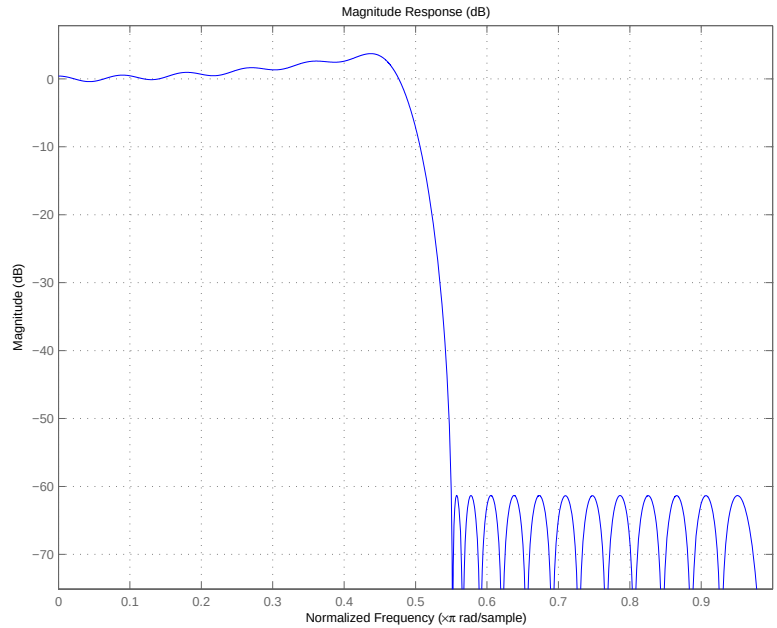
```
fs = 96e3;   % Input sampling frequency.
fpass = 4e3; % Frequency band of interest.
m = 6;  % Decimation factor.
hcic = design(fdesign.decimator(m,'cic',1,fpass,60,fs));
hd = cascade(dfilt.scalar(1/gain(hcic)),hcic);
hd(2) = design(fdesign.ciccomp(hcic.differentialdelay, ...
            hcic.numberofsections,fpass,4.5e3,.1,60,fs/m));
fvtool(hd(1),hd(2),...
cascade(hd(1),hd(2)),'Fs',[96e3 96e3/m 96e3])
```

You see the results in the preceding plot.

Examples

Designed to compensate for the roll-off inherent in CIC filters, CIC compensators can improve the performance of your CIC design. This example designs a compensator `d` with five sections and a differential delay equal to one. The plot displayed after the code shows the increasing gain in the passband that is characteristic of CIC compensators, to overcome the droop in the CIC filter passband. Ideally, cascading the CIC compensator with the CIC filter results in a lowpass filter with flat passband response and narrow transition region.

```
h = fdesign.ciccomp;
set(h, 'NumberOfSections', 5, 'DifferentialDelay', 1);
hd = equiripple(h);
fvtool(hd);
```



This compensator would work for a decimator or interpolator that had differential delay of 1 and 5 sections.

See Also

`fdesign.decimator`, `fdesign.interpolator`

Purpose

Decimator filter specification object

Syntax

```
d = fdesign.decimator(m)
d = fdesign.decimator(m,design)
d = fdesign.decimator(m,design,spec)
d = fdesign.decimator(...,spec,specvalue1,specvalue2,...)
d = fdesign.decimator(...,fs)
d = fdesign.decimator(...,magunits)
```

Description

`d = fdesign.decimator(m)` constructs a decimating filter specification object `d`, applying default values for the properties `fp`, `fst`, `ap`, and `ast` and using the default design, Nyquist. Specify `m`, the decimation factor, as an integer. When you omit the input argument `m`, `fdesign.decimator` sets the decimation factor `m` to 2.

Using `fdesign.decimator` with a design method generates an `mfilt` object.

`d = fdesign.decimator(m,design)` constructs a decimator with the decimation factor `m` and the design type you specify in `design`. By using the `design` input argument, you can choose the sort of filter that results from using the decimator specifications object. `design` accepts the following strings that define the filter response.

design String	Description
`arbmag`	Sets the design for the decimator specifications object to Arbitrary Magnitude.
`arbmagnphase`	Sets the design for the decimator specifications object to Arbitrary Magnitude and Phase.
`bandpass`	Sets the design for the decimator specifications object to bandpass.
`bandstop`	Sets the design for the decimator specifications object to bandstop.
`cic`	Sets the design for the decimator specifications object to CIC filter.

design String	Description
`ciccomp`	Sets the design for the decimator specifications object to CIC compensator.
`halfband`	Sets the design for the decimator specifications object to halfband.
`highpass`	Sets the design for the decimator specifications object to highpass.
`isinclp`	Sets the design for the decimator specifications object to inverse-sinc lowpass.
`lowpass`	Sets the design for the decimator specifications object to lowpass.
`nyquist`	Sets the design for the decimator specifications object to Nyquist.

Notice the entries in the first column. They match the design method names. However, when you create your specifications object, the `Response` property contains the full name of the response, such as `CIC Compensator` or `Inverse-Sinc Lowpass`, rather than the shorter method names `isinclp` or `ciccomp`. So, when designing a new filter object, use the design String name shown in the left column of the table. To change the `Response` property value for an existing specifications object, use the full response name.

`d = fdesign.decimator(m,`*`design`*`,`*`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`* depend on the design type of the specifications object.

When you add the `spec` input argument, you must also add the `design` input argument.

Because you are designing multirate filters, the specification strings available are not the same as the specifications for designing single-rate

filters with such design methods as `fdesign.lowpass`. The strings are not case sensitive.

The decimation factor `m` is not in the specification strings. Various design types provide different specifications, as shown in this table.

Design Type	Valid Specification Strings
`Arbitrary Magnitude`	• `n,f,a` (default string) • `n,b,f,a`
`Arbitrary Magnitude and Phase`	• `n,f,h` (default string) • `n,b,f,h`
`Bandpass`	• `fst1,fp1,fp2,fst2,ast1,ap,ast2` (default string) • `n,fc1,fc2` • `n,fst1,fp1,fp2,fst2`
`Bandstop`	• `n,fc1,fc2` • `n,fp1,fst1,fst2,fp2` • `fp1,fst1,fst2,fp2,ap1,ast,ap2` (default string)
`CIC`	• `fp,ast` (default and only string)
`CIC Compensator`	• `fp,fst,ap,ast` (default string) • `n,fc,ap,ast` • `n,fp,ap,ast` • `n,fp,fst` • `n,fst,ap,ast`

Design Type	Valid Specification Strings
`Halfband`	• `tw,ast` (default string) • `n,tw` • `n` • `n,ast`
`Highpass`	• `fst,fp,ast,ap` (default string) • `n,fc` • `n,fc,ast,ap` • `n,fp,ast,ap` • `n,fst,fp,ap` • `n,fst,fp,ast` • `n,fst,ast,ap` • `n,fst,fp`
`Inverse-Sinc Lowpass`	• `fp,fst,ap,ast` (default string) • `n,fc,ap,ast` • `n,fst,ap,ast` • `n,fp,ap,ast` • `n,fp,fst`

Design Type	Valid Specification Strings
`Lowpass`	• `fp,fst,ap,ast` (default string) • `n,fc` • `n,fc,ap,ast` • `n,fp,ap,ast` • `n,fp,fst` • `n,fp,fst,ap` • `n,fp,fst,ast` • `n,fst,ap,ast`
`Nyquist`	• `tw,ast` (default string) • `n,tw` • `n` • `n,ast`

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.
- `ap` — amount of ripple allowed in the pass band 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. Bandpass and bandstop filters use this option.
- `ap2` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass2. Bandpass and bandstop filters use this option.
- `ast` — attenuation in the first stop band in decibels (the default units). Also called Astop.

- `ast1` — attenuation in the first stop band in decibels (the default units). Also called Astop1. Bandpass and bandstop filters use this option.
- `ast2` — attenuation in the first stop band in decibels (the default units). Also called Astop2. Bandpass and bandstop filters use this option.
- `b` — number of bands in the multiband filter
- `f` — frequency vector. Frequency values in `f` specify locations where you provide specific filter response amplitudes. When you provide `f` you must also provide `a`.
- `fc1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. Bandpass and bandstop filters use this option.
- `fc2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. Bandpass and bandstop filters use this option.
- `fp1` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass1. Bandpass and bandstop filters use this option.
- `fp2` — frequency at the end of the pass band. Specified in normalized frequency units. Also called Fpass2. Bandpass and bandstop filters use this option.
- `fst1` — frequency at the end of the first stop band. Specified in normalized frequency units. Also called Fstop1. Bandpass and bandstop filters use this option.
- `fst2` — frequency at the start of the second stop band. Specified in normalized frequency units. Also called Fstop2. Bandpass and bandstop filters use this option.
- `h` — complex frequency response values
- `n` — filter order.

- `tw` — width of the transition region between the pass and stop bands. Both halfband and Nyquist filters use this option.

`d = fdesign.decimator(...,spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.decimator(...,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.decimator(...,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

These examples show how to construct decimating filter specification objects. First, create a default specifications object without using input arguments except for the decimation factor `m`.

```
d = fdesign.decimator(2,'nyquist',2,0.1,80) % Set tw=0.1, and ast=80.

d =

          MultirateType: 'Decimator'
               Response: 'Nyquist'
       DecimationFactor: 2
          Specification: 'TW,Ast'
            Description: {'Transition Width';
                          'Stopband Attenuation (decibels)'}
```

```
    NormalizedFrequency: true
        TransitionWidth: 0.1
                  Astop: 80
```

Now create an object by passing a specification type string 'fst1,fp1,fp2,fst2,ast1,ap,ast2' and a design — the resulting object uses default values for the filter specifications. You must provide the design input argument, bandpass in this example, when you include a specification.

```
d=fdesign.decimator(8,'bandpass','fst1,fp1,fp2,fst2,...
ast1,ap,ast2')

d =

          MultirateType: 'Decimator'
               Response: 'Bandpass'
       DecimationFactor: 8
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7x1 cell}
    NormalizedFrequency: true
                  Fstop1: 0.35
                  Fpass1: 0.45
                  Fpass2: 0.55
                  Fstop2: 0.65
                  Astop1: 60
                   Apass: 1
                  Astop2: 60
```

Create another decimating filter specification object, passing the specification values to the object rather than accepting the default values for fp,fst,ap,ast.

```
d=fdesign.decimator(3,'lowpass',.45,0.55,.1,60)

d =

           MultirateType: 'Decimator'
```

```
              Response: 'Lowpass'
     DecimationFactor: 3
        Specification: 'Fp,Fst,Ap,Ast'
          Description: {4x1 cell}
NormalizedFrequency: true
                Fpass: 0.45
                Fstop: 0.55
                Apass: 0.1
                Astop: 60
```

Now pass the filter specifications that correspond to the specifications — `n,fc,ap,ast`.

```
d=fdesign.decimator(3,'ciccomp',1,2,'n,fc,ap,ast',...
20,0.45,.05,50)

d =

           MultirateType: 'Decimator'
                Response: 'CIC Compensator'
       DecimationFactor: 3
          Specification: 'N,Fc,Ap,Ast'
            Description: {4x1 cell}
       NumberOfSections: 2
     DifferentialDelay: 1
    NormalizedFrequency: true
            FilterOrder: 20
                Fcutoff: 0.45
                  Apass: 0.05
                  Astop: 50
```

Now design a decimator using the `kaiserwin` design method.

```
hm = kaiserwin(d)
```

Pass a new specification type for the filter, specifying the filter order. Note that the inputs must include the differential delay `dd` with the CIC input argument to design a CIC specification object.

```
m = 5;
dd = 2;
d = fdesign.decimator(m,'cic',dd,'fp,ast',0.55,55)

d =

         MultirateType: 'Decimator'
              Response: 'CIC'
       DecimationFactor: 5
          Specification: 'Fp,Ast'
            Description: {'Passband Frequency';'
                          Stopband Attenuation(decibels)'}
     DifferentialDelay: 2
    NormalizedFrequency: true
                  Fpass: 0.55
```

In this example, you specify a sampling frequency as the last input argument. Here is it 1000 Hz.

```
d=fdesign.decimator(8,'bandpass','fst1,fp1,fp2,fst2,...
ast1,ap,ast2',0.25,0.35,.55,.65,50,.05,50,1e3)

d =

          MultirateType: 'Decimator'
               Response: 'Bandpass'
       DecimationFactor: 8
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7x1 cell}
    NormalizedFrequency: false
                     Fs: 1000
                 Fstop1: 0.25
                 Fpass1: 0.35
                 Fpass2: 0.55
                 Fstop2: 0.65
                 Astop1: 50
                  Apass: 0.05
```

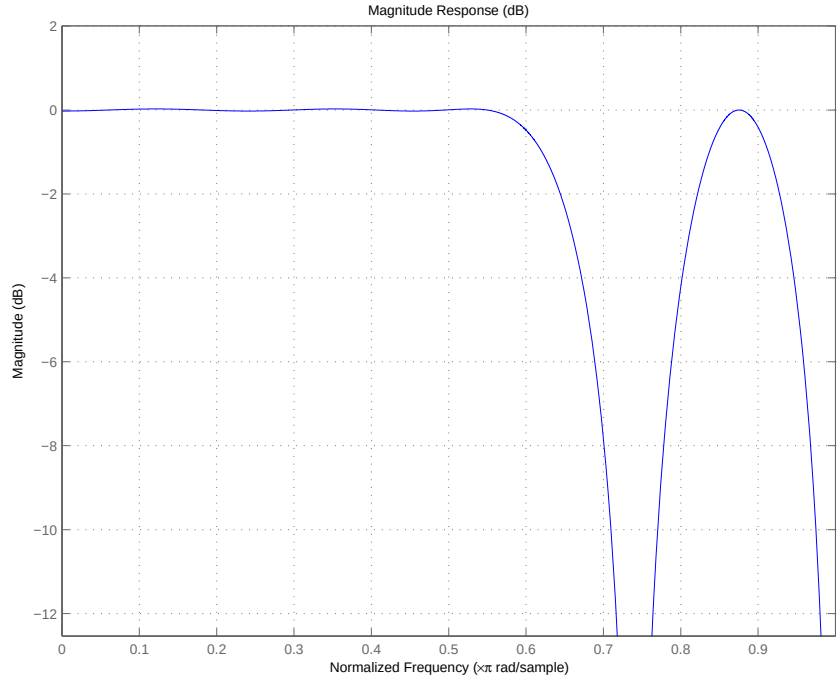
```
Astop2: 50
```

In this, the last example, use the `linear` option for the filter specification object and specify the stopband ripple attenuation in linear format.

```
hs = fdesign.decimator(4,'lowpass','n,fst,ap,ast',15,0.55,.05,50,...
     1e-3,'linear') % 1e-3 = 60decibels.

hs =

               Response: 'Lowpass decimator'
          Specification: 'TW,Ast'
            Description: {'Transition Width';'
                           Stopband Attenuation (decibels)'}
       DecimationFactor: 4
    NormalizedFrequency: false
                     Fs: 500
        TransitionWidth: 0.1
                  Astop: 60
```

Design the filter and display the magnitude response in FVTool.

```
designmethods(hs);
equiripple(hs);  % Starts FVTool to display the response.
```



See Also

`fdesign`, `fdesign.arbmag`, `fdesign.arbmagnphase`, `fdesign.interpolator`, `fdesign.rsrc`

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:

- `n` — full band differentiator (default).
- `n,fp,fst` — 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.

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.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

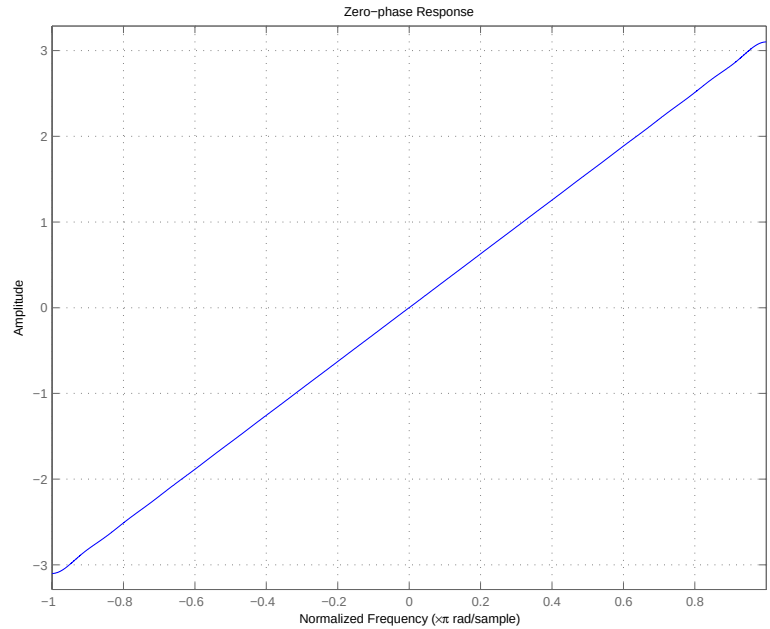
When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

The toolbox lets you design a range of differentiators. These examples present a few possible designs. The first example designs a 33rd-order

full band differentiator. The FVTool plot following the code shows the resulting 33rd-order filter.

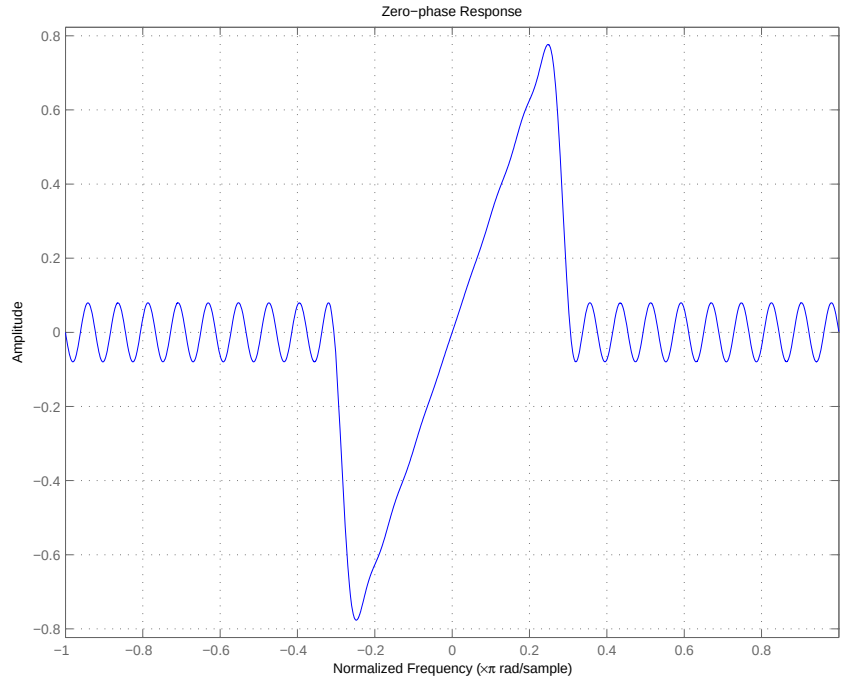
```
d = fdesign.differentiator(33); % Filter order is 33.
designmethods(d);

hd = design(d,'firls');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')

Design Methods for class fdesign.differentiator (N):

equiripple
firls
```



For the second example, design a narrow band differentiator. Differentiate the first 25 percent of the frequencies in the Nyquist range and filter the higher frequencies.

```
d = fdesign.differentiator('n,fp,fst',54,.25,.3);
designmethods(d);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase');
set(hf,'frequencyrange','[-fs/2, fs/2)')
```

Here is the view from FVTool.

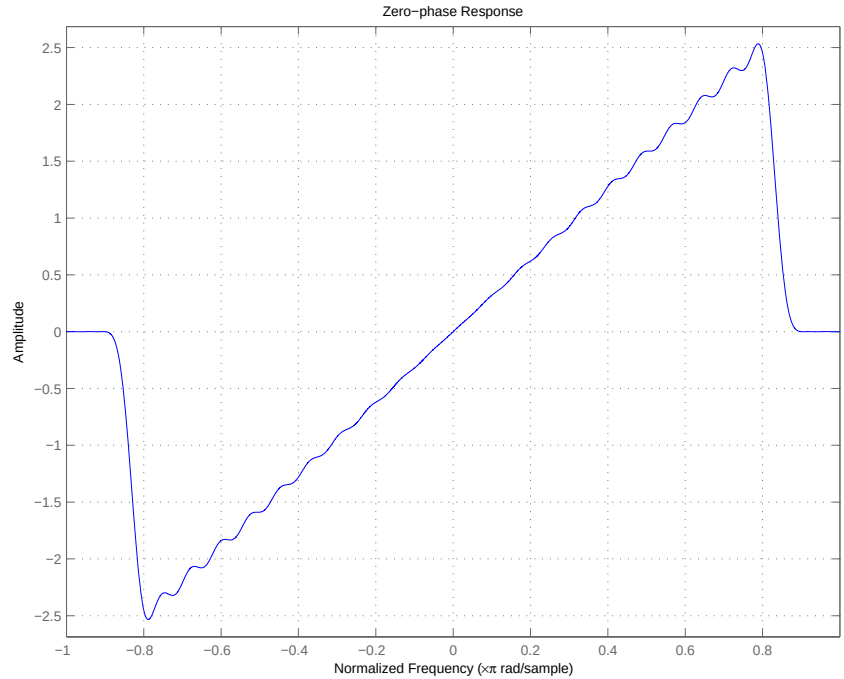


Finally, design a minimum-order, wide-band differentiator.

```
d = fdesign.differentiator('fp,fst,ap,ast',.8,.9,1,80);
```

```
designmethods(d);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase','frequencyrange')
```

FVTool returns this plot.



See Also

`design`, `fdesign`, `setspecs`

fdesign.fracdelay

Purpose

Fractional delay filter specification object

Syntax

```
d = fdesign.fracdelay(delta)
d = fdesign.fracdelay(delta,'N')
d = fdesign.fracdelay(delta,'N',n)
d = fdesign.fracdelay(delta,n)
d = fdesign.fracdelay(...,fs)
```

Description

`d = fdesign.fracdelay(delta)` constructs a default fractional delay filter designer `d` with the filter order set to 3 and the delay value set to `delta`. The fractional delay `delta` must be between 0 and 1 samples.

`d = fdesign.fracdelay(delta,'N')` initializes the filter designer specification string to `N`, where `N` specifies the fractional delay filter order and defaults to filter order of 3.

Use `designmethods(d)` to get a list of the design methods available for a specification string.

`d = fdesign.fracdelay(delta,'N',n)` initializes the filter designer to specification string `N` and sets the filter order to `n`.

`d = fdesign.fracdelay(delta,n)` assumes the default specification `N`, filter order, and sets the filter order to the value you provide in input `n`.

`d = fdesign.fracdelay(...,fs)` adds the argument `fs`, specified in units of Hertz (Hz) to define the sampling frequency. In this case, specify the fractional delay `delta` to be between 0 and 1/`fs`.

Examples

Design a second-order fractional delay filter of 0.2 samples using the Lagrange method. Implement the filter using a Farrow fractional delay (`fd`) structure.

```
d = fdesign.fracdelay(0.2,'N',2);
hd = design(d,'lagrange','filterstructure','fd');
fvtool(hd,'analysis','grpdelay')
```

Design a cubic fractional delay filter with a sampling frequency of 8 kHz and fractional delay of 50 microseconds using the Lagrange method.

```
d = fdesign.fracdelay(50e-6,'N',3,8000);
Hd = design(d, 'lagrange', 'FilterStructure', 'fd');
fvtool(Hd)
```

See Also

design, designopts, fdesign, setspecs

fdesign.halfband

Purpose

Halfband filter specification object

Syntax

```
d = fdesign.halfband
d = fdesign.halfband('type',type)
d = fdesign.halfband(spec)
d = fdesign.halfband(spec,specvalue1,specvalue2,...)
d = fdesign.halfband(specvalue1,specvalue2)
d = fdesign.halfband(...,fs)
d = fdesign.halfband(...,magunits)
```

Description

`d = fdesign.halfband` constructs a halfband filter specification object `d`, applying default values for the properties `tw` and `ast`.

Using `fdesign.halfband` with a design method generates a `dfilt` object.

`d = fdesign.halfband('type',type)` initializes the filter designer 'Type' property with `type`. "`type`" must be either `lowpass` or `highpass` and is not case sensitive.

`d = fdesign.halfband(spec)` constructs object `d` and sets its '`Specification`' 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.

- `tw,ast` (default `spec`)
- `n,tw`
- `n`
- `n,ast`

The string entries are defined as follows:

- `ast` — attenuation in the stop band in decibels (the default units).
- `n` — filter order.

- `tw` — width of the transition region between the pass and stop bands. Specified in normalized frequency units.

By default, all frequency specifications are assumed to be in normalized frequency units. Moreover, all magnitude specifications are assumed to be in dB. Different specification types may have different design methods available.

The filter design methods that apply to a halfband filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string.

`d = fdesign.halfband(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.halfband(specvalue1,specvalue2)` constructs an object `d` assuming the default `Specification` property string `tw,ast`, using the values you provide for the input arguments `specvalue1` and `specvalue2` for `tw` and `ast`.

`d = fdesign.halfband(...,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.halfband(...,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

These examples show how to construct a halfband filter specification object. First, create a default specifications object without using input arguments.

```
>> d=fdesign.halfband

d =

                Response: 'Halfband'
           Specification: 'TW,Ast'
             Description: {'Transition Width';'Stopband Attenuation (dB
                    Type: 'Lowpass'
    NormalizedFrequency: true
        TransitionWidth: 0.1
                  Astop: 80
```

Now create an object by passing a specification type string 'n,ast' — the resulting object uses default values for n and ast.

```
>> d=fdesign.halfband('n,ast')

d =

              Response: 'Halfband'
         Specification: 'N,Ast'
           Description: {'Filter Order';'Stopband Attenuation (dB)'}
                  Type: 'Lowpass'
    NormalizedFrequency: true
           FilterOrder: 10
                 Astop: 80
```

Create another halfband filter object, passing the specification values to the object rather than accepting the default values for n and ast.

```
>> d = fdesign.halfband('n,ast', 42, 80)

d =
```

```
              Response: 'Halfband'
         Specification: 'N,Ast'
           Description: {'Filter Order';'Stopband Attenuation (dB)'}
                  Type: 'Lowpass'
    NormalizedFrequency: true
           FilterOrder: 42
                 Astop: 80
```

For another example, pass the filter values that correspond to the default `Specification` — `n,ast`.

```
>> d = fdesign.halfband(.01, 80)

d =

              Response: 'Halfband'
         Specification: 'TW,Ast'
           Description: {'Transition Width';'Stopband Attenuation
                  Type: 'Lowpass'
    NormalizedFrequency: true
        TransitionWidth: 0.01
                 Astop: 80
```

This example designs an equiripple FIR filter, starting by passing a new specification type and specification values to `fdesign.halfband`.

```
>> hs = fdesign.halfband('n,ast',80,70)

hs =

              Response: 'Halfband'
         Specification: 'N,Ast'
           Description: {'Filter Order';'Stopband Attenuation (dB)
                  Type: 'Lowpass'
    NormalizedFrequency: true
           FilterOrder: 80
                 Astop: 70
```

```
equiripple(hs); % Opens FVTool automatically.
```

In this example, pass the specifications for the filter, and then design a least-squares FIR filter from the object, using `firls` as the design method.

```
>> hs = fdesign.halfband('n,tw', 42, .04)

hs =

               Response: 'Halfband'
          Specification: 'N,TW'
            Description: {'Filter Order';'Transition Width'}
                   Type: 'Lowpass'
    NormalizedFrequency: true
            FilterOrder: 42
        TransitionWidth: 0.04

>> designmethods(hs)


Design Methods for class fdesign.halfband (N,TW):


ellip
iirlinphase
equiripple
firls
kaiserwin

>> hd=firls(hs)

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x43 double]
```

```
       PersistentMemory: false
```

In this example we design a 80th order equiripple halfband highpass filter with 70dB of stopband attenuation.

```
d = fdesign.halfband('Type','Highpass','N,Ast',80,70);
   Hd = design(d,'equiripple');
```

See Also

`fdesign`, `fdesign.decimator`, `design`, `fdesign.interpolator`, `fdesign.nyquist`, `setspecs`

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 properties `fst`, `fp`, `ast` and `ap`.

Using `fdesign.highpass` with a design method generates a `dfilt` object.

`d = fdesign.highpass(spec)` constructs object `d` and sets its 'Specification' 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.

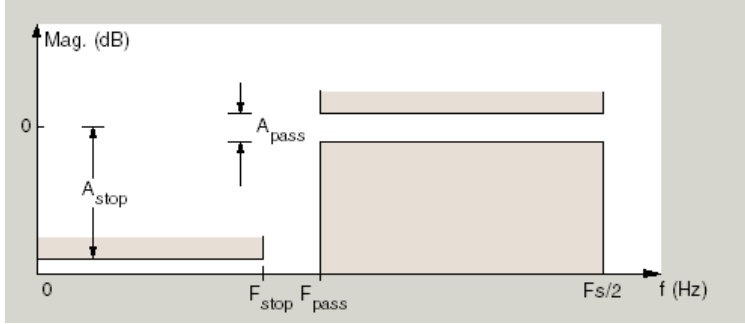
- `fst,fp,ast,ap` (default `spec`)
- `n,f3db`
- `n,f3db,ap`
- `n,f3db,ast`
- `n,f3db,ast,ap`
- `n,f3db,fp`
- `n,fc`
- `n,fc,ast,ap`
- `n,fp,ap`
- `n,fp,ast,ap`
- `n,fst,ast`

- `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 3 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 `fst1` 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.

`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 values for the default `Specification` property string, using the specifications you provide as input arguments specvalue1,specvalue2,specvalue3,specvalue4.

`d = fdesign.highpass(...,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.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

These examples how to construct a highpass filter specification object. First, create a default specifications object without using input arguments.

```
d=fdesign.highpass

d =

               Response: 'Minimum-order highpass'
          Specification: 'Fst,Fp,Ast,Ap'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fstop: 0.4500
                  Fpass: 0.5500
                  Astop: 60
                  Apass: 1
```

This time, pass the specifications that correspond to the default `Specification` string.

```
hs = fdesign.highpass(.4,.5,80,1);

hs =

               Response: 'Minimum-order highpass'
          Specification: 'Fst,Fp,Ast,Ap'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fstop: 0.4000
                  Fpass: 0.5000
                  Astop: 80
                  Apass: 1
```

Now create an object by passing a specification type string `'n,fc'` — the resulting object uses default values for `n` and `fc`.

```
d=fdesign.highpass('n,fc')

d =

               Response: 'Highpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: true
            FilterOrder: 10
                Fcutoff: 0.5000
```

Create the same filter, passing the values for `n` and `fc` rather than accepting the default values. You can add include the sampling frequency `fs` as the final input argument. Adding `fs` puts all the frequency specifications into linear frequency format, rather than normalized frequency.

```
d=fdesign.highpass('n,fc',10,9600,48000)

d =

               Response: 'Highpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
            FilterOrder: 10
                Fcutoff: 9600
```

Finally, pass values for the filter specifications that match the default `Specification` string — `fp = 10`, `fst = 12`, `ast = 80` and `ap = 0.5`. Add the sampling frequency on the end.

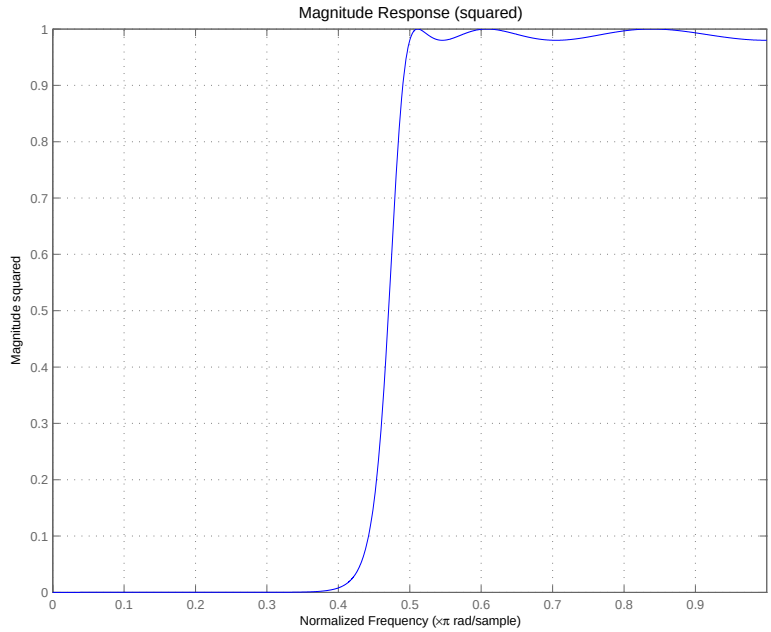
```
d=fdesign.highpass(10,12,80,0.5,48000)
```

```
d =

               Response: 'Minimum-order highpass'
          Specification: 'Fst,Fp,Ast,Ap'
            Description: {4x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
                  Fstop: 10
                  Fpass: 12
                  Astop: 80
```

To demonstrate the `magunits` input option, pass the magnitude specifications in squared units and include the `squared` input argument for `magunits`.

```
hs = fdesign.highpass(.4, .5, .02, .98, 'squared');
hd = cheby1(hs);
fvtool(hd,'MagnitudeDisplay','Magnitude Squared');
```

The following figure shows the filter response.



See Also

`fdesign`, `fdesign.bandpass`, `fdesign.bandstop`, `fdesign.lowpass`

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 31.

`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 string you provide is not case sensitive:

- `n,tw` — default spec string.
- `tw,ap` — minimum-order Hilbert filter.

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 to use. 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

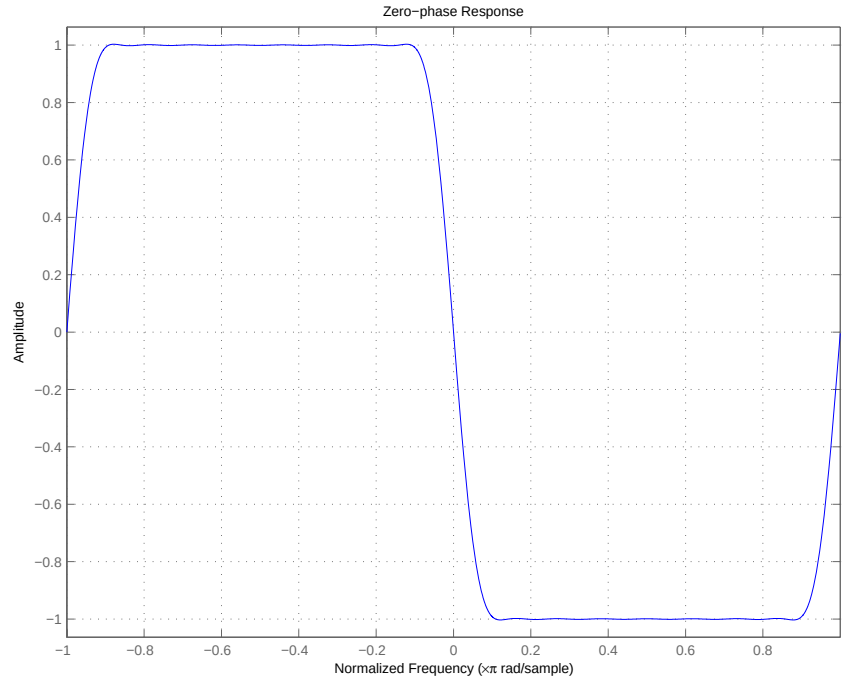
When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

The toolbox lets you design a range of Hilbert filters. These examples present a few possible designs. The first example designs a 30th-order type III Hilbert transformer filter. The FVTool plot following the code shows the resulting filter.

```
d = fdesign.hilbert(30,0.2); % n,tw specification string.
designmethods(d);
```

```
hd = design(d,'firls');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')


Design Methods for class fdesign.hilbert (N,TW):

ellip
iirlinphase
equiripple
firls
```

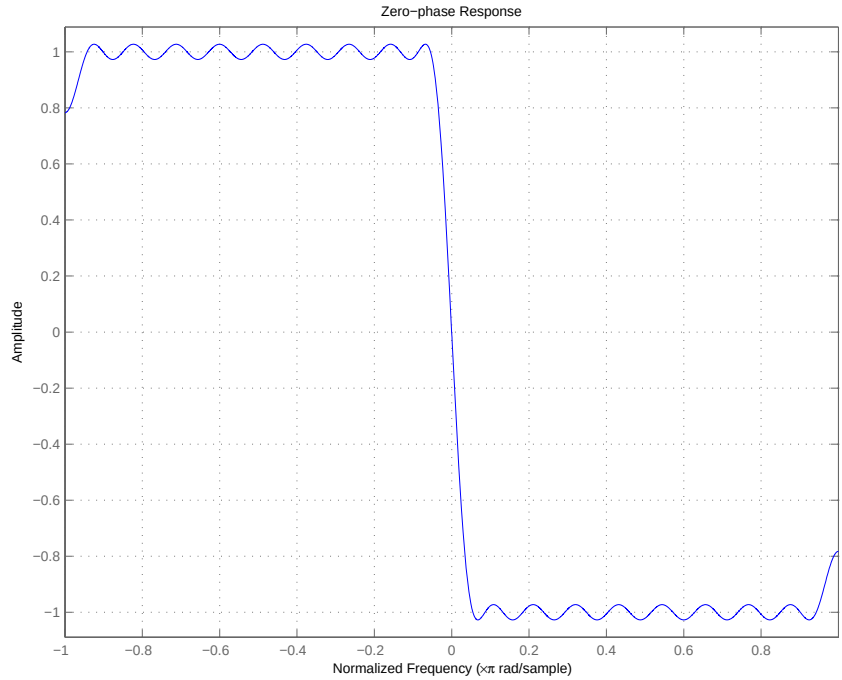


For the second example, design a 35th-order type IV Hilbert transformer.

```
d = fdesign.hilbert('n,tw',35,0.1);
```

```
designmethods(d);
hd = design(d,'equiripple');
hf = fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange')
set(hf,'frequencyrange','[-fs/2, fs/2)')
```

Here is the view from FVTool.

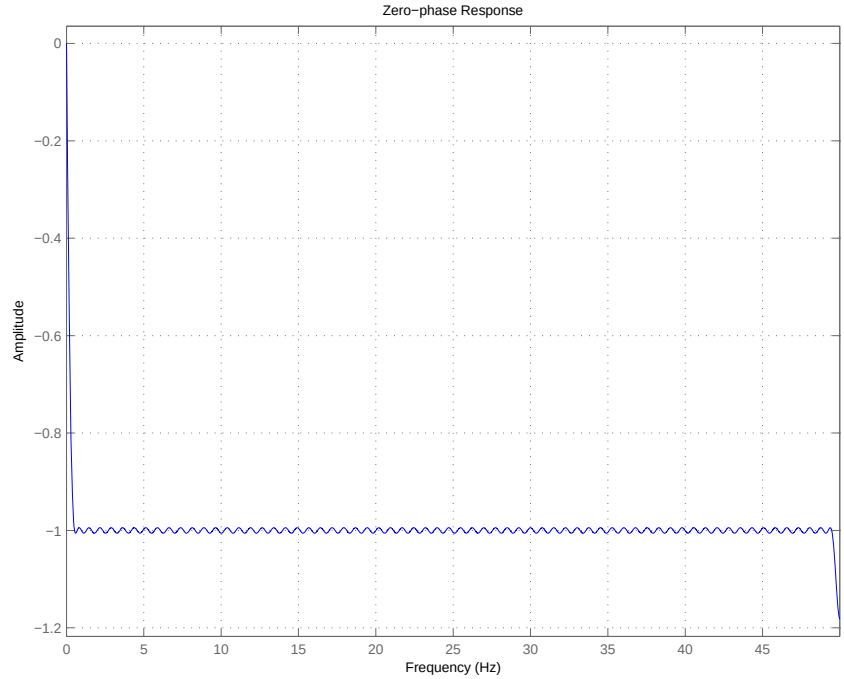


Finally, design a minimum-order transformer that has a sampling frequency of 100 Hz — add `Fs` as an input argument in Hz.

```
d = fdesign.hilbert('tw,ap',1,0.1,100); % Fs = 100 Hz.
designmethods(d);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase');
```

```
set(hf,'frequencyrange','[-fs/2, fs/2)')
```

FVTool returns this plot.



See Also

`design`, `fdesign`, `setspecs`

fdesign.interpolator

Purpose

Interpolator filter specification

Syntax

```
d = fdesign.interpolator(l)
d = fdesign.interpolator(l,design)
d = fdesign.interpolator(l,design,spec)
d =
fdesign.interpolator(...,spec,specvalue1,specvalue2,...)
d = fdesign.interpolator(...,fs)
d = fdesign.interpolator(...,magunits)
```

Description

`d = fdesign.interpolator(l)` constructs an interpolating filter specification object `d`, applying default values for the properties `fp`, `fst`, `ap`, and `ast` and using the default design, Nyquist. Specify `l`, the interpolation factor, as an integer. When you omit the input argument `l`, `fdesign.interpolator` sets the interpolation factor `l` to 3.

Using `fdesign.interpolator` with a design method generates an `mfilt` object.

`d = fdesign.interpolator(l,design)` constructs an interpolator with the interpolation factor `l` and the response you specify in `design`. By using the `design` input argument, you can choose the sort of filter that results from using the interpolator specifications object. `design` accepts the following strings that define the filter response.

design String	Description
`arbmag`	Sets the response for the interpolator specifications object to Arbitrary Magnitude.
`arbmangnphase`	Sets the response for the interpolator specifications object to Arbitrary Magnitude and Phase.
`bandpass`	Sets the response for the interpolator specifications object to bandpass.
`bandstop`	Sets the response for the interpolator specifications object to bandstop.

design String	Description
`cic`	Sets the response for the interpolator specifications object to CIC filter.
`ciccomp`	Sets the response for the interpolator specifications object to CIC compensator.
`halfband`	Sets the response for the interpolator specifications object to halfband.
`highpass`	Sets the response for the interpolator specifications object to highpass.
`isinclp`	Sets the response for the interpolator specifications object to inverse-sinc lowpass.
`lowpass`	Sets the response for the interpolator specifications object to lowpass.
`nyquist`	Sets the response for the interpolator specifications object to Nyquist.

`d = fdesign.interpolator(l,`*`design`*`,`*`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`* depend on the design type of the specifications object.

When you add the `spec` input argument, you must also add the `design` input argument.

Because you are designing multirate filters, the specification strings available are not the same as the specifications for designing single-rate filters with such design methods as `fdesign.lowpass`. The strings are not case sensitive.

The interpolation factor `l` is not in the specification strings. Various design types provide different specifications, as shown in this table.

Design Type	Valid Specification Strings
Arbitrary Magnitude	• `n,b,f,a` • `n,f,a` (default string)
Arbitrary Magnitude and Phase	• `n,b,f,h` • `n,f,h` (default string)
`Bandpass`	• `fst1,fp1,fp2,fst2,ast1,ap,ast2` (default string) • `n,fc1,fc2` • `n,fst1,fp1,fp2,fst2`
`Bandstop`	• `n,fc1,fc2` • `n,fp1,fst1,fst2,fp2` • `fp1,fst1,fst2,fp2,ap1,ast,ap2` (default string)
`CIC`	• `fp,ast` (default and only string)
`CIC Compensator`	• `fp,fst,ap,ast` (default string) • `n,fc,ap,ast` • `n,fp,ap,ast` • `n,fp,fst` • `n,fst,ap,ast`
`Halfband`	• `tw,ast` (default string) • `n,tw` • `n` • `n,ast`

Design Type	Valid Specification Strings
`Highpass`	• `fst,fp,ast,ap` (default string) • `n,fc` • `n,fc,ast,ap` • `n,fp,ast,ap` • `n,fst,fp,ap` • `n,fst,fp,ast` • `n,fst,ast,ap` • `n,fst,fp`
`Inverse-Sinc Lowpass`	• `fp,fst,ap,ast` (default string) • `n,fc,ap,ast` • `n,fst,ap,ast` • `n,fp,ap,ast` • `n,fp,fst`
`Lowpass`	• `fp,fst,ap,ast` (default string)
`Nyquist`	• `tw,ast` (default string) • `n,tw` • `n` • `n,ast`

The string entries are defined as follows:

- `a` — magnitude response at the frequencies in `f`. Usually this is a vector of values with the same length as `f`.
- `ap` — amount of ripple allowed in the pass band 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. Bandpass and bandstop filters use this option.
- `ap2` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass2. Bandpass and bandstop filters use this option.
- `ast` — attenuation in the first stop band in decibels (the default units). Also called Astop.
- `ast1` — attenuation in the first stop band in decibels (the default units). Also called Astop1. Bandpass and bandstop filters use this option.
- `ast2` — attenuation in the first stop band in decibels (the default units). Also called Astop2. Bandpass and bandstop filters use this option.
- `b` — number of filter bands.
- `f` — vector of specific frequency points in the filter response. In combination with `a`, this specifies the desired filter response.
- `fc1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. Bandpass and bandstop filters use this option.
- `fc2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. Bandpass and bandstop filters use this option.
- `fp1` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass1. Bandpass and bandstop filters use this option.
- `fp2` — frequency at the end of the pass band. Specified in normalized frequency units. Also called Fpass2. Bandpass and bandstop filters use this option.

- `fst1` — frequency at the end of the first stop band. Specified in normalized frequency units. Also called Fstop1. Bandpass and bandstop filters use this option.
- `fst2` — frequency at the start of the second stop band. Specified in normalized frequency units. Also called Fstop2. Bandpass and bandstop filters use this option.
- `h` — complex frequency response values.
- `n` — filter order.
- `tw` — width of the transition region between the pass and stop bands. Halfband, Hilbert, and Nyquist filters use this option.

`d = fdesign.interpolator(...,`*`spec`*`,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.interpolator(...,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.interpolator(...,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

These examples show how to construct interpolating filter specification objects. First, create a default specifications object without using input arguments except for the interpolation factor `l`.

```
l = 2;
d = fdesign.interpolator(2)

d =

          MultirateType: 'Interpolator'
               Response: 'Nyquist'
       DecimationFactor: 2
          Specification: 'TW,Ast'
            Description: {'Transition Width';'
                          Stopband Attenuation (dB)'}
    NormalizedFrequency: true
        TransitionWidth: 0.1
                  Astop: 80
```

Now create an object by passing a specification string '`fst1,fp1,fp2,fst2,ast1,ap,ast2`' and a design — the resulting object uses default values for all of the filter specifications. You must provide the design input argument when you include a specification.

```
d=fdesign.interpolator(8,'bandpass','fst1,fp1,fp2,fst2,...
ast1,ap,ast2')

d =

           MultirateType: 'Interpolator'
                Response: 'Bandpass'
        DecimationFactor: 8
           Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
             Description: {7x1 cell}
     NormalizedFrequency: true
                 Fstop1: 0.35
                 Fpass1: 0.45
                 Fpass2: 0.55
                 Fstop2: 0.65
                 Astop1: 60
                  Apass: 1
```

```
                Astop2: 60
```

Create another interpolating filter object, passing the specification values to the object rather than accepting the default values for, in this case, `fp`,`fst`,`ap`,`ast`.

```
d=fdesign.interpolator(3,'lowpass',.45,0.55,.1,60)

d =

          MultirateType: 'Interpolator'
               Response: 'Lowpass'
       DecimationFactor: 3
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.45
                  Fstop: 0.55
                  Apass: 0.1
                  Astop: 60
```

Now pass the filter specifications that correspond to the specifications — `n`,`fc`,`ap`,`ast`.

```
d=fdesign.interpolator(3,'ciccomp',1,2,'n,fc,ap,ast',...
20,0.45,.05,50)

d =

          MultirateType: 'Interpolator'
               Response: 'CIC Compensator'
       DecimationFactor: 3
          Specification: 'N,Fc,Ap,Ast'
            Description: {4x1 cell}
      NumberOfSections: 2
     DifferentialDelay: 1
    NormalizedFrequency: true
            FilterOrder: 20
```

```
          Fcutoff: 0.45
            Apass: 0.05
            Astop: 50
```

With the specifications object in your workspace, design an interpolator using the `kaiserwin` design method.

```
hm = design(d,'kaiserwin')
```

Pass a new specification type for the filter, specifying the filter order.

```
d = fdesign.interpolator(5,'CIC','fp,ast',0.55,55)

d =

          MultirateType: 'Interpolator'
               Response: 'CIC'
       DecimationFactor: 5
          Specification: 'Fp,Aa'
            Description: {'Passband Frequency';'Stopband Attenuation(dB)'}
      DifferentialDelay: 1
    NormalizedFrequency: true
                  Fpass: 0.55
```

In this example, you specify a sampling frequency as the right most input argument. Here, it is set to 1000 Hz.

```
d=fdesign.interpolator(8,'bandpass','fst1,fp1,fp2,fst2,...
ast1,ap,ast2',0.25,0.35,.55,.65,50,.05,1e3)

d =

          MultirateType: 'Interpolator'
               Response: 'Bandpass'
       DecimationFactor: 8
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7x1 cell}
    NormalizedFrequency: false
```

```
    Fs: 1000
Fstop1: 0.25
Fpass1: 0.35
Fpass2: 0.55
Fstop2: 0.65
Astop1: 50
 Apass: 0.05
Astop2: 50
```

In this, the last example, use the `linear` option for the filter specification object and specify the stopband ripple attenuation in linear form.

```
d = fdesign.interpolator(4,'lowpass','n,fst,ap,ast',15,0.55,.05,...
     1e3,'linear') % 1e3 = 60dB.

d =

               Response: 'Lowpass interpolator'
          Specification: 'TW,Ast'
            Description: {'Transition Width';'Stopband Attenuation (dB)'}
       DecimationFactor: 4
    NormalizedFrequency: false
                     Fs: 500
        TransitionWidth: 0.1
                  Astop: 60
```

Design the filter and display the magnitude response in FVTool.

```
designmethods(d);
design(d,'equiripple');  % Opens FVTool.
```

Now design a CIC interpolator for a signal sampled at 19200 Hz. Specify the differential delay of 2 and set the attenuation of information beyond 50 Hz to be at least 80 dB.

The filter object sampling frequency is (`l` x `fs`) where `fs` is the sampling frequency of the input signal.

```
dd  = 2;     % Differential delay.
```

```
fp  = 50;    % Passband of interest.
ast = 80;    % Minimum attenuation of alias components in passband.
fs  = 600;   % Sampling frequency for input signal.
l   = 32;    % Interpolation factor.
d   = fdesign.interpolator(l,'cic',dd,'fp,ast',fp,ast,l*fs);
d =

          MultirateType: 'Interpolator'
    InterpolationFactor: 32
               Response: 'CIC'
          Specification: 'Fp,Ast'
            Description: {'Passband Frequency';'Imaging Attenuation(dB)'}
      DifferentialDelay: 2
    NormalizedFrequency: false
                     Fs: 19200
                  Fs_in: 600
                 Fs_out: 19200
                  Fpass: 50
                  Astop: 80
hm = design(d);  %Use the default design method.
hm

hm =

        FilterStructure: 'Cascaded Integrator-Comb Interpolator'
             Arithmetic: 'fixed'
      DifferentialDelay: 2
       NumberOfSections: 2
    InterpolationFactor: 32
       PersistentMemory: false

        InputWordLength: 16
        InputFracLength: 15

        FilterInternals: 'FullPrecision'
```

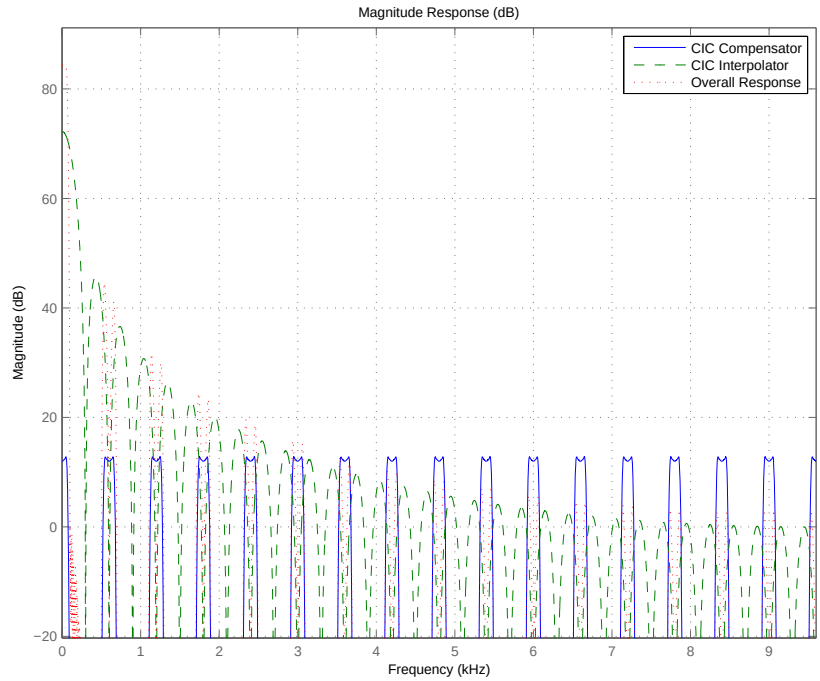
This next example results in a minimum-order CIC compensator that interpolates by 4 and compensates for the droop in the passband for the CIC filter `hm` from the previous example.

```
nsecs = hm.numberofsections;
d = fdesign.interpolator(4,'ciccomp',dd,nsecs,...
50,100,0.1,80,fs);
hmc = design(d,'equiripple');
hmc.arithmetic = 'fixed';
```

`hmc` is designed to compensate for `hm`. To see the effect of the compensating CIC filter, use FVTool to analyze both filters individually and include the compound filter response by cascading `hm` and `hmc`.

```
fvtool(hmc,hm,cascade(hmc,hm),'fs',[fs,l*fs,l*fs],...
'showreference','off');
legend('CIC Compensator','CIC Interpolator',...
'Overall Response');
```

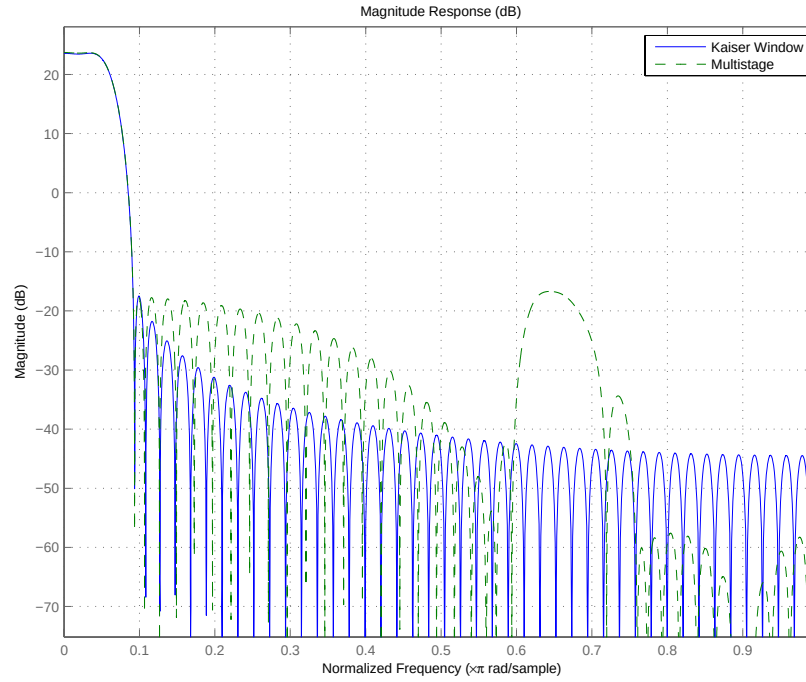
FVTool returns with this plot.



For the third example, use `fdesign.interpolator` to design a minimum-order Nyquist interpolator that uses a Kaiser window. For comparison, design a multistage interpolator as well and compare the responses.

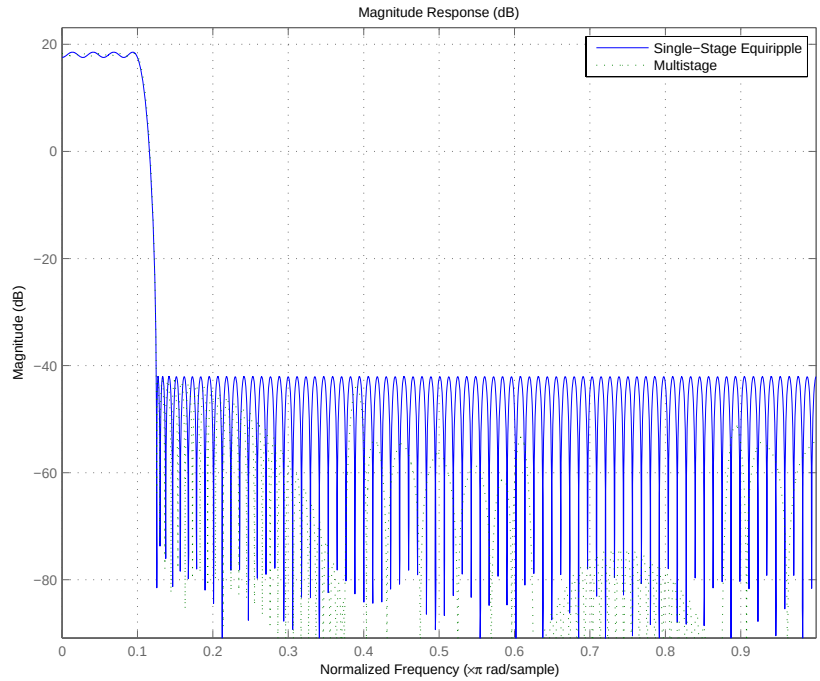
```
l   = 15;   % Set the interpolation factor and the Nyquist band.
tw  = 0.05; % Specify the normalized transition width.
ast = 40;   % Set the minimum stopband attenuation in dB.
d   = fdesign.interpolator(l,'nyquist',l,tw,ast);
hm  = design(d,'kaiserwin');
hm2 = design(d,'multistage'); % Design the multistage interpolator.
fvtool(hm,hm2);
legend('Kaiser Window','Multistage')
```

FVTool shows both responses.



Design a lowpass interpolator for an interpolation factor of 8. Compare the single-stage equiripple design to a multistage design with the same interpolation factor.

```
l = 8; % Interpolation factor.
d = fdesign.interpolator(l,'lowpass');
hm(1) = design(d,'equiripple');
% Use halfband filters whenever possible.
hm(2) = design(d,'multistage','usehalfbands',true);
fvtool(hm);
legend('Single-Stage Equiripple','Multistage')
```



See Also

`fdesign`, `fdesign.arbmag`, `fdesign.arbmagnphase`, `fdesign.decimator`, `fdesign.rsrc`, `setspecs`

Purpose

Inverse-sinc filter specification

Syntax

```
d = fdesign.isinclp
d = fdesign.isinclp(spec)
d = fdesign.isinclp(spec,specvalue1,specvalue2,...)
d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,
   specvalue4)
d = fdesign.isinclp(...,fs)
d = fdesign.isinclp(...,magunits)
```

Description

`d = fdesign.isinclp` constructs an inverse-sinc lowpass filter specification object `d`, applying default values for the properties `tw` and `ast`.

Using `fdesign.isinclp` with a design method generates a `dfilt` object.

`d = fdesign.isinclp(spec)` constructs object `d` and sets its 'Specification' 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.

- `fp,fst,ap,ast` (default `spec`)
- `n,fst,ap,ast`
- `n,fp,fst`

The string entries are defined as follows:

- `ast` — attenuation in the first stop band in decibels (the default units). Also called Astop.
- `ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.
- `fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.

- `fst` — frequency at the end of the first stop band. Specified in normalized frequency units. Also called Fstop.
- `n` — filter order.

The filter design methods that apply to an inverse-sinc lowpass filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string.

`d = fdesign.isinclp(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `d` assuming the default `Specification` property string `fp,fst,ap,ast`, using the values you provide in specvalue1,specvalue2, specvalue3, and specvalue4.

`d = fdesign.isinclp(...,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.isinclp(...,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

Pass the specifications for the default specification — `fp,fst,ap,ast` — as input arguments to the specifications object.

```
d = fdesign.isinclp(.4,.5,.01,40);
designmethods(d)
hd = design(d,'equiripple');
fvtool(hd);
```

FVTool shows the classic inverse-sinc filter response.

See Also

`fdesign`, `fdesign.bandpass`, `fdesign.bandstop`, `fdesign.halfband`, `fdesign.highpass`, `fdesign.lowpass`, `fdesign.nyquist`

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 properties `fp`, `fst`, `ap`, and `ast`.

Using the `fdesign.lowpass` specification object with a design method generates a `dfilt` object.

`d = fdesign.lowpass(spec)` constructs object `d` and sets its '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.

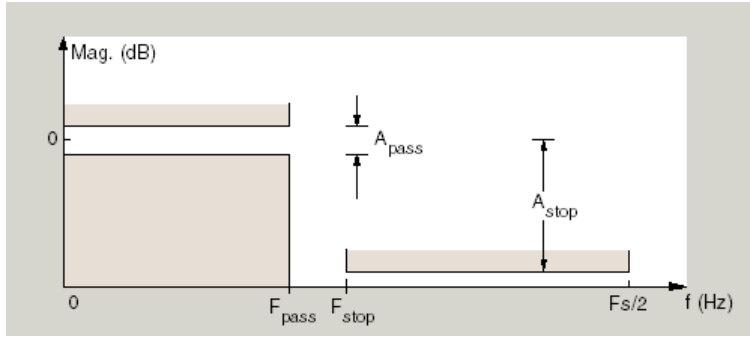
- `fp,fst,ap,ast` (default `spec`)
- `n,f3db`
- `n,f3db,ap`
- `n,f3db,ap,ast`
- `n,f3db,ast`
- `n,f3db,fst`
- `n,fc`
- `n,fc,ap,ast`
- `n,fp,ap`
- `n,fp,ap,ast`
- `n,fp,fst,ap`

- `n,fp,f3db`
- `n,fp,fst`
- `n,fp,fst,ap`
- `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 3 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.

The filter design methods that apply to a lowpass filter specification object change depending on the `Specification` string. Here are all the valid strings for lowpass filter specification objects.

- `fp,fst,ap,ast`
- `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,f3db`
- `n,fp,fst`
- `n,fp,fst,ap`

- `n,fp,fst,ast`
- `n,fst,ap,ast`
- `n,fst,ast`
- `n,fp,ap,ast`
- `nb,na,fp,fst`

`d = fdesign.lowpass(`*`spec`*`,specvalue1,specvalue2,...)` constructs an object `d` and sets its 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 `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

These examples how to construct a lowpass filter specification object. First, create a default lowpass filter object without using input arguments.

```
d=fdesign.lowpass

d =

               Response: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.4500
                  Fstop: 0.5500
                  Apass: 1
                  Astop: 60
```

Now create an object by passing specifications for the passband and stopband edge frequencies and the passband and stopband attenuations — the resulting object uses the input values for `fp`, `fst`, `ap`, and `ast`.

```
hs = fdesign.lowpass(.4,.5,1,80);
hs

hs =

               Response: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.4000
                  Fstop: 0.5000
                  Apass: 1
                  Astop: 80
```

Create another filter object, passing the values for `n` and `fc` rather than accepting the default values. You can add include the sampling frequency `fs` as the final input argument.

```
d=fdesign.lowpass('n,fc',10, 9600,48000)

d =

               Response: 'Lowpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
            FilterOrder: 10
                Fcutoff: 9600
```

Finally, pass values for the filter specifications that match the default `Specification` string entries — `fp = 0.4`, `fst = 0.5`, `ast = 80` and `ap = 1.0`. Add the sampling frequency on the end.

```
hs = fdesign.lowpass(.4,.5,1,80)

hs =

               Response: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.4000
                  Fstop: 0.5000
                  Apass: 1
                  Astop: 80
```

Finally, the next examples add the sampling frequency specification in Hz, and then the `magunits` option.

```
hs = fdesign.lowpass('N,Fp,Ap', 10, 9600, .5, 48000);
```

and

```
hsmag = fdesign.lowpass(.4, .5, .98, .02, 'squared');
```

Using the last example filter object, create a highpass filter.

```
hd = design(hsmag,'cheby1';
```

See Also

`fdesign`, `fdesign.bandpass`, `fdesign.bandstop`, `fdesign.highpass`

Purpose

Notch filter specification

Syntax

```
d = fdesign.notch(specstring, value1, value2, ...)
d = fdesign.notch(n,f0,q)
d = fdesign.notch(...,Fs)
d = fdesign.notch(...,MAGUNITS)
```

Description

`d = fdesign.notch(specstring, value1, value2, ...)` constructs a notch filter specification object `d`, with a specification string set to `specstring` and values provided for all members of the `specstring`. The possible specification strings, which are not case sensitive, are listed as follows:

- 'N,F0,Q' (default)
- 'N,F0,Q,Ap'
- 'N,F0,Q,Ast'
- 'N,F0,Q,Ap,Ast'
- 'N,F0,BW'
- 'N,F0,BW,Ap'
- 'N,F0,BW,Ast'
- 'N,F0,BW,Ap,Ast'

where the variables are defined as follows:

- N - Filter Order (must be even)
- F0 - Center Frequency
- Q - Quality Factor
- BW - 3-dB Bandwidth
- Ap - Passband Ripple (decibels)
- Ast - Stopband Attenuation (decibels)

Different specification strings, resulting in different specification objects, may have different design methods available. Use the function `designmethods` to get a list of design methods available for a given specification. For example:

```
>> d  = fdesign.notch('N,F0,Q,Ap',6,0.5,10,1);
>> designmethods(d)


Design Methods for class fdesign.notch (N,F0,Q,Ap):


cheby1
```

`d = fdesign.notch(n,f0,q)` constructs a notch filter specification object using the default `specstring ('N,F0,Q')` and setting the corresponding values to `n`, `f0`, and `q`.

By default, all frequency specifications are assumed to be in normalized frequency units. All magnitude specifications are assumed to be in decibels.

`d = fdesign.notch(...,Fs)` constructs a notch filter specification object while providing the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other values provided. If you specify an `Fs`, it is assumed to be in Hz, as are all the other frequency values provided.

`d = fdesign.notch(...,MAGUNITS)` constructs a notch filter specification while providing the units for any magnitude specification given. `MAGUNITS` can be one of the following: `'linear'`, `'dB'`, or `'squared'`. If this argument is omitted, `'dB'` is assumed. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If `Fs` is provided, `MAGUNITS` must follow `Fs` in the input argument list.

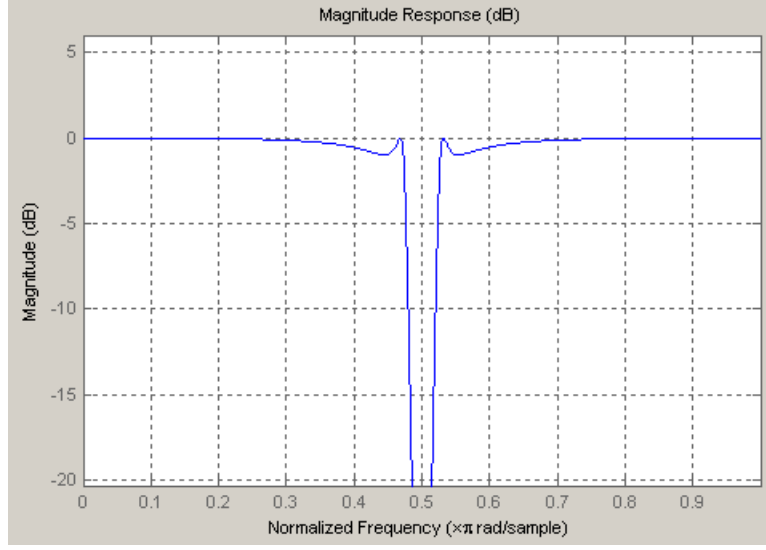
Examples

Design a notching filter with a passband ripple of 1 dB.

```
d  = fdesign.notch('N,F0,Q,Ap',6,0.5,10,1);
```

```
Hd = design(d);
fvtool(Hd)
```

This produces a filter with the magnitude response shown in the following figure.



See Also

`fdesign`, `fdesign.peak`

fdesign.nyquist

Purpose

Nyquist filter specification

Syntax

```
d = fdesign.nyquist
d = fdesign.nyquist(l,spec)
d = fdesign.nyquist(l,spec,specvalue1,specvalue2,...)
d = fdesign.nyquist(l,specvalue1,specvalue2)
d = fdesign.nyquist(...,fs)
d = fdesign.nyquist(...,magunits)
```

Description

`d = fdesign.nyquist` constructs a Nyquist or L-band filter specification object `d`, applying default values for the properties `tw` and `ast`. By default, the filter object designs a minimum-order half-band (L=2) Nyquist filter.

Using `fdesign.nyquist` with a design method generates a `dfilt` object.

`d = fdesign.nyquist(l,spec)` constructs object `d` and sets its `Specification` property to *`spec`*. Use `l` to specify the desired value for L. L = 2 design a half-band FIR filter, L = 3 a third-band FIR filter, and so on. When you use a Nyquist filter as an interpolator, l or L is the interpolation factor. The first input argument must be l when you are not using the default syntax `d = fdesign.nyquist`.

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.

- `tw,ast` (default `spec`)
- `n,tw`
- `n`
- `n,ast`

The string entries are defined as follows:

- `ast` — attenuation in the stop band in decibels (the default units).
- `n` — filter order.

- `tw` — width of the transition region between the pass and stop bands. Specified in normalized frequency units.

The filter design methods that apply to an interpolating filter specification object change depending on the `Specification` string. Paired with each string in the following table are the design methods for interpolating filter specification objects that use that string.

Specification String	Applicable Design Method
`tw,ast`	`kaiserwin`
`n,tw`	`kaiserwin`
`n`	`window`
`n,ast`	`kaiserwin`

`d = fdesign.nyquist(l,`*`spec`*`,specvalue1,specvalue2,...)` constructs an object `d` and sets its specification to *`spec`*, and the specification values to `specvalue1`, `specvalue2`, and so on at construction time.

`d = fdesign.nyquist(l,specvalue1,specvalue2)` constructs an object `d` with the values you provide in `l`, `specvalue1`,`specvalue2` as the values for `l`, `tw` and `ast`.

`d = fdesign.nyquist(...,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.nyquist(...,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.

Limitations of the Nyquist fdesign Object

Using Nyquist filter specification objects with the `equiripple` design method imposes a few limitations on the resulting filter, caused by the `equiripple` design algorithm.

- When you request a minimum-order design from `equiripple` with your Nyquist object, the design algorithm might not converge and can fail with a filter convergence error.
- When you specify the order of your desired filter, and use the `equiripple` design method, the design might not converge.
- Generally, the following specifications, alone or in combination with one another, can cause filter convergence problems with Nyquist objects and the `equiripple` design method.
  - very high order
  - small transition width
  - very large stopband attenuation

Note that halfband filters (filters where band = 2) do not exhibit convergence problems.

When convergence issues arise, either in the cases mentioned or in others, you might be able to design your filter with the `kaiserwin` method.

In addition, if you use Nyquist objects to design decimators or interpolators (where the interpolation or decimation factor is not a prime number), using multistage filter designs might be your best approach.

Examples

These examples show how to construct a Nyquist filter specification object. First, create a default specifications object without using input arguments.

```
d=fdesign.nyquist

d =

               Response: 'Nyquist'
          Specification: 'TW,Ast'
            Description: {'Transition Width';'Stopband Attenuation (dB)'}
                   Band: 2
    NormalizedFrequency: true
        TransitionWidth: 0.1
                  Astop: 80
```

Now create an object by passing a specification type string 'n,ast' — the resulting object uses default values for `n` and `ast`.

```
d=fdesign.nyquist(2,'n,ast')

d =

               Response: 'Nyquist'
          Specification: 'N,Ast'
            Description: {'Filter Order';'Stopband Attenuation (dB)'}
                   Band: 2
    NormalizedFrequency: true
            FilterOrder: 10
                  Astop: 80
```

Create another Nyquist filter object, passing the specification values to the object rather than accepting the default values for `n` and `ast`.

```
d=fdesign.nyquist(3,'n,ast',42,80)

d =
```

```
                Response: 'Nyquist'
           Specification: 'N,Ast'
             Description: {'Filter Order';'Stopband Attenuation (dB)'}
                    Band: 3
      NormalizedFrequency: true
             FilterOrder: 42
                   Astop: 80
```

Finally, pass the filter specifications that correspond to the default `Specification` — `tw,ast`. When you pass only the values, `fdesign.nyquist` assumes the default `Specification` string.

```
d = fdesign.nyquist(4,.01,80)

d =

                Response: 'Nyquist'
           Specification: 'TW,Ast'
             Description: {'Transition Width';'Stopband Attenuation (dB)'}
                    Band: 4
      NormalizedFrequency: true
          TransitionWidth: 0.01
                   Astop: 80
```

Now design a Nyquist filter using the `kaiserwin` design method.

```
hd = design(d,'kaiserwin')

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x1005 double]
    PersistentMemory: false
```

See Also

`fdesign`, `fdesign.interpolator`, `fdesign.halfband`, `fdesign.interpolator`, `fdesign.rsrc`

Purpose

Octave filter specification

Syntax

```
d = fdesign.octave(l)
d = fdesign.octave(l, MASK)
d = fdesign.octave(l, MASK, spec)
d = fdesign.octave(..., Fs)
```

Description

`d = fdesign.octave(l)` constructs an octave filter specification object `d`, with `l` bands per octave. The default value for `l` is `1`.

`d = fdesign.octave(l, MASK)` constructs an octave filter specification object `d` with `l` bands per octave and `MASK` specification for the FVTool. The available values for mask are:

- `'class 0'`
- `'class 1'`
- `'class 2'`

`d = fdesign.octave(l, MASK, spec)` constructs an octave filter specification object `d` with `l` bands per octave, `MASK` specification for the FVTool, and the `spec` specification string. The specification strings available are:

- `'N, F0'`

(not case sensitive), where:

- `N` is the filter order
- `F0` is the center frequency.

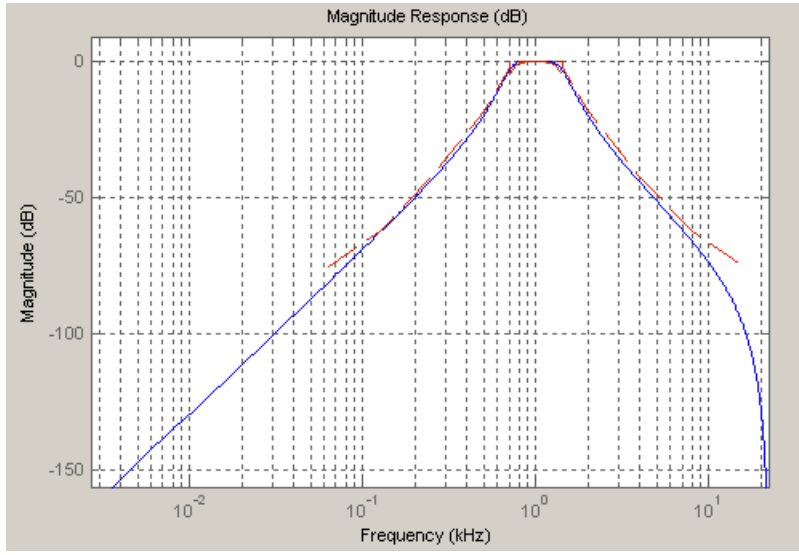
The center frequency is typically specified in normalized frequency units, unless a sampling frequency in Hz is included in the specification: `d = fdesign.octave(..., Fs)`. In this case, all frequencies must be specified in Hz, with the center frequency falling between 20 Hz and 20 kHz (the audio range).

Examples

Design an sixth order, octave-band class 0 filter with a center frequency of 1000 Hz and, a sampling frequency of 44.1 kHz.

```
d = fdesign.octave(1,'Class 0','N,F0',6,1000,44100)
Hd = design(d)
fvtool(Hd)
```

The following figure shows the magnitude response plot of the filter. The logarithmic scale for frequency is automatically set by FVTool for the octave filters.



See Also

`fdesign`

Purpose

Parametric equalizer filter specification

Syntax

```
d = fdesign.parameq(spec, specvalue1, specvalue2, ...)
d = fdesign.parameq(... fs)
```

Description

`d = fdesign.parameq(spec, specvalue1, specvalue2, ...)` constructs a parametric equalizer filter design object, where `spec` is a non-case sensitive specification string. The choices for `spec` are as follows:

- `'FO, BW, BWp, Gref, GO, GBW, Gp'` (minimum order default)
- `'FO, BW, BWst, Gref, GO, GBW, Gst'`
- `'FO, BW, BWp, Gref, GO, GBW, Gp, Gst'`
- `'N, FO, BW, Gref, GO, GBW'`
- `'N, FO, BW, Gref, GO, GBW, Gp'`
- `'N, FO ,BW, Gref, GO, GBW, Gst'`
- `'N, FO, BW, Gref, GO, GBW, Gp, Gst'`
- `'N, Flow, Fhigh, Gref, GO, GBW'`
- `'N, Flow, Fhigh, Gref, GO, GBW, Gp'`
- `'N, Flow, Fhigh, Gref, GO, GBW, Gst'`
- `'N, Flow, Fhigh, Gref, GO, GBW, Gp, Gst'`

where the parameters are defined as follows:

- F0 — Center Frequency
- BW — Bandwidth
- BWp — Passband Bandwidth
- BWst — Stopband Bandwidth
- Gref — Reference Gain (decibels)

- G0 — Center Frequency Gain (decibels)
- GBW — Gain at which Bandwidth (BW) is measured (decibels)
- Gp — Passband Gain (decibels)
- Gst — Stopband Gain (decibels)
- N — Filter Order
- Flow - Lower Frequency at Gain GBW
- Fhigh - Higher Frequency at Gain GBW

Regardless of the specification string chosen, there are some conditions that apply to the specification parameters. These are as follows:

- Specifications for parametric equalizers must be given in decibels
- To boost the input signal, set `G0 > Gref`; to cut, set `Gref > G0`
- For boost: `G0 > Gp > GBW > Gst > Gref`; For cut: `G0 < Gp < GBW < Gst < Gref`
- Bandwidth must satisfy: `BWst > BW > BWp`

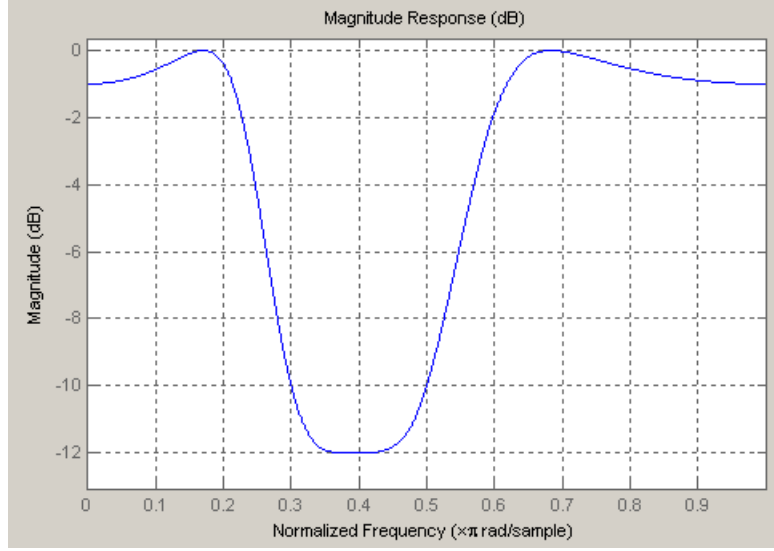
`d = fdesign.parameq(... fs)` adds the input sampling frequency. Fs must be specified as a scalar trailing the other numerical values provided, and is assumed to be in Hz.

Examples

Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB:

```
d = fdesign.parameq('N,Flow,Fhigh,Gref,G0,GBW,Gst',...
      4,.3,.5,0,-12,-10,-1);
Hd = design(d,'cheby2');
fvtool(Hd)
```

The magnitude response is shown in the following figure.


Magnitude Response (dB)
Magnitude (dB)
0
-2
-4
-6
-8
-10
-12
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)

fdesign.peak

Purpose

Peak filter specification

Syntax

```
d = fdesign.peak(specstring, value1, value2, ...)
d = fdesign.peak(n,f0,q)
d = fdesign.peak(...,Fs)
d = fdesign.peak(...,MAGUNITS)
```

Description

`d = fdesign.peak(specstring, value1, value2, ...)` constructs a peaking filter specification object `d`, with a specification string set to `specstring` and values provided for all members of the `specstring`. The possible specification strings, which are not case sensitive, are listed as follows:

- 'N,F0,Q' (default)
- 'N,F0,Q,Ap'
- 'N,F0,Q,Ast'
- 'N,F0,Q,Ap,Ast'
- 'N,F0,BW'
- 'N,F0,BW,Ap'
- 'N,F0,BW,Ast'
- 'N,F0,BW,Ap,Ast'

where the variables are defined as follows:

- N - Filter Order (must be even)
- F0 - Center Frequency
- Q - Quality Factor
- BW - 3-dB Bandwidth
- Ap - Passband Ripple (decibels)
- Ast - Stopband Attenuation (decibels)

Different specification strings, resulting in different specification objects, may have different design methods available. Use the function `designmethods` to get a list of design methods available for a given specification. For example:

```
>> d  = fdesign.peak('N,F0,Q,Ap',6,0.5,10,1);
>> designmethods(d)


Design Methods for class fdesign.peak (N,F0,Q,Ap):


cheby1
```

`d = fdesign.peak(n,f0,q)` constructs a peaking filter specification object using the default `specstring ('N,F0,Q')` and setting the corresponding values to `n`, `f0`, and `q`.

By default, all frequency specifications are assumed to be in normalized frequency units. All magnitude specifications are assumed to be in decibels.

`d = fdesign.peak(...,Fs)` constructs a peak filter specification object while providing the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other values provided. If you specify an `Fs`, it is assumed to be in Hz, as all the other frequency values provided.

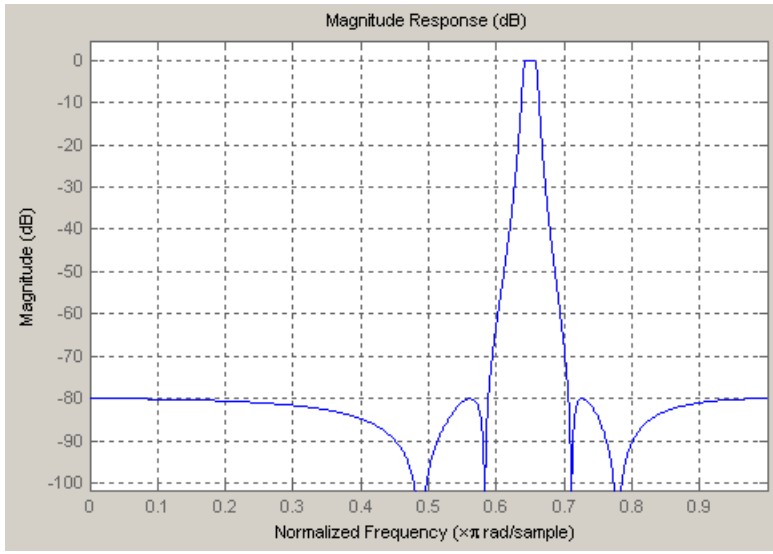
`d = fdesign.peak(...,MAGUNITS)` constructs a notch filter specification while providing the units for any magnitude specification given. `MAGUNITS` can be one of the following: `'linear'`, `'dB'`, or `'squared'`. If this argument is omitted, `'dB'` is assumed. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If `Fs` is provided, `MAGUNITS` must follow `Fs` in the input argument list.

Examples

Design a Chebyshev Type II peaking filter with a stopband attenuation of 80 dB:

```
d = fdesign.peak('N,F0,BW,Ast',8,.65,.02,80);
Hd = design(d,'cheby2');
fvtool(Hd)
```

This design produces a filter with the magnitude response shown in the following figure.



See Also

`fdesign`, `fdesign.notch`

Purpose

Construct polynomial sample-rate converter (POLYSRC) filter designer

Syntax

```
d = fdesign.polysrc(l,m)
d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np)
d = fdesign.polysrc(...,Fs)
```

Description

`d = fdesign.polysrc(l,m)` constructs a polynomial sample-rate converter filter designer D with an interpolation factor L and a decimation factor M. L defaults to 3. M defaults to 2. L and M can be arbitrary positive numbers.

`d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np)` initializes the filter designer specification with Np and sets the polynomial order to the value Np. If omitted Np defaults to 3.

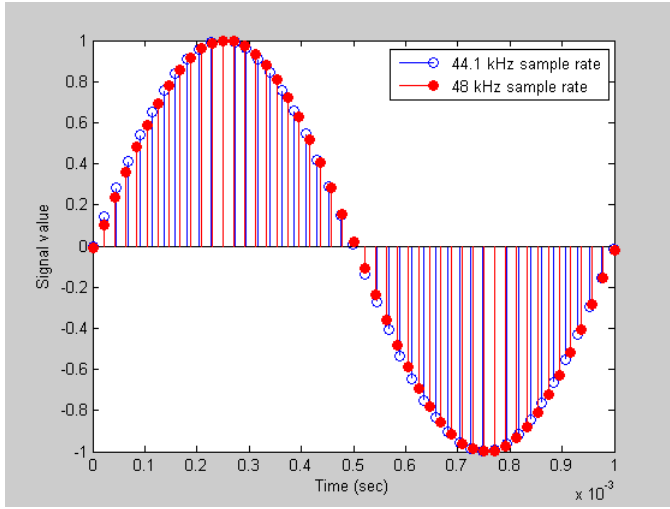
`d = fdesign.polysrc(...,Fs)` specifies the sampling frequency (in Hz).

Examples

This example shows how to design sample-rate converter that uses a 3rd order Lagrange interpolation filter to convert from 44.1kHz to 48kHz:

```
[L,M] = rat(48/44.1);
f = fdesign.polysrc(L,M,'Fractional Delay','Np',3);
Hm = design(f,'lagrange');
% Original sampling frequency
Fs = 44.1e3;
% 9408 samples, 0.213 seconds long
n = 0:9407;
% Original signal, sinusoid at 1kHz
x  = sin(2*pi*1e3/Fs*n);
% 10241 samples, still 0.213 seconds
y = filter(Hm,x);
% Plot original sampled at 44.1kHz
stem(n(1:45)/Fs,x(1:45))
hold on
% Plot fractionally interpolated signal (48kHz) in red
stem((n(3:51)-2)/(Fs*L/M),y(3:51),'r','filled')
xlabel('Time (sec)');ylabel('Signal value')
legend('44.1 kHz sample rate','48 kHz sample rate')
```

This code generates the following figure.



For more information about Farrow SRCs, see the “Efficient Sample Rate Conversion between Arbitrary Factors” demo.

See Also

`fdesign`

Purpose

Rational-factor sample-rate converter specification

Syntax

```
d = fdesign.rsrc(l,m)
d = fdesign.rsrc(...,design)
d = fdesign.rsrc(...,design,spec)
d = fdesign.rsrc(...,spec,specvalue1,specvalue2,...)
d = fdesign.rsrc(...,fs)
d = fdesign.rsrc(...,magunits)
```

Description

`d = fdesign.rsrc(l,m)` constructs a rational-factor sample-rate convertor filter specification object `d`, applying default values for the properties `tw` and `ast` and using the default design, Nyquist. Specify `l` and m, the interpolation and decimation factors, as integers.

`l/m` is the rational-factor for the rate change. When you omit the input argument `l` or `m` or both, `fdesign.rsrc` sets the values to defaults — the interpolation factor (if omitted) to 3 and the decimation factor (if omitted) to 2. The default rate change factor is 3/2 .

Using `fdesign.rsrc` with a design method generates an `mfilt` object.

`d = fdesign.rsrc(...,design)` constructs an rational-factor sample-rate converter with the interpolation factor `l`, decimation factor `m`, and the response you specify in `design`. Using the `design` input argument lets you choose the sort of filter that results from using the rational-factor sample-rate converter specifications object. `design` accepts the following strings that define the filter response.

`design` String	Description
`arbmag`	Sets the design for the rational-factor sample-rate converter specifications object to Arbitrary Magnitude.
`arbmagnphase`	Sets the design for the rational-factor sample-rate converter specifications object to Arbitrary Magnitude and Phase.

`design` String	Description
`bandpass`	Sets the design for the rational-factor sample-rate converter specifications object to bandpass.
`bandstop`	Sets the design for the rational-factor sample-rate converter specifications object to bandstop.
`cic`	Sets the design for the rational-factor sample-rate converter specifications object to CIC filter.
`ciccomp`	Sets the design for the rational-factor sample-rate converter specifications object to CIC compensator.
`halfband`	Sets the design for the rational-factor sample-rate converter specifications object to halfband.
`highpass`	Sets the design for the rational-factor sample-rate converter specifications object to highpass.
`isinclp`	Sets the design for the rational-factor sample-rate converter specifications object to inverse-sinc lowpass.
`lowpass`	Sets the design for the rational-factor sample-rate converter specifications object to lowpass.
`nyquist`	Sets the design for the rational-factor sample-rate converter specifications object to Nyquist.

`d = fdesign.rsrc(...,`*`design`*`,`*`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* depend on the design type of the specifications object.

When you add the `spec` input argument, you must also add the `design` input argument.

Because you are designing multirate filters, the specification strings available are not the same as the specifications for designing single-rate filters with such design methods as `fdesign.lowpass`. The strings are not case sensitive.

The interpolation factor `l` is not in the specification strings. Various design types provide different specifications. as shown in this table. In the third column, you see the filter design methods that apply to specifications objects that use the specification string in column two.

Design Type	Valid Specification Strings
`Arbitrary Magnitude`	• `n,f,a` (default string) • `n,b,f,a`
`Arbitrary Magnitude and Phase`	• `n,f,h` (default string) • `n,b,f,h`
`Bandpass`	• `fst1,fp1,fp2,fst2,ast1,ap,ast2` (default string) • `n,fc1,fc2` • `n,fst1,fp1,fp2,fst2`
`Bandstop`	• `n,fc1,fc2` • `n,fp1,fst1,fst2,fp2` • `fp1,fst1,fst2,fp2,ap1,ast,ap2` (default string)
`CIC`	• `fp,ast` (default and only string)

Design Type	Valid Specification Strings
`CIC Compensator`	• `fp,fst,ap,ast` (default string) • `n,fc,ap,ast` • `n,fp,ap,ast` • `n,fp,fst` • `n,fst,ap,ast`
`Halfband`	• `tw,ast` (default string) • `n,tw` • `n` • `n,ast`
`Highpass`	• `fst,fp,ast,ap` (default string) • `n,fc` • `n,fc,ast,ap` • `n,fp,ast,ap` • `n,fst,fp,ap` • `n,fst,fp,ast` • `n,fst,ast,ap` • `n,fst,fp`
`Inverse-Sinc Lowpass`	• `fp,fst,ap,ast` (default string) • `n,fc,ap,ast` • `n,fp,fst`

Design Type	Valid Specification Strings
`Lowpass`	• `fp,fst,ap,ast` (default string) • `n,fc` • `n,fc,ap,ast` • `n,fp,ap,ast` • `n,fp,fst` • `n,fp,fst,ap` • `n,fp,fst,ast` • `n,fst,ap,ast`
`Nyquist`	• `tw,ast` (default string) • `n,tw` • `n` • `n,ast`

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.
- `ap` — amount of ripple allowed in the pass band 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. Bandpass and bandstop filters use this option.
- `ap2` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass2. Bandpass and bandstop filters use this option.
- `ast` — attenuation in the first stop band in decibels (the default units). Also called Astop.

- `ast1` — attenuation in the first stop band in decibels (the default units). Also called Astop1. Bandpass and bandstop filters use this option.
- `ast2` — attenuation in the first stop band in decibels (the default units). Also called Astop2. Bandpass and bandstop filters use this option.
- `b` — number of bands in the multiband filter
- `f` — frequency vector. Frequency values in `f` specify locations where you provide specific filter response amplitudes. When you provide `f` you must also provide `a`.
- `fc1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. Bandpass and bandstop filters use this option.
- `fc2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. Bandpass and bandstop filters use this option.
- `fp1` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass1. Bandpass and bandstop filters use this option.
- `fp2` — frequency at the end of the pass band. Specified in normalized frequency units. Also called Fpass2. Bandpass and bandstop filters use this option.
- `fst1` — frequency at the end of the first stop band. Specified in normalized frequency units. Also called Fstop1. Bandpass and bandstop filters use this option.
- `fst2` — frequency at the start of the second stop band. Specified in normalized frequency units. Also called Fstop2. Bandpass and bandstop filters use this option.
- `h` — complex frequency response values.
- `n` — filter order.

- `tw` — width of the transition region between the pass and stop bands. Both halfband and Nyquist filters use this option.

`d = fdesign.rsrc(...,spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.rsrc(...,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.rsrc(...,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

This series of examples demonstrates progressively more complete techniques for creating rational sample-rate change filters. First, pass the filter design specifications directly to the Nyquist design type. Then use `kaiserwin`, one of the valid design methods, to design the rate change filter.

```
d = fdesign.rsrc(5,3,'nyquist',5,.05,40);
designmethods(d)
hm = design(d,'kaiserwin'); % Use Kaiser window to design
 rate changer.
```

For this example, specify the filter order (12) when you create the specifications object `d`.

```
d = fdesign.rsrc(5,3,'nyquist',5,'n,tw',12)
```

Expand the input arguments by specify a sampling frequency for the filter. Recall that the sampling frequency for rate changers refers to the input sample rate times the interpolation factor.

```
d = fdesign.rsrc(5,3,'nyquist',5,'n,tw',12,0.1,5)
designmethods(d);
design(d,'equiripple'); % Opens FVTool.
```

Specify a stopband ripple in linear units.

```
d = fdesign.rsrc(4,7,'nyquist',5,'tw,ast',.1,1e-3,5,...
'linear') % 1e-3 = 60dB attenuation in the stopband.
```

See Also

`design`, `designmethods`, `fdesign.decimator`, `fdesign.interpolator`, `setspecs`, `fdesign.arbmag`, `fdesign.arbmagnphase`

Purpose

Frequency-domain coefficients

Syntax

```
c = fftcoeffs(hd)
c = fftcoeffs(ha)
```

Description

`c = fftcoeffs(hd)` return the frequency-domain coefficients used when filtering with the `dfilt.fftfir` object. c contains the coefficients

`c = fftcoeffs(ha)` return the frequency-domain coefficients used when filtering with `adaptfilt` objects.

`fftcoeffs` applies to the following adaptive filter algorithms:

- `adaptfilt.fdaf`
- `adaptfilt.pbfdaf`
- `adaptfilt.pbufdaf`
- `adaptfilt.ufdaf`

Examples

This example demonstrates returning the FFT coefficients from the discrete-time filter `hd`.

```
b = [0.05 0.9 0.05];
len = 50;
hd = dfilt.fftfir(b,len)

hd =

        FilterStructure: 'Overlap-Add FIR'
              Numerator: [0.0500 0.9000 0.0500]
            BlockLength: 50
    NonProcessedSamples: []
       PersistentMemory: false

c=fftcoeffs(hd)

c =
```

```
 1.0000
 0.9920 + 0.1204i
 0.9681 + 0.2386i
 0.9289 + 0.3523i
 0.8753 + 0.4594i
 0.8084 + 0.5580i
 0.7297 + 0.6464i
 0.6408 + 0.7233i
 0.5435 + 0.7874i
 0.4398 + 0.8381i
 0.3317 + 0.8747i
 0.2211 + 0.8971i
 0.1099 + 0.9054i
      0 + 0.9000i
-0.1070 + 0.8815i
-0.2097 + 0.8506i
-0.3066 + 0.8084i
-0.3967 + 0.7558i
-0.4790 + 0.6939i
-0.5528 + 0.6240i
-0.6176 + 0.5472i
-0.6730 + 0.4645i
-0.7185 + 0.3771i
-0.7541 + 0.2860i
-0.7796 + 0.1921i
-0.7949 + 0.0965i
-0.8000
-0.7949 - 0.0965i
-0.7796 - 0.1921i
-0.7541 - 0.2860i
-0.7185 - 0.3771i
-0.6730 - 0.4645i
-0.6176 - 0.5472i
-0.5528 - 0.6240i
-0.4790 - 0.6939i
-0.3967 - 0.7558i
```

```
-0.3066 - 0.8084i
-0.2097 - 0.8506i
-0.1070 - 0.8815i
      0 - 0.9000i
 0.1099 - 0.9054i
 0.2211 - 0.8971i
 0.3317 - 0.8747i
 0.4398 - 0.8381i
 0.5435 - 0.7874i
 0.6408 - 0.7233i
 0.7297 - 0.6464i
 0.8084 - 0.5580i
 0.8753 - 0.4594i
 0.9289 - 0.3523i
 0.9681 - 0.2386i
 0.9920 - 0.1204i
```

Similarly, you can use `fftcoeffs` with the adaptive filters algorithms listed above. Start by constructing an adaptive filter `ha`.

```
d  = 16;                    % Number of samples of delay.
b  = exp(j*pi/4)*[-0.7 1];  % Numerator coefficients of channel.
a  = [1 -0.7];              % Denominator coefficients of channel.
ntr= 1000;                  % Number of iterations.
s  = sign(randn(1,ntr+d)) +...
j*sign(randn(1,ntr+d));     % Baseband QPSK signal.
n  = 0.1*(randn(1,ntr+d) + j*randn(1,ntr+d));  % Noise signal.
r  = filter(b,a,s)+n;       % Received signal.
x  = r(1+d:ntr+d);          % Input signal (received signal).
d  = s(1:ntr);              % Desired signal (delayed QPSK signal).
del = 1;                    % Initial FFT input powers.
mu  = 0.1;                  % Step size.
lam = 0.9;                  % Averaging factor.
d   = 8;                    % Block size.
ha = adaptfilt.pbufdaf(32,mu,1,del,lam,n);
```

Here are the coefficients before you filter a signal.

```
c=fftcoeffs(ha)

c =

  Columns 1 through 13

   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0

  Columns 14 through 16

      0     0     0
      0     0     0
      0     0     0
      0     0     0
```

Filtering a signal y produces complex nonzero coefficients that you use `fftcoeffs` to see.

```
[y,e] = filter(ha,x,d);
c=fftcoeffs(ha)

c =

  Columns 1 through 4

  0.1425 - 0.0957i  0.0487 - 0.0503i -0.0479 + 0.0315i  0.0769 - 0.0435i
  0.7264 - 0.7605i -0.7423 - 0.6382i  0.1758 + 0.6679i  0.2018 - 0.6544i
  0.1604 + 0.0747i -0.0709 + 0.2610i -0.1634 + 0.2929i -0.1488 + 0.3610i
 -0.0396 + 0.0416i  0.0985 + 0.0095i  0.0733 + 0.0011i  0.0700 + 0.0348i

  Columns 5 through 8

 -0.0604 + 0.1767i  0.0732 - 0.0648i -0.0870 + 0.0383i  0.0298 - 0.0852i
 -0.1665 + 0.3741i  0.3174 - 0.5234i -0.1990 + 0.4150i  0.3657 - 0.4760i
```

```
-0.2198 + 0.4273i -0.2690 + 0.3981i -0.2820 + 0.3095i -0.3633 + 0.3517i
-0.0537 - 0.0855i -0.0190 + 0.0336i  0.0091 - 0.0061i -0.0299 + 0.0001i

 Columns 9 through 12

-0.0437 + 0.0676i  0.0499 - 0.0164i -0.0397 + 0.0165i  0.0455 - 0.0085i
-0.3293 + 0.3076i  0.4986 - 0.3949i -0.3300 + 0.3448i  0.5492 - 0.2633i
-0.2671 + 0.3238i -0.3813 + 0.2999i -0.4130 + 0.2333i -0.2910 + 0.2823i
-0.0300 + 0.0236i -0.0103 + 0.0438i  0.0244 + 0.0476i  0.1043 + 0.0359i

 Columns 13 through 16

-0.0602 + 0.1189i -0.0227 - 0.1076i -0.0282 + 0.0634i  0.0170 - 0.0464i
-0.4385 + 0.0549i  0.5232 - 0.1904i -0.6414 - 0.1717i  0.5580 + 0.6477i
-0.4511 + 0.3217i -0.4301 + 0.1765i -0.2805 + 0.1270i -0.2531 + 0.0299i
 0.1076 - 0.0383i -0.0166 + 0.0020i  0.0004 - 0.0376i  0.0071 - 0.0714i
```

See Also

adaptfilt.fdaf, adaptfilt.pbfdaf, adaptfilt.pbufdaf, adaptfilt.ufdaf

Purpose

Filter data with filter object

Syntax

Fixed-Point Filter Syntaxes

```
y = filter(hd,x)
y = filter(hd,x,dim)
```

Adaptive Filter Syntax

```
y = filter(ha,x,d)
[y,e] = filter(ha,x,d)
```

Multirate Filter Syntax

```
y = filter(hm,x)
y = filter(hm,x,dim)
```

Description

This reference page contains three sections that describe the syntaxes for the filter objects:

- Fixed-Point Filter Syntaxes
- “Adaptive Filter Syntaxes” on page 2-677
- “Multirate Filter Syntaxes” on page 2-678

Fixed-Point Filter Syntaxes

`y = filter(hd,x)` filters a vector of real or complex input data `x` through a fixed-point filter `hd`, producing filtered output data `y`. The vectors `x` and `y` have the same length. filter stores the final conditions for the filter in the `States` property of `hd` — `hd.states`.

When you set the property `PersistentMemory` to `false` (the default setting), the initial conditions for the filter are set to zero before filtering starts. To use nonzero initial conditions for `hd`, set `PersistentMemory` to `true`. Then set `hd.states` to a vector of `nstates(hd)` elements, one element for each state to set. If you specify a scalar for `hd.states`, `filter` expands the scalar to a vector of the proper length for the states. All elements of the expanded vector have the value of the scalar.

If `x` is a matrix, `y = filter(hd,x)` filters along each column of `x` to produce a matrix `y` of independent channels. If `x` is a multidimensional

array, `y = filter(hd,x)` filters `x` along the first nonsingleton dimension of `x`.

To use nonzero initial conditions when you are filtering a matrix `x`, set the filter states to a matrix of initial condition values. Set the initial conditions by setting the `States` property for the filter (`hd.states`) to a matrix of `nstates(hd)` rows and `size(x,2)` columns.

`y = filter(hd,x,dim)` applies the filter `hd` to the input data located along the specific dimension of `x` specified by `dim`.

When you are filtering multichannel data, `dim` lets you specify which dimension of the input matrix to filter along — whether a row represents a channel or a column represents a channel. When you provide the `dim` input argument, the filter operates along the dimension specified by `dim`. When your input data `x` is a vector or matrix and `dim` is 1, each column of `x` is treated as a one input channel. When `dim` is 2, the filter treats each row of the input `x` as a channel.

To filter multichannel data in a loop environment, you must use the `dim` input argument to set the proper processing dimension.

You specify the initial conditions for each channel individually, when needed, by setting `hm.states` to a matrix of `nstates(hd)` rows (one row containing the states for one channel of input data) and `size(x,2)` columns (one column containing the filter states for each channel).

Adaptive Filter Syntaxes

`y = filter(ha,x,d)` filters a vector of real or complex input data `x` through an adaptive filter object `ha`, producing the estimated desired response data `y` from the process of adapting the filter. The vectors `x` and `y` have the same length. Use `d` for the desired signal. Note that `d` and `x` must be the same length signal chains.

`[y,e] = filter(ha,x,d)` produces the estimated desired response data `y` and the prediction error `e` (refer to previous syntax for more information).

Multirate Filter Syntaxes

`y = filter(hd,x)` filters a vector of real or complex input data `x` through a fixed-point filter `hd`, producing filtered output data `y`. The vectors `x` and `y` have the same length. filter stores the final conditions for the filter in the `States` property of `hd` — `hd.states`.

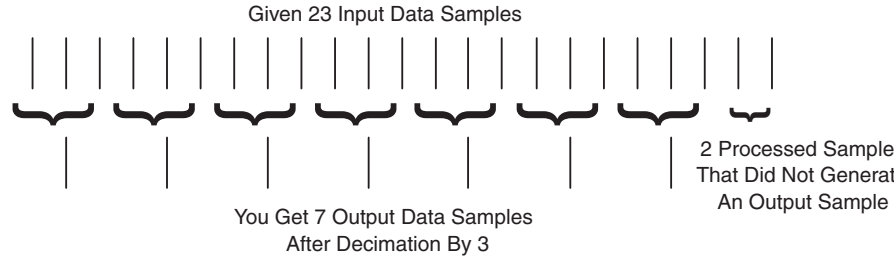
`y = filter(hm,x,dim)` applies the filter `hd` to the input data located along the specific dimension of `x` specified by `dim`.

When you are filtering multichannel data, `dim` lets you specify which dimension of the input matrix to filter along — whether a row represents a channel or a column represents a channel. When you provide the `dim` input argument, the filter operates along the dimension specified by `dim`. When your input data `x` is a vector or matrix and `dim` is 1, each column of `x` is treated as a one input channel. When `dim` is 2, the filter treats each row of the input `x` as a channel.

To filter multichannel data in a loop environment, you must use the `dim` input argument to set the processing dimension.

You specify the initial conditions for each channel individually, when needed, by setting `hm.states` to a matrix of `nstates(hm)` rows (one row containing the states for one channel of input data) and `size(x,2)` columns (one column containing the filter states for each channel).

The number of data samples in your input data set `x` does not need to be a multiple of the rate change factor `r` for the object. When the rate change factor is not an even divisor of the number of input samples `x`, `filter` processes the samples as shown in the following figure, where the rate change factor is 3 and the number of input samples is 23. Decimators always take the first input sample to generate the first output sample. After that, the next output sample comes after each `r` number of input samples.



Examples

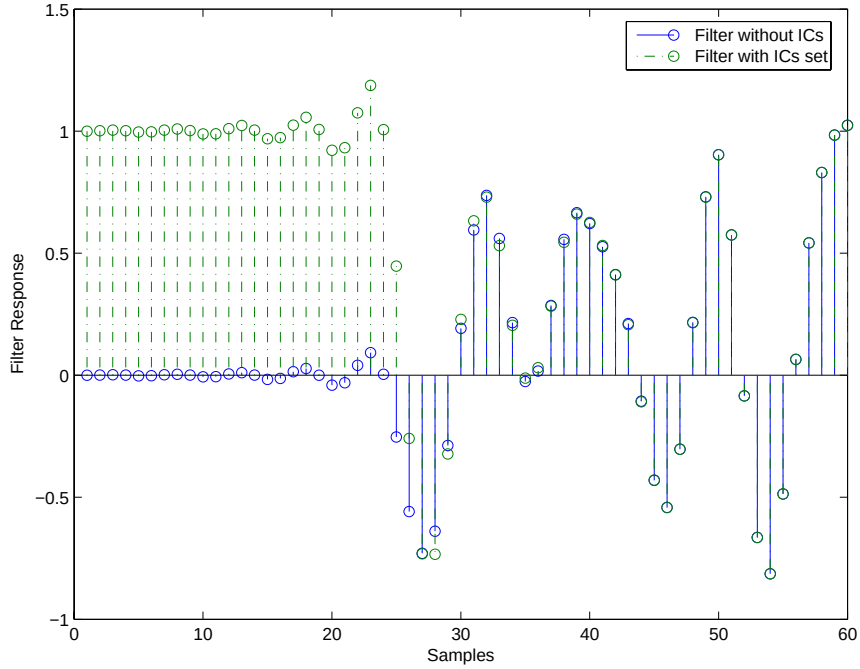
Filter a signal using a filter with various initial conditions (IC) or no initial conditions.

```
x = randn(100,1);    % Original signal.
b = fir1(50,.4);     % 50th-order linear-phase FIR filter.
hd = dfilt.dffir(b);    % Direct-form FIR implementation.

% Do not set specific initial conditions.

y1 = filter(hd,x);   % 'PersistentMemory'='false'(default).
zf = hd.states;      % Final conditions.
```

Now use nonzero initial conditions by setting ICs after before you filter.

```
hd.persistentmemory = true;
hd.states = 1;      % Uses scalar expansion.
y2 = filter(hd,x);
stem([y1 y2])       % Different sequences at beginning.
```

Looking at the stem plot shows that the sequences are different at the beginning of the filter process.



Here is one way to use `filter` with streaming data.

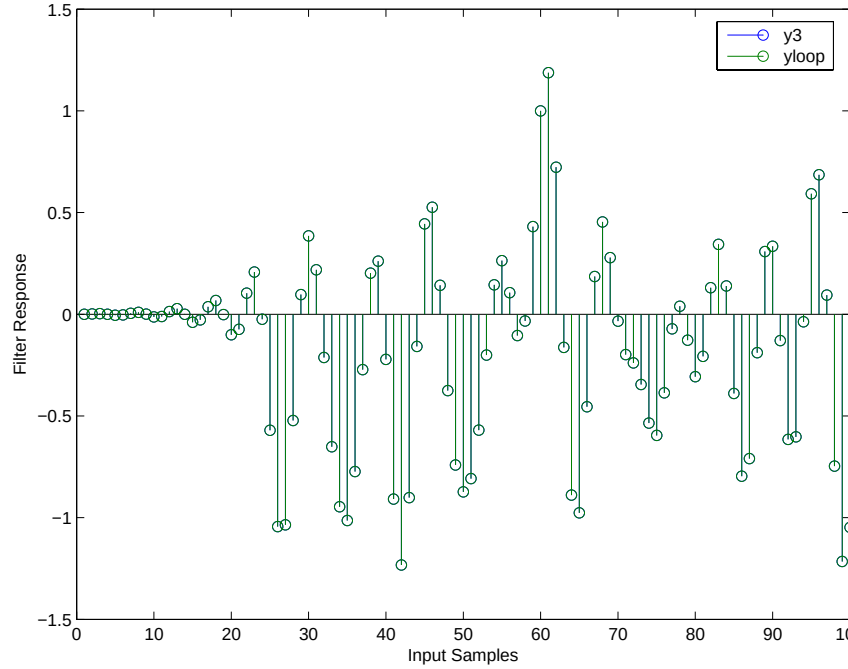
```
reset(hd);           % Clear filter history.
y3 = filter(hd,x);   % Filter entire signal in one block.
```

As an experiment, repeat the process, filtering the data as sections, rather than in streaming form.

```
reset(hd);              % Clear filter history.
yloop = zeros(100,1)    % Preallocate output array.
xblock = reshape(x,[20 5]);
for i=1:5,
   yloop = [yloop; filter(hd,xblock(:,i))];
end
```

Use a stem plot to see the comparison between streaming and block-by-block filtering.

```
stem([y3 yloop]);
```



Filtering the signal section-by-section is equivalent to filtering the entire signal at once.

To show the similarity between filtering with discrete-time and with multirate filters, this example demonstrates multirate filtering.

```
Fs = 44.1e3;             % Original sampling frequency: 44.1kHz.
n = [0:10239].';         % 10240 samples, 0.232 second long signal.
x  = sin(2*pi*1e3/Fs*n); % Original signal, sinusoid at 1kHz.
m = 2;                   % Decimation factor.
hm = mfilt.firdecim(m);  % Use the default filter.
```

First, filter without setting initial conditions.

```
y1 = filter(hm,x);       % PersistentMemory is false (default).
zf = hm.states;          % Final conditions.
```

This time, set nonzero initial conditions before filtering the data.

```
hm.persistentmemory = true;
hm.states = 1;       % Uses scalar expansion to set ICs.
y2 = filter(Hm,x);
stem([y1(1:60) y2(1:60)]) % Show the filtering results.
```

Note the different sequences at the start of filtering.

Finally, try filtering streaming data.

```
reset(hm);          % Clear the filter history.
y3 = filter(hm,x);  % Filter entire signal in one block.
```

As with the discrete-time filter, filtering the signal section by section is equivalent to filtering the entire signal at once.

```
reset(hm);            % Clear filter history again.
yloop = zeros(100,1)  % Preallocate output array.
xblock = reshape(x,[2048 5]);
for i=1:5,
   yloop = [yloop; filter(Hm,xblock(:,i))];end
```

Algorithm

Quantized Filters

The `filter` command implements fixed- or floating-point arithmetic on the quantized filter structure you specify.

The algorithm applied by `filter` when you use a discrete-time filter object on an input signal depends on the response you chose for the filter, such as lowpass or Nyquist or bandstop. To learn more about each filter algorithm, refer to the literature reference provided on the appropriate discrete-time filter reference page.

Note `dfilt/filter` does not normalize the filter coefficients automatically. Function `filter` supplied by MATLAB does normalize the coefficients.

Adaptive Filters

The algorithm used by `filter` when you apply an adaptive filter object to a signal depends on the algorithm you chose for your adaptive filter. To learn more about each adaptive filter algorithm, refer to the literature reference provided on the appropriate `adaptfilt.`*`algorithm`* reference page.

Multirate Filters

The algorithm applied by `filter` when you apply a multirate filter objects to signals depends on the algorithm you chose for the filter — the form of the multirate filter, such as decimator or interpolator. To learn more about each filter algorithm, refer to the literature reference provided on the appropriate multirate filter reference page.

See Also

`adaptfilt`, `impz`, `mfilt`, `nstates`

`dfilt` in Signal Processing Toolbox™ documentation

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

filterbuilder

Purpose

GUI-based filter design

Syntax

```
filterbuilder('response')
filterbuilder(h)
```

Description

`filterbuilder('response')` opens the filter design dialog box to design a filter with the specified response. Enter the string to specify the response surrounded by single quotes.

Response String	Description of Resulting Filter Design
`arbmag`	Arbitrary magnitude and phase filter
`bandpass` or `bp`	Bandpass filter
`bandstop` or `bs`	Bandstop filter
`cic`	CIC filter
`ciccomp`	CIC compensator
`diff`	Differentiator filter
`fracdelay`	Fractional delay filter
`halfband` or `hb`	Halfband filter
`highpass` or `hp`	Highpass filter
`hilb`	Hilbert filter
`isinclp`	Inverse sinc lowpass filter
`lowpass` or `lp`	Lowpass filter (default)
`notch`	Notch filter
`nyquist`	Nyquist filter
`octave`	Octave filter

Response String	Description of Resulting Filter Design
`parameq`	Parametric Equalizer filter
`peak`	Peak filter

`filterbuilder(h)` launches the appropriate filter design dialog box for the filter object `h`. For example, when `h` is a bandpass filter, `filterbuilder(h)` opens the bandpass filter design dialog box.

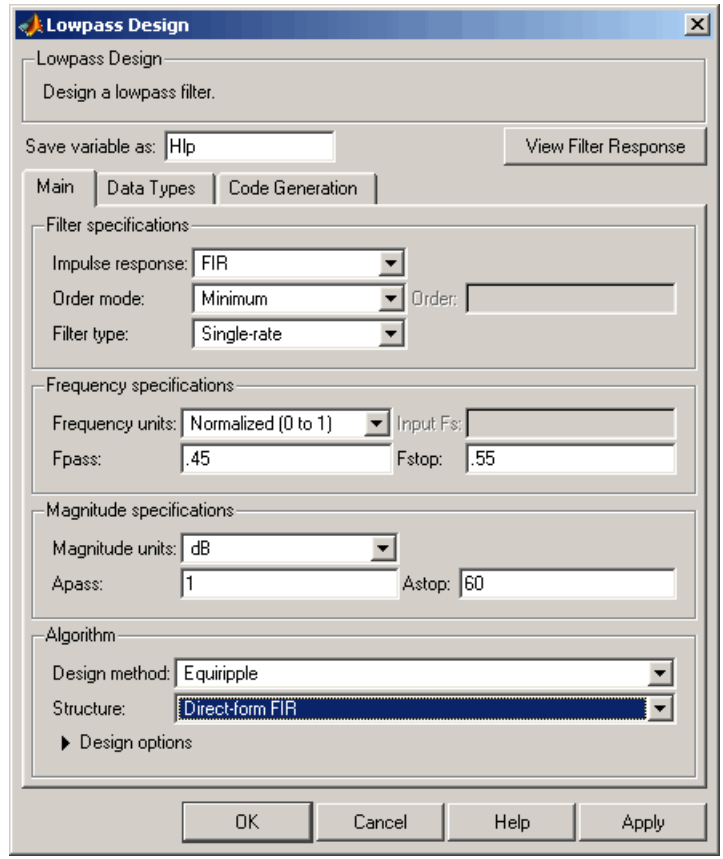
To use this syntax to edit or change a filter `h`, you must have used `filterbuilder` to design `h` or `h` must be a `dfilt` or `mfilt` object.

`filterbuilder` provides a graphical interface to the `fdesign` filter design methods, providing the same capabilities for design in an interactive environment.

Note Because they do not change the filter structure, the magnitude specifications and design method are tunable when using `filterbuilder`.

Filterbuilder Dialog Box

Although the main pane of the filterbuilder dialog box varies depending on the filter response type, 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.

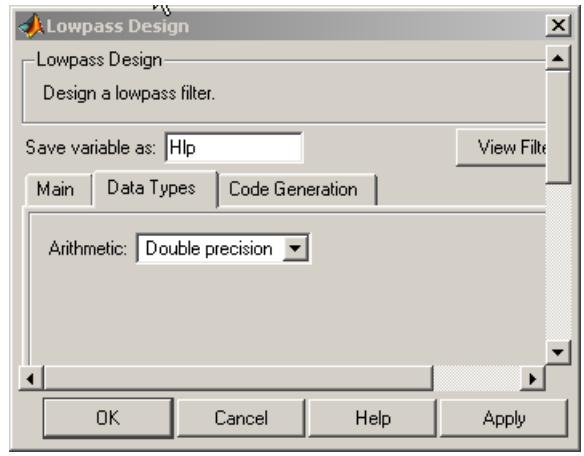
- **Save variable as** — When you click **Apply** to apply your changes or **OK** to close this dialog box, `filterbuilder` saves the current filter to your MATLAB workspace as a filter object with the name you enter.
- **View Filter Response** — Displays the magnitude response for the current filter specifications and design method by opening the Filter Visualization Tool (`fvtool`) from Signal Processing Toolbox™ software. For more information about FVTool, refer to Signal Processing Toolbox documentation.

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.

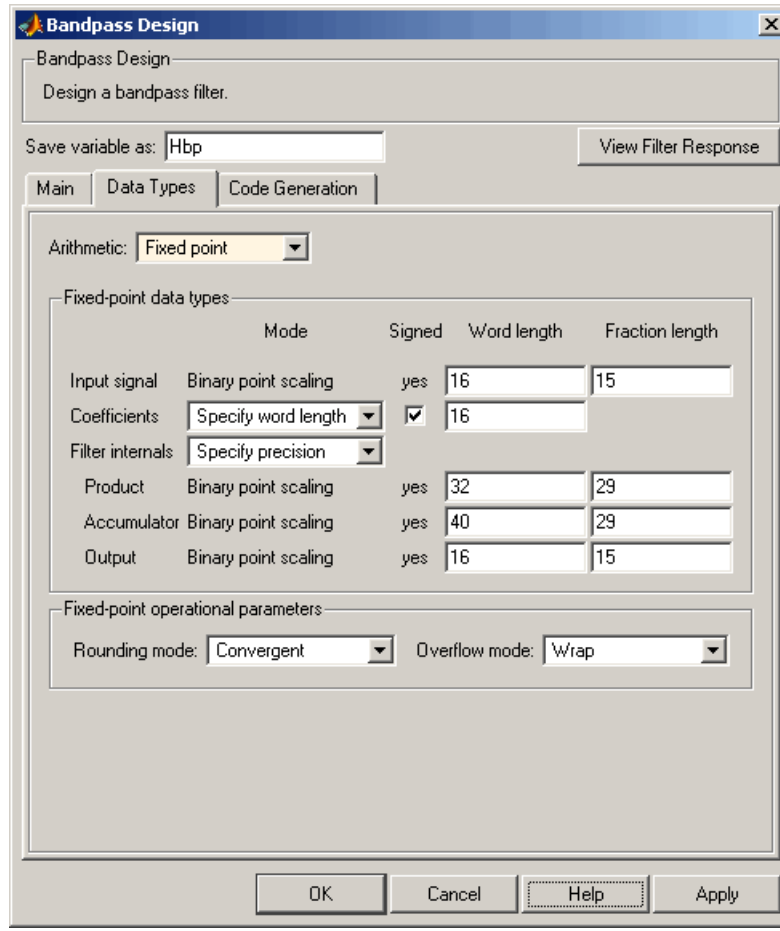


The **Arithmetic** drop down box allows the choice of `Double precision`, `Single precision`, or `Fixed point`. Some of these options may be unavailable depending on the filter parameters. The following table describes these options.

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 create a filter, `double precision` is the default value for the Arithmetic property.

Arithmetic List Entry	Effect on the Filter
`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 \&tm_fixedpointtoolbox; software Toolbox along with Filter Design Toolbox™ software.

The following figure shows the **Data Types** pane after you select `Fixed point` for **Arithmetic**.



Not all parameters described in the following section apply to all filters. For example, FIR filters do not have the **Section input** and **Section output** parameters.

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).

- `Ceiling` — Round up to the next allowable quantized value.
- `Convergent` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.
- `Zero` — Round negative numbers up and positive numbers down to the next allowable quantized value.
- `Floor` — Round down to the next allowable quantized value.
- `Nearest` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.

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.

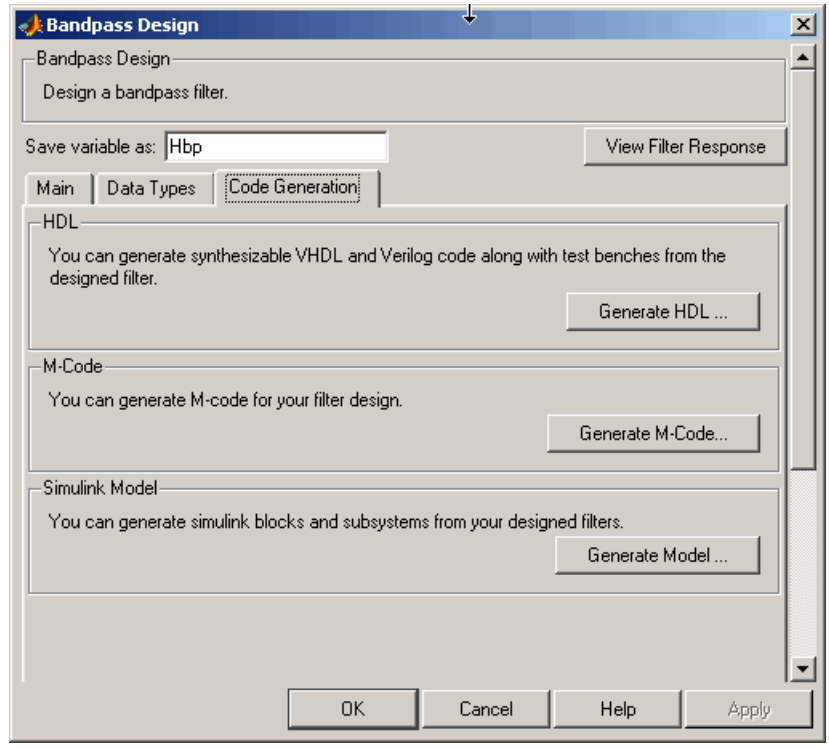
The effect of clearing or selecting **Cast before sum** is as follows:

- 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. You can generate VHDL and Verilog

code from the designed filter. You can generated M-Code. You can also choose to create or update a Simulink® model from the designed filter. The following section explains these options.



HDL

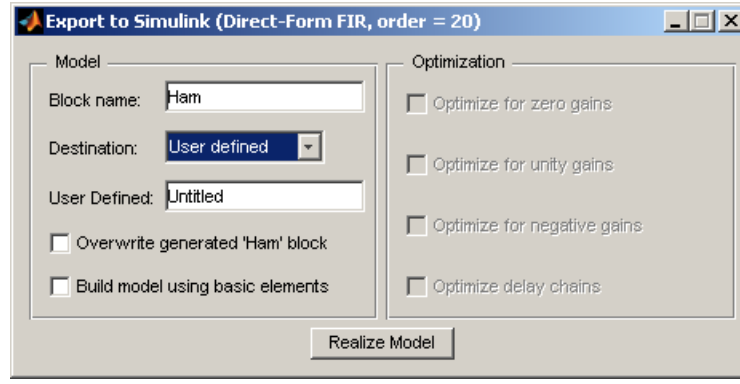
For more information on this option, see “Opening the Generate HDL Dialog Box from the filterbuilder GUI” documentation, where all the parameters on the sub dialog box are explained in detail.

M-Code

Clicking on the **Generate M-Code** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable M-file.

Simulink Model

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, Filter Design Toolbox software overwrites an existing block with the name specified in **Block Name**; when cleared, creates a new block with the same name.

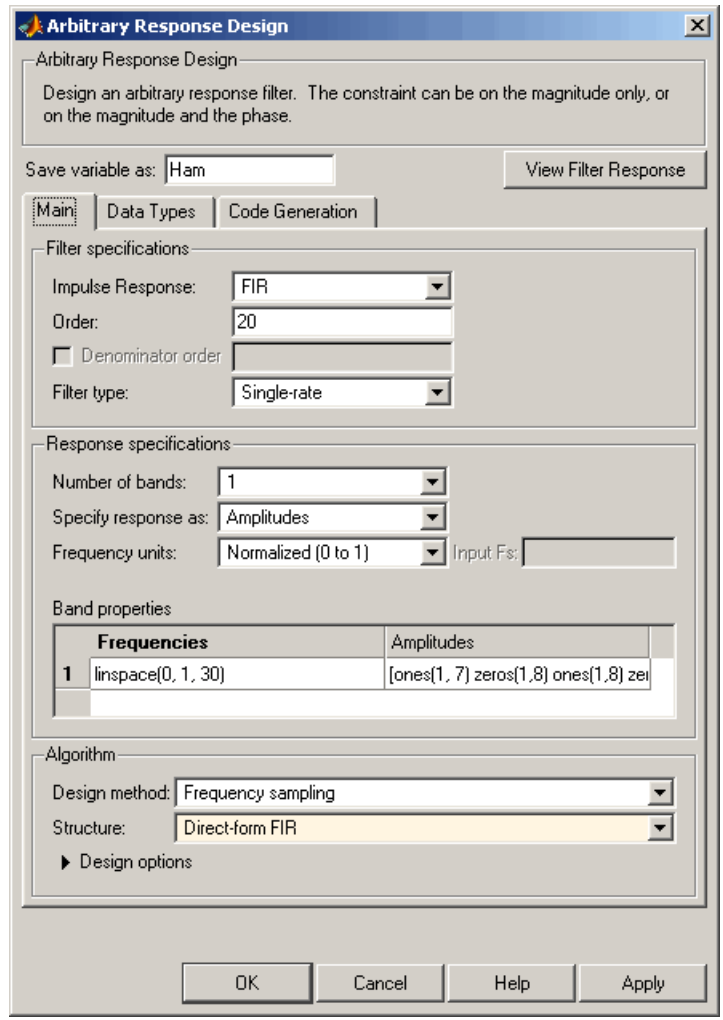
- **Build model using basic elements** — When this check box is selected, Filter Design Toolbox software builds the model using only basic blocks.
- **Optimize for zero gains** — When this check box is selected, Filter Design Toolbox software removes all zero gain blocks from the model.
- **Optimize for unity gains** — When this check box is selected, Filter Design Toolbox software replaces all unity gains with direct connections.
- **Optimize for negative gains** — When this check box is selected, Filter Design 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, Filter Design Toolbox software replaces cascaded delay blocks with a single integer delay block with an equivalent total delay.
- **Realize Model** — Filter Design Toolbox software builds the model with the set parameters.

Main Pane

Most of this pane contains parameters specific to the filter type. These are described in detail in the following sections:

- "Arbitrary Response Design Dialog Box — Main Pane" on page 2-700
- "Bandpass Filter Design Dialog Box — Main Pane" on page 2-704
- "Bandstop Filter Design Dialog Box — Main Pane" on page 2-712
- "CIC Filter Design Dialog Box — Main Pane" on page 2-720
- "CIC Compensator Filter Design Dialog Box — Main Pane" on page 2-723
- "Differentiator Filter Design Dialog Box — Main Pane" on page 2-729

Arbitrary Response 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 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.

Order

Enter the order for FIR filter, or the order of the numerator for the IIR filter.

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 option is available for FIR filters only. 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.

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`, or `Frequency response`.

Frequency units

Specify frequency units as either `Normalized`, which means normalized by the input sampling frequency, or select from `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 the frequency units are selected.

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 the response chosen in the **Specify response as** drop-down box is `Amplitudes`.
- **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

Design Method

Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

Structure

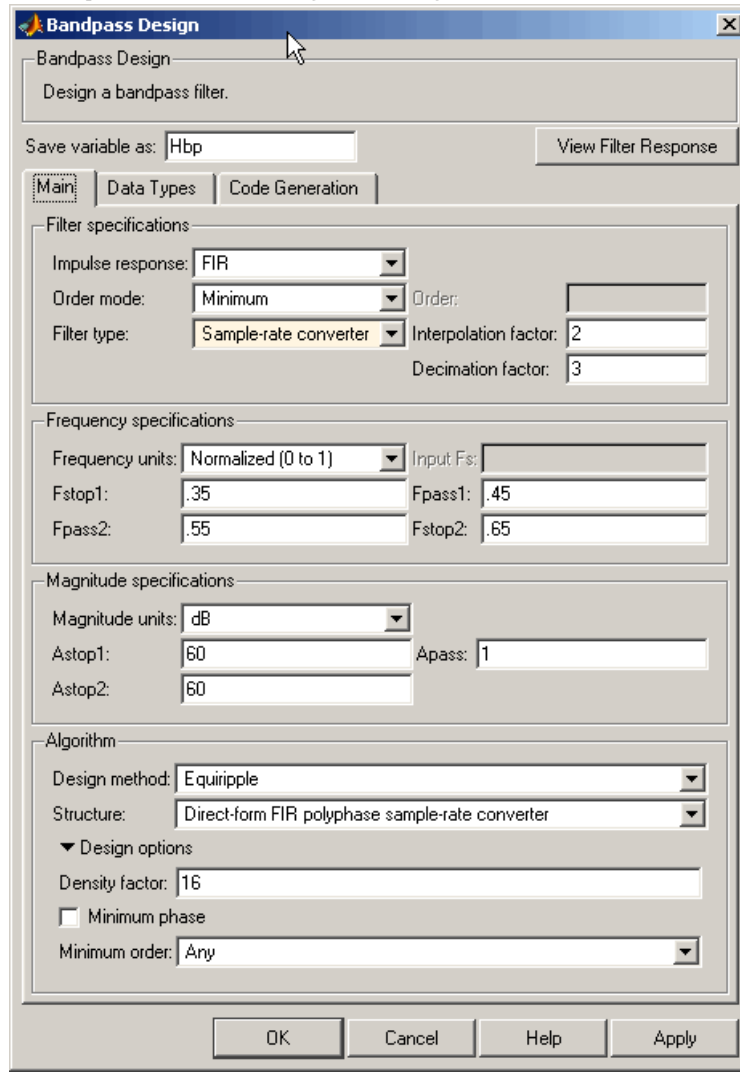
Select the structure for the filter, available for the design method selected in the previous box.

Design Options

Available for some design methods, these options usually include the following:

- **Density factor** — Controls the density of the frequency grid over which the design method optimization evaluates your filter response function
- **Weights** — Controls the relative importance applied to meeting the error specification in each band, telling the design algorithm how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. This vector must have the same number of elements as the frequencies vector specified in **Band properties**.

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 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.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down box. Selecting `Specify` enables the **Order** option (explained in the following descriptions) 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 **Filter order mode**.

Decimation Factor

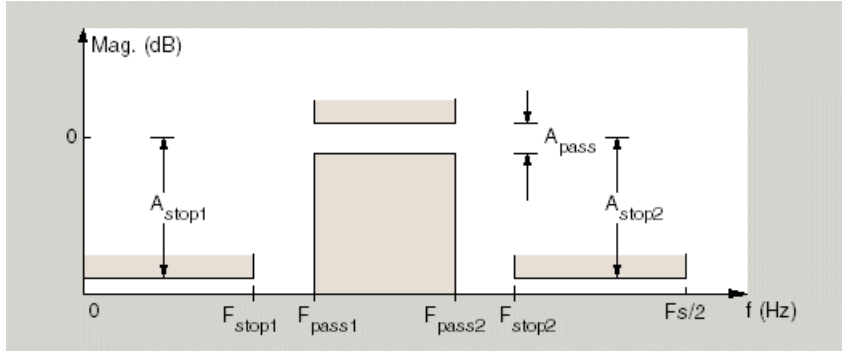
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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.

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 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.
- `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.

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 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.

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.

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 20 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:

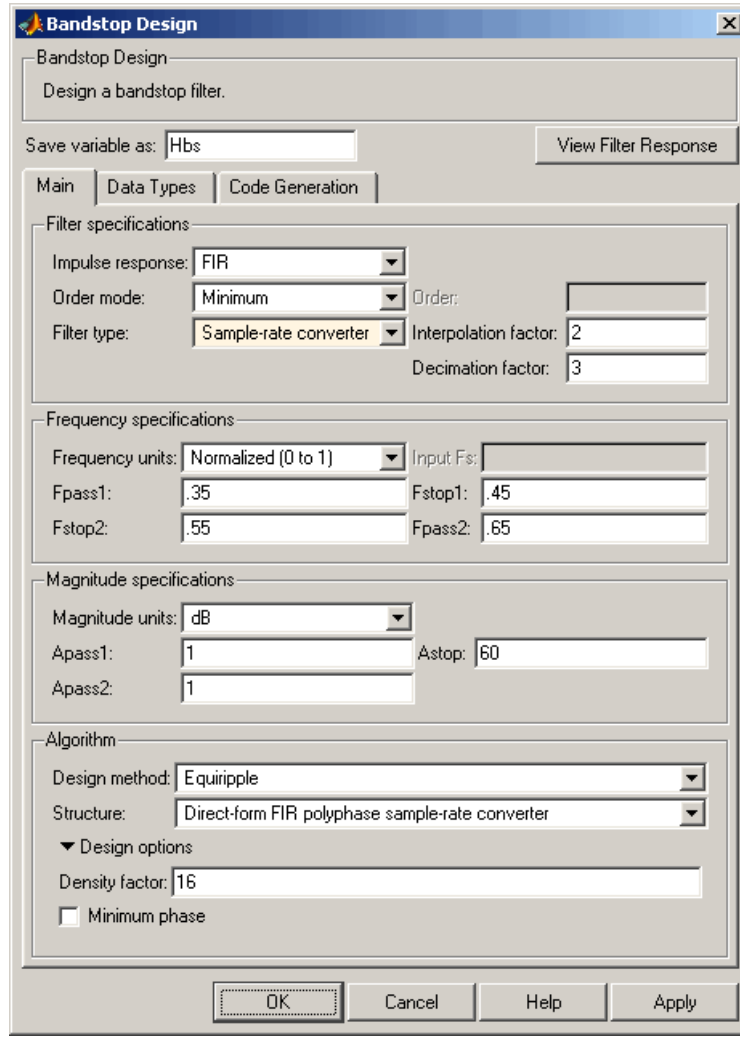
- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband 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.

Bandstop 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 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.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

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

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 **Filter order mode**.

Decimation Factor

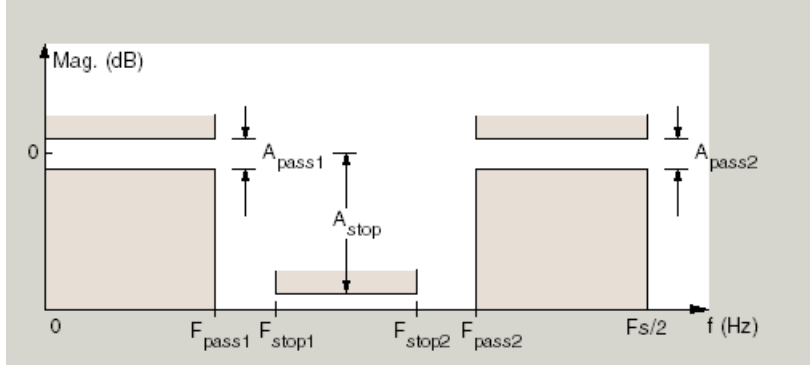
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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. 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 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.
- `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.

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 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.

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.

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 20 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:

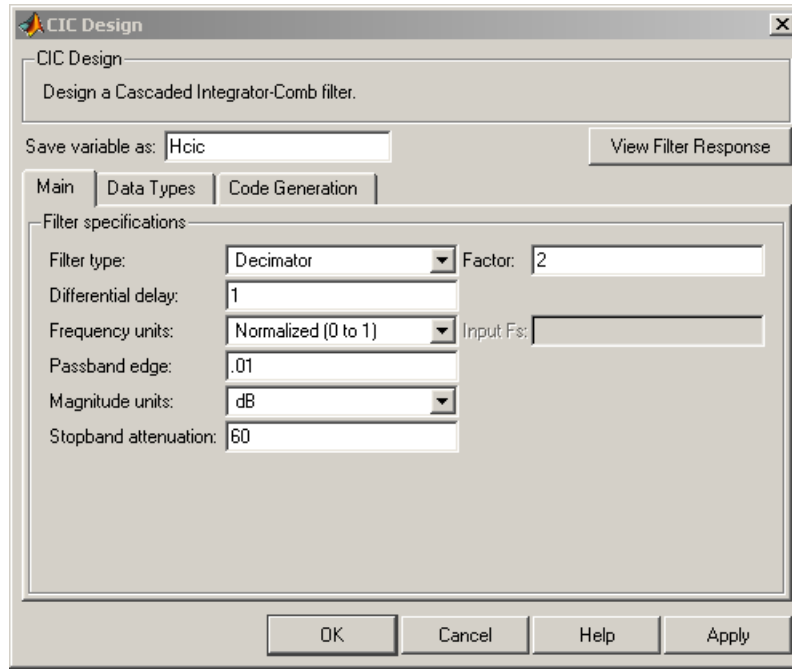
- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband 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.

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. The default value is `1`. Most CIC filters use 1 or 2. Differential delay changes both the shape and number of nulls in the filter response. The delay value also affects the null locations. Increasing the delay increases the number and sharpness of the nulls and response between nulls. Generally, 1 or 2 work best as values for the delay.

Factor

When you select `decimator` or `interpolator` for **Filter type**, enter the decimation or interpolation factor for your filter in this field. You must enter a positive integer for the factor. The default factor value is 2.

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 units

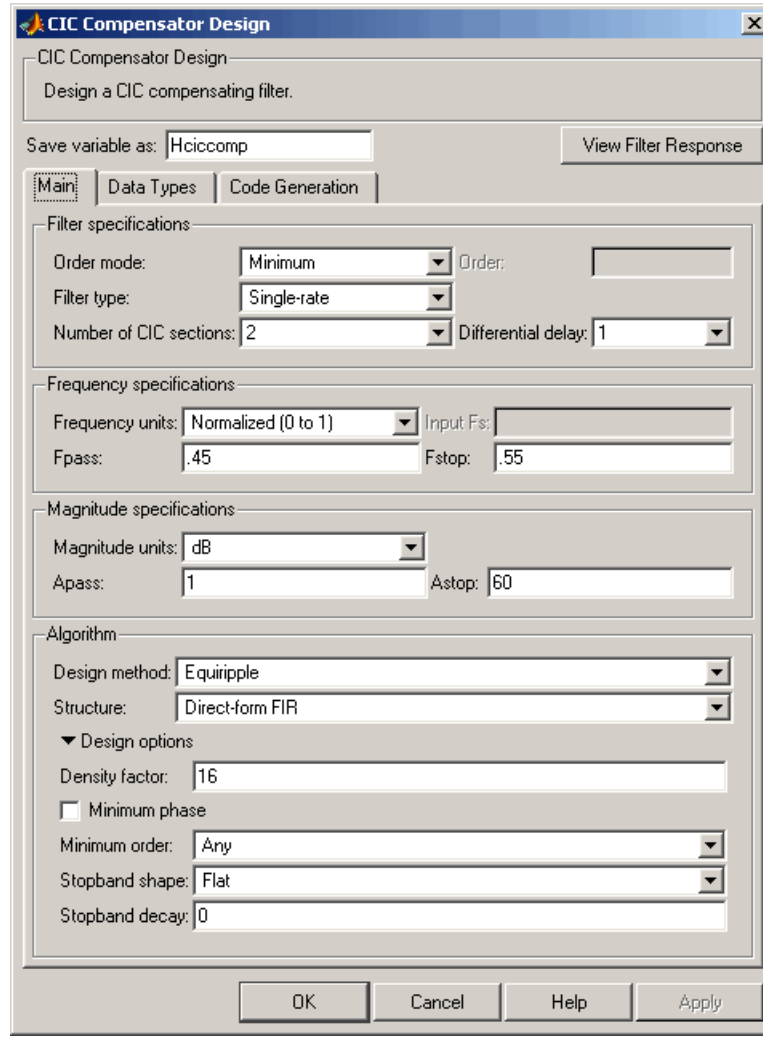
Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `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.

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.

Filter order mode

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

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 **Filter 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.

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.

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 20 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**.

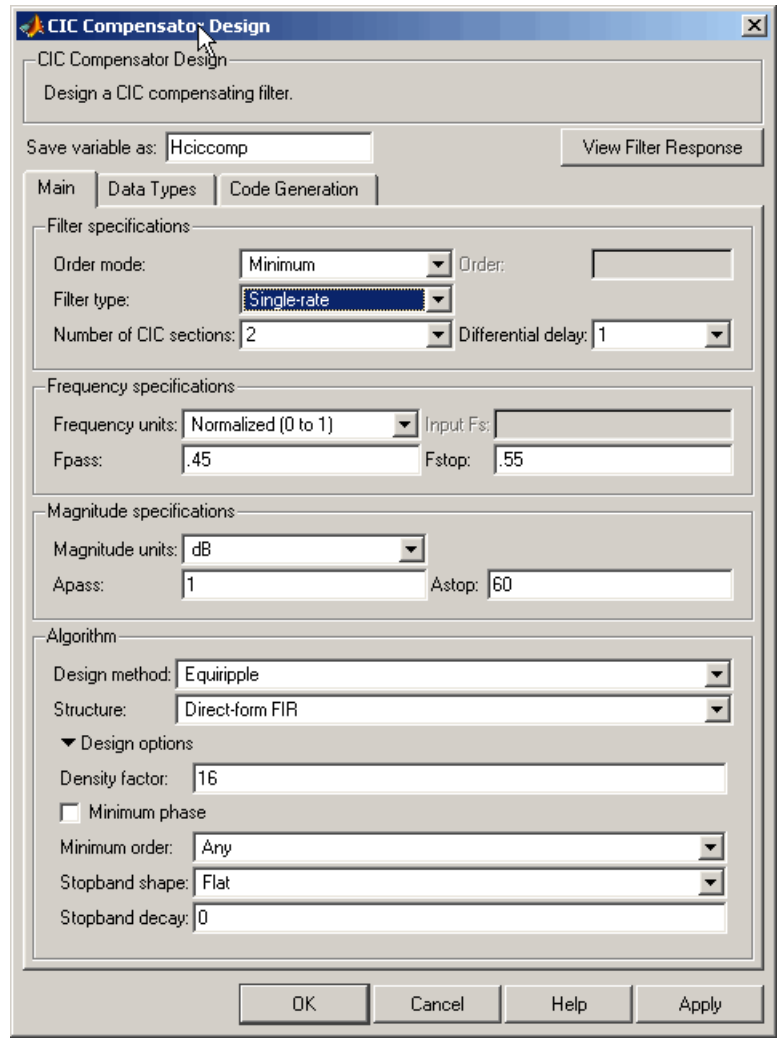
Stopband Decay

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.

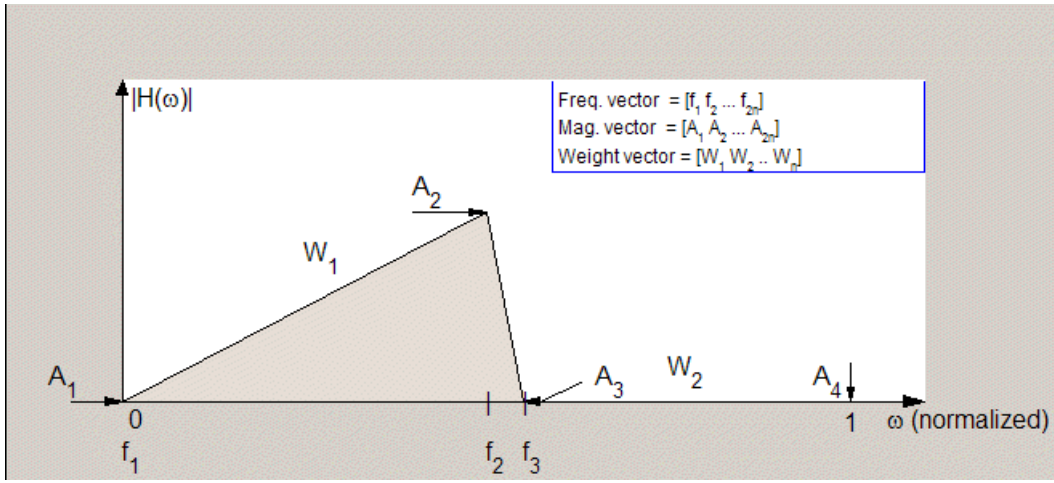
`filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Differentiator 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. 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.

Filter order mode

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

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 **Filter 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 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.

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.

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.

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 20 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**.

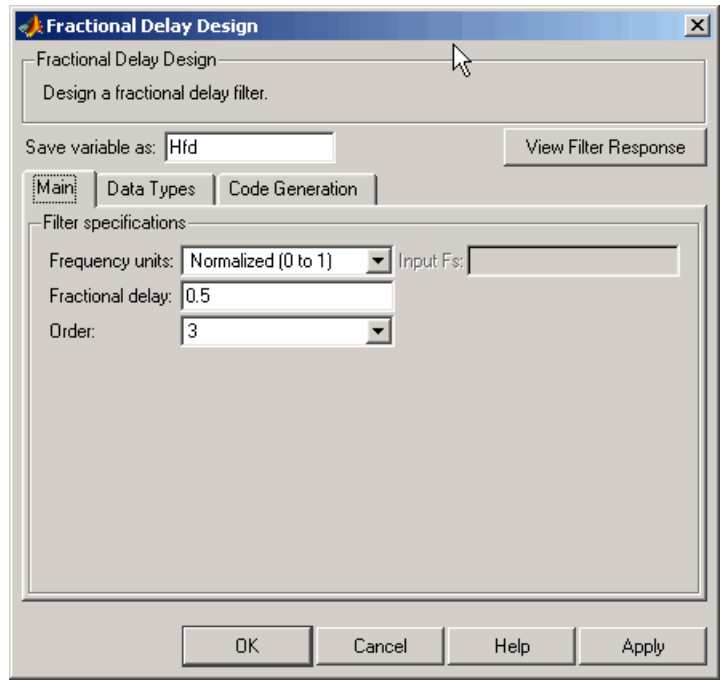
Stopband Decay

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.

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 **Filter 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.

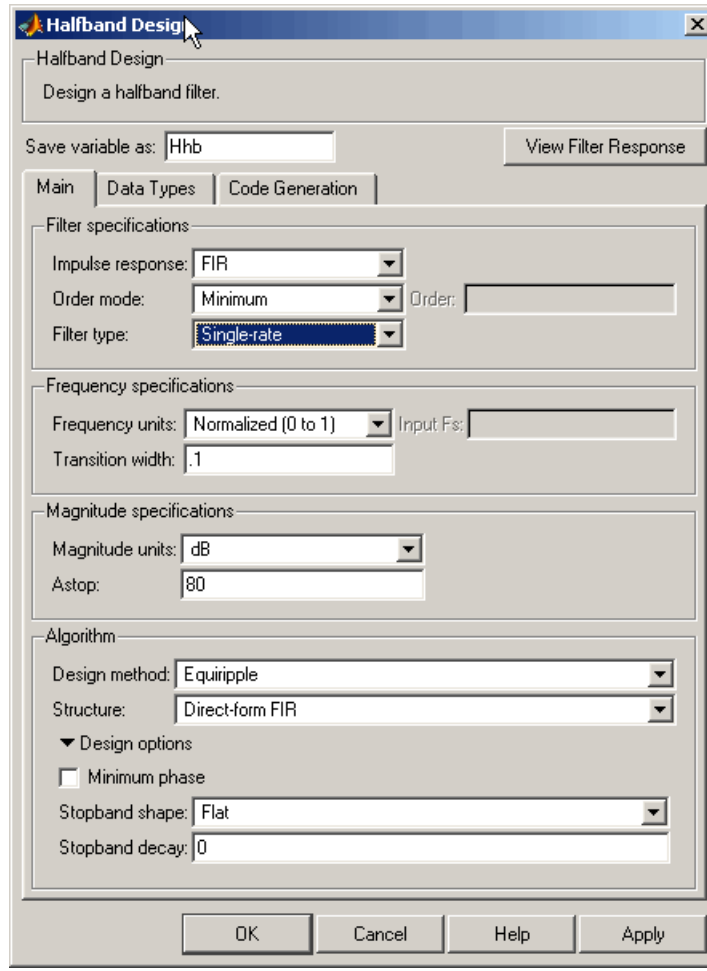
Frequency units

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



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 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.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

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

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 **Filter order mode**.

Decimation Factor

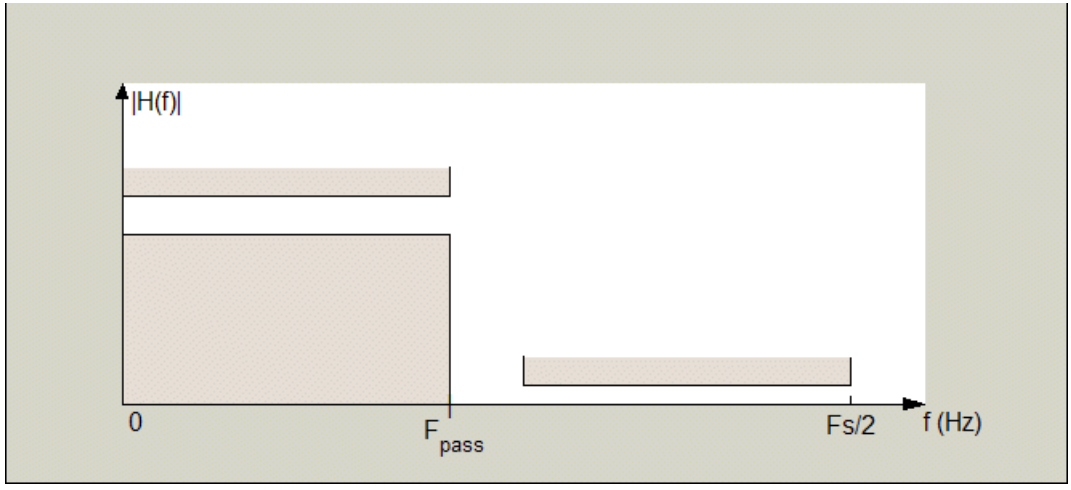
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 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).
- `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.

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.

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 20 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**.

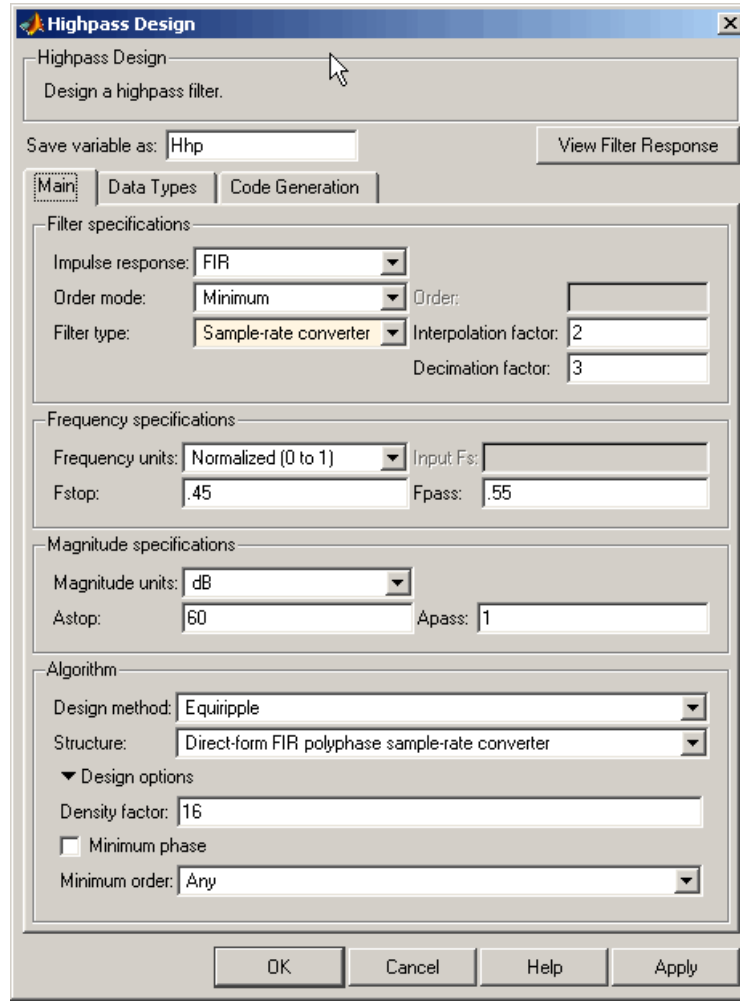
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

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.

Highpass 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 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.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

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

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 **Filter order mode**.

Decimation Factor

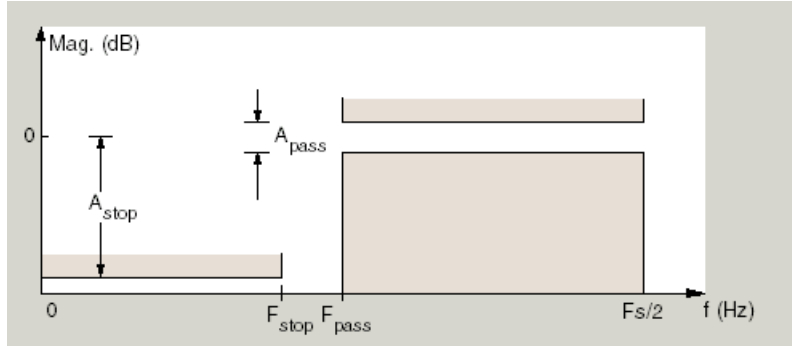
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 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.

- `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.
- `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.

Fstop

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 in **Frequency units**.

Fpass

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**.

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.

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.

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 20 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:

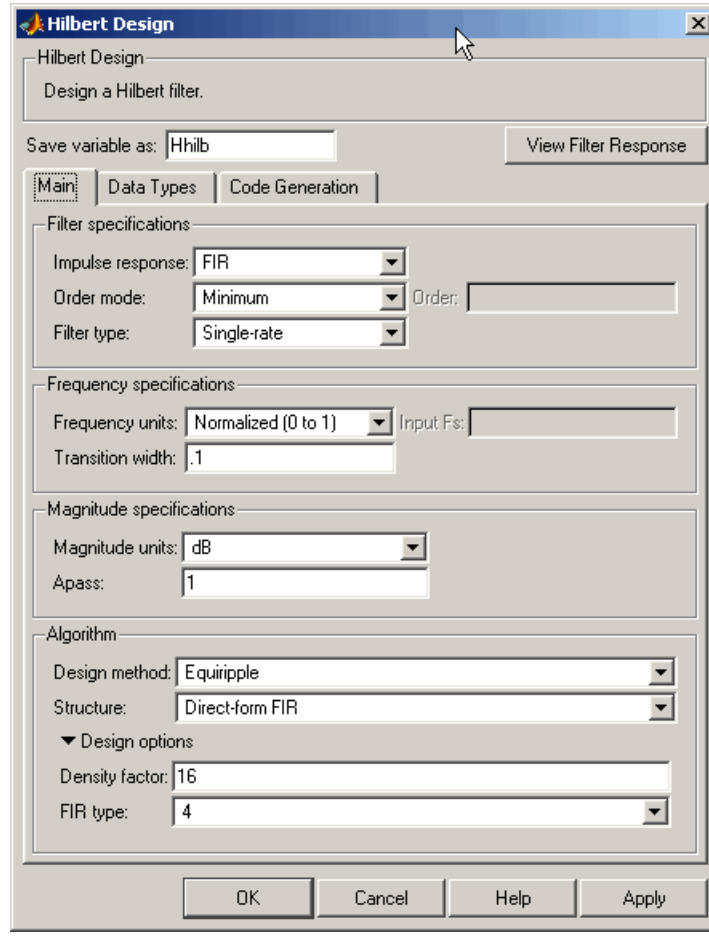
- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband 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.

Hilbert 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 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.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

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

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 **Filter order mode**.

Decimation Factor

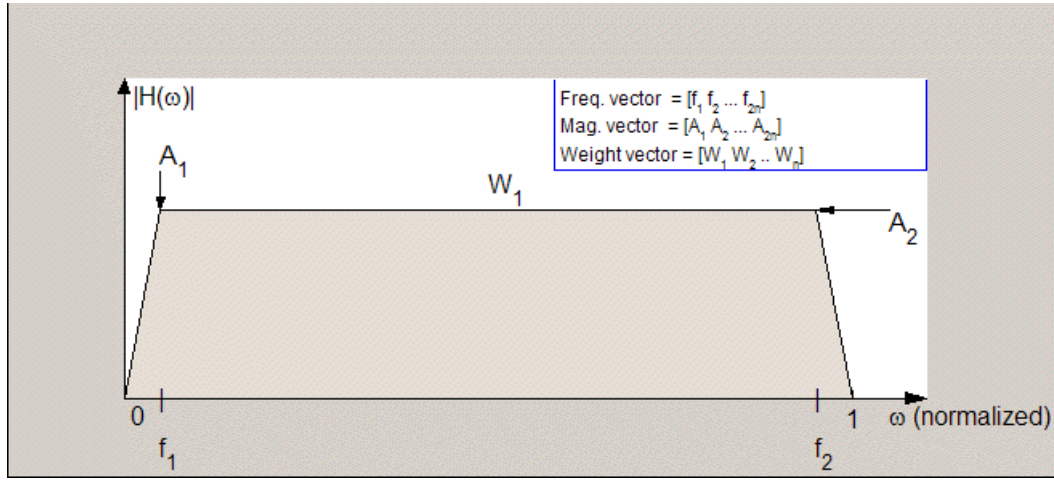
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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.

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.

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 20 represents a

reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

FIR Type

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

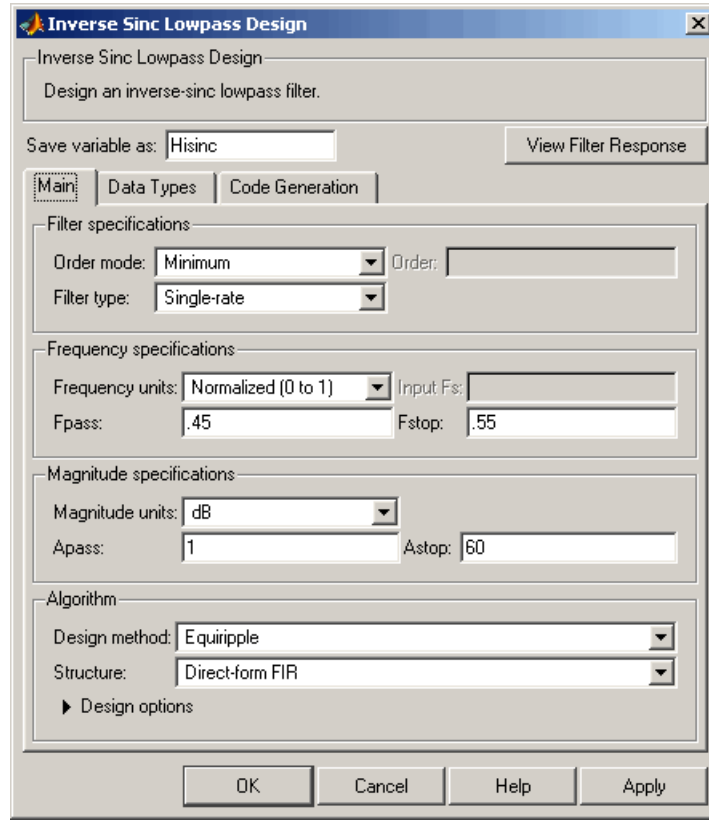
Select either `3` or `4` from the drop-down list.

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.

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.

Filter order mode

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

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 **Filter order mode**.

Decimation Factor

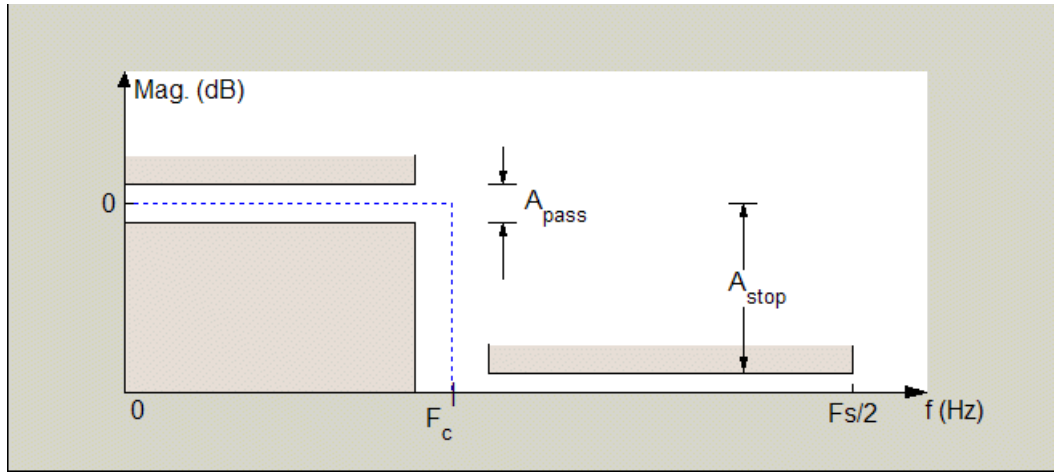
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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

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 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.
- `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.

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.

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.

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 20 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**.

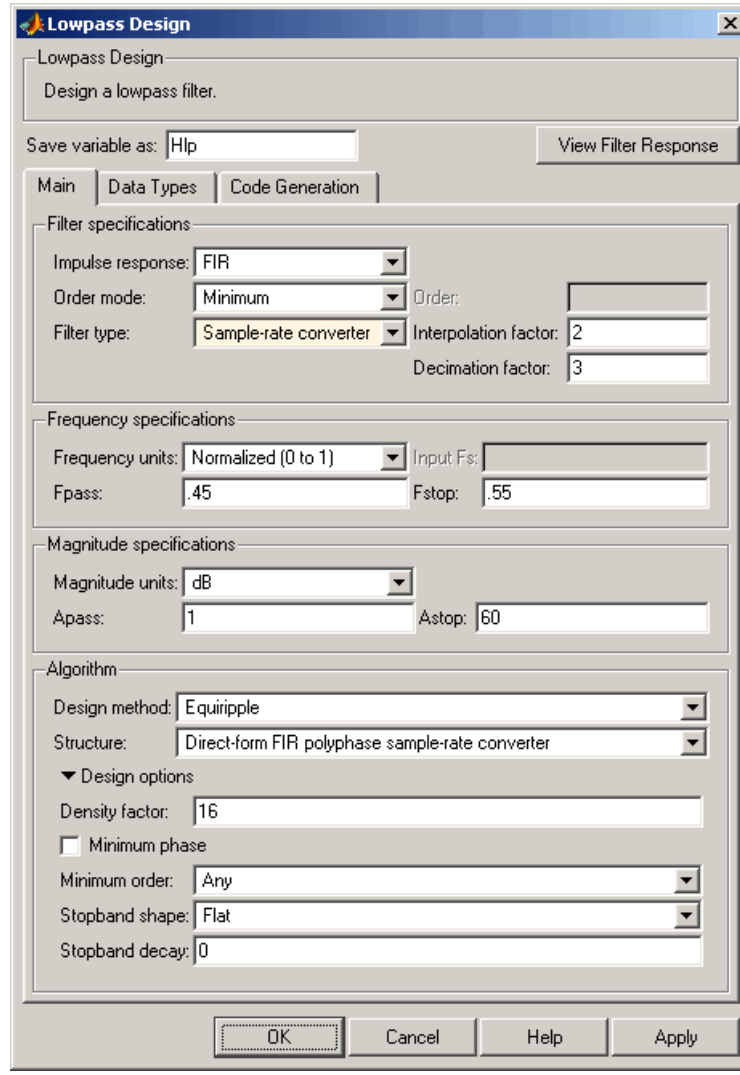
Stopband Decay

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.

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 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.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

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

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 **Filter order mode**.

Decimation Factor

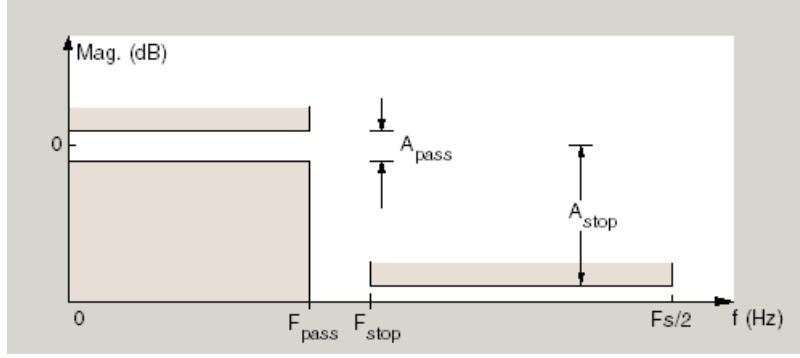
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 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.

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.

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.

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 20 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**.

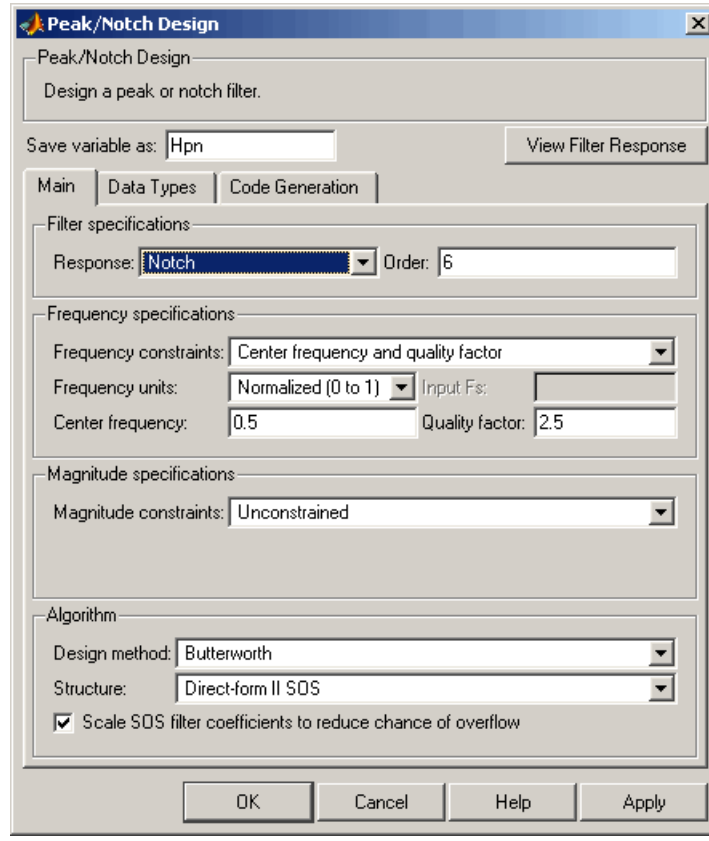
Stopband Decay

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.

Notch/Peak Filter Design Dialog Box — Main Pane

Main Pane



Filter Specifications

In this are 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. The rest of the parameters that specify are equivalent for either filter type.

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 specified previously.

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 in this 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.

Structure

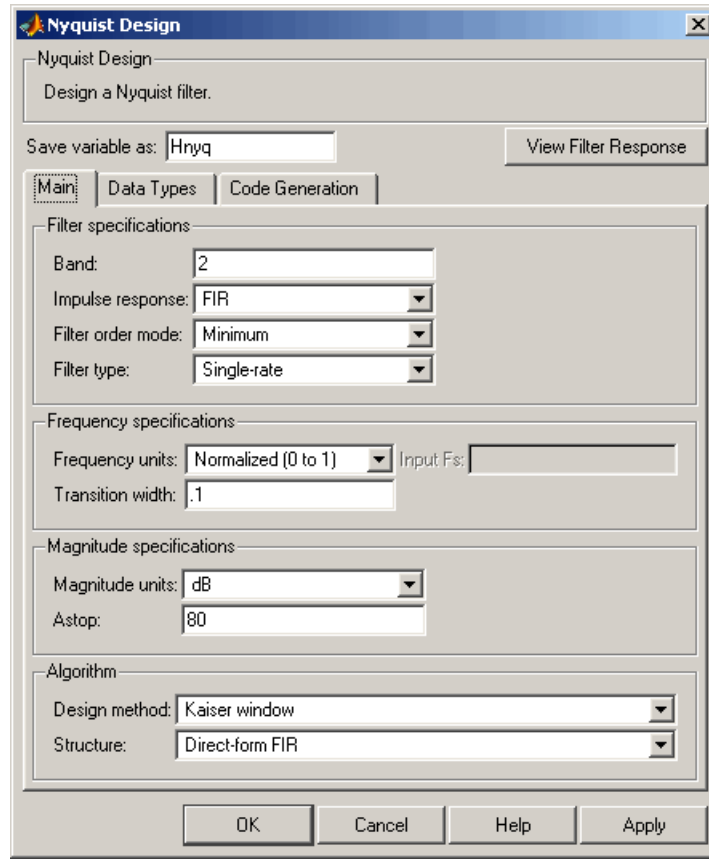
Lists all available filter structures for the filter specifications and design method you select. The typical options are:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

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.

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 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.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

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

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 **Filter order mode**.

Decimation Factor

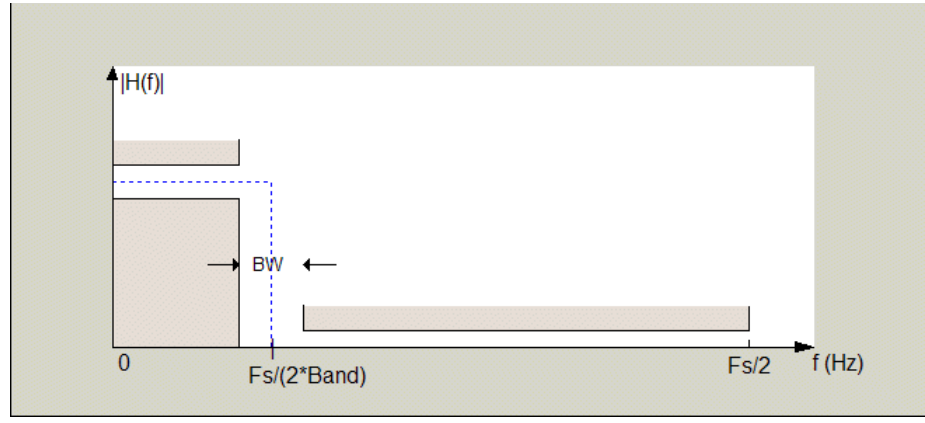
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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.

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.

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 20 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.

Octave Filter Design Dialog Box — Main Pane

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.

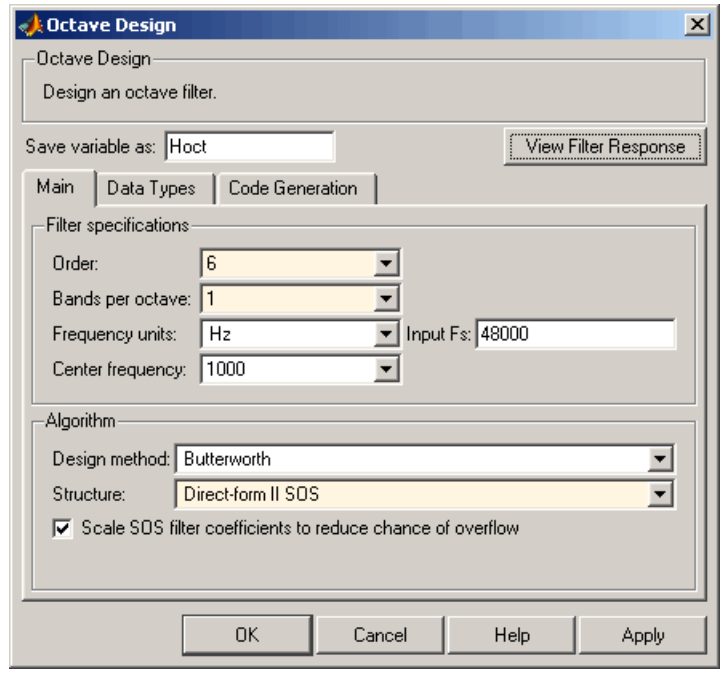
Algorithm

Design Method
Butterworth is the design method used for this type of filter.

Structure
Specify filter structure. Choose from:

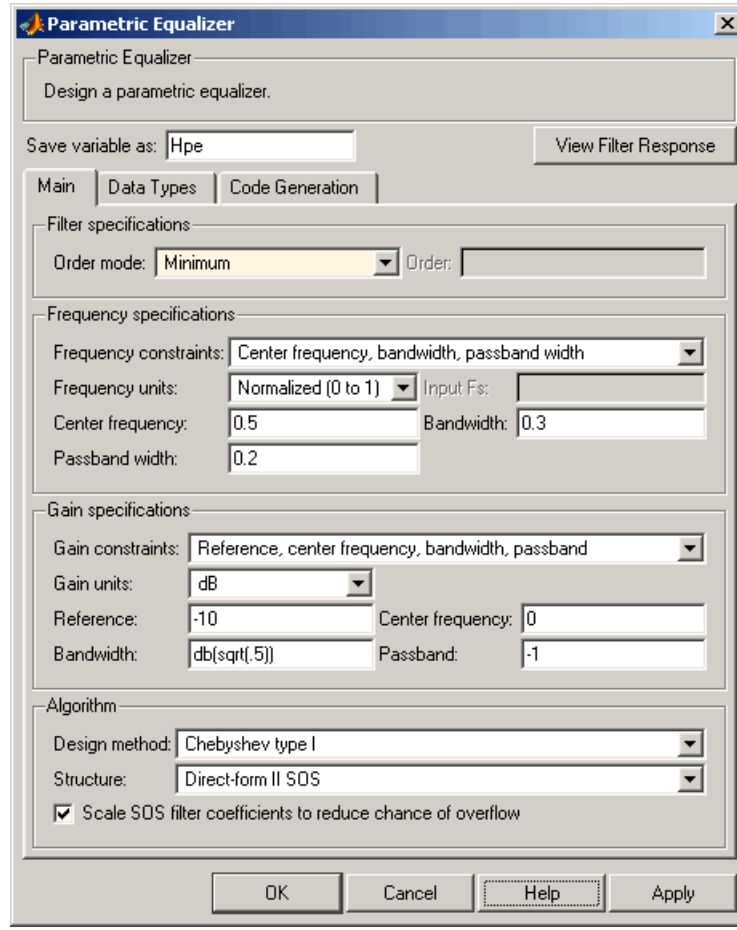
- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Scale SOS filter coefficients to reduce chance of overflow
Select the check box to scale the filter coefficients.


Octave Design
Octave Design
Design an octave filter.
Save variable as:
Hoct
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Order:
6
Bands per octave:
1
Frequency units:
Hz
Input Fs:
48000
Center frequency:
1000
Algorithm
Design method:
Butterworth
Structure:
Direct-form II SOS
Scale SOS filter coefficients to reduce chance of overflow
OK
Cancel
Help
Apply

Parametric Equalizer Filter Design Dialog Box — Main Pane

Filter Specifications



Filter Specifications

Order mode

Select `Minimum` for minimum filter order, 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, then the available frequency constraints are: `Center frequency, bandwidth, passband width` and `Center frequency, bandwidth, stopband width`. However, if you select a specific filter order, then the list of frequency constraints consists of: `Center frequency, bandwidth` and `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. Also, once you choose the frequency constraints the input boxes in this area change to reflect the selection.

Frequency constraints

Select the specification array to represent 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 custom specified order only)
- `Low frequency, high frequency` (available for a custom 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, either normalized, or in the units selected previously.

Bandwidth

Enter the bandwidth.

Passband width

Enter the passband width. 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

Enter the stopband width. 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. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`

High frequency

Enter the high frequency. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`

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

Enter the reference gain.

Bandwidth

Enter the bandwidth.

Center frequency

Enter the center frequency (in the units specified in **Frequency units** input box)

Passband

Specify the passband gain. This input is enabled only for specific configurations.

Stopband

Specify the stopband gain. This input is enabled only for specific configurations.

Algorithm

Design method

Select the design method from the drop-down list. Different methods are available depending on the chosen filter constraints.

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

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Purpose

Store CIC filter states

Description

`filtstates.cic` objects hold the states information for CIC filters. Once you create a CIC filter, the states for the filter are stored in `filtstates.cic` objects, and you can access them and change them as you would any property of the filter. This arrangement parallels that of the `filtstates` object that IIR direct-form I filters use (refer to `filtstates` for more information).

Each `States` property in the CIC filter comprises two properties — `Numerator` and `Comb` — that hold `filtstates.cic` objects. Within the `filtstates.cic` objects are the numerator-related and comb-related filter states. The states are column vectors, where each column represents the states for one section of the filter. For example, a CIC filter with four decimator sections and four interpolator sections has `filtstates.cic` objects that contain four columns of states each.

Examples

To show you the `filtstates.cic` object, create a CIC decimator and filter a signal.

```
hm=mfilt.cicdecim(5,2,4)

hm =

  FilterStructure: 'Cascaded Integrator-Comb Decimator'
       Arithmetic: 'fixed'
DifferentialDelay: 2
 NumberOfSections: 4
 DecimationFactor: 5
 PersistentMemory: false

  InputWordLength: 16
  InputFracLength: 15

SectionWordLengthMode: 'MinWordLengths'

hm.persistentMemory=true
```

```
hm =

 FilterStructure: 'Cascaded Integrator-Comb Decimator'
      Arithmetic: 'fixed'
DifferentialDelay: 2
 NumberOfSections: 4
 DecimationFactor: 5
 PersistentMemory: true
           States: Integrator: [4x1 States]
                         Comb: [4x1 States]
      InputOffset: 0

  InputWordLength: 16
  InputFracLength: 15

SectionWordLengthMode: 'MinWordLengths'
```

Use `hm` to filter some input data.

```
fs = 44.1e3;            % Original sampling frequency: 44.1kHz.
n = 0:10239;            % 10240 samples, 0.232 second long signal.
x  = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1kHz.
y=filter(hm,x)
```

`hm` has nonzero states now.

```
s=hm.states

s =

    Integrator: [4x1 States]
          Comb: [4x1 States]

s.Integrator

ans =
```

```
  1.0e+003 *

    0.0043
   -2.0347
   -0.4175
    0.8206

s.Comb

ans =

  1.0e+003 *

   -3.1301
   -0.8493
   -2.5474
    1.7888
   -1.6253
    3.1981
    0.4729
    3.4559
```

You can use `int` to see the states as 32-bit integers.

```
int(s.Integrator)

ans =

      142435
    -8334019
     -427469
      210081
```

`whos` shows you the `filtstates.cic` object.

```
whos
  Name      Size                    Bytes  Class
```

```
  Fs        1x1                        8  double array
  ans       4x1                       16  int32 array
  hm        1x1                           mfilt.cicdecim
  n         1x10240                81920  double array
  s         1x1                           filtstates.cic
  x         1x10240                81920  double array
  y         1x2048                        embedded.fi

Grand total is 20488 elements using 163864 bytes
```

See Also

`mfilt`, `mfilt.cicdecim`, `mfilt.cicinterp`

`filtstates` in Signal Processing Toolbox™ documentation

Purpose

Constrained-band equiripple FIR filter

Syntax

```
b = fircband(n,f,a,w,c)
b = fircband(n,f,a,s)
b = fircband(...,'1')
b = fircband(...,'minphase')
b = fircband(..., 'check')
b = fircband(...,{lgrid})
[b,err] = fircband(...)
[b,err,res] = fircband(...)
```

Description

`fircband` is a minimax filter design algorithm that you use to design the following types of real FIR filters:

- Types 1-4 linear phase
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase,
- Minimum order (even or odd), extra ripple
- Maximal ripple
- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain
- Arbitrary shape frequency response curve filters

`b = fircband(n,f,a,w,c)` designs filters having constrained error magnitudes (ripples). `c` is a cell array of strings of the same length as `w`. The entries of `c` must be either '`c` to indicate that the corresponding

element in `w` is a constraint (the ripple for that band cannot exceed that value) or '`w`' indicating that the corresponding entry in `w` is a weight. There must be at least one unconstrained band — `c` must contain at least one **`w`** entry. For instance, Example 1 below uses a weight of one in the passband, and constrains the stopband ripple not to exceed 0.2 (about 14 dB).

A hint about using constrained values: if your constrained filter does not touch the constraints, increase the error weighting you apply to the unconstrained bands.

Notice that, when you work with constrained stopbands, you enter the stopband constraint according to the standard conversion formula for power — the resulting filter attenuation or constraint equals 20*log(*constraint*) where *constraint* is the value you enter in the function. For example, to set 20 dB of attenuation, use a value for the constraint equal to 0.1. This applies to constrained stopbands only.

`b = fircband(n,f,a,s)` is used to design filters with special properties at certain frequency points. `s` is a cell array of strings and must be the same length as `f` and `a`. Entries of `s` must be one of:

- '**`n`**' — normal frequency point.
- '**`s`**' — single-point band. The frequency band is given by a single point. You must specify the corresponding gain at this frequency point in `a`.
- '**`f`**' — forced frequency point. Forces the gain at the specified frequency band to be the value specified.
- '**`i`**' — indeterminate frequency point. Use this argument when bands abut one another (no transition region).

`b = fircband(...,'1')` designs a type 1 filter (even-order symmetric). You could also specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), or type 4 (odd-order antisymmetric) filters. Note there are restrictions on `a` at `f = 0` or `f = 1` for types 2, 3, and 4.

`b = fircband(...,'minphase')` designs a minimum-phase FIR filter. There is also '**maxphase**'.

`b = fircband(..., 'check')` produces a warning when there are potential transition-region anomalies in the filter response.

`b = fircband(...,{lgrid})`, where {`lgrid`} is a scalar cell array containing an integer, controls the density of the frequency grid.

`[b,err] = fircband(...)` returns the unweighted approximation error magnitudes. `err` has one element for each independent approximation error.

`[b,err,res] = fircband(...)` returns a structure `res` of optional results computed by `fircband`, and contains the following fields:

Structure Field	Contents
`res.fgrid`	Vector containing the frequency grid used in the filter design optimization
`res.des`	Desired response on `fgrid`
`res.wt`	Weights on `fgrid`
`res.h`	Actual frequency response on the frequency grid
`res.error`	Error at each point (desired response - actual response) on the frequency grid
`res.iextr`	Vector of indices into `fgrid` of external frequencies
`res.fextr`	Vector of extremely frequencies
`res.order`	Filter order

Structure Field	Contents
`res.edgecheck`	Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK , 0 = probable transition-region anomaly , -1 = edge not checked. Computed when you specify the `'check'` input option in the function syntax.
`res.iterations`	Number of Remez iterations for the optimization
`res.evals`	Number of function evaluations for the optimization

Examples

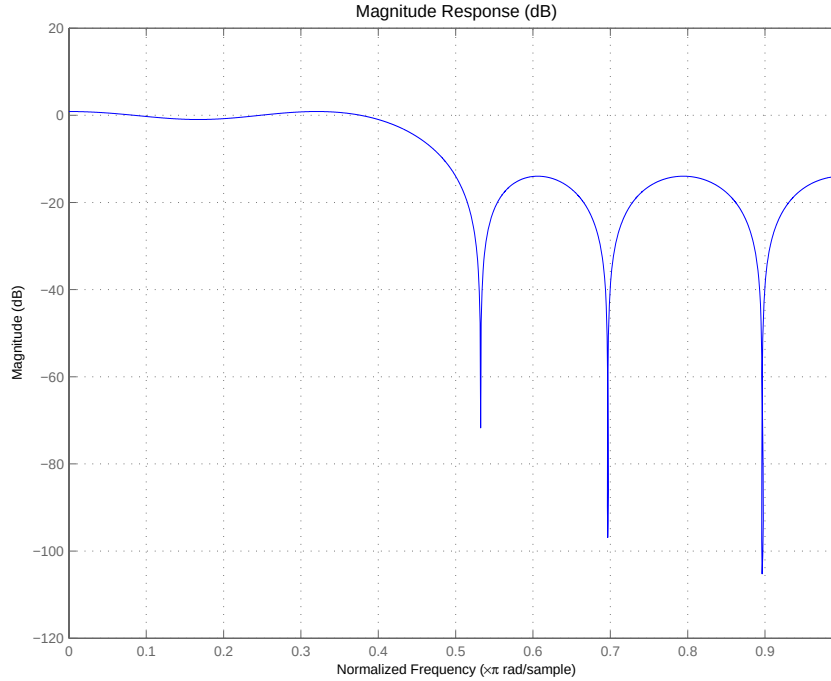
Two examples of designing filters with constrained bands.

Example 1

design a 12th-order lowpass filter with a constraint on the filter response.

```
b = fircband(12,[0 0.4 0.5 1], [1 1 0 0], ...
[1 0.2], {'w' 'c'});
```

Using `fvtool` to display the result `b` shows you the response of the filter you designed.

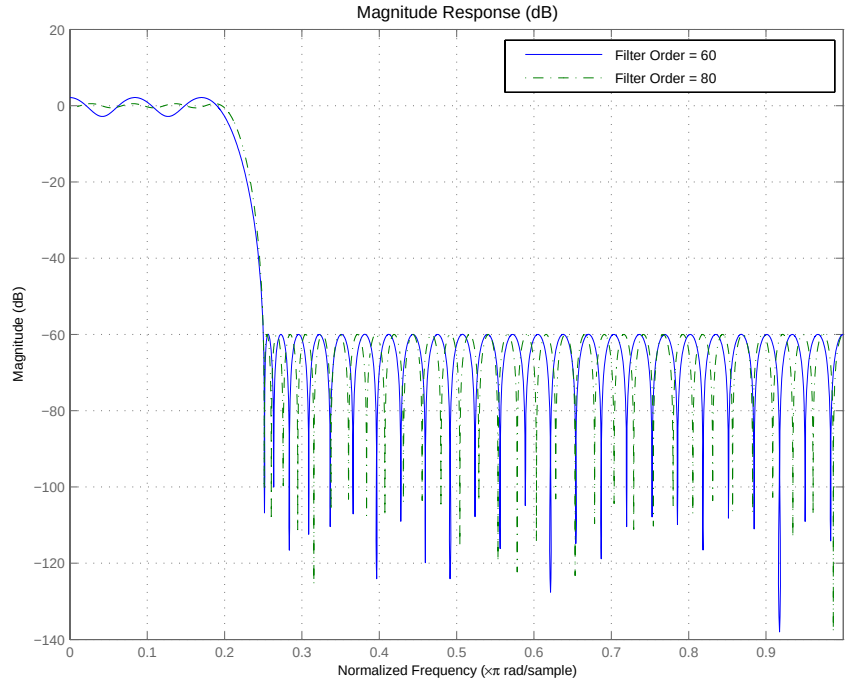


Example 2

design two filters of different order with the stopband constrained to 60 dB. Use excess order (80) in the second filter to improve the passband ripple.

```
b1=fircband(60,[0 .2 .25 1],[1 1 0 0],...
[1 .001],{'w','c'});
b2=fircband(80,[0 .2 .25 1],[1 1 0 0],...
[1 .001],{'w','c'});
fvtool(b1,1,b2,1)
```

To set the stopband constraint to 60 dB, enter `0.001`, since 20*log(0.001) = -60, or 60 dB of signal attenuation.



See Also

`firceqrip`, `firgr`, `firls`

`firpm` in Signal Processing Toolbox™ documentation

Also refer to "Constrained Band Equiripple FIR Filter Design" in Demos

Purpose

Equiripple FIR interpolators

Syntax

```
b = fireqint(n,l,alpha)
b = fireqint(n,l,alpha,w)
b = fireqint('minorder',l,alpha,r)
b = fireqint({'minorder',initord},l,alpha,r)
```

Description

`b = fireqint(n,l,alpha)` designs an FIR equiripple filter useful for interpolating input signals. `n` is the filter order and it must be an integer. `l`, also an integer, is the interpolation factor. `alpha` is the bandlimitedness factor, identical to the same feature in `intfilt`.

`alpha` is inversely proportional to the transition bandwidth of the filter. It also affects the bandwith of the don't-care regions in the stopband. Specifying `alpha` allows you to control how much of the Nyquist interval your input signal occupies. This can be beneficial for signals to be interpolated because it allows you to increase the transition band width without affecting the interpolation, resulting in better stopband attenuation for a given `l`. If you set `alpha` to 1, `fireqint` assumes that your signal occupies the entire Nyquist interval. Setting `alpha` to a value 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.

The signal to be interpolated is assumed to have zero (or negligible) power in the frequency band between (`alpha`*π) and π. `alpha` must therefore be a positive scalar between 0 and 1. `fireqint` treat such bands as don't-care regions for assessing filter design.

`b = fireqint(n,l,alpha,w)` allows you to specify a vector of weights in `w`. The number of weights required in `w` is given by 1 + `floor`(`l`/2). The weights in `w` are applied to the passband ripple and stopband attenuations. Using weights (values between 0 and 1) enables you to specify different attenuations in different parts of the stopband, as well as providing the ability to adjust the compromise between passband ripple and stopband attenuation.

`b = fireqint('minorder',l,alpha,r)` allows you to design a minimum-order filter that meets the design specifications. `r` is a vector

of maximum deviations or ripples from the ideal filter magnitude response. When you use the input argument **minorder**, you must provide the vector `r`. The number of elements required in `r` is given by 1 + `floor(l/2)`.

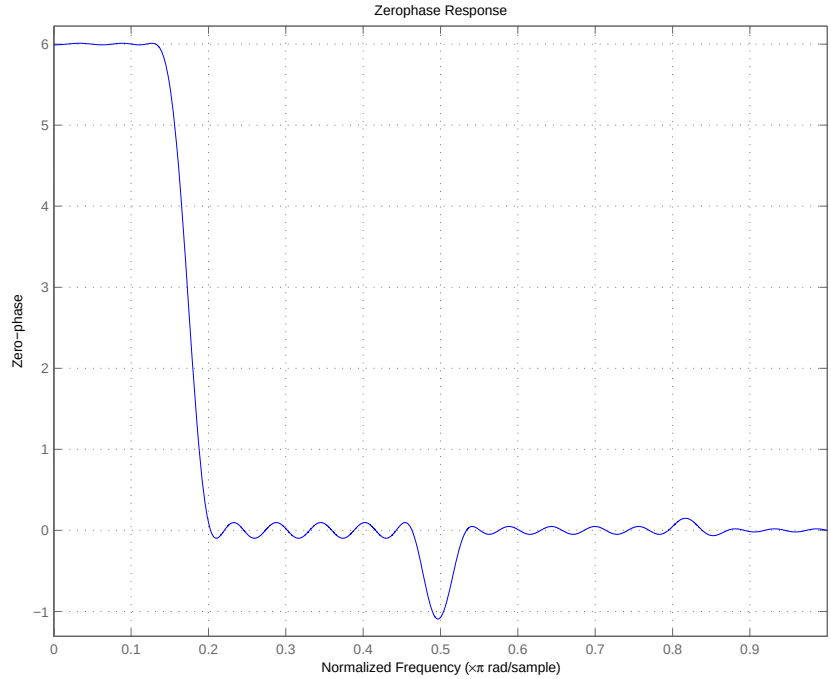
`b = fireqint({'minorder',initord},l,alpha,r)` adds the argument `initord` so you can provide an initial estimate of the filter order. Any positive integer is valid here. Again, you must provide `r`, the vector of maximum deviations or ripples, from the ideal filter magnitude response.

Examples

Design a minimum order interpolation filter for `l` = 6 and `alpha` = 0.8. A vector of ripples must be supplied with the input argument **minorder**.

```
b = fireqint('minorder',6,.8,[0.01 .1 .05 .02]);
hm = mfilt.firinterp(6,b); % Create a polyphase interpolator filter
zerophase(hm);
```

Here is the resulting plot of the zerophase response for `hm`.



For `hm`, the minimum order filter with the requested design specifications, here is the filter information.

```
hm =

FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
          Arithmetic: 'double'
           Numerator: [1x70 double]
InterpolationFactor: 6
  PersistentMemory: false
```

See Also

`firgr`, `firhalfband`, `firls`, `firnyquist`, `mfilt`

`intfilt` in Signal Processing Toolbox™ documentation

Purpose

Constrained, equiripple FIR filter

Syntax

```
hd = firceqrip(n,wo,del)
hd = firceqrip(...,'slope',r)
hd = firceqrip(...,'passedge')
hd = firceqrip(...,'stopedge')
hd = firceqrip(...,'high')
hd = firceqrip(...,'min')
hd = firceqrip(...,'invsinc',c))
```

Description

`hd = firceqrip(n,wo,del)` design an order `n` filter (filter length equal `n + 1`) lowpass FIR filter with linear phase.

`firceqrip` produces the same equiripple lowpass filters that `firpm` produces using the Parks-McClellan algorithm. The difference is how you specify the filter characteristics for the function.

Input argument `wo` specifies the cutoff frequency. The two-element vector `del` specifies the peak or maximum error allowed in the passband and stopbands. Enter `[d1 d2]` for `del` where `d1` sets the passband error and `d2` sets the stopband error. Since `firceqrip` works in the normalized frequency domain, you must set `wo` to be between 0 and 1 ($0 < wo < 1$).

`hd = firceqrip(...,'slope',r)` uses the input keyword '**slope**' and input argument `r` to design a filter with a stopband that does not demonstrate equiripple characteristics. `r` determines the slope of the stopband in decibels when $r > 0$.

In this constrained equiripple design approach, you can specify a stopband slope (increasing attenuation with increasing frequency). Enter your desired slope in decibels as a positive value. Larger slope values create increasing attenuation of the stopband as frequency increases.

Slope is defined in the following ways:

- For filters specified in linear frequency, the slope is defined over every Fs/2 frequency bands.

- For filters specified in normalized frequency, the slope is defined over π rad/sample.

Here is a description of how slope works. The algorithm defines slope in decibels per half of the Nyquist interval. If you are working in normalized frequency and you set the slope to 40 dB, the stopband attenuation increases by 40 dB every rad/sample.

Try setting `r` to `10` to see the effect on the filter frequency response. In the Examples section, example 3 designs a filter with `r` equal to 20.

`hd = firceqrip(...,'passedge')` designs a filter where `wo` specifies the frequency at which the passband starts to roll off.

`hd = firceqrip(...,'stopedge')` designs a filter where `wo` specifies the frequency at which the stopband begins.

`hd = firceqrip(...,'high')` designs a high pass FIR filter instead of a lowpass filter.

`hd = firceqrip(...,'min')` designs an FIR filter with minimum phase.

`hd = firceqrip(...,'invsinc',c))` designs a lowpass filter whose passband has the shape of the `inverse sinc` function. For this syntax, keyword **`invsinc`** applies the`inverse sinc` function as defined by whether `c` is a scalar or a two-element vector:

- When you use `c` as a scalar with the **`invsinc`** keyword, `firceqrip` applies the function 1/sinc(`c`*`w`), where `w` is the normalized frequency, to the passband.
- When you use `c` as a two-element vector entered as `[c p]`, with the **`invsinc`** keyword, `firceqrip` applies the function 1/sinc(`c`*`w`)^{p} to the passband, where `w` is the normalized frequency.

In both cases, `c` must meet the condition `c` < 1/w_o.

When you use a cascaded-integrated comb (CIC) filter in series with this FIR filter, argument `p` lets you compensate for the droop in the passband of the CIC filter. Setting `p` equal to the number of stages

in your CIC generally produces an FIR filter whose passband neatly compensates for the CIC passband shape.

To let you specify precisely the FIR filter to design, use any or all of the optional input arguments together. Any ordering of the optional arguments works — order is not important in the function call. Refer to Examples 3 and 4 to see multiple optional input arguments being used.

Note If the `wo` you specify is too small or too large, or if either `c` or `p` is too large, your filter specifications may be unfeasible, causing the design algorithm to fail to generate your filter.

Examples

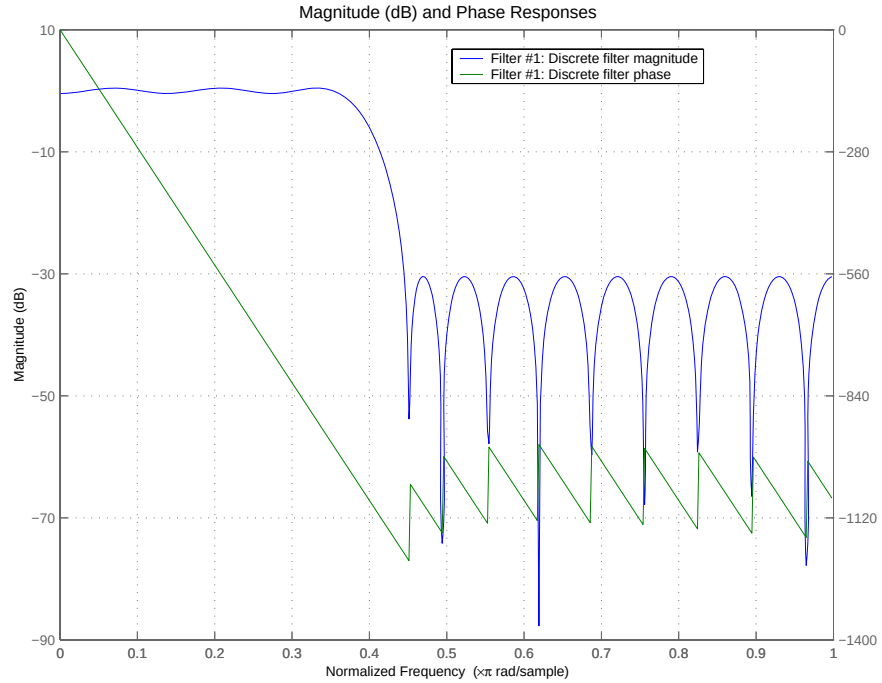
To introduce a few of the variations on FIR filters that you design with `firceqrip`, these five examples cover both the default syntax `hd = firceqrip(n,wo,del)` and some of the optional input arguments. For each example, the input arguments `n`, `wo`, and `del` remain the same.

Example 1

Design an order = 30 FIR filter without using optional input arguments or keywords.

```
hd = firceqrip(n,wo,del); fvtool(hd)
```

Both the phase and magnitude response for the resulting lowpass filter appear in the plot shown here.



Example 2

Design an order = 30 FIR filter with the **stopedge** keyword to define the response at the edge of the filter stopband.

```
hd = firceqrip(n,wo,del,'stopedge ); fvtool(hd,1)
```

Example 3

Design an order = 30 FIR filter with the **slope** keyword and r = 20.

```
hd = firceqrip(n,wo,del,'slope ,20, stopedge ); fvtool(hd)
```

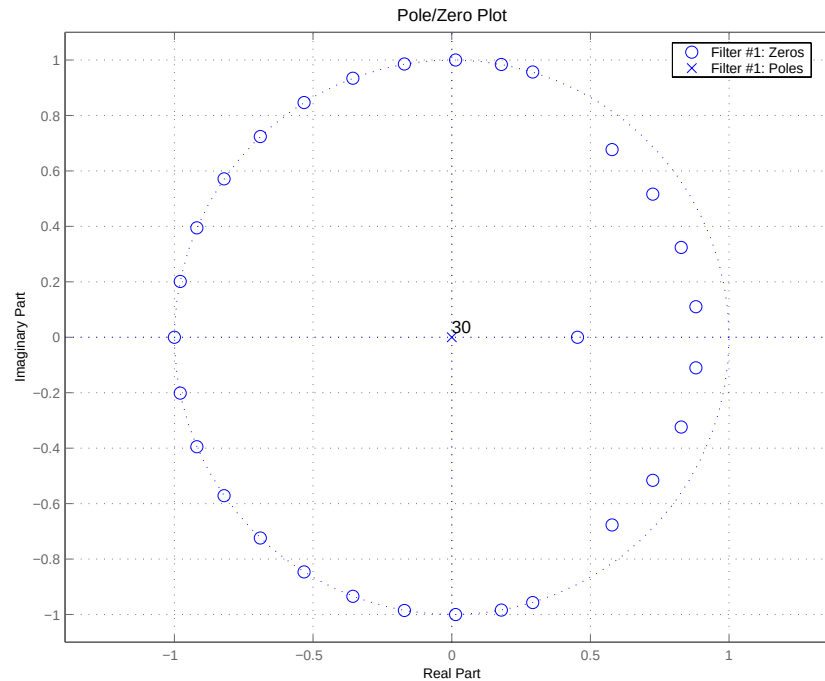
Example 4

Design an order = 30 FIR filter defining the stopband and specifying that the resulting filter is minimum phase with the **`min`** keyword.

```
hd = firceqrip(n,wo,del,'stopedge , min ); fvtool(hd)
```

Comparing this filter to the filter in Example 1, the cutoff frequency `wo = 0.4` now applies to the edge of the stopband rather than the point at which the frequency response magnitude is 0.5.

Viewing the zero-pole plot shown here reveals this is a minimum phase FIR filter — the zeros lie on or inside the unit circle, z = 1.

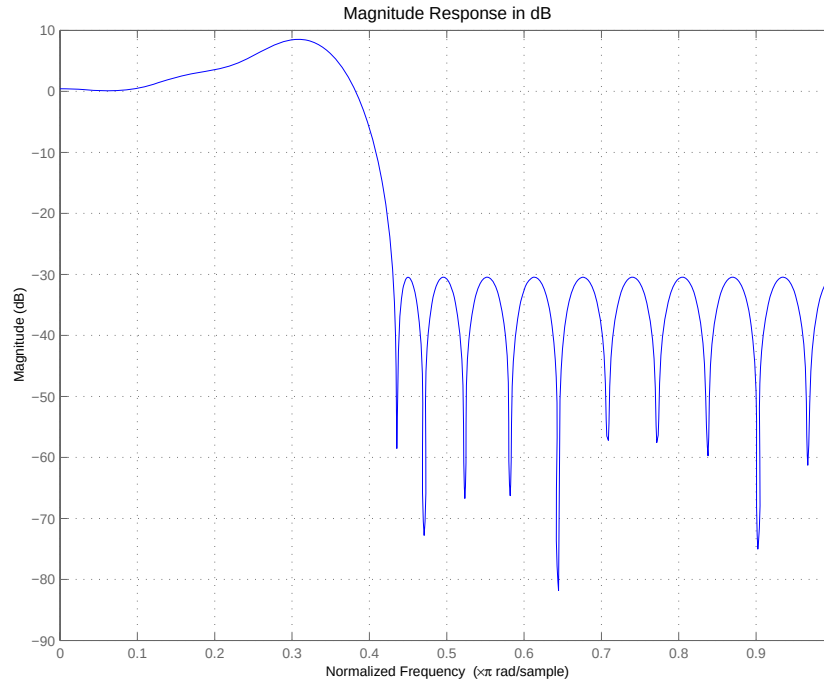


Example 5

Design an order = 30 FIR filter with the **`invsinc`** keyword to shape the filter passband with an inverse sinc function.

```
hd = firceqrip(n,wo,del,'invsinc ,[2 1.5]); fvtool(hd,1)
```

With the inverse sinc function being applied defined as $1/\text{sinc}(2*w)^{1.5}$, the figure shows the reshaping of the passband that results from using the **`invsinc`** keyword option, and entering `c` as the two-element vector `[2 1.5]`.



See Also

`firhalfband`, `firnyquist`, `firgr`, `ifir`, `iirgrpdelay`, `iirlpnorm`, `iirlpnormc`

`fircls`, `firls`, `firpm` in Signal Processing Toolbox™ documentation

Purpose

Parks-McClellan FIR filter

Syntax

```
b = firgr(n,f,a,w)
b = firgr(n,f,a,'hilbert')
b = firgr(m,f,a,r),
b = firgr({m,ni},f,a,r)
b = firgr(n,f,a,w,e)
b = firgr(n,f,a,s)
b = firgr(n,f,a,s,w,e)
b = firgr(...,'1')
b = firgr(...,'minphase')
b = firgr(..., 'check')
b = firgr(...,{lgrid}),
[b,err] = firgr(...)
[b,err,res] = firgr(...)
b = firgr(n,f,fresp,w)
b = firgr(n,f,{fresp,p1,p2,...},w)
b = firgr(n,f,a,w)
```

Description

`firgr` is a minimax filter design algorithm you use to design the following types of real FIR filters:

- Types 1-4 linear phase:
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase
- Minimum order (even or odd)
- Extra ripple
- Maximal ripple

- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain
- Arbitrary shape frequency response curve filters

`b = firgr(n,f,a,w)` returns a length `n+1` linear phase FIR filter which has the best approximation to the desired frequency response described by `f` and `a` in the minimax sense. `w` is a vector of weights, one per band. When you omit `w`, all bands are weighted equally. For more information on the input arguments, refer to `firpm` in *Signal Processing Toolbox™ User's Guide*.

`b = firgr(n,f,a,'hilbert')` and `b = firgr(n,f,a,'differentiator')` design FIR Hilbert transformers and differentiators. For more information on designing these filters, refer to `firpm` in *Signal Processing Toolbox User's Guide*.

`b = firgr(m,f,a,r)`, where `m` is one of '`minorder`', '`mineven`' or '`minodd`', designs filters repeatedly until the minimum order filter, as specified in `m`, that meets the specifications is found. `r` is a vector containing the peak ripple per frequency band. You must specify `r`. When you specify '`mineven`' or '`minodd`', the minimum even or odd order filter is found.

`b = firgr({m,ni},f,a,r)` where `m` is one of '`minorder`', '`mineven`' or '`minodd`', uses `ni` as the initial estimate of the filter order. `ni` is optional for common filter designs, but it must be specified for designs in which `firpmord` cannot be used, such as while designing differentiators or Hilbert transformers.

`b = firgr(n,f,a,w,e)` specifies independent approximation errors for different bands. Use this syntax to design extra ripple or maximal ripple filters. These filters have interesting properties such as having the minimum transition width. `e` is a cell array of strings specifying the approximation errors to use. Its length must equal the number of bands. Entries of `e` must be in the form '`e#`' where # indicates which approximation error to use for the corresponding band. For example,

when `e = {'e1','e2','e1'}`, the first and third bands use the same approximation error `'e1'` and the second band uses a different one `'e2'`. Note that when all bands use the same approximation error, such as `{'e1','e1','e1',...}`, it is equivalent to omitting `e`, as in `b = firgr(n,f,a,w)`.

`b = firgr(n,f,a,s)` is used to design filters with special properties at certain frequency points. `s` is a cell array of strings and must be the same length as `f` and `a`. Entries of `s` must be one of:

- `'n'` — normal frequency point.
- `'s'` — single-point band. The frequency "band" is given by a single point. The corresponding gain at this frequency point must be specified in `a`.
- `'f'` — forced frequency point. Forces the gain at the specified frequency band to be the value specified.
- `'i'` — indeterminate frequency point. Use this argument when adjacent bands abut one another (no transition region).

For example, the following command designs a bandstop filter with zero-valued single-point stop bands (notches) at 0.25 and 0.55.

```
b = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],...
[1 1 0 1 1 0 1 1],{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'})
```

`b = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...{'n' 'i' 'f' 'n' 'n' 'n'})` designs a highpass filter with the gain at 0.06 forced to be zero. The band edge at 0.055 is indeterminate since the first two bands actually touch. The other band edges are normal.

`b = firgr(n,f,a,s,w,e)` specifies weights and independent approximation errors for filters with special properties. The weights and properties are included in vectors `w` and `e`. Sometimes, you may need to use independent approximation errors to get designs with forced values to converge. For example,

```
b = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

`b = firgr(...,'1')` designs a type 1 filter (even-order symmetric). You can specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), and type 4 (odd-order antisymmetric) filters as well. Note that restrictions apply to a at f = 0 or f = 1 for FIR filter types 2, 3, and 4.

`b = firgr(...,'minphase')` designs a minimum-phase FIR filter. You can use the argument `'maxphase'` to design a maximum phase FIR filter.

`b = firgr(..., 'check')` returns a warning when there are potential transition-region anomalies.

`b = firgr(...,{lgrid})`, where `{lgrid}` is a scalar cell array. The value of the scalar controls the density of the frequency grid by setting the number of samples used along the frequency axis.

`[b,err] = firgr(...)` returns the unweighted approximation error magnitudes. `err` contains one element for each independent approximation error returned by the function.

`[b,err,res] = firgr(...)` returns the structure `res` comprising optional results computed by `firgr`. `res` contains the following fields.

Structure Field	Contents
`res.fgrid`	Vector containing the frequency grid used in the filter design optimization
`res.des`	Desired response on `fgrid`
`res.wt`	Weights on `fgrid`
`res.h`	Actual frequency response on the frequency grid
`res.error`	Error at each point (desired response - actual response) on the frequency grid

Structure Field	Contents
`res.iextr`	Vector of indices into `fgrid` of external frequencies
`res.fextr`	Vector of external frequencies
`res.order`	Filter order
`res.edgecheck`	Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK, 0 = probable transition-region anomaly, -1 = edge not checked. Computed when you specify the `'check'` input option in the function syntax.
`res.iterations`	Number of `s` iterations for the optimization
`res.evals`	Number of function evaluations for the optimization

`firgr` is also a "function function," allowing you to write a function that defines the desired frequency response.

`b = firgr(n,f,fresp,w)` returns a length N + 1 FIR filter which has the best approximation to the desired frequency response as returned by the user-defined function `fresp`. Use the following `firgr` syntax to call `fresp`:

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

where:

- `fresp` is the string variable that identifies the function that you use to define your desired filter frequency response.
- `n` is the filter order.
- `f` is the vector of frequency band edges which must appear monotonically between 0 and 1, where 1 is one-half of the sampling frequency. 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.

- `gf` is a vector of grid points that have been chosen over each specified frequency band by `firgr`, and determines the frequencies at which `firgr` evaluates the response function.
- `w` is a vector of real, positive weights, one per band, for use during optimization. `w` is optional in the call to `firgr`. If you do not specify `w`, it is set to unity weighting before being passed to `fresp`.
- `dh` and `dw` are the desired frequency response and optimization weight vectors, evaluated at each frequency in grid `gf`.

`firgr` includes a predefined frequency response function named `'firpmfrf2'`. You can write your own based on the simpler `'firpmfrf'`. See the help for `private/firpmfrf` for more information.

`b = firgr(n,f,{`*`fresp`*`,p1,p2,...},w)` specifies optional arguments `p1`, `p2`,..., `pn` to be passed to the response function `fresp`.

`b = firgr(n,f,a,w)` is a synonym for `b = firgr(n,f,{'firpmfrf2',a},w)`, where `a` is a vector containing your specified response amplitudes at each band edge in `f`. By default, `firgr` designs symmetric (even) FIR filters. `'firpmfrf2'` is the predefined frequency response function. If you do not specify your own frequency response function (the `fresp` string variable), `firgr` uses ’`firpmfrf2'`.

`b = firgr(...,'h')` and `b = firgr(...,'d')` design antisymmetric (odd) filters. When you omit the `'h'` or `'d'` arguments from the `firgr` command syntax, each frequency response function `fresp` can tell `firgr` to design either an even or odd filter. Use the command syntax `sym = `*`fresp`*`('defaults',{n,f,[],w,p1,p2,...})`.

`firgr` expects `fresp` to return `sym = 'even'` or `sym = 'odd'`. If `fresp` does not support this call, `firgr` assumes even symmetry.

For more information about the input arguments to `firgr`, refer to `firpm`.

Examples

These examples demonstrate some filters you might design using `firgr`.

Example 1

design an FIR filter with two single-band notches at 0.25 and 0.55

```
b1 = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],[1 1 0 1 1 0 1 1],...
{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'});
```

Example 2

design a highpass filter whose gain at 0.06 is forced to be zero. The gain at 0.055 is indeterminate since it should abut the band.

```
b2 = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'});
```

Example 3

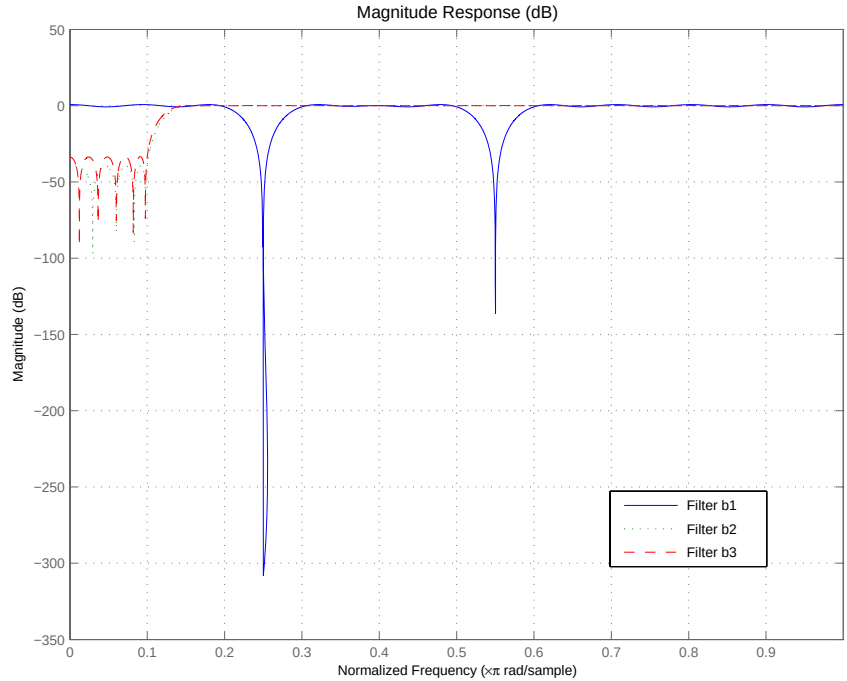
design a second highpass filter with forced values and independent approximation errors.

```
b3 = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1], ...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

Use the filter visualization tool to view the results of the filters created in these examples.

```
fvtool(b1,1,b2,1,b3,1)
```

Here is the figure from FVTool.



See Also

`butter`, `cheby1`, `cheby2`, `ellip`, `freqz`, `filter`, `firls`, `fircls`, and `firpm` in Signal Processing Toolbox documentation

References

Shpak, D.J. and A. Antoniou, "A generalized Remez method for the design of FIR digital filters," *IEEE® Trans. Circuits and Systems*, pp. 161-174, Feb. 1990.

firhalfband

Purpose

Halfband FIR filter

Syntax

```
b = firhalfband(n,fp)
b = firhalfband(n,win)
b = firhalfband(n,dev,'dev')
b = firhalfband('minorder',fp,dev)
b = firhalfband('minorder',fp,dev,'kaiser')
b = firhalfband(...,'high')
b = firhalfband(...,'minphase')
```

Description

`b = firhalfband(n,fp)` designs a lowpass halfband FIR filter of order `n` with an equiripple characteristic. `n` must be an even integer. `fp` determines the passband edge frequency, and it must satisfy 0 < `fp` < 1/2, where 1/2 corresponds to $\pi/2$ rad/sample.

`b = firhalfband(n,win)` designs a lowpass Nth-order filter using the truncated, windowed-impulse response method instead of the equiripple method. `win` is an `n+1` length vector. The ideal impulse response is truncated to length `n + 1`, and then multiplied point-by-point with the window specified in `win`.

`b = firhalfband(n,dev,'dev')` designs an Nth-order lowpass halfband filter with an equiripple characteristic. Input argument `dev` sets the value for the maximum passband and stopband ripple allowed.

`b = firhalfband('minorder',fp,dev)` designs a lowpass minimum-order filter, with passband edge `fp`. The peak ripple is constrained by the scalar `dev`. This design uses the equiripple method.

`b = firhalfband('minorder',fp,dev,'kaiser')` designs a lowpass minimum-order filter, with passband edge `fp`. The peak ripple is constrained by the scalar `dev`. This design uses the Kaiser window method.

`b = firhalfband(...,'high')` returns a highpass halfband FIR filter.

`b = firhalfband(...,'minphase')` designs a minimum-phase FIR filter such that the filter is a spectral factor of a halfband filter (recall that `h = conv(b,fliplr(b))` is a halfband filter). This can be useful for designing perfect reconstruction, two-channel FIR filter

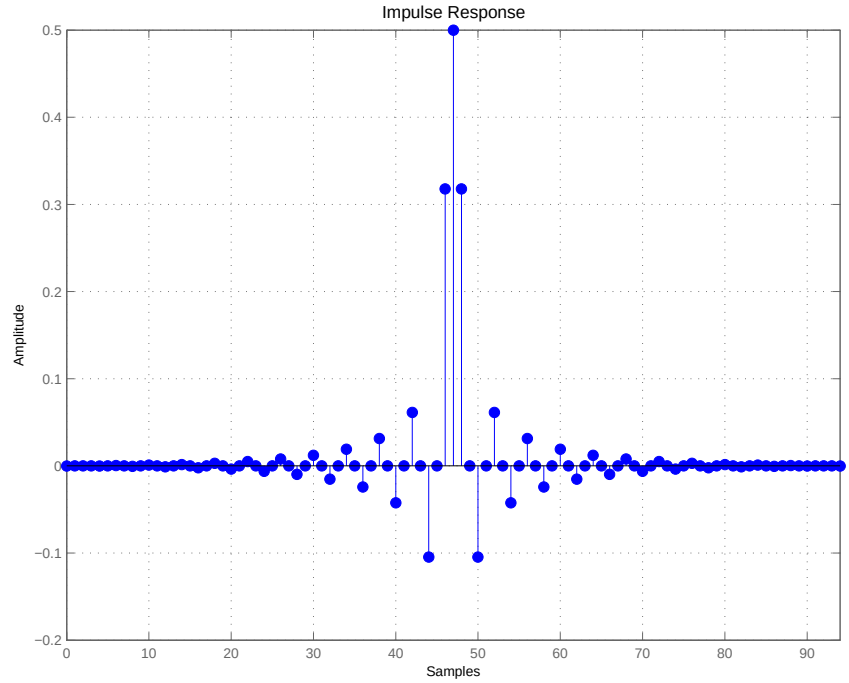
banks. The **minphase** option for `firhalfband` is not available for the window-based halfband filter designs — `b = firhalfband(n,win)` and `b = firhalfband('minorder',fp,dev,'kaiser')`.

In the minimum phase cases, the filter order must be odd.

Examples

This example designs a minimum order halfband filter with specified maximum ripple:

```
b = firhalfband('minorder',.45,0.0001);
h = dfilt.dfsymfir(b);
impz(b) % Impulse response is zero for every other sample
```



The next example designs a halfband filter with specified maximum ripple of 0.0001 dB in the pass and stop bands.

```
b = firhalfband(98,0.0001,'dev');
h = mfilt.firdecim(2,b); % Create a polyphase decimator
freqz(h); % 80 dB attenuation in the stopband
```

See Also

`firnyquist`, `firgr`

`fir1`, `firls`, `firpm` in Signal Processing Toolbox™ documentation

References

Saramaki, T, "Finite Impulse Response Filter Design," *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Purpose

Convert FIR Type I lowpass to FIR Type 1 lowpass with inverse bandwidth

Syntax

```
g = firlp2lp(b)
```

Description

`g = firlp2lp(b)` transforms the Type I lowpass FIR filter `b` with zero-phase response Hr(w) to a Type I lowpass FIR filter `g` with zero-phase response [1 - Hr(π-w)].

When `b` is a narrowband filter, `g` will be a wideband filter and vice versa. The passband and stopband ripples of `g` will be equal to the stopband and passband ripples of `b`.

Examples

Overlay the original narrowband lowpass and the resulting wideband lowpass

```
b = firgr(36,[0 .2 .25 1],[1 1 0 0],[1 5]);
zerophase(b);
hold on
h = firlp2lp(b);
zerophase(h); hold off
```

See Also

`firlp2hp`

`zerophase` in Signal Processing Toolbox™ documentation

References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

firlp2hp

Purpose

Convert FIR lowpass filter to Type I FIR highpass filter

Syntax

```
g = firlp2hp(b)
g = firlp2hp(b,'narrow')
g = firlp2hp(b,'wide')
```

Description

`g = firlp2hp(b)` transforms the lowpass FIR filter `b` into a Type I highpass FIR filter `g` with zero-phase response Hr(π-w). Filter `b` can be any FIR filter, including a nonlinear-phase filter.

The passband and stopband ripples of `g` will be equal to the passband and stopband ripples of `b`.

`g = firlp2hp(b,'narrow')` transforms the lowpass FIR filter `b` into a Type I narrow band highpass FIR filter `g` with zero-phase response Hr(π-w). `b` can be any FIR filter, including a nonlinear-phase filter.

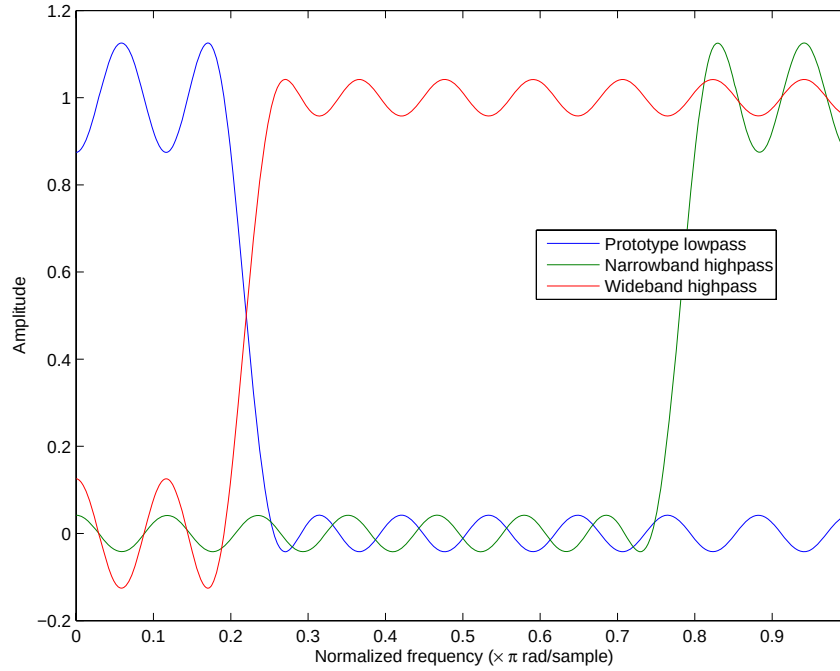
`g = firlp2hp(b,'wide')` transforms the Type I lowpass FIR filter `b` with zero-phase response Hr(w) into a Type I wide band highpass FIR filter `g` with zero-phase response 1 - Hr(w). Note the restriction that `b` must be a Type I linear-phase filter.

For this case, the passband and stopband ripples of `g` will be equal to the stopband and passband ripples of `b`.

Examples

Overlay the original narrowband lowpass (the prototype filter) and the post-conversion narrowband highpass and wideband highpass filters to compare and assess the conversion. The following plot shows the results.

```
b = firgr(36,[0 .2 .25 1],[1 1 0 0],[1 3]);
zerophase(b); hold on;
h = firlp2hp(b);
zerophase(h);
g = firlp2hp(b,'wide');
zerophase(g); hold off
```



See Also

`firlp2lp`

`zerophase` in Signal Processing Toolbox™ documentation

References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

firlpnorm

Purpose

Least P-norm optimal FIR filter

Syntax

```
b = firlpnorm(n,f,edges,a)
b = firlpnorm(n,f,edges,a,w)
b = firlpnorm(n,f,edges,a,w,p)
b = firlpnorm(n,f,edges,a,w,p,dens)
b = firlpnorm(n,f,edges,a,w,p,dens,initnum)
b = firlpnorm(...,'minphase')
[b,err] = firlpnorm(...)
```

Description

`b = firlpnorm(n,f,edges,a)` returns a filter of numerator order `n` which represents the best approximation to the frequency response described by `f` and `a` in the least-Pth norm sense. P is set to 128 by default, which essentially equivalent to the infinity norm. Vector `edges` specifies the band-edge frequencies for multiband designs. `firlpnorm` uses an unconstrained quasi-Newton algorithm to design the specified filter.

`f` and `a` must have the same number of elements, which can exceed the number of elements in `edges`. This lets you specify filters with any gain contour within each band. However, the frequencies in `edges` must also be in vector `f`. Always use `freqz` to check the resulting filter.

Note `firlpnorm` uses a nonlinear optimization routine that may not converge in some filter design cases. Furthermore the algorithm is not well-suited for certain large-order (order > 100) filter designs.

`b = firlpnorm(n,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `firlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. For example,

```
b = firlpnorm(20,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband, and with emphasis placed on minimizing the error in the stopband.

`b = firlpnorm(n,f,edges,a,w,p)` where `p` is a two-element vector `[pmin pmax]` lets you specify the minimum and maximum values of `p` used in the least-pth algorithm. Default is `[2 128]` which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even numbers. The design algorithm starts optimizing the filter with `pmin` and moves toward an optimal filter in the `pmax` sense. When `p` is the string '**`inspect`**', `firlpnorm` does not optimize the resulting filter. You might use this feature to inspect the initial zero placement.

`b = firlpnorm(n,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `[dens*(n+1)]`. The default is 20. You can specify `dens` as a single-element cell array. The grid is equally spaced.

`b = firlpnorm(n,f,edges,a,w,p,dens,initnum)` lets you determine the initial estimate of the filter numerator coefficients in vector `initnum`. This can prove helpful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox™ software can be used for generating `initnum`.

`b = firlpnorm(...,'minphase')` where string '`minphase`' is the last argument in the argument list generates a minimum-phase FIR filter. By default, `firlpnorm` design mixed-phase filters. Specifying input option '`minphase`' causes `firlpnorm` to use a different optimization method to design the minimum-phase filter. As a result of the different optimization used, the minimum-phase filter can yield slightly different results.

`[b,err] = firlpnorm(...)` returns the least-pth approximation error `err`.

Examples

To demonstrate `firlpnorm`, here are two examples — the first designs a lowpass filter and the second a highpass, minimum-phase filter.

```
% Lowpass filter with a peak of 1.4 in the passband.
b = firlpnorm(22,[0 .15 .4 .5 1],[0 .4 .5 1],[1 1.4 1 0 0],...
```

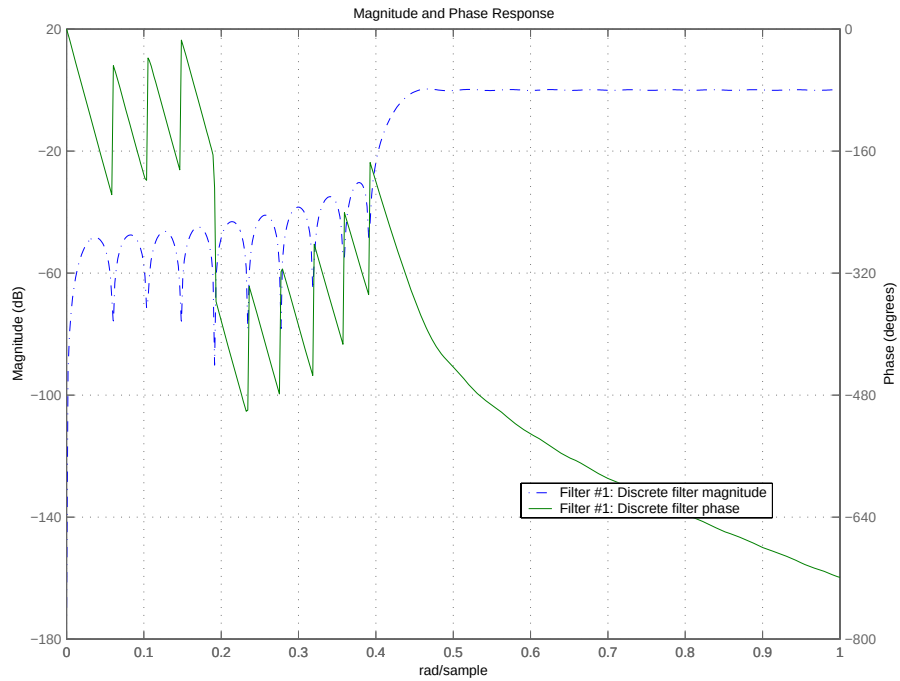
```
[1 1 1 2 2]);
fvtool(b)
```

From the figure you see the resulting filter is lowpass, with the desired 1.4 peak in the passband (notice the 1.4 specified in vector a).

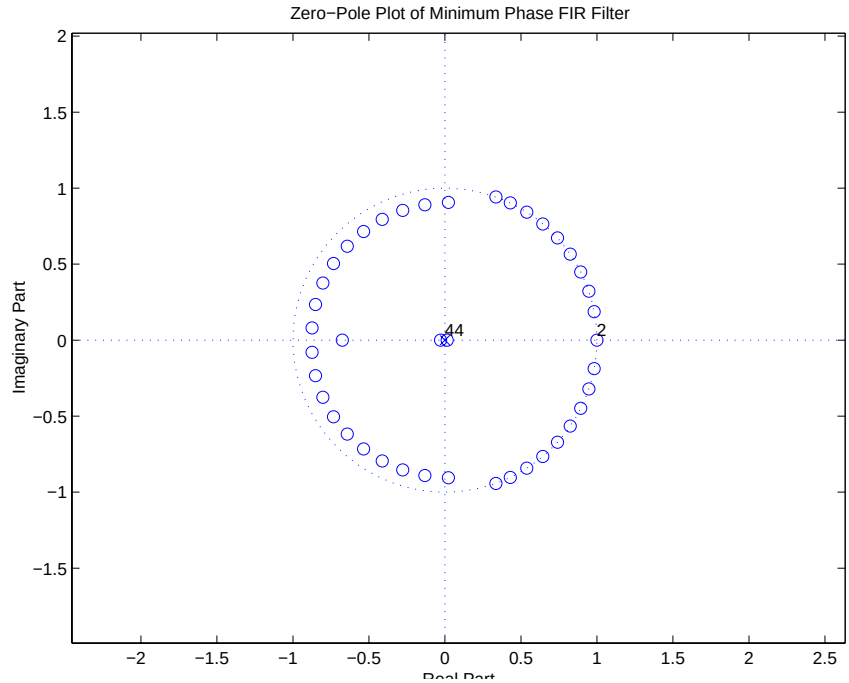
Now for the minimum-phase filter.

```
% Highpass minimum-phase filter optimized for the 4-norm.
b = firlpnorm(44,[0 .4 .45 1],[0 .4 .45 1],[0 0 1 1],[5 1 1 1],...
[2 4],'minphase');
fvtool(b)
```

As shown in the next figure, this is a minimum-phase, highpass filter.



The next zero-pole plot shows the minimum phase nature more clearly.



See Also

`firgr`, `iirgrpdelay`, `iirlpnorm`, `iirlpnormc`

`filter`, `fvtool`, `freqz`, `zplane` in Signal Processing Toolbox documentation

References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Purpose

RLS filter from specification object

Syntax

```
hd = firls(d)
```

Description

`hd = firls(d)` designs a discrete-time FIR filter using a least-squares error minimization method. Only halfband and interpolation specifications objects with `Specification` of 'n,tw' or `'pl,tw'` work as specifications objects for `firls`.

`hd` is either a `dfilt` object (a single-rate digital filter) or an `mfilt` object (a multirate digital filter) depending on the `Specification` property of the filter specification object `d` and the filter specification object type — halfband or interpolator.

Examples

Here are two examples of using `firls` to design filters. The first example returns a single-rate halfband filter using 120 as the filter order..

```
d = fdesign.halfband('n,tw',120,.04);
hd = firls(d);
```

Now use `firls` to design a multirate halfband interpolator filter.

```
d = fdesign.interpolator(2,'pl,tw',60,.04); % 60 is the polyphase
                                            % length.
hm = firls(d);
```

See Also

`equiripple`, `kaiserwin`

Purpose

Minimum-phase FIR spectral factor

Syntax

```
h = firminphase(b)
h = firminphase(b,nz)
```

Description

`h = firminphase(b)` computes the minimum-phase FIR spectral factor `h` of a linear-phase FIR filter `b`. Filter `b` must be real, have even order, and have nonnegative zero-phase response.

`h = firminphase(b,nz)` specifies the number of zeros, `nz`, of `b` that lie on the unit circle. You must specify `nz` as an even number to compute the minimum-phase spectral factor because every root on the unit circle must have even multiplicity. Including `nz` can help `firminphase` calculate the required FIR spectral factor. Zeros with multiplicity greater than two on the unit circle cause problems in the spectral factor determination.

Note You can find the maximum-phase spectral factor, `g`, by reversing `h`, such that $g = fliplr(h)$, and $b = conv(h, g)$.

Example

This example designs a constrained least squares filter with a nonnegative zero-phase response, and then uses `firminphase` to compute the minimum-phase spectral factor.

```
f   = [0 0.4 0.8 1];
a   = [0 1 0];
up  = [0.02 1.02  0.01];
lo  = [0 0.98 0]; % The zeros insure nonnegative zero-phase resp.
n   = 32;
b   = fircls(n,f,a,up,lo);
h   = firminphase(b);
```

See Also

`firgr`

`fircls`, `zerophase` in Signal Processing Toolbox™ documentation

References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Purpose

Lowpass Nyquist (Lth-band) FIR filter

Syntax

```
b = firnyquist(n,l,r)
b = firnyquist('minorder',l,r,dev)
b = firnyquist(n,l,r,decay)
b = firnyquist(n,l,r,'nonnegative')
b = firnyquist(n,l,r,'minphase')
```

Description

`b = firnyquist(n,l,r)` designs an Nth order, Lth band, Nyquist FIR filter with a roll-off factor `r` and an equiripple characteristic.

The rolloff factor `r` is related to the normalized transition width `tw` by $tw = 2\pi(r/l)$ (rad/sample). The order, `n`, must be even. `l` must be an integer greater than one. If `l` is not specified, it defaults to `4`. `r` must satisfy `0< r < 1`. If `r` is not specified, it defaults to `0.5`.

`b = firnyquist('minorder',l,r,dev)` designs a minimum-order, Lth band Nyquist FIR filter with a roll-off factor `r` using the Kaiser window. The peak ripple is constrained by the scalar `dev`.

`b = firnyquist(n,l,r,decay)` designs an Nth order (`n`), Lth band (`l`), Nyquist FIR filter where the scalar `decay`, specifies the rate of decay in the stopband. `decay` must be nonnegative. If you omit or leave it empty, `decay` defaults to `0` which yields an equiripple stopband. `A` nonequiripple stopband (`decay` ≠0) may be desirable for decimation purposes.

`b = firnyquist(n,l,r,'nonnegative')` returns an FIR filter with nonnegative zero-phase response. This filter can be spectrally factored into minimum-phase and maximum-phase “square-root” filters. This allows you to use the spectral factors in applications such as matched-filtering.

`b = firnyquist(n,l,r,'minphase')` returns the minimum-phase spectral factor `bmin` of order `n`. `bmin` meets the condition `b=conv(bmin,bmax)` so that `b` is an Lth band FIR Nyquist filter of order 2n with filter roll-off factor `r`. Obtain `bmax`, the maximum phase spectral factor by reversing the coefficients of `bmin`. For example, `bmax = bmin(end:-1:1)`.

Examples

Example 1

This example designs a minimum phase factor of a Nyquist filter.

```
bmin = firnyquist(47,10,.45,'minphase');
b = firnyquist(2*47,10,.45,'nonnegative');
[h,w,s] = freqz(b); hmin = freqz(bmin);
fvtool(b,1,bmin,1);
```

Example 2

This example compares filters with different decay rates.

```
b1 = firnyquist(72,8,.3,0); % Equiripple
b2 = firnyquist(72,8,.3,.5);
b3 = firnyquist(72,8,.3,1);
fvtool(b1,1,b2,1,b3,1);
```

See Also

`firhalfband`, `firgr`, `firls`, `firminphase`

`firrcos`, `firls` in Signal Processing Toolbox™ documentation

References

T. Saramaki, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Purpose

Two-channel FIR filter bank for perfect reconstruction

Syntax

```
[h0,h1,g0,g1] = firpr2chfb(n,fp)
[h0,h1,g0,g1] = firpr2chfb(n,dev,'dev')
[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)
```

Description

`[h0,h1,g0,g1] = firpr2chfb(n,fp)` designs four FIR filters for the analysis sections (`h0` and `h1`) and synthesis section is (`g0` and `g1`) of a two-channel perfect reconstruction filter bank. The design corresponds to the orthogonal filter banks also known as power-symmetric filter banks.

`n` is the order of all four filters. It must be an odd integer. `fp` is the passband-edge for the lowpass filters `h0` and `g0`. The passband-edge argument `fp` must be less than 0.5. `h1` and `g1` are highpass filters with the passband-edge given by (1-`fp`).

`[h0,h1,g0,g1] = firpr2chfb(n,dev,'dev')` designs the four filters such that the maximum stopband ripple of `h0` is given by the scalar `dev`. Specify `dev` in linear units, not decibels. The stopband-ripple of `h1` is also be given by `dev`, while the maximum stopband-ripple for both `g0` and `g1` is (2*`dev`).

`[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)` designs the four filters such that `h0` meets the passband-edge specification `fp` and the stopband-ripple `dev` using minimum order filters to meet the specification.

Algorithm

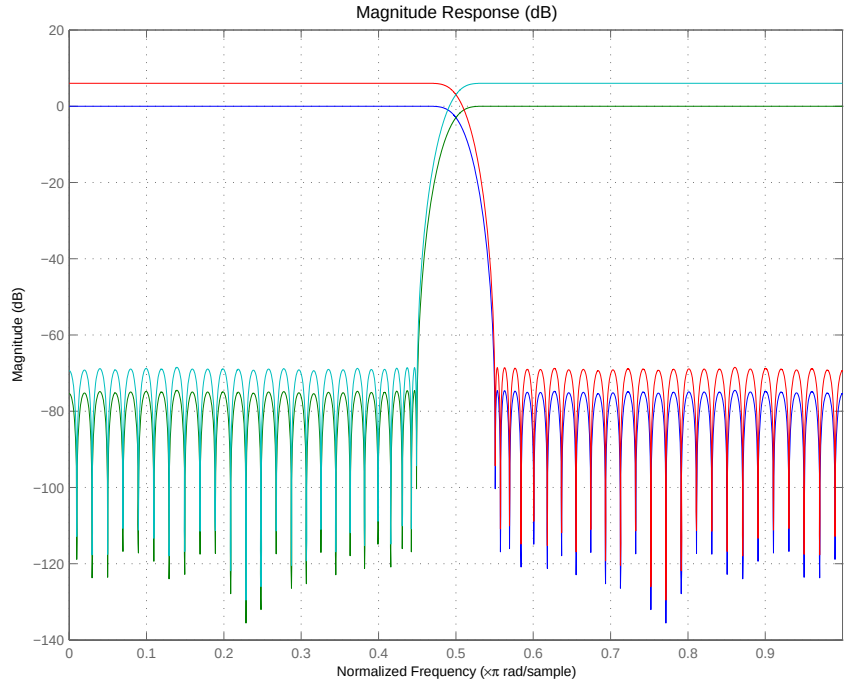
For perfect reconstruction, filters that compose the filter bank must fulfill these conditions.

Examples

Design a filter bank with filters of order `n` equal to 99 and passband edges of 0.45 and 0.55.

```
n = 99;
[h0,h1,g0,g1] = firpr2chfb(n,.45);
fvtool(h0,1,h1,1,g0,1,g1,1);
```

Here are the filters, showing clearly the passband edges.



Use the following stem plots to verify perfect reconstruction using the filter bank created by `firpr2chfb`.

```
stem(1/2*conv(g0,h0)+1/2*conv(g1,h1))
n=0:n;
stem(1/2*conv((-1).^n.*h0,g0)+1/2*conv((-1).^n.*h1,g1))
stem(1/2*conv((-1).^n.*g0,h0)+1/2*conv((-1).^n.*g1,h1))
stem(1/2*conv((-1).^n.*g0,(-1).^n.*h0)+...
1/2*conv((-1).^n.*g1,(-1).^n.*h1))
stem(conv((-1).^n.*h1,h0)-conv((-1).^n.*h0,h1))
```

See Also

`firceqrip`, `firgr`, `firhalfband`, `firnyquist`

Purpose

Type of linear phase FIR filter

Syntax

```
t = firtype(hd)
t = firtype(hm)
```

Description

The next sections describe common `firtype` operation with discrete-time and multirate filters.

Discrete-Time Filters

`t = firtype(hd)` determines the type (1 through 4) of a discrete-time FIR filter object `hd`, returning the type number in `t`. Filter `hd` must be both real and have linear phase.

Filter types 1 through 4 are defined as follows:

- Type 1 — even order symmetric coefficients
- Type 2 — odd order symmetric coefficients
- Type 3 — even order antisymmetric coefficients
- Type 4 — odd order antisymmetric coefficients

When `hd` is a cascade or parallel filter and therefore has multiple stages, each stage must be a real FIR filter with linear phase. In this case, `t` is a cell array containing the filter type of each stage.

Multirate Filters

`t = firtype(hm)` determines the type (1 through 4) of the multirate filter object `hm`. The filter must be real and have linear phase.

Filter types 1 through 4 are defined as follows:

- Type 1 — even order symmetric coefficients
- Type 2 — odd order symmetric coefficients
- Type 3 — even order antisymmetric coefficients
- Type 4 — odd order antisymmetric coefficients

When `hm` has multiple sections, all sections must be real FIR filters with linear phase. In this case, `t` is a cell array containing the filter type of each section.

Examples

Determine the type of the default interpolator for L=4.

```
l = 4;
hm = mfilt.firinterp(l);
firtype(hm)
ans =

     1
```

See Also

`islinphase`

Purpose

Estimate fixed-point filter frequency response through filtering

Syntax

```
[h,w] = freqrespest(hd,L)
[h,w] = freqrespest(hd,L,param1,value1,param2,
value2,...)
freqrespest(hd,L,opts)
```

Description

`[h,w] = freqrespest(hd,L)` estimates the frequency response of filter `hd` by filtering a set of input data and then forming the ratio between output data and input data. The test input data comprises sinusoids with uniformly distributed random frequencies.

Use this filter-based technique for judging the performance of fixed-point filters. Because you can compare a filtering-based frequency response estimate for a fixed-point filter to the response of a similar filter that uses quantized coefficients, but applies floating-point arithmetic internally. This comparison determines whether the fixed-point filter performance closely matches the floating-point, quantized coefficients version of the filter.

`L` is the number of trials to use to compute the estimate. If you do not specify this value, `L` defaults to 10. More trials generates a more accurate estimate of the response, but require more time to compute the estimate.

`h` is the estimate of the complex frequency response. `w` contains the vector of frequencies at which `h` is estimated.

Refer to example 2 for one way to plot `h` with `w`.

`[h,w] = freqrespest(hd,L,param1,value1,param2, value2,...)` uses parameter value (PV) pairs as input arguments to specify optional parameters for the test. These parameters are the valid PV pairs. Enter the parameter names as string input arguments in single quotation marks. The following table provides valid parameters for `[h, w]`.

Parameter Name	Default Value	Description
`NFFT`	`512`	Number of FFT points to use.
`NormalizedFrequency`	`true`	Indicates whether to use normalized frequency or linear frequency. Values are `true` (use normalized frequency), or `false` (use linear frequency). When you specify `false`, you must supply the sampling frequency `Fs`.
`Fs`	`normalized`	Specifies the sampling frequency when `NormalizedFrequency` is `false`. No default value. You must set `NormalizedFrequency` to `false` before setting a value for `Fs`.
`SpectrumRange`	`half`	Specifies whether to use the whole spectrum or half. `half` is the default, and the valid values are `half` and `whole`.
`CenterDC`	`false`	Specifies whether to set the center of the spectrum to the DC value in the output plot. If you select `true`, both the negative and positive values appear in the plot. If you select `false` DC appears at the origin of the axes.

`freqrespest(hd,L,opts)` uses an object `opts` to specify the optional input parameters instead of directly specifying PV pairs as input arguments. Create `opts` with

```
opts = freqrespopts(hd);
```

Because `opts` is an object, you use `set` to change the parameter values in `opts` before you use it with `freqrespest`. For example, you could specify a new sample rate with

```
set(opts,'fs',48e3); % Same as opts.fs=48e3
```

`freqrespest` can also compute the frequency response of double-precision floating filters that cannot be converted to transfer-function form without introducing significant round off errors which affect the `freqz` frequency response computation. Examples of these kinds of filters include state-space or lattice filters, in particular high-order filters.

Examples

These examples demonstrate some uses for `freqrespest`.

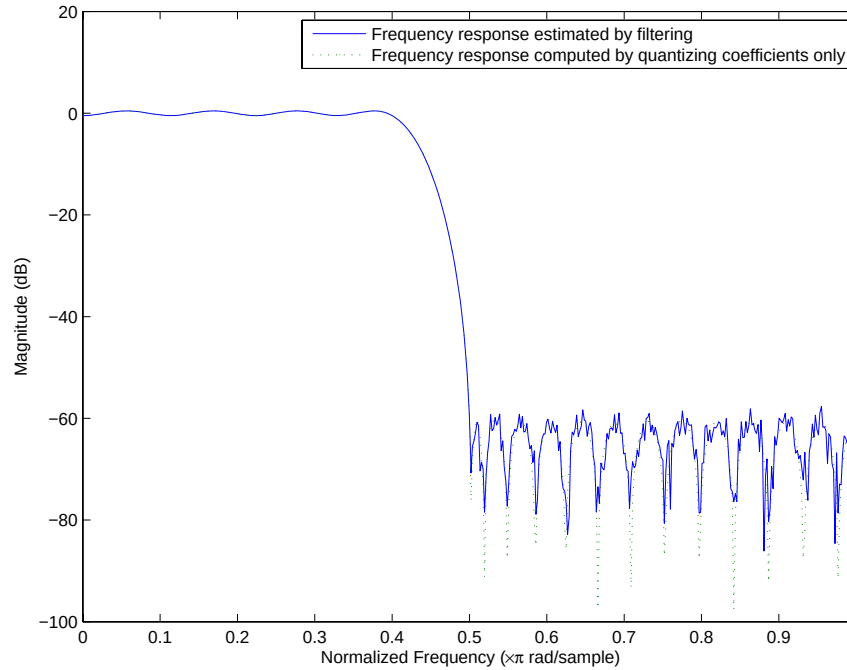
Start by estimating the frequency response of a fixed-point FIR filter that has filter internals set to `full precision`.

```
hd = design(fdesign.lowpass(.4,.5,1,60),'equiripple');
hd.arithmetic = 'fixed';
[h,w] = freqrespest(hd); % This should be about the same as freqz.
```

Continuing with filter `hd`, change the value of the `filterinternals` property to `specifyprecision` and then specify the word lengths and precision (the fraction lengths) applied to the output from internal addition and multiplication operations. After you set the word and fraction lengths, use `freqrespest` to compute the frequency response estimate for the fixed-point filter.

```
hd.filterinternals = 'specifyprecision';
hd.outputwordlength=16;
hd.outputfraclength=15;
hd.productwordlength=16;
```

```
hd.productfraclength=15;
hd.accumwordlength=16;
hd.accumfraclength=15;
[h,w] = freqrespest(hd,2);
[h2,w2] = freqz(hd,512);
plot(w/pi,20*log10(abs([h,h2])))
legend('Frequency response estimated by filtering',...
'Freq. response computed by quantizing coefficients only');
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
```



`freqrespest` works with state-space filters as well. This example estimates the frequency response of a state-space filter.

```
fs = 315000;
```

```
wp = [320 3800]/(fs/2);
ws = [50 19000]/(fs/2);
rp=0.15; rs=60;
[n,wn]=cheb1ord(wp,ws,rp,rs);
[a,b,c,d] = cheby1(n,rp,wn);
hd = dfilt.statespace(a,b,c,d);
% Compare the following to freqz(hd,8192)
freqrespest(hd,1,'nfft',8192);
```

See Also

dfilt, freqrespopts, freqz, limitcycle, noisepsd, scale

freqrespopts

Purpose

freqrespest parameters and values

Syntax

```
opts = freqrespopts(hd)
```

Description

`opts = freqrespopts(hd)` uses the settings in filter `hd` to create an object `opts` that contains parameters and values for estimating the filter frequency response. You pass `opts` as an input argument to `freqrespest` to specify values for the input parameters.

With `freqrespopts` you can use the same settings for `freqrespest` with multiple filters without having to specify all of the parameters as input arguments to `freqrespest`.

Examples

This example shows `freqrespopts` in use for setting options for `freqrespest`. `hd` and `hd2` are bandpass filters that use different design methods. The `opts` object makes it easier to set the same conditions for the frequency response estimate in `freqrespest`.

```
d=fdesign.bandpass('fst1,fp1,fp2,fst2,ast1,ap,ast2',...
0.25,0.3,0.45,0.5,60,0.1,60);

hd=design(d,'butter');
hd.arithmetic='fixed';
hd2=design(d,'cheby2')
hd2.arithmetic='fixed';
opts=freqrespopts(hd)

opts =

                    NFFT: 512
    NormalizedFrequency: true
                     Fs: 'Normalized'
           SpectrumRange: 'Half'
                CenterDC: false

opts.NFFT=256; % Same as set(opts,'nfft',256).
opts.NormalizedFrequency=false;
```

```
opts.fs=1.5e3;
opts.CenterDC=true

opts =

                    NFFT: 256
    NormalizedFrequency: false
                     Fs: 1500
           SpectrumRange: 'Whole'
                CenterDC: true
```

With `opts` configured as needed, use it as an input argument for `freqrespest`.

```
[h2,w2]=freqrespest(hd2,20,opts);
[h1,w1]=freqrespest(hd,20,opts);
```

See Also

`freqrespest`, `noisepsd`, `noisepsdopts`, `norm`, `scale`

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 `fspecifications object` `h`.

`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 chebwin, or hamming. Click `window` for the full list of windows available or refer to `window` 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 final example shows a cell array input argument.
- The window vector itself.

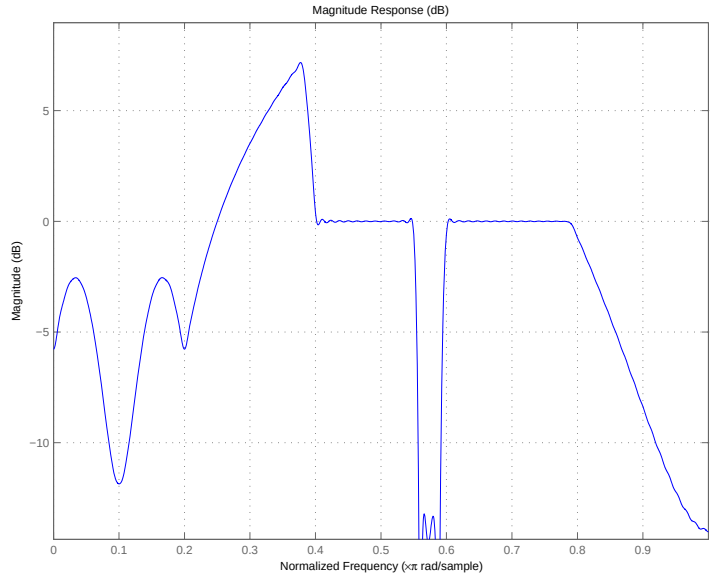
Examples

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)
```

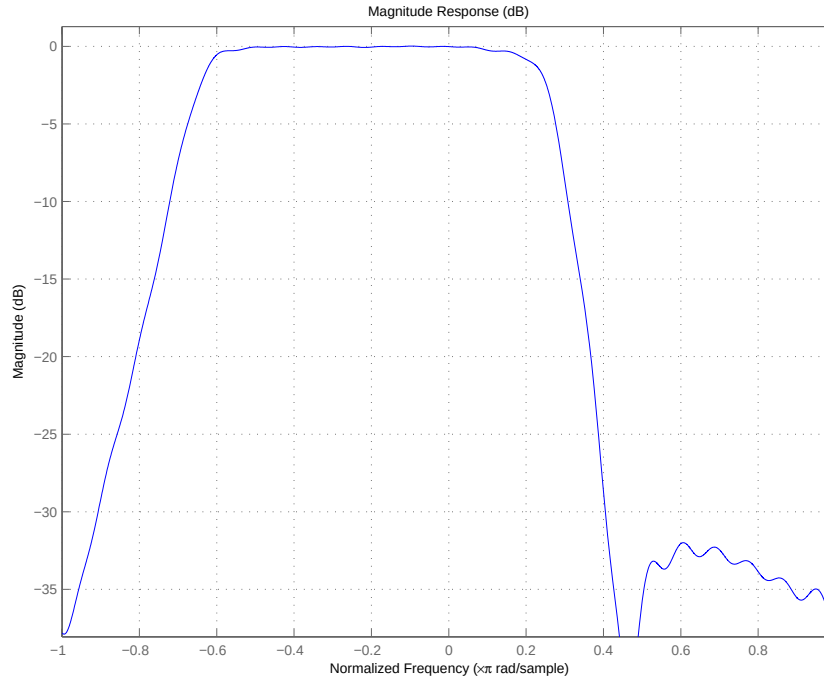
The plot from FVTool shows the response for `hd`.



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. `design(d,...)` returns a complex filter for `hdc` because the frequency vector includes negative frequency values.



See Also

`design`, `designmethods`, `fdesign.arbmag`, `help window` in the Signal Processing Toolbox documentation

Purpose

Frequency response of filter

Syntax

```
[h,w] = freqz(ha)
[h,w] = freqz(ha,n)
freqz(ha)
[h,w] = freqz(hd)
[h,w] = freqz(hd,n)
freqz(hd)
[h,w] = freqz(hm)
[h,w] = freqz(hd,n)
freqz(hd)
```

Description

The next sections describe common `freqz` operation with adaptive, discrete-time, and multirate filters. For more input options, refer to `freqz` in Signal Processing Toolbox™ documentation.

Adaptive Filters

For adaptive filters, `freqz` returns the instantaneous frequency response based on the current filter coefficients.

`[h,w] = freqz(ha)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the adaptive filter `ha`. When `ha` is a vector of adaptive filters, `freqz` returns the matrix `h`. Each column of `h` corresponds to one filter in the vector `ha`.

`[h,w] = freqz(ha,n)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the adaptive filter `ha`. `freqz` uses the transfer function associated with the adaptive filter to calculate the frequency response of the filter with the current coefficient values. The vectors `h` and `w` are both of length `n`. The 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 8192 samples for the FFT.

`freqz(ha)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the adaptive filter `ha`. If `ha` is a vector of filters, `freqz` plots the magnitude response and phase for each filter in the vector.

Discrete-Time Filters

`[h,w] = freqz(hd)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the discrete-time filter `hd`. When `hd` is a vector of discrete-time filters, `freqz` returns the matrix `h`. Each column of `h` corresponds to one filter in the vector `hd`.

`[h,w] = freqz(hd,n)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the discrete-time filter `hd`. `freqz` uses the transfer function associated with the discrete-time filter to calculate the frequency response of the filter with the current coefficient values. The vectors `h` and `w` are both of length `n`. The 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 8192 samples for the FFT.

`freqz(hd)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the adaptive filter `hd`. If `hd` is a vector of filters, `freqz` plots the magnitude response and phase for each filter in the vector.

Multirate Filters

`[h,w] = freqz(hm)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the multirate filter `hd`. When `hd` is a vector of multirate filters, `freqz` returns the matrix `h`. Each column of `h` corresponds to one filter in the vector `hd`.

`[h,w] = freqz(hd,n)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the multirate filter `hd`. `freqz` uses the `transfer` function associated with the multirate filter to calculate the frequency response of the filter with the current coefficient values. The vectors `h` and `w` are both of length `n`. The 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 8192 samples for the FFT.

`freqz(hd)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the adaptive filter `hd`. If `hd` is a vector of filters, `freqz` plots the magnitude response and phase for each filter in the vector.

Remarks

There are several ways of analyzing the frequency response of filters. `freqz` accounts for quantization effects in the filter coefficients, but does not account for quantization effects in filtering arithmetic. To account for the quantization effects in filtering arithmetic, refer to function `noisepsd`.

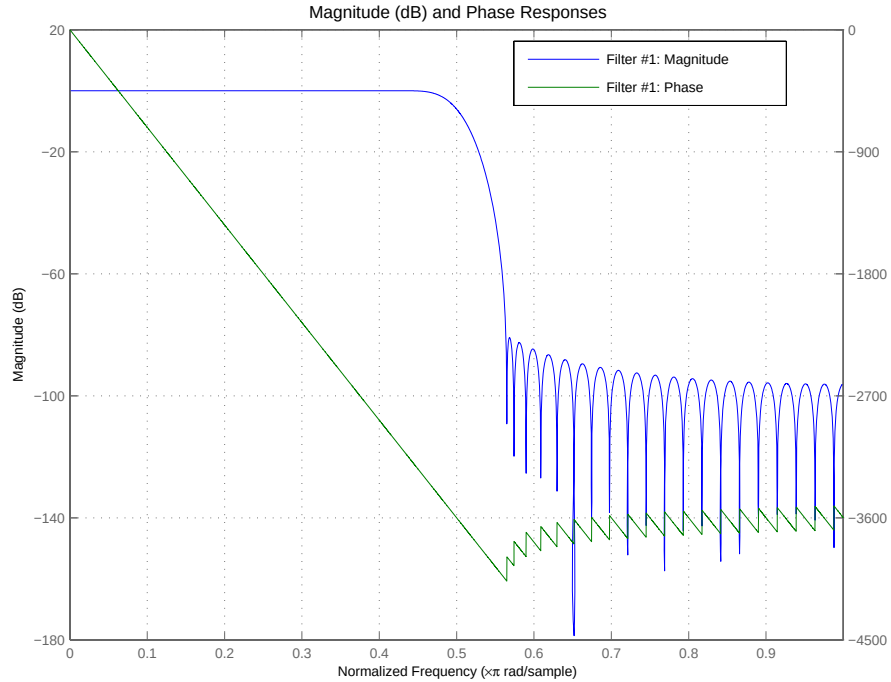
Algorithm

`freqz` calculates the frequency response for a filter from the filter transfer function $Hq(z)$. The complex-valued frequency response is calculated by evaluating $Hq(e^{j\omega})$ at discrete values of w specified by the syntax you use. The integer input argument `n` determines the number of equally-spaced points around the upper half of the unit circle at which `freqz` evaluates the frequency response. The frequency ranges from 0 to π radians per sample when you do not supply a sampling frequency as an input argument. When you supply the scalar sampling frequency `fs` as an input argument to `freqz`, the frequency ranges from 0 to `fs`/2 Hz.

Examples

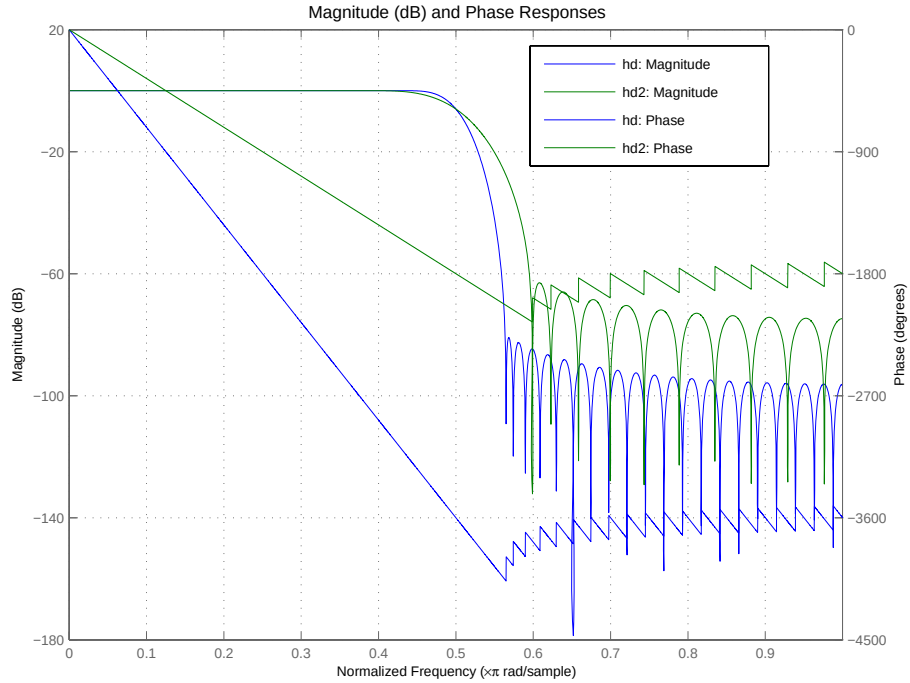
Plot the estimated frequency response of a filter. This example uses discrete-time filters, but any `adaptfilt`, `dfilt`, or `mfilt` object would work. First plot the results for one filter.

```
b = fir1(80,0.5,kaiser(81,8));
hd = dfilt.dffir(b);
freqz(hd);
```



If you have more than one filter, you can plot them on the same figure using a vector of filters.

```
b = fir1(40,0.5,kaiser(41,6));
hd2 = dfilt.dffir(b);
h = [hd hd2];
freqz(h);
```



See Also

`adaptfilt`, `dfilt`, `mfilt`

`fvtool` in Signal Processing Toolbox documentation

Purpose

CIC filter gain

Syntax

```
gain(hm)
gain(hm,j)
```

Description

`gain(hm)` returns the gain of `hm`, the CIC decimation or interpolation filter.

When `hm` is a decimator, `gain` returns the gain for the overall CIC decimator.

When `hm` is an interpolator, the CIC interpolator inserts zeros into the input data stream, reducing the filter overall gain by $1/R$, where R is the interpolation factor, to account for the added zero valued samples.

Therefore, the gain of a CIC interpolator is $(RM)^N/R$, where N is the number of filter sections and M is the filter differential delay. `gain(hm)` returns this value. The example below presents this case.

`gain(hm,j)` returns the gain of the `j`th section of a CIC interpolation filter. When you omit `j`, `gain` assumes that `j` is 2*N, where N is the number of sections, and returns the gain of the last section of the filter. This syntax does not apply when `hm` is a decimator.

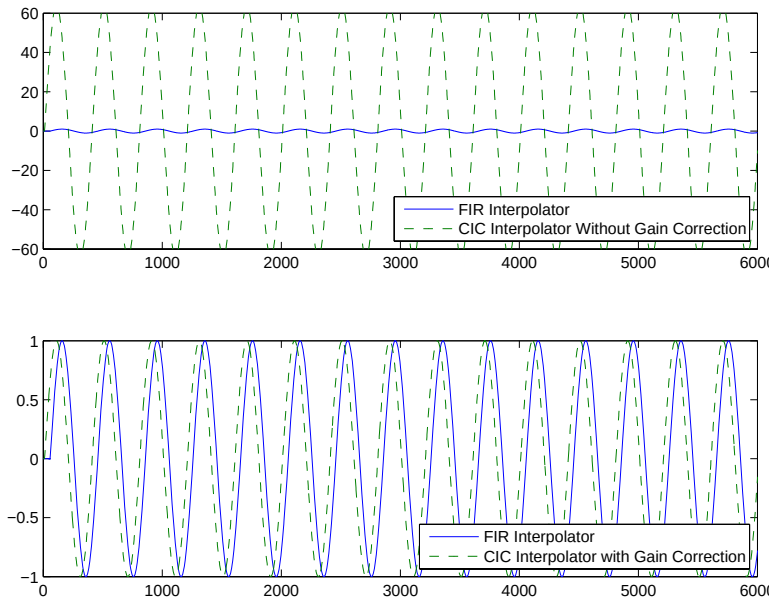
Examples

To compare the performance of two interpolators, one a CIC filter and the other an FIR filter, use gain to adjust the CIC filter output amplitude to match the FIR filter output amplitude. Start by creating an input data set — a sinusoidal signal `x`.

```
fs = 1000;          % Input sampling frequency.
t = 0:1/fs:1.5;     % Signal length = 1501 samples.
x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.

l = 4; % Interpolation factor for FIR filter.
d = fdesign.interpolator(l);
hm = design(d,'multistage');
ym = filter(hm,x);

r = 4; % Interpolation factor for the CIC filter.
```

```
d = fdesign.interpolator(r,'cic');
hcic = design(d,'multisection');
ycic = filter(hcic,x);
gaincic = gain(hcic);
subplot(211);
plot(1:length(ym),[ym; double(ycic)]);
subplot(212)
plot(1:length(ym),[ym; double(ycic)/gain(hcic)]);
```

After correcting for the gain induced by the CIC interpolator, the figure below shows the filters provide nearly identical interpolation.



See Also

`scale`

Purpose

Filter group delay

Syntax

```
[gd,w] = grpdelay(ha)
[gd,w] = grpdelay(ha,n)
grpdelay(ha)
[gd,w] = grpdelay(hd)
[gd,w] = grpdelay(hd,n)
grpdelay(hd)
[gd,w] = grpdelay(hm)
[gd,w] = grpdelay(hm,n)
grpdelay(hm)
```

Description

The next sections describe common `grpdelay` operation with adaptive, discrete-time, and multirate filters. For more input options, refer to `grpdelay` in Signal Processing Toolbox™ documentation.

Adaptive Filters

For adaptive filters, `grpdelay` returns the instantaneous group delay based on the current filter coefficients.

`[gd,w] = grpdelay(ha)` returns the group delay vector `gd` and the corresponding frequency vector `w` for the adaptive filter `ha`. When `ha` is a vector of adaptive filters, `grpdelay` returns the matrix `gd`. Each column of `gd` corresponds to one filter in the vector `ha`. If you provide a row vector of frequency points `f` as an input argument, each row of `gd` corresponds to one filter in the vector.

Function `grpdelay` uses the transfer function associated with the adaptive filter to calculate the group delay of the filter with the current coefficient values. The vectors `gd` and `w` are both of length `n`. The 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 8192 samples for the FFT.

`[gd,w] = grpdelay(ha,n)` returns length `n` vectors vector `gd` containing the current group delay for the adaptive filter `ha` and the

vector w which contains the frequencies in radians at which grpdelay calculated the delay. Group delay is

$$-\frac{d}{dw}(angle(w))$$

The frequency response is evaluated at n points equally spaced around the upper half of the unit circle. For FIR filters where n is a power of two, the computation is done faster using FFTs. When you do not specify n, it defaults to 8192.

grpdelay(ha) uses FVTool to plot the group delay of the adaptive filter ha. If ha is a vector of filters, grpdelay plots the magnitude response and phase for each filter in the vector.

Discrete-Time Filters

[gd,w] = grpdelay(hd) returns the group delay vector gd and the corresponding frequency vector w for the discrete-time filter hd. When hd is a vector of discrete-time filters, grpdelay returns the matrix gd. Each column of gd corresponds to one filter in the vector hd. If you provide a row vector of frequency points f as an input argument, each row of gd corresponds to each filter in the vector.

Function grpdelay uses the transfer function associated with the discrete-time filter to calculate the group delay of the filter. The vectors gd and w are both of length n. The 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 8192 samples for the FFT.

[gd,w] = grpdelay(hd,n) returns length n vectors vector gd containing the current group delay for the discrete-time filter hd and the vector w which contains the frequencies in radians at which grpdelay calculated the delay. Group delay is

$$-\frac{d}{dw}(angle(w))$$

The frequency response is evaluated at n points equally spaced around the upper half of the unit circle. For FIR filters where n is a power

of two, the computation is done faster using FFTs. When you do not specify `n`, it defaults to 8192.

`grpdelay(hd)` uses FVTool to plot the group delay of the discrete-time filter `hd`. If `hd` is a vector of filters, `grpdelay` plots the magnitude response and phase for each filter in the vector.

Multirate Filters

`[gd,w] = grpdelay(hm)` returns the group delay vector `gd` and the corresponding frequency vector `w` for the multirate filter `hm`. When `hm` is a vector of multirate filters, `grpdelay` returns the matrix `gd`. Each column of `gd` corresponds to one filter in the vector `hm`. If you provide a row vector of frequency points `f` as an input argument, each row of `gd` corresponds to one filter in the vector.

Function `grpdelay` uses the transfer function associated with the multirate filter to calculate the group delay of the filter. The vectors `gd` and `w` are both of length `n`. The 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 8192 samples for the FFT.

`[gd,w] = grpdelay(hm,n)` returns length `n` vectors vector `gd` containing the group delay for the multirate filter `hm` and the vector `w` which contains the frequencies in radians at which `grpdelay` calculated the delay. Group delay is

$$-\frac{d}{dw}(angle(w))$$

The frequency response is evaluated at `n` points equally spaced around the upper half of the unit circle. For FIR filters where `n` is a power of two, the computation is done faster using FFTs. When you do not specify `n`, it defaults to 8192.

`grpdelay(hm)` uses FVTool to plot the magnitude and unwrapped phase of the group delay of the multirate filter `hm`. If `ha` is a vector of filters, `grpdelay` plots the group delay for each filter in the vector.

See Also

phasez, zerophase

Purpose

Help for design method with filter specification

Syntax

```
help(d,'designmethod')
```

Description

`help(d,'designmethod')` displays help in the Command Window for the design algorithm `designmethod` for the current specifications of the filter specification object `d`. The string you enter for `designmethod` must be one of the strings returned by `designmethods` for `d`, the design object.

Examples

Get specific help for designing lowpass Butterworth filters. The first lowpass filter uses the default specification string '`Fp,Fst,Ap,Ast`' and returns help text specific to the specification string.

```
d = fdesign.lowpass;
designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):

butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage

help(d,'butter')

DESIGN Design a Butterworth IIR filter.
HD = DESIGN(D, 'butter') designs a Butterworth filter specified
by the FDESIGN object D.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter
with the structure STRUCTURE.  STRUCTURE is 'df2sos' by default
and can be any of the following.
```

```
'df1sos'
'df2sos'
'df1tsos'
'df2tsos'

HD = DESIGN(..., 'MatchExactly', MATCH) designs a Butterworth
filter and matches the frequency and magnitude specification for
the band MATCH exactly. The other band will exceed the
specification.  MATCH can be 'stopband' or 'passband' and is
'stopband' by default.

% Example #1 - Compare passband and stopband MatchExactly.
h     = fdesign.lowpass('Fp,Fst,Ap,Ast', .1, .3, 1, 60);
Hd    = design(h, 'butter', 'MatchExactly', 'passband');
Hd(2) = design(h, 'butter', 'MatchExactly', 'stopband');

% Compare the passband edges in FVTool.
fvtool(Hd);
axis([.09 .11 -2 0]);
```

Note the discussion of the `MatchExactly` input option. When you use a design object that uses a different specification string, such as `'N,F3dB'`, the help content for the `butter` design method changes.

In this case, the `MatchExactly` option does not appear in the help because it is not an available input argument for the specification string `'N,F3dB'`.

```
d=fdesign.lowpass('N,F3dB')

d =
                Response: 'Lowpass'
           Specification: 'N,F3dB'
             Description: {'Filter Order';'3dB Frequency'}
    NormalizedFrequency: true
             FilterOrder: 10
                   F3dB: 0.5
```

```
   designmethods(d)

   Design Methods for class fdesign.lowpass (N,F3dB):

   butter

   help(d,'butter
```

```
DESIGN Design a Butterworth IIR filter.

HD = DESIGN(D, 'butter') designs a Butterworth filter
specified by the FDESIGN object D.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a
filter with the structure STRUCTURE. STRUCTURE is 'df2sos'
by default and can be any of the following.

'df1sos'

'df2sos'

'df1tsos'

'df2tsos'

% Example #1 - Design a lowpass Butterworth filter in the
DF2TSOS structure.

h = fdesign.lowpass('N,F3dB');

Hd = design(h, 'butter', 'FilterStructure', 'df2tsos');
```

See Also

fdesign, design, designmethods, designopts

Purpose

Interpolated FIR filter from filter specification

Syntax

```
hd = ifir(d)
hd = design(d,'ifir',designoption,value,designoption,...
value,...)
```

Description

`hd = ifir(d)` designs an FIR filter from design object `d`, using the interpolated FIR method. `ifir` returns `hd` as a cascade of two filters that act together to meet the specifications in `d`. The resulting filter is particularly efficient, having a low number of multipliers. However, if `ifir` determines that a single-stage filter would be more efficient than the default two-stage design, it returns `hd` as a single-stage filter. `ifir` only creates linear phase filters. Generally, `ifir` uses an advanced optimization algorithm to create highly efficient FIR filters.

`ifir` returns `hd` as either a single-rate `dfilt` object or a multirate `mfilt` object (when you have Filter Design Toolbox™ software installed), based on the specifications you provide in `d`, the filter specification object.

specifications supplied in the object `h`.

`hd = design(d,'ifir',designoption,value,designoption,...
value,...)` returns an interpolated 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 `ifir`, refer to the command line help system. For example, to get specific information about using `ifir` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'ifir')
```

Note For help about how you use `ifir` to design filters without using design objects, enter

```
help ifir
```

at the MATLAB prompt.

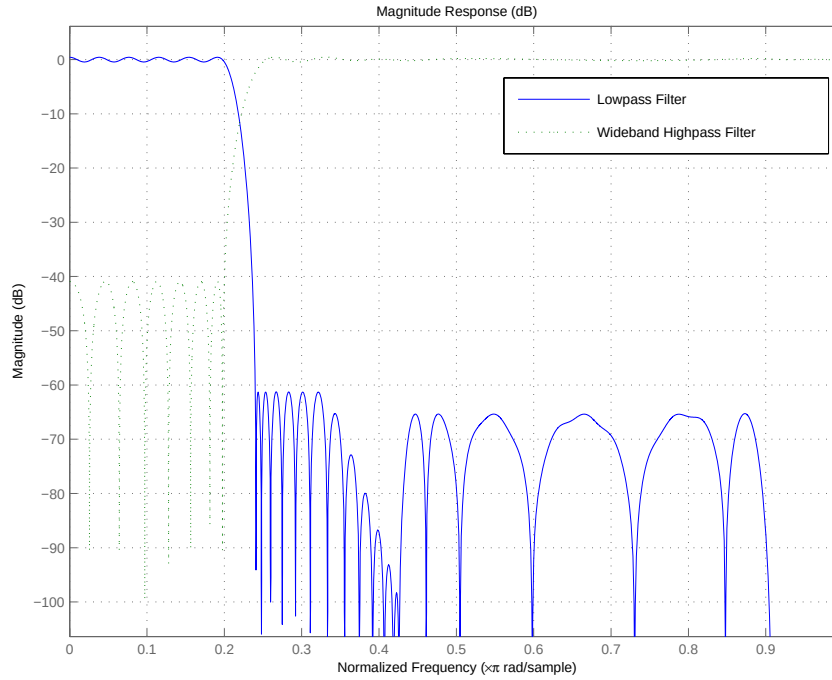
Examples

Use `fdesign.lowpass` and `fdesign.highpass` to design a lowpass filter and a wideband highpass filter. After designing the filters, use FVTool to plot the response curves for both.

```
fpass = 0.2;
fstop = 0.24;
d1 = fdesign.lowpass(fpass, fstop);
hd1 = design(d1,'ifir');
fstop = 0.2;
fpass = 0.25;
astop = 40;
apass = 1;
d2 = fdesign.highpass(fstop,fpass,astop,apass);
hd2 = design(d2,'ifir');
```

Here are the magnitude response curves for both filters.

```
fvtool(hd1,hd2)
```



See Also

`fdesign`, `firgr`

`fir1`, `firls`, `firpm` in Signal Processing Toolbox™ documentation

Purpose

Transform IIR complex bandpass filter to IIR complex bandpass filter with different characteristics

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations, W_{t1}, and W_{t2} respectively. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle; e.g., in the adaptive system.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

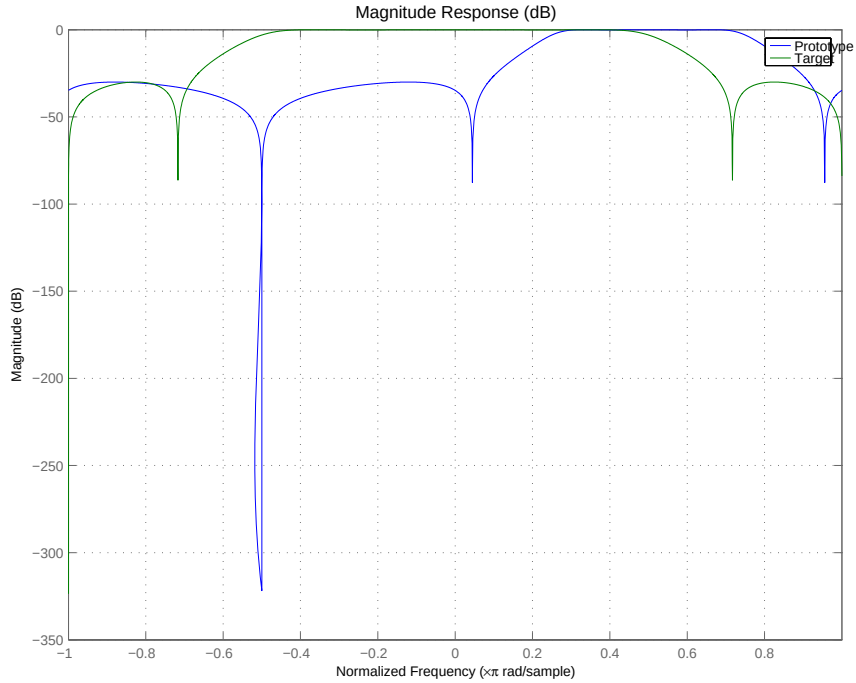
Create a complex passband from 0.25 to 0.75:

```
[b, a] = iirlp2bpc (b, a, 0.5, [0.25,0.75]);
[num, den] = iirbpc2bpc(b, a, [0.25, 0.75], [-0.5, 0.5]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Using FVTool to plot the filters shows you the comparison, presented in this figure.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpassbpc2bpc`, `zpkbpc2bpc`

Purpose

IIR comb notch or peak filter

Syntax

```
[num,den] = iircomb(n,bw)
[num,den] = iircomb(n,bw,ab)
[num,den] = iircomb(...,'type')
```

Description

`[num,den] = iircomb(n,bw)` returns a digital notching filter with order `n` and with the width of the filter notch at -3 dB set to `bw`, the filter bandwidth. The filter order must be a positive integer. `n` also defines the number of notches in the filter across the frequency range from 0 to 2π — the number of notches equals `n`+1.

For the notching filter, the transfer function takes the form

$$H(z) = b \times \frac{1 - z^{-n}}{1 - az^{-n}}$$

where a and b are the filter coefficients and n is the filter order or the number of notches in the filter minus 1.

The quality factor (Q factor) q for the filter is related to the filter bandwidth by $q = \omega_0/bw$ where ω_0 is the frequency to remove from the signal.

`[num,den] = iircomb(n,bw,ab)` returns a digital notching filter whose bandwidth, `bw`, is specified at a level of `-ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default -3 dB point, such as -6 dB or 0 dB.

`[num,den] = iircomb(...,'type')` returns a digital filter of the specified type. The input argument `type` can be either

- `'notch'` to design an IIR notch filter. Notch filters attenuate the response at the specified frequencies. This is the default type. When you omit the `type` input argument, `iircomb` returns a notch filter.
- `'peak'` to design an IIR peaking filter. Peaking filters boost the signal at the specified frequencies.

The transfer function for peaking filters is

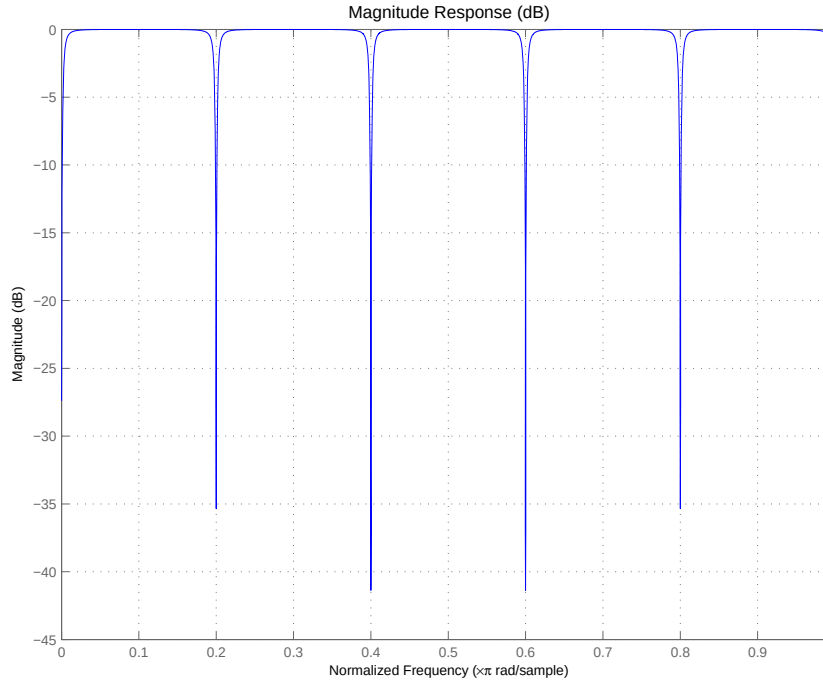
$$H(z) = b \times \frac{1 + z^{-n}}{1 - az^{-n}}$$

Examples

Design and plot an IIR notch filter with 11 notches (equal to filter order plus 1) that removes a 60 Hz tone (`f0`) from a signal at 600 Hz (`fs`). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth.

```
fs = 600; fo = 60;  q = 35; bw = (fo/(fs/2))/q;
[b,a] = iircomb(fs/fo,bw,'notch'); % Note type flag 'notch'
fvtool(b,a);
```

Using the Filter Visualization Tool (FVTool) generates the following plot showing the filter notches. Note the notches are evenly spaced and one falls at exactly 60 Hz.



See Also

`firgr`, `iirnotch`, `iirpeak`

Purpose

IIR frequency transformation of filter

Syntax

```
[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)
```

Description

`[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)` returns the numerator and denominator vectors, `OutNum` and `OutDen`, of the target filter, which is the result of transforming the prototype filter specified by the numerator, `OrigNum`, and denominator, `OrigDen`, with the mapping filter given by the numerator, `FTFNum`, and the denominator, `FTFDen`. If the allpass mapping filter is not specified, then the function returns an original filter.

Examples

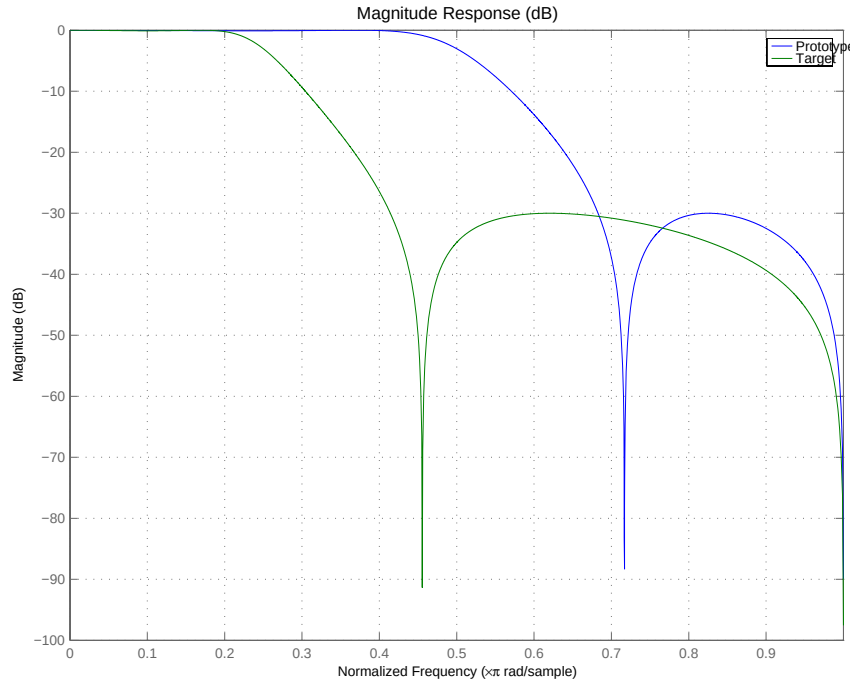
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[num, den] = iirftransf(b, a, AlpNum, AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Here's the comparison between the filters.



Arguments

Variable	Description
OrigNum	Numerator of the prototype lowpass filter
OrigDen	Denominator of the prototype lowpass filter
FTFNum	Numerator of the mapping filter
FTFDen	Denominator of the mapping filter
OutNum	Numerator of the target filter
OutDen	Denominator of the target filter

See Also

`zpkftransf`

Purpose

Optimal IIR filter with prescribed group-delay

Syntax

```
[num,den] = iirgrpdelay(n,f,edges,a)
[num,den] = iirgrpdelay(n,f,edges,a,w)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,
   tau)
[num,den,tau] = iirgrpdelay(n,f,edges,a,w)
```

Description

`[num,den] = iirgrpdelay(n,f,edges,a)` returns an allpass IIR filter of order `n` (`n` must be even) which is the best approximation to the relative group-delay response described by `f` and `a` in the least-pth sense. `f` is a vector of frequencies between 0 and 1 and `a` is specified in samples. The vector `edges` specifies the band-edge frequencies for multi-band designs. `iirgrpdelay` uses a constrained Newton-type algorithm. Always check your resulting filter using `grpdelay` or `freqz`.

`[num,den] = iirgrpdelay(n,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point and must be the same length as `f` and `a`). Entries in `w` tell `iirgrpdelay` how much emphasis to put on minimizing the error in the vicinity of each specified frequency point relative to the other points.

`f` and `a` must have the same number of elements. `f` and `a` can contains more elements than the vector `edges` contains. This lets you use `f` and `a` to specify a filter that has any group-delay contour within each band.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius)` returns a filter having a maximum pole radius equal to `radius`, where 0<`radius`<1. `radius` defaults to 0.999999. Filters whose pole radius you constrain to be less than 1.0 can better retain transfer function accuracy after quantization.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)`, where `p` is a two-element vector [`pmin pmax`], lets you determine the minimum and maximum values of `p` used in the least-pth algorithm. `p` defaults to [2

128] which yields filters very similar to the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even. If `p` is the string `'inspect'`, no optimization occurs. You might use this feature to inspect the initial pole/zero placement.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens)` specifies the grid density `dens` used in the optimization process. The number of grid points is `(dens*(n+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden)` allows you to specify the initial estimate of the denominator coefficients in vector `initden`. This can be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox™ software can be used for generating `initden`.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,tau)` allows the initial estimate of the group delay offset to be specified by the value of `tau`, in samples.

`[num,den,tau] = iirgrpdelay(n,f,edges,a,w)` returns the resulting group delay offset. In all cases, the resulting filter has a group delay that approximates `[a + tau]`. Allpass filters can have only positive group delay and a non-zero value of tau accounts for any additional group delay that is needed to meet the shape of the contour specified by `(f,a)`. The default for `tau` is `max(a)`.

Hint: If the zeros or poles cluster together, your filter order may be too low or the pole radius may be too small (overly constrained). Try increasing `n` or `radius`.

For group-delay equalization of an IIR filter, compute `a` by subtracting the filter's group delay from its maximum group delay. For example,

```
[be,ae] = ellip(4,1,40,0.2);
f = 0:0.001:0.2;
g = grpdelay(be,ae,f,2);   % Equalize only the passband.
a = max(g)-g;
```

```
[num,den]=iirgrpdelay(8, f, [0 0.2], a);
```

See Also

freqz, filter, grpdelay, iirlpnorm, iirlpnormc, zplane

References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

iirlinphase

Purpose

Quasi-linear phase IIR filter from halfband filter specification

Syntax

```
hd = design(d,'iirlinphase')
hd = design(...,'filterstructure',structure)
```

Description

`hd = design(d,'iirlinphase')` designs a quasi-linear phase filter `hd` specified by the filter specification object `d`.

`hd = design(...,'filterstructure',structure)` returns a filter with the structure specified by `structure`. By default, the filter structure is `df2sos` (direct-form II with second-order sections). You can substitute one of the following strings for `structure` to specify the structure of `hd`.

Structure String	Filter Structure
`df1sos`	Direct-form I IIR filter with second-order sections
`df2sos`	Direct-form II IIR filter with second-order sections
`df1tsos`	Transposed direct-form I IIR filter with second-order sections
`df2tsos`	Transposed direct-form II IIR filter with second-order sections

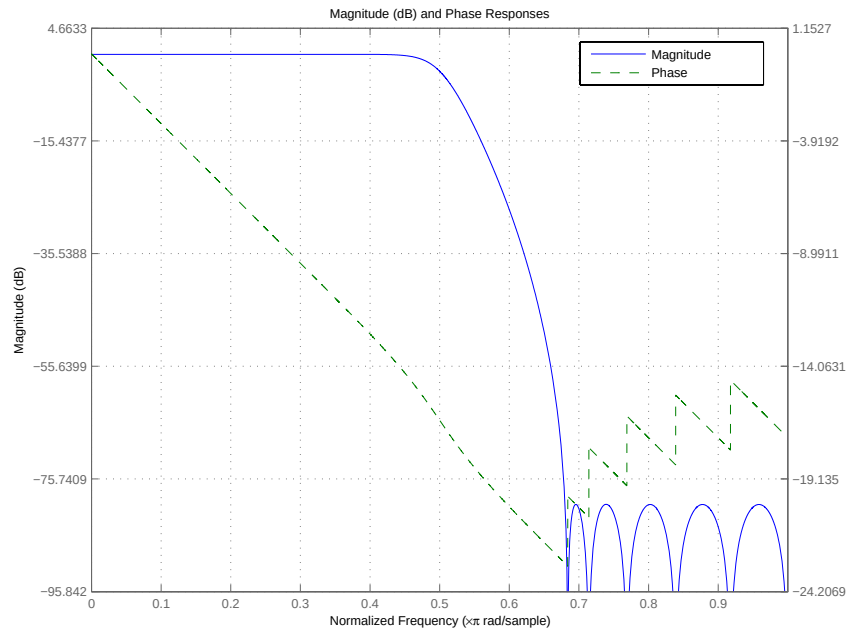
Examples

Design a quasi-linear phase, minimum-order halfband IIR filter with transition width of 0.36 and stopband attenuation of at least 80 dB.

```
tw = 0.36;
ast = 80;
d = fdesign.halfband('tw,ast',tw,ast); % Transition width,
                                       % stopband attenuation.
hd = design(d,'iirlinphase');

fvtool(hd)
```

Notice the characteristic halfband nature of the ripple in the stopband. If you measure the resulting filter, you see it meets the specifications.

```
measure(hd)

ans =

Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.32
3-dB Point         : 0.5
6-dB Point         : 0.51911
Stopband Edge      : 0.68
Passband Ripple    : 4.0866e-008 dB
Stopband Atten.    : 80.2642 dB
Transition Width   : 0.36
```



See Also

`fdesign.halfband`

Purpose

Transform IIR lowpass filter to IIR bandpass filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)
[G,AllpassNum,AllpassDen] = iirlp2bp(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "DC Mobility," meaning that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_ts.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature: the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Real lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies.

`[G,AllpassNum,AllpassDen] = iirlp2bp(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a real bandpass magnitude response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b,a] = ellip(3, 0.1, 30, 0.409);
```

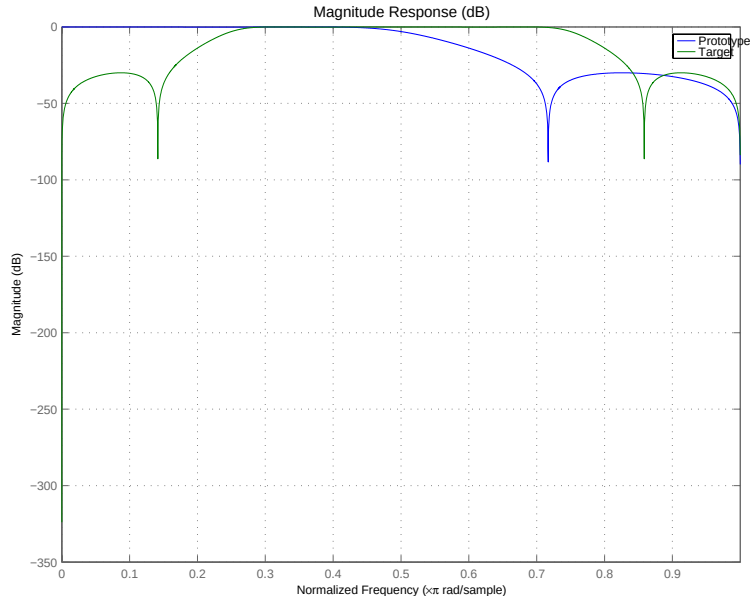
Create the real bandpass filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies W_{t1}=0.25 and W_{t2}=0.75:

```
[num,den] = iirlp2bp(b,a,0.5,[0.25,0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b,a,num,den);
```

You can compare the results in this figure to verify the transformation.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpasslp2bp`, `zpklp2bp`

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE® Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters,' *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose

IIR lowpass to complex bandpass transformation

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)
[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; for example real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

`[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a bandpass magnitude response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

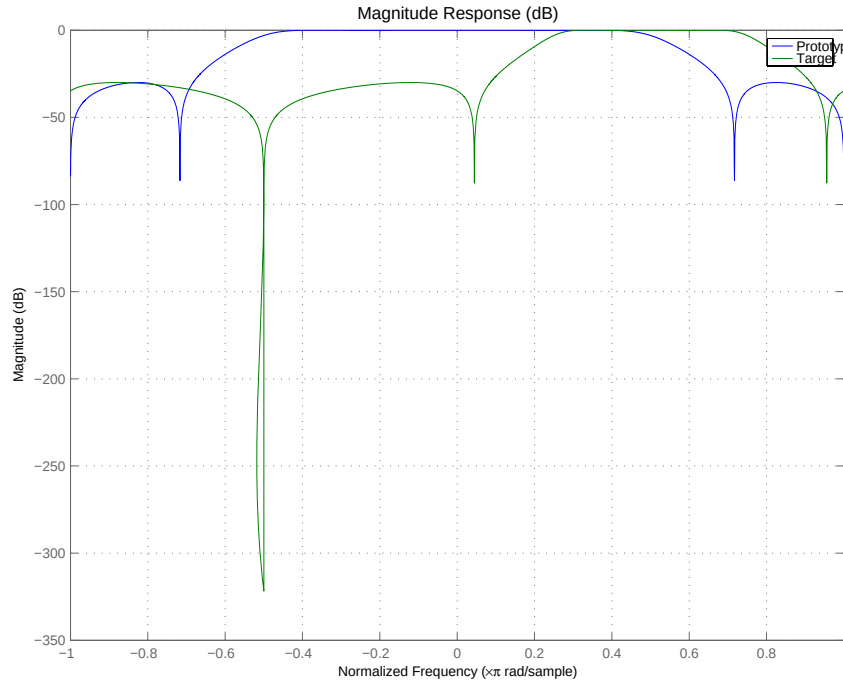
Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a complex bandpass filter:

```
[num, den] = iirlp2bpc(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Plotting the prototype and target filters together in FVTool lets you compare the filters.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter

Variable	Description
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`iirftransf`, `allpasslp2bpc`, `zpklp2bpc`

Purpose

Transform IIR lowpass filter to IIR bandstop filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bs(B,A,Wo,Wt)
[G,AllpassNum,AllpassDen] = iirlp2bs(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bs(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_ts.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

`[G,AllpassNum,AllpassDen] = iirlp2bs(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a bandstop magnitude response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping

filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

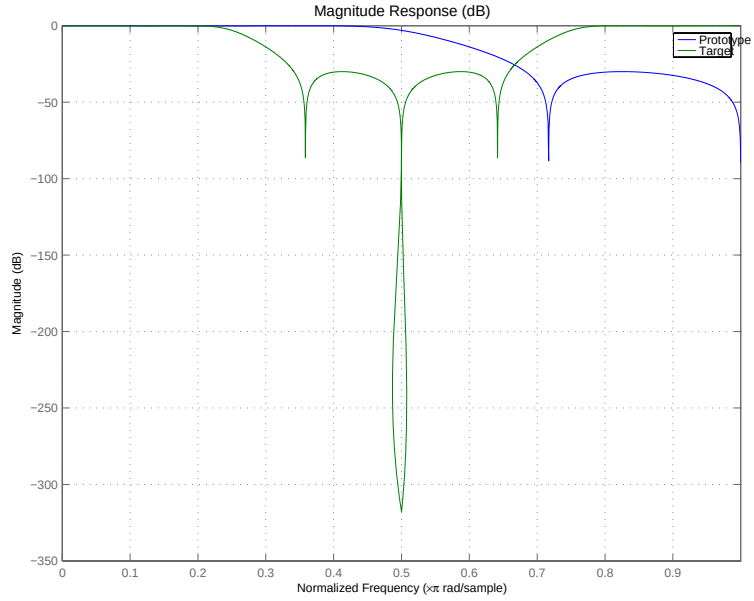
```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create the real bandstop filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies W_{t1}=0.25 and W_{t2}=0.75:

```
[num, den] = iirlp2bs(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```



With both filters plotted in the figure, you see clearly the results of the transformation.

Arguments

Variable	Description
`B`	Numerator of the prototype lowpass filter
`A`	Denominator of the prototype lowpass filter
`Wo`	Frequency value to be transformed from the prototype filter
`Wt`	Desired frequency locations in the transformed target filter
`Num`	Numerator of the target filter
`Den`	Denominator of the target filter
`AllpassNum`	Numerator of the mapping filter
`AllpassDen`	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpasslp2bs`, `zpklp2bs`

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE® Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., “Design of bandpass digital filters,” *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose

Transform IIR lowpass filter to IIR complex bandstop filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)
[G,AllpassNum,AllpassDen] = iirlp2bsc(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and the denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandstop filters for band

attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

`[G,AllpassNum,AllpassDen] = iirlp2bsc(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a bandstop magnitude response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

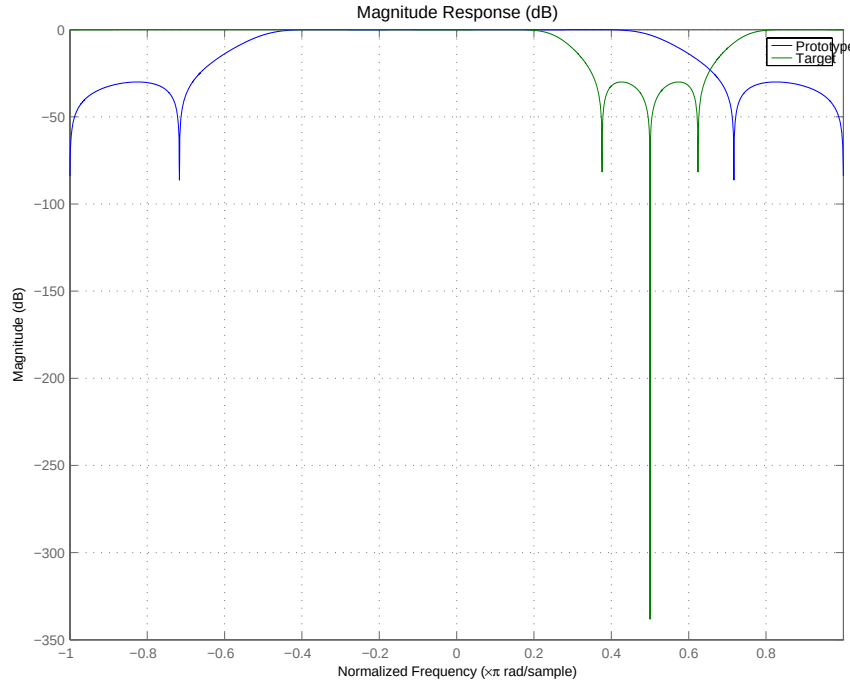
Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a complex bandstop filter:

```
[num, den] = iirlp2bsc(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

The last command in the example plots both filters in the same window so you can compare the results.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Variable	Description
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`iirftransf`, `allpasslp2bsc`, `zpklp2bsc`.

Purpose

Transform lowpass IIR filter to highpass filter

Syntax

```
[num,den] = iirlp2hp(b,a,wc,wd)
[G,AllpassNum,AllpassDen] = iirlp2hp(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

Description

`[num,den] = iirlp2hp(b,a,wc,wd)` with input arguments `b` and `a`, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, `iirlp2bp` transforms the magnitude response from lowpass to highpass. `num` and `den` return the coefficients for the transformed highpass filter. For `wc`, enter a selected frequency from your lowpass filter. You use the chosen frequency to define the magnitude response value you want in the highpass filter. Enter one frequency for the highpass filter — the value that defines the location of the transformed point — in `wd`. Note that all frequencies are normalized between zero and one. Notice also that the filter order does not change when you transform to a highpass filter.

When you select `wc` and designate `wd`, the transformation algorithm sets the magnitude response at the `wd` values of your bandstop filter to be the same as the magnitude response of your lowpass filter at `wc`. Filter performance between the values in `wd` is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as `wc`. Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version. Without designing the highpass filter from the beginning.

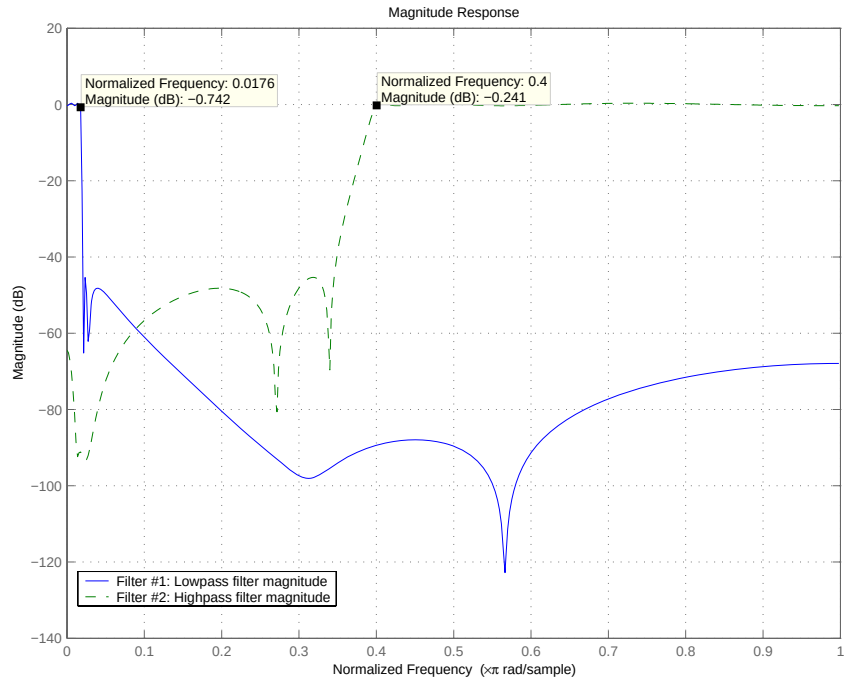
In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

`[G,AllpassNum,AllpassDen] = iirlp2hp(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a highpass magnitude response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

This example transforms an IIR filter from lowpass to high pass by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter. To generate a highpass filter whose passband flattens out at 0.4, select the frequency in the lowpass filter where the passband starts to rolloff (`wc = 0.0175`) and move it to the new location at `wd = 0.4`.

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1],...
[0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0],...
[1 1 1 1 10 10]);
wc = 0.0175;
wd = 0.4;
[num,den] = iirlp2hp(b,a,wc,wd);
fvtool(b,a,num,den);
```



In the figure showing the magnitude responses for the two filters, the transition band for the highpass filter is essentially the mirror image of the transition for the lowpass filter from 0.0175 to 0.025, stretched out over a wider frequency range. In the passbands, the filter share common ripple characteristics and magnitude.

See Also

`iirlp2bp`, `iirlp2bs`, `iirlp2lp`, `firlp2lp`, `firlp2hp`

References

Mitra, Sanjit K., *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

iirlp2lp

Purpose

Transform lowpass IIR filter to different lowpass filter

Syntax

```
[num,den] = iirlp2hp(b,a,wc,wd)
[G,AllpassNum,AllpassDen] = iirlp2lp(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

Description

`[num,den] = iirlp2hp(b,a,wc,wd)` with input arguments `b` and `a`, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, `iirlp2bp` transforms the magnitude response from lowpass to highpass. `num` and `den` return the coefficients for the transformed highpass filter. For `wc`, enter a selected frequency from your lowpass filter. You use the chosen frequency to define the magnitude response value you want in the highpass filter. Enter one frequency for the highpass filter — the value that defines the location of the transformed point — in `wd`. Note that all frequencies are normalized between zero and one. Notice also that the filter order does not change when you transform to a highpass filter.

When you select `wc` and designate `wd`, the transformation algorithm sets the magnitude response at the `wd` values of your bandstop filter to be the same as the magnitude response of your lowpass filter at `wc`. Filter performance between the values in `wd` is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as `wc`. Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version. Without designing the highpass filter from the beginning.

In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

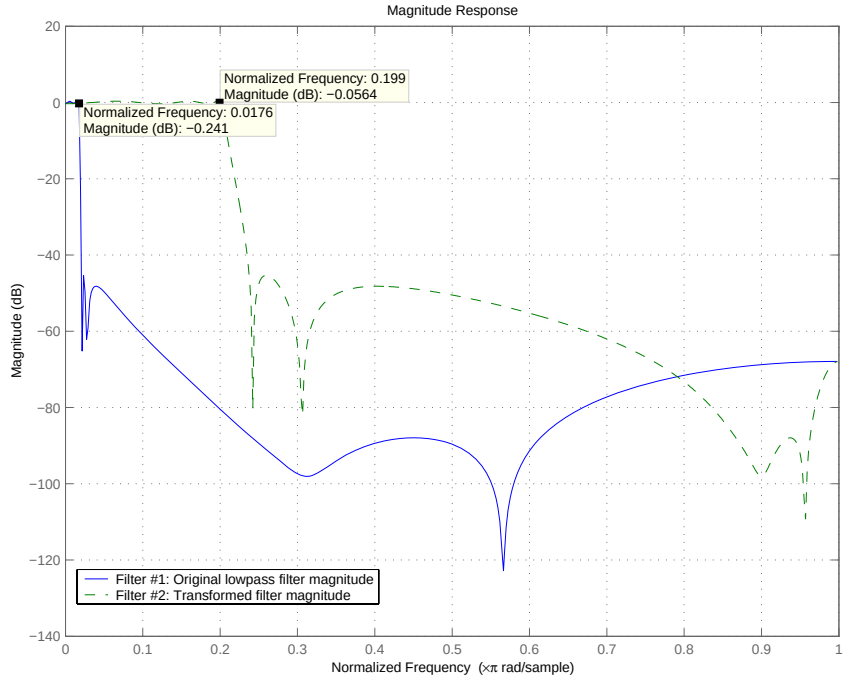
`[G,AllpassNum,AllpassDen] = iirlp2lp(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a lowpass magnitude response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

This example transforms an IIR filter from lowpass to high pass by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter. To generate a lowpass filter whose passband extends out to 0.2, select the frequency in the lowpass filter where the passband starts to rolloff (`wc` = 0.0175) and move it to the new location at `wd` = 0.2.

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1],...
[0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0],...
[1 1 1 1 10 10]);
wc = 0.0175;
wd = 0.2;
[num,den] = iirlp2lp(b,a,wc,wd);
fvtool(b,a,num,den);
```

Moving the edge of the passband from 0.0175 to 0.2 results in a new lowpass filter whose peak response in-band is the same as the original filter: same ripple, same absolute magnitude.



The rolloff is slightly less steep and the stopband profiles are the same for both filters; the new filter stopband is a “stretched” version of the original, as is the passband of the new filter.

See Also

`iirlp2bp`, `iirlp2bs`, `iirlp2hp`, `firlp2lp`, `firlp2hp`

References

Mitra, Sanjit K, *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

Purpose

Transform IIR lowpass filter to IIR M-band filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen]=iirlp2mb(B,A,Wo,Wt,Pass)
[G,AllpassNum,AllpassDen] = iirlp2mb(Hd,Wo,Wt)
[G,AllpassNum,AllpassDen] = iirlp2mb(...,Pass)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `M`th-order real lowpass to real multiple bandpass frequency mapping. By default the DC feature is kept at its original location.

`[Num,Den,AllpassNum,AllpassDen]=iirlp2mb(B,A,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter do not change in the target filter. It is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

`[G,AllpassNum,AllpassDen] = iirlp2mb(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with an IIR real M-band filter frequency response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

`[G,AllpassNum,AllpassDen] = iirlp2mb(...,Pass)` returns transformed `dfilt` object `G` with an IIR real M-band filter frequency response. This syntax allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Example 1

Create the real multiband filter with two passbands:

```
[num1, den1] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10);
[num2, den2] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10, 'pass');
```

The second code snippet uses the `pass` option to select the Nyquist mobility option. In this case the resulting filter is the same.

Example 2

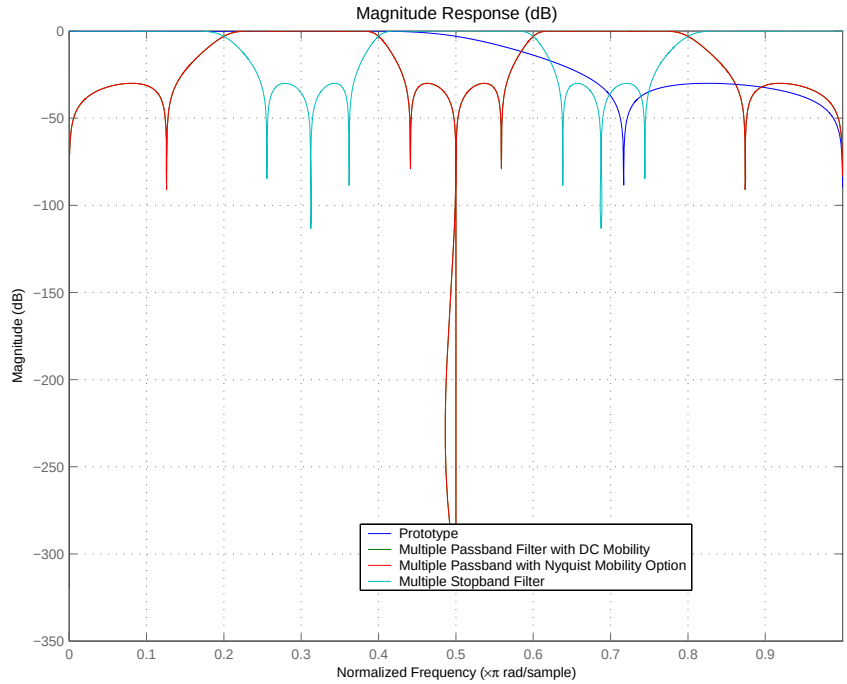
Create the real multiband filter with two stopbands:

```
[num3, den3] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with target filters:

```
fvtool(b, a, num1, den1, num2, den2, num3, den3);
```

Combining all of the filters, prototypes and targets, on one figure makes comparing them straightforward. Passbands for the filters in example 1 appear separately in the figure, although they overlap to a degree that makes them hard to identify — they have identical coefficients.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
Num	Numerator of the target filter

Variable	Description
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpasslp2mb`, `zpklp2mb`

References

Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R. A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Mass., Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, "An extension of the Schur Algorithm for frequency transformations," *Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

iirlp2mbc

Purpose

Transform IIR lowpass filter to IIR complex M-band filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2mbc(B,A,Wo,Wc)
[G,AllpassNum,AllpassDen] = iirlp2mbc(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2mbc(B,A,Wo,Wc)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `M`th-order real lowpass to complex multibandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

`[G,AllpassNum,AllpassDen] = iirlp2mbc(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with an IIR complex M-band filter frequency response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

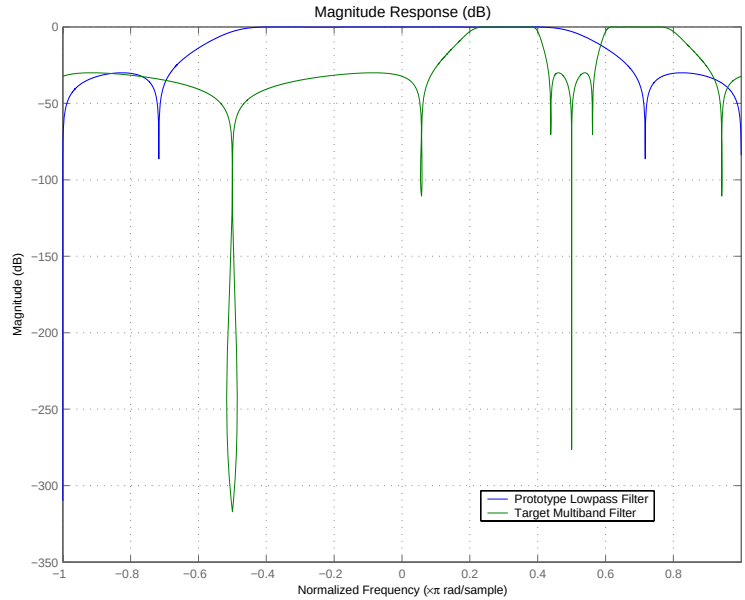
```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Now create a complex multiband filter with two passbands:

```
[num1, den1] = iirlp2mbc(b, a, 0.5, [2 4 6 8]/10);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num1, den1);
```



You see in the figure that `iirlp2mbc` replicates the desired feature at 0.5 in the lowpass filter at four locations in the multiband filter.

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter.
A	Denominator of the prototype lowpass filter.
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wc	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter.

Variable	Description
Den	Denominator of the target filter.
AllpassNum	Numerator of the mapping filter.
AllpassDen	Denominator of the mapping filter.

See Also

`iirftransf`, `allpasslp2mbc`, `zpklp2mbc`

Purpose

Transform IIR lowpass filter to IIR complex N-point filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt)
[G,AllpassNum,AllpassDen] = iirlp2xc(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to complex multipoint frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., a stopband edge, DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

`[G,AllpassNum,AllpassDen] = iirlp2xc(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with an IIR complex N-point filter frequency response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

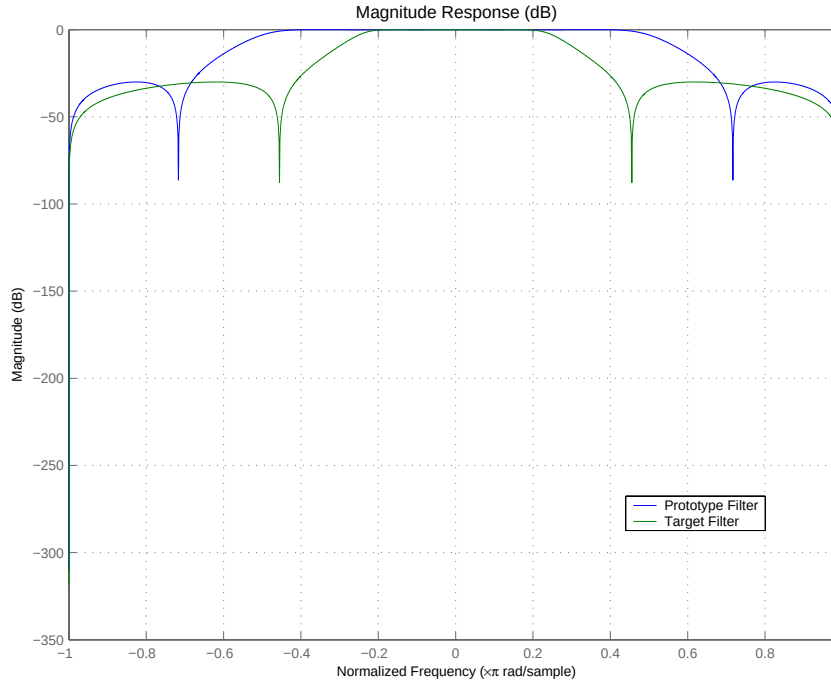
Create the complex bandpass filter from the real lowpass filter:

```
[num, den] = iirlp2xc(b, a, [-0.5 0.5], [-0.25 0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Reviewing the coefficients and the figure produced by the example shows that the target filter has complex coefficients and is indeed a bandpass filter as expected.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter.
A	Denominator of the prototype lowpass filter.
Wo	Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter.

Variable	Description
Den	Denominator of the target filter.
AllpassNum	Numerator of the mapping filter.
AllpassDen	Denominator of the mapping filter.

See Also

`iirftransf`, `allpasslp2xc`, `zpklp2xc`

Purpose

Transform IIR lowpass filter to IIR real N-point filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt,Pass)
[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt),
where Hd is a dfilt object
[G,AllpassNum,AllpassDen] = iirlp2bpc(...,Pass)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to real multipoint frequency transformation, where `N` is the number of features being mapped. By default the DC feature is kept at its original location.

`[Num,Den,AllpassNum,AllpassDen]= iirlp2xn(B,A,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after

the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

`[G,AllpassNum,AllpassDen] = iirlp2xn(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with an IIR real N-point filter frequency response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

`[G,AllpassNum,AllpassDen] = iirlp2xn(...,Pass)` returns transformed `dfilt` object `G` with an IIR real N-point filter frequency response. This syntax allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

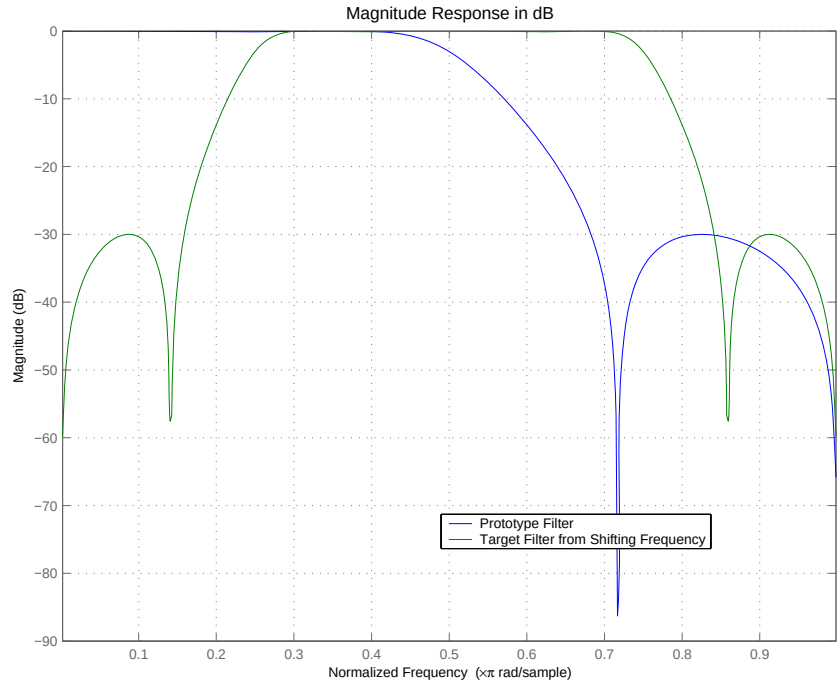
Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a real bandpass filter:

```
[num, den] = iirlp2xn(b, a, [-0.5 0.5], [0.25 0.75], ...
pass');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

`iirlp2xn` has created the desired bandpass filter with the cutoff locations specified in the command.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
Num	Numerator of the target filter

Variable	Description
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpasslp2xn`, `zpklp2xn`

References

Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

Purpose

Least P-norm optimal IIR filter

Syntax

```
[num,den] = iirlpnorm(n,d,f,edges,a)
[num,den] = iirlpnorm(n,d,f,edges,a,w)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)
```

Description

`[num,den] = iirlpnorm(n,d,f,edges,a)` returns a filter having a numerator order `n` and denominator order `d` which is the best approximation to the desired frequency response described by `f` and `a` in the least-pth sense. The vector `edges` specifies the band-edge frequencies for multi-band designs. An unconstrained quasi-Newton algorithm is employed and any poles or zeros that lie outside of the unit circle are reflected back inside. `n` and `d` should be chosen so that the zeros and poles are used effectively. See the "Hints" on page 2-924 section. Always use `freqz` to check the resulting filter.

`[num,den] = iirlpnorm(n,d,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `iirlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. `f` and `a` must have the same number of elements, which may exceed the number of elements in `edges`. This allows for the specification of filters having any gain contour within each band. The frequencies specified in `edges` must also appear in the vector `f`. For example,

```
[num,den] = iirlpnorm(5,12,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

is a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p)` where `p` is a two-element vector `[pmin pmax]` allows for the specification of the minimum and maximum values of `p` used in the least-pth algorithm. Default is [2 128] which essentially yields the L-infinity, or Chebyshev, norm. `Pmin` and `pmax` should be even. If `p` is the string `'inspect'`, no

optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `(dens*(n+d+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. This may be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox™ software can be used for generating `initnum` and `initden`.

Hints

- This is a weighted least-pth optimization.
- Check the radii and locations of the poles and zeros for your filter. If the zeros are on the unit circle and the poles are well inside the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.
- Similarly, if several poles have a large radii and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weighting in the passband.

See Also

`iirlpnormc`, `filter`, `freqz`, `iirgrpdelay`, `zplane`

References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

Purpose

Constrained least Pth-norm optimal IIR filter

Syntax

```
[num,den] = iirlpnormc(n,d,f,edges,a)
[num,den] = iirlpnormc(n,d,f,edges,a,w)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens,...
initnum,initden)
[num,den,err] = iirlpnormc(...)
[num,den,err,sos,g] = iirlpnormc(...)
```

Description

`[num,den] = iirlpnormc(n,d,f,edges,a)` returns a filter having numerator order `n` and denominator order `d` which is the best approximation to the desired frequency response described by `f` and `a` in the least-pth sense. The vector `edges` specifies the band-edge frequencies for multi-band designs. A constrained Newton-type algorithm is employed. `n` and `d` should be chosen so that the zeros and poles are used effectively. See the Hints section. Always check the resulting filter using `fvtool`.

`[num,den] = iirlpnormc(n,d,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `iirlpnormc` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. `f` and `a` must have the same number of elements, which can exceed the number of elements in `edges`. This allows for the specification of filters having any gain contour within each band. The frequencies specified in `edges` must also appear in the vector `f`. For example,

```
[num,den] = iirlpnormc(5,5,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)` returns a filter having a maximum pole radius of `radius` where 0<`radius`<1. `radius`

defaults to 0.999999. Filters that have a reduced pole radius may retain better transfer function accuracy after you quantize them.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)` where `p` is a two-element vector `[pmin pmax]` allows for the specification of the minimum and maximum values of `p` used in the least-pth algorithm. Default is [2 128] which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even. If `p` is the string `'inspect'`, no optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `(dens*(n+d+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens,...initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. This may be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox™ software can be used for generating `initnum` and `initden`.

`[num,den,err] = iirlpnormc(...)` returns the least-Pth approximation error `err`.

`[num,den,err,sos,g] = iirlpnormc(...)` returns the second-order section representation in the matrix SOS and gain G. For numerical reasons you may find SOS and G beneficial in some cases.

Hints

- This is a weighted least-pth optimization.
- Check the radii and location of the resulting poles and zeros.
- If the zeros are all on the unit circle and the poles are well inside of the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.

- Similarly, if several poles have a large radius and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weight in the passband.
- If you reduce the pole radius, you might need to increase the order of the denominator.

The message

```
Poorly conditioned matrix. See the "help" file.
```

indicates that `iirlpnormc` cannot accurately compute the optimization because either:

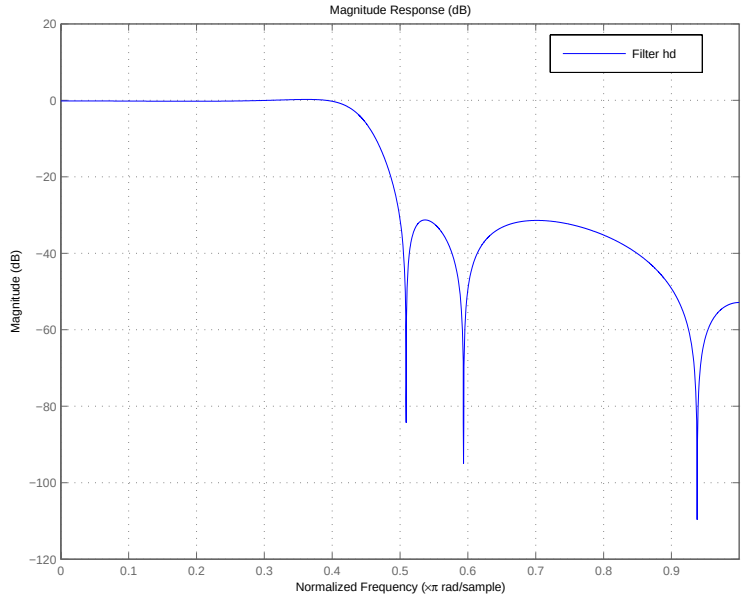
1. The approximation error is extremely small (try reducing the number of poles or zeros — refer to the hints above).
2. The filter specifications have huge variation, such as `a=[1 1e9 0 0]`.

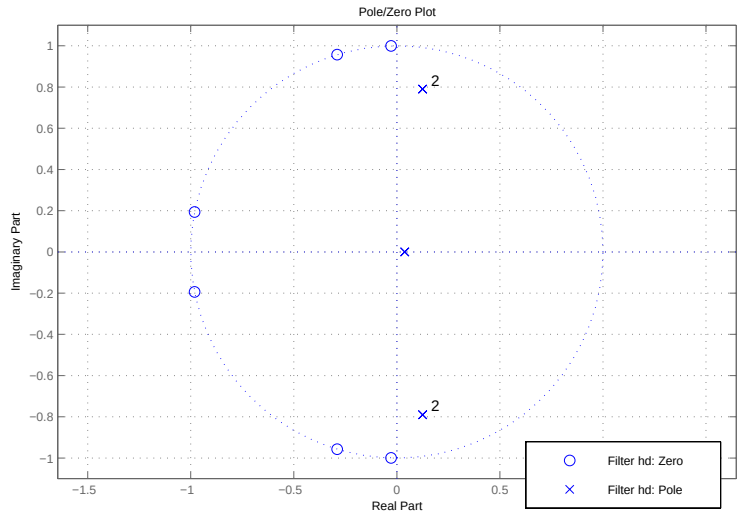
Examples

This example returns a lowpass filter whose pole radius is constrained to 0.8

```
[b,a,err,s,g] = iirlpnormc(6,6,[0 .4 .5 1],[0 .4 .5 1],...
[1 1 0 0],[1 1 1 1],.8);
hd = dfilt.df1sos(s,g); % Construct second-order sections filter.
fvtool(hd); % View filter's magnitude response
```

From the magnitude response shown here you see the lowpass nature of the filter. The pole/zero plot following shows that the poles are constrained to 0.8 as specified in the command.


Magnitude Response (dB)
Filter hd
Magnitude (dB)
Normalized Frequency (×π rad/sample)


Pole/Zero Plot
Imaginary Part
Real Part
2
2
Filter hd: Zero
Filter hd: Pole

See Also

`freqz`, `filter`, `iirgrpdelay`, `iirlpnorm`, `zplane`

References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

Purpose

RLS IIR filter from specification object

Syntax

```
hd = design(d,'iirls')
hd = design(d,'iirls',designoption,value,designoption,value,
   ...)
```

Description

`hd = design(d,'iirls')` designs a least-squares filter specified by the filter specification object `d`.

Note The `iirls` algorithm might not be well behaved in all cases. Experience is your best guide to determining if the resulting filter meets your needs. When you use `iirls` to design a filter, review the filter carefully to ensure that it is appropriate for your use.

`hd = design(d,'iirls',designoption,value,designoption,value,...)` returns a least-squares IIR 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 `iirls`, refer to the command line help system. For example, to get specific information about using `iirls` with `d`, the specification object, enter the following at the MATLAB prompt.

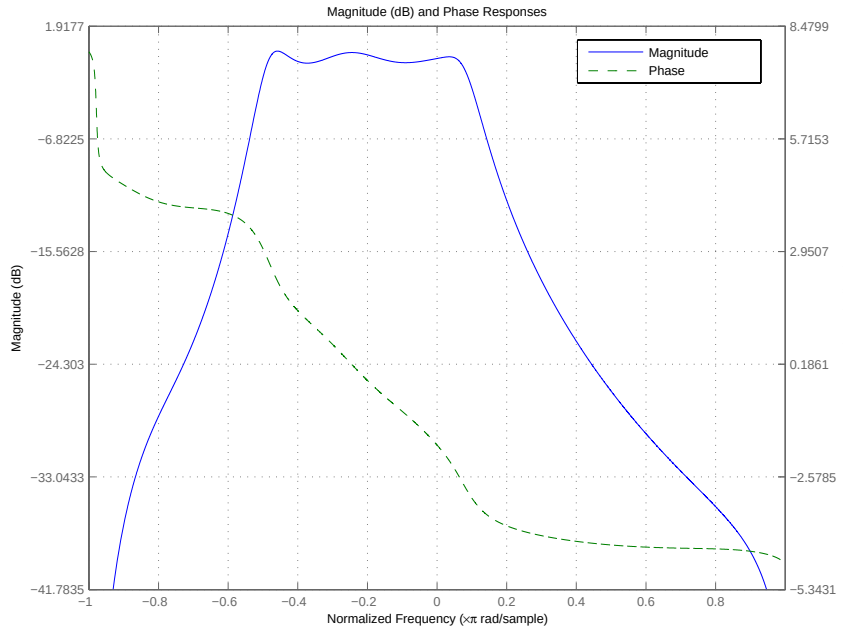
```
help(d,'iirls')
```

Examples

Starting from an arbitrary magnitude and phase design object `d`, generate a complex bandpass filter of order = 5. To make the example a little easier to do, use the default values for F, and H, the frequency vector and the complex desired frequency response.

```
d = fdesign.arbmagnphase('N,F,H',5);
d =

           Response: 'Arbitrary Magnitude and Phase'
      Specification: 'N,F,H'
        Description: {'Filter Order';'Frequency Vector';'
                       Complex Desired Frequency Response'
 NormalizedFrequency: true
         FilterOrder: 5
         Frequencies: [1x655 double]
        FreqResponse: [1x655 double]

design(d,'iirls'); % Opens FVTool to show the filter.
```

Displaying both the phase and magnitude response in FVTool shows you the filter.



See Also

`fdesign.arbmag`, `fdesign.arbmagnphase`, `firls`

Purpose

Second-order IIR notch filter

Syntax

```
[num,den] = iirnotch(w0,bw)
[num,den] = iirnotch(w0,bw,ab)
```

Description

`[num,den] = iirnotch(w0,bw)` turns a digital notching filter with the notch located at `w0`, and with the bandwidth at the -3 dB point set to `bw`. To design the filter, `w0` must meet the condition 0.0 `w0` 1.0, where 1.0 corresponds to π adians per sample in the frequency range.

The quality factor (Q factor) `q` for the filter is related to the filter bandwidth by `q w0/bw` where ω_0 is `w0`, the frequency to remove from the signal.

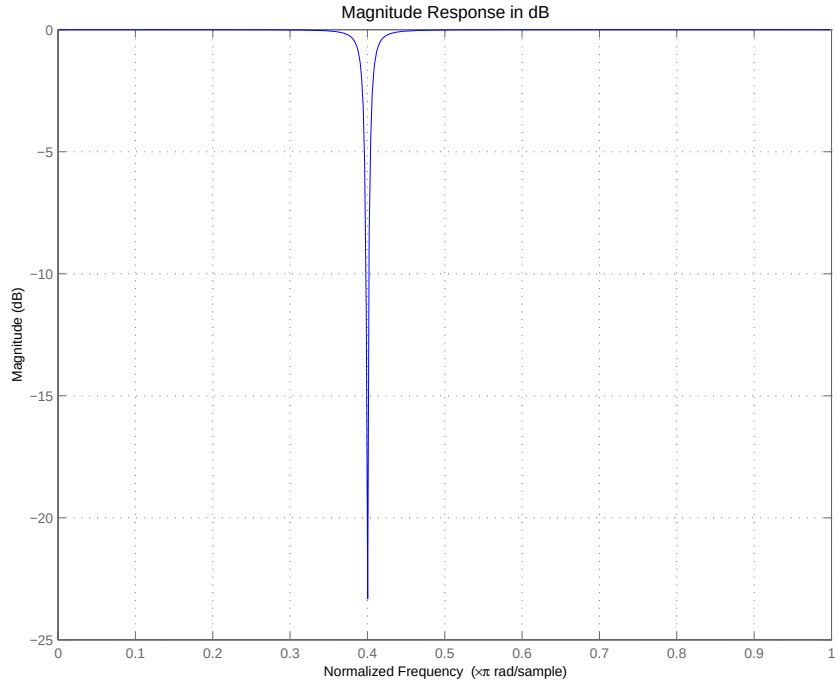
`[num,den] = iirnotch(w0,bw,ab)` returns a digital notching filter whose bandwidth, `bw`, is specified at a level of `-ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default -3 dB point, such as -6 dB or 0 dB.

Examples

Design and plot an IIR notch filter that removes a 60 Hz tone (`f0`) from a signal at 300 Hz (`fs`). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth:

```
wo = 60/(300/2);  bw = wo/35;
[b,a] = iirnotch(wo,bw);
fvtool(b,a);
```

Shown in the next plot, the notch filter has the desired bandwidth with the notch located at 60 Hz, or 0.4π radians per sample. Compare this plot to the comb filter plot shown on the reference page for `iircomb`.



See Also

`firgr`, `iircomb`, `iirpeak`

Purpose

Second-order IIR peak or resonator filter

Syntax

```
[num,den] = iirpeak(w0,bw)
[num,den] = iirpeak(w0,bw,ab)
```

Description

`[num,den] = iirpeak(w0,bw)` turns a second-order digital peaking filter with the peak located at `w0`, and with the bandwidth at the +3 dB point set to `bw`. To design the filter, `w0` must meet the condition 0.0 < `w0` < 1.0, where 1.0 corresponds to π radians per sample in the frequency range.

The quality factor (Q factor) `q` for the filter is related to the filter bandwidth by `q = w0/bw` where ω_0 is `w0` the signal frequency to boost.

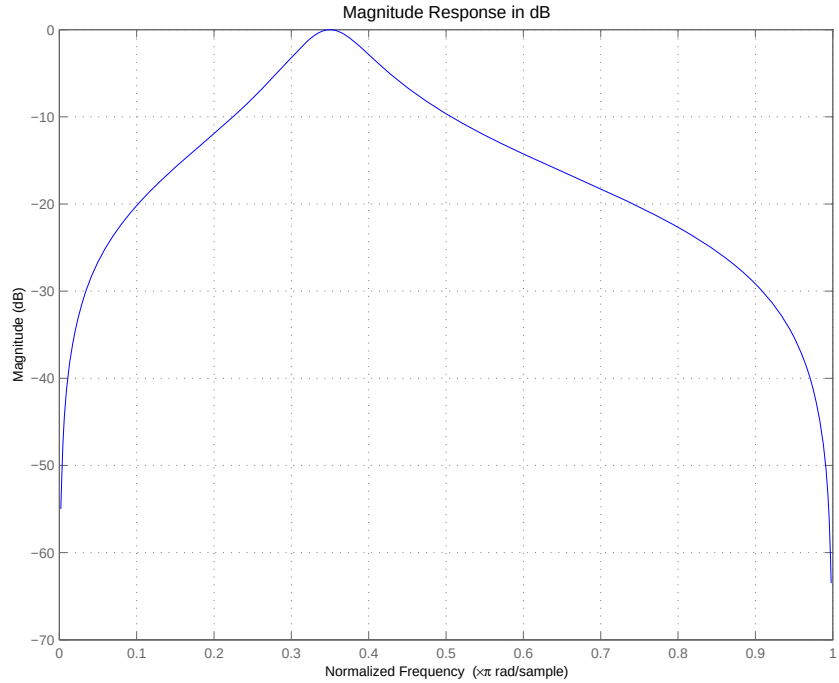
`[num,den] = iirpeak(w0,bw,ab)` returns a digital peaking filter whose bandwidth, `bw`, is specified at a level of +`ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default +3 dB point, such as +6 dB or 0 dB.

Examples

Design and plot an IIR peaking filter that boosts the frequency at 1.75 Khz in a signal and has bandwidth of 500 Hz at the -3 dB point:

```
fs = 10000; wo = 1750/(fs/2);  bw = 500/(fs/2);
[b,a] = iirpeak(wo,bw);
fvtool(b,a);
```

Shown in the next plot, the peak filter has the desired gain and bandwidth at 1.75 KHz.



See Also

`firgr`, `iircomb`, `iirnotch`

Purpose

Power complementary IIR filter

Syntax

```
[bp,ap] = iirpowcomp(b,a)
[bp,ap,c] = iirpowcomp(b,a)
```

Description

`[bp,ap] = iirpowcomp(b,a)` returns the coefficients of the power complementary IIR filter g(z) = bp(z)/ap(z) in vectors `bp` and `ap`, given the coefficients of the IIR filter h(z) = b(z)/a(z) in vectors `b` and `a`. `b` must be symmetric (Hermitian) or antisymmetric (antihermitian) and of the same length as `a`. The two power complementary filters satisfy the relation

$|H(w)|^2 + |G(w)|^2 = 1$.

`[bp,ap,c] = iirpowcomp(b,a)` where `c` is a complex scalar of magnitude = 1, forces `bp` to satisfy the generalized hermitian property

`conj(bp(end:-1:1)) = c*bp`.

When `c` is omitted, it is chosen as follows:

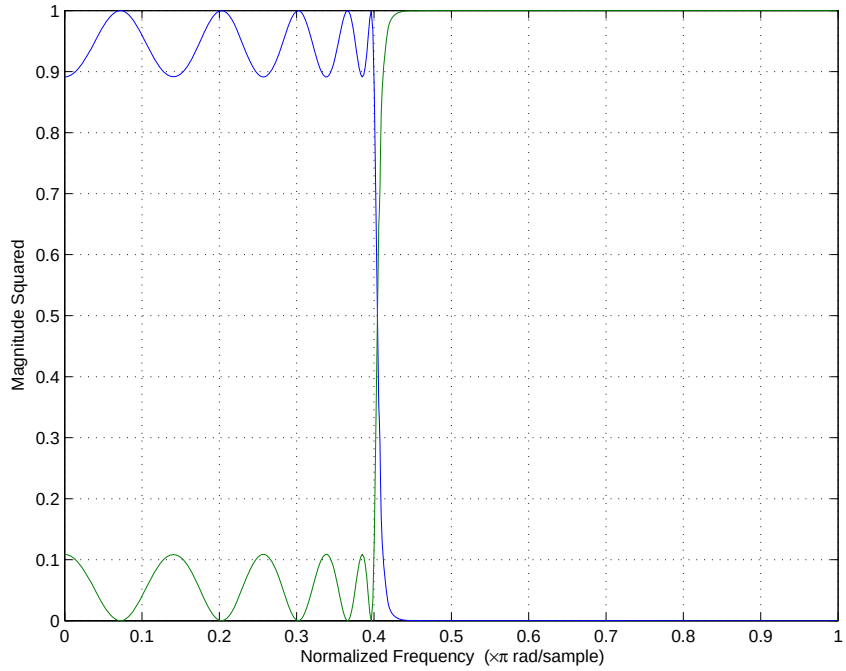
- When `b` is real, chooses C as 1 or -1, whichever yields `bp` real
- When `b` is complex, C defaults to 1

`ap` is always equal to `a`.

Examples

```
[b,a]=cheby1(10,.5,.4);
[bp,ap]=iirpowcomp(b,a);
[h,w,s]=freqz(b,a); [h1,w,s]=freqz(bp,ap);
s.plot='mag'; s.yunits='sq';freqzplot([h h1],w,s)
```

The next figure presents the results of applying `iirpowcomp` to the Chebyshev filter — the power complementary version of the original filter.



See Also

`tf2ca`, `tf2cl`, `ca2tf`, `cl2tf`

Purpose

Upsample IIR filter by integer factor

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter being transformed from any prototype by applying an `N`th-order rateup frequency transformation, where `N` is the upsample ratio. Transformation creates `N` equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

The relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

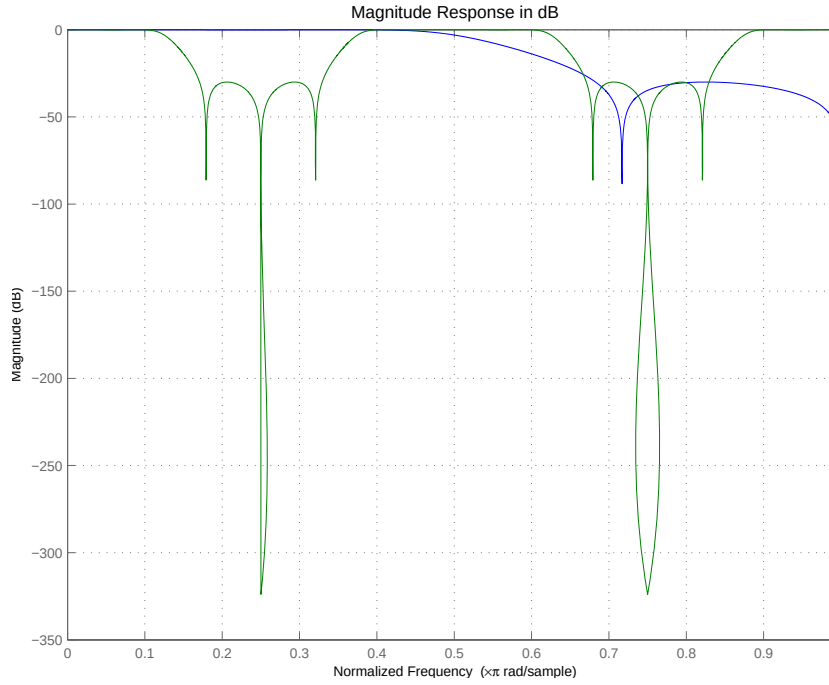
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
[num, den] = iirrateup(b, a, 4);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

As shown in the figure produced by FVTool, the transformed filter appears as expected.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
N	Frequency multiplication ratio
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`iirftransf`, `allpassrateup`, `zpkrateup`

iirshift

Purpose

Shift frequency response of IIR filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real shift frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator of the allpass mapping filter, `AllpassDen`. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

This transformation places one selected feature of an original filter located at frequency W_o to the required target frequency location, W_t. This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them from the beginning.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

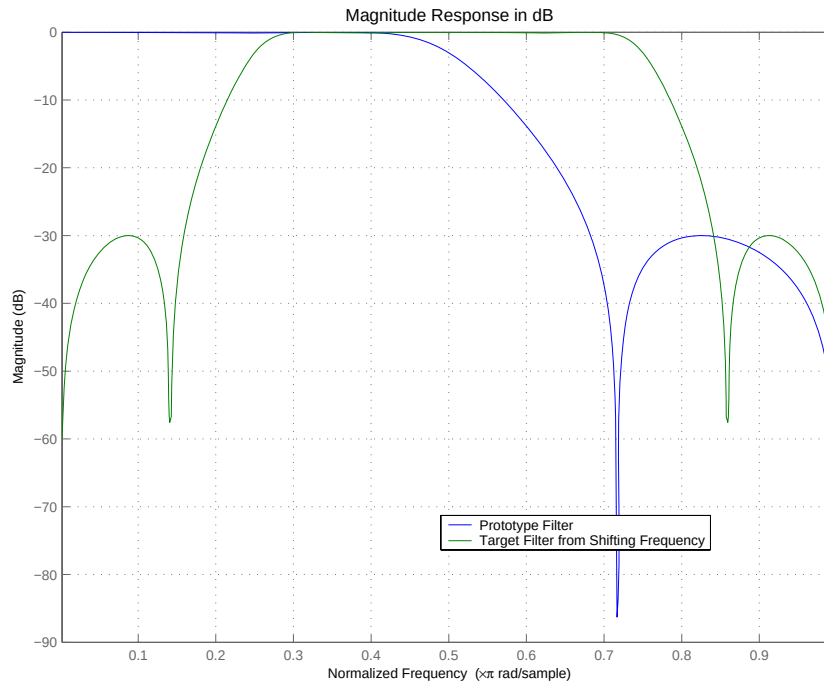
Perform the real frequency shift by defining where the selected feature of the prototype filter, originally at W_o=0.5, should be placed in the target filter, W_t=0.75:

```
Wo = 0.5; Wt = 0.75;
[num, den] = iirshift(b, a, Wo, Wt);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Shifting the specified feature from the prototype to the target generates the response shown in the figure.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpassshift`, `zpkshift`.

Purpose

Shift frequency response of IIR complex filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at W_o, placed at W_t in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)` calculates the allpass filter for doing the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)` calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

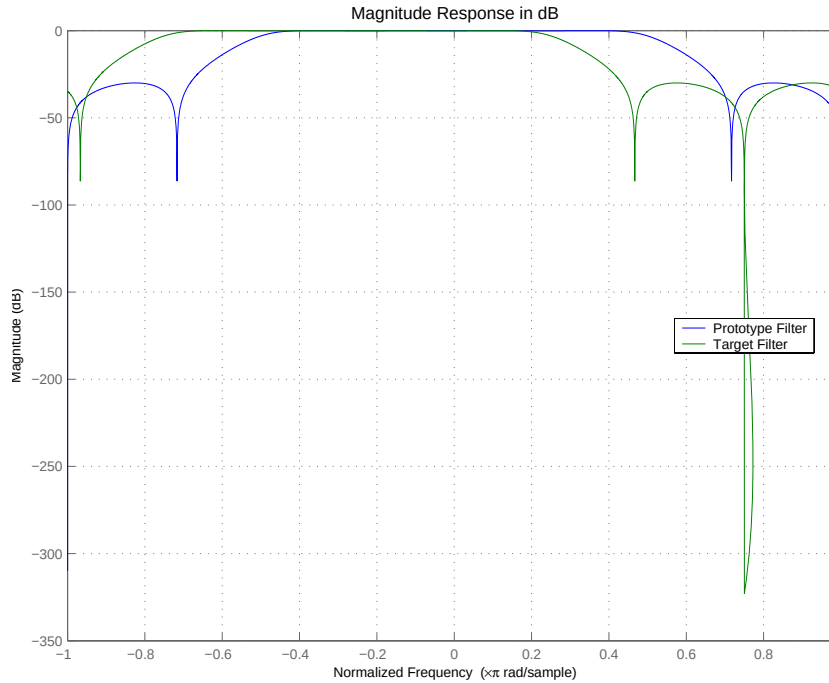
Rotate all features of the prototype filter in the frequency domain by the same amount by specifying where the selected feature of an original filter, W_o= `0.5`, should appear in the target filter, W_t= `0.25`:

```
[num, den] = iirshiftc(b, a, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

After applying the shift, the selected feature from the original filter is just where it should be, at W_t = 0.25.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpassshiftc`, `zpkshiftc`

References

Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE® Transactions on Education*, vol. 32, pp. 314-318, August 1989.

impz

Purpose

Filter impulse response

Syntax

```
[h,t] = impz(ha)
[h,t] = impz(...,fs)
impz(ha,...)
[h,t] = impz(hd)
impz(hd)
[h,t] = impz(hm)
impz(hm)
```

Description

The next sections describe common `impz` operation with adaptive, discrete-time, and multirate filters. For more input options, refer to `impz` in Signal Processing Toolbox™ documentation.

- "Discrete-Time Filters" on page 2-949
- "Multirate Filters" on page 2-949

Adaptive Filters

For adaptive filters, `impz` returns the instantaneous impulse response based on the current filter coefficients.

`[h,t] = impz(ha)` computes the instantaneous impulse response of the adaptive filter `ha` choosing the number of samples for you, and returns the response in column vector `h` and a vector of times or sample intervals in `t` where (`t = [0 1 2...]'`).

`[h,t] = impz(...,fs)` returns a matrix `h` if `ha` is a vector. Each column of the matrix corresponds to one filter in the vector. When `ha` is a vector of adaptive filters, `impz` returns the matrix `h`. Each column of `h` corresponds to one filter in the vector `ha`. If you provide a sampling frequency `fs` as an input argument, `impz` uses `fs` in when determining the impulse response.

`impz(ha,...)` uses FVTool to plot the impulse response of the adaptive filter `ha`. If `ha` is a vector of filters, `impz` plots the response and for each filter in the vector.

Discrete-Time Filters

`[h,t] = impz(hd)` computes the instantaneous impulse response of the discrete-time filter `hd` choosing the number of samples for you, and returns the response in column vector `h` and a vector of times or sample intervals in `t` where (`t = [0 1 2...]'`). `impz` returns a matrix `h` if `hd` is a vector. Each column of the matrix corresponds to one filter in the vector. When `hd` is a vector of discrete-time filters, `impz` returns the matrix `h`. Each column of `h` corresponds to one filter in the vector `hd`.

`impz(hd)` uses FVTool to plot the impulse response of the discrete-time filter `hd`. If `hd` is a vector of filters, `impz` plots the response and for each filter in the vector.

Multirate Filters

`[h,t] = impz(hm)` computes the instantaneous impulse response of the multirate filter `hm` choosing the number of samples for you, and returns the response in column vector `h` and a vector of times or sample intervals in `t` where (`t = [0 1 2...]'`). `[h,t] = impz(hm)` returns a matrix `h` if `hm` is a vector. Each column of the matrix corresponds to one filter in the vector. When `hm` is a vector of multirate filters, `impz` returns the matrix `h`. Each column of `h` corresponds to one filter in the vector `ha`.

`impz(hm)` uses FVTool to plot the impulse response of the multirate filter `hm`. If `ha` is a vector of filters, `impz` plots the response and for each filter in the vector.

Note that the multirate filter impulse response is computed relative to the rate at which the filter is running. When you specify `fs` (the sampling rate) as an input argument, `impz` assumes the filter is running at that rate.

For multistage cascades, `impz` forms a single-stage multirate filter that is equivalent to the cascade and computes the response relative to the rate at which the equivalent filter is running. `impz` does not support all multistage cascades. Only cascades for which it is possible to derive an equivalent single-stage filter are allowed for analysis.

As an example, consider a 2-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an

interpolation factor of 4. An equivalent single-stage filter with an overall interpolation factor of 8 can be found. `impz` uses the equivalent filter for the analysis. If a sampling frequency `fs` is specified as an input argument to `impz`, the function interprets `fs` as the rate at which the equivalent filter is running.

Note `impz` works for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

Examples

Create a discrete-time filter for a fourth-order, low-pass elliptic filter with a cutoff frequency of 0.4 times the Nyquist frequency. Use a second-order sections structure to resist quantization errors. Plot the first 50 samples of the impulse response, along with the reference impulse response.

```
% Create a design object for the prototype filter.

d = fdesign.lowpass(.4,.5,1,80)

d =

               Response: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
                  Fpass: 0.4000
                  Fstop: 0.5000
                  Apass: 1
                  Astop: 80
```

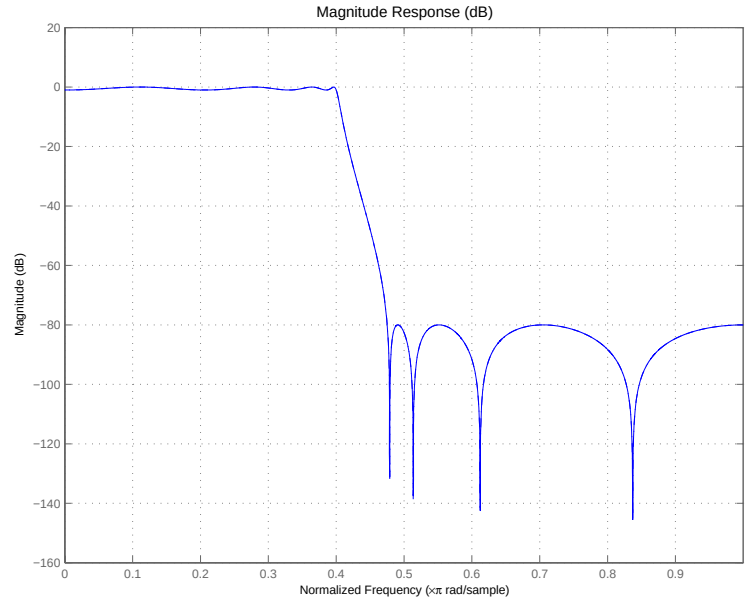
Use `ellip` to design the discrete-time filter in second-order section form, with minimum-order.

```
hd=design(d,'ellip')
```

```
hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
       sosMatrix: [4x6 double]
     ScaleValues: [5x1 double]
ResetBeforeFiltering: 'on'
          States: [2x4 double]
```

Convert hd to fixed-point and check the impulse response
`hd.arithmetic = 'fixed';`

```
impz(hd)
```



See Also

`filter`

Purpose

Information about filter

Syntax

```
s = info(h)
info(h)
info(h, 'short')
s = info(h, 'long')
info(h, 'long')
```

Description

`s = info(h)` or `info(h)` or `info(h, 'short')` returns very basic information about the filter. The particulars depend on the filter type and structure.

`s = info(h, 'long')` or `info(h, 'long')` returns the following information about the filter:

- Specifications such as the filter structure and filter order
- Information about the design method and options
- Performance measurements for the filter response, such as the passband cutoff or stopband attenuation, included in the`measure` method.
- Cost of implementing the filter in terms of operations required to apply the filter to data, included in the `cost` method.

When the filter uses fixed-point arithmetic, the `info` returns additional information about the filter, including the arithmetic setting and details about the filter internals.

Examples

In the following example shows how to obtain information about a filter. Note that the short version of the available information is obtained by default.

```
>> d = fdesign.lowpass;
>> f = design(d);
>> info(f, 'short')
Discrete-Time FIR Filter (real)
-------------------------------
```

```
Filter Structure  : Direct-Form FIR
Filter Length     : 43
Stable            : Yes
Linear Phase      : Yes (Type 1)

>> info (f)
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 43
Stable            : Yes
Linear Phase      : Yes (Type 1)

>> info (f, 'long')
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 43
Stable            : Yes
Linear Phase      : Yes (Type 1)

Design Method Information
Design Algorithm : equiripple

Design Options
DensityFactor : 16
MinOrder      : any
MinPhase      : false
StopbandDecay : 0
StopbandShape : flat

Design Specifications
Sampling Frequency : N/A (normalized frequency)
Response           : Lowpass
Specification      : Fp,Fst,Ap,Ast
Passband Edge      : 0.45
Stopband Edge      : 0.55
```

```
Passband Ripple    : 1 dB
Stopband Atten.    : 60 dB

Measurements
Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.45
3-dB Point         : 0.46956
6-dB Point         : 0.48313
Stopband Edge      : 0.55
Passband Ripple    : 0.8919 dB
Stopband Atten.    : 60.9681 dB
Transition Width   : 0.1

Implementation Cost
Number of Multipliers : 43
Number of Adders      : 42
Number of States      : 42
MultPerInputSample    : 43
AddPerInputSample     : 42
```

See Also

`coeffs`, `isfir`, `isstable`, `islinphase`

`dfilt` in Signal Processing Toolbox™ documentation

Purpose

States from CIC filter

Syntax

```
integerstates = int(hm.states)
```

Description

`integerstates = int(hm.states)` returns the states of a CIC filter in matrix form, rather than as the native `filtstates` object. An important point about `int` is that it quantizes the state values to the smallest number of bits possible while maintaining the values accurately.

Examples

For many users, the states of multirate filters are most useful as a matrix, but the CIC filters store the states as objects. Here is how you get the states from you CIC filter as a matrix.

```
hm = mfilt.cicdecim;
hs = hm.states; % Returns a FILTSTATES.CIC object hs.
states = int(hs); % Convert object hs to a signed integer matrix.
```

After using int to convert the states object to a matrix, here is what you get.

Before converting:

```
hm.states

ans =

    Integrator: [2x1 States]
          Comb: [2x1 States]
```

After the conversion and assigning the states to `states`:

```
states

states =

           0           0
           0           0
```

See Also `filtstates.cic`, `mfilt.cicdecim`, `mfilt.cicinterp`

Purpose

Determine whether filter is allpass

Syntax

```
isallpass(hd)
isallpass(hd,tolerance)
```

Description

`isallpass(hd)` determines whether the filter object `hd` is an allpass filter, returning `1` if true and `0` if false.

`isallpass(hd,tolerance)` uses input argument `tolerance` to determine whether the numerator and denominator transfer functions for the filter are close enough in value to be considered equal, and thus allpass, returning `1` if true (the difference between the numbers is less than `tolerance`) and `0` if not.

Given an allpass filter with this transfer function

$$H(z) = \frac{a_n + \ldots + a_1 z^{-(n-1)} + z^{-n}}{1 + a_1 z^{-1} + \ldots + a_n z^{-n}}$$

if the numerator and denominator transfer functions are equal, the filter is allpass. The `tolerance` input argument lets you determine how closely the transfer functions have to match to be considered equal. This might be most helpful in fixed-point allpass filters.

Lattice coupled allpass filters always have allpass sections, this function always returns `1` for filters whose structure is `latticeca`.

Examples

Use `dfilt.allpass` to construct an allpass filter and test whether the filter is allpass.

```
c=[.8,1.5,0.4, 0.7]; % Allpass coefficients.
hd=dfilt.allpass(c)

hd =

         FilterStructure: 'Minimum-Multiplier Allpass'
    AllpassCoefficients: [.8,1.5,0.4, 0.7]
```

```
       PersistentMemory: false
                 States: [0;0;0;0;0;0;0;0]
    NumSamplesProcessed: 0

isallpass(hd)

ans =

     1
```

See Also

isfir, islinphase, ismaxphase, isminphase, isreal, issos, isstable

Purpose

Determine whether filter is FIR

Syntax

```
isfir(h)
```

Description

`isfir(h)` determines whether filter `h` is an FIR filter, returning `1`when the filter is an FIR filter, and `0` when it is IIR. `isfir` applies to `dfilt`, `mfilt`, and `adaptfilt` objects.

To determine whether `h` is an FIR filter, `isfir(h)` inspects filter `h` and determines whether the filter, in transfer function form, has a scalar denominator. If it does, it is an FIR filter.

Examples

```
d = fdesign.lowpass;
h = design(d);
isfir(h)
ans =

     1
```

returns `1` for the status of filter `h`; the filter is an FIR structure with denominator reference coefficient equal to 1.

For multirate filters, `isfir` works the same way.

```
d = fdesign.interpolator(5); % Interpolate by 5.
h = design(d); % Use the default design method.
isfir(h)

ans =

     1
```

Use `isfir` with adaptive filters as well.

See Also

`isallpass`, `islinphase`, `ismaxphase`, `isminphase`, `isreal`, `issos`, `isstable`

islinphase

Purpose

Determine whether filter is linear phase

Syntax

```
islinphase(h)
islinphase(h,tolerance)
```

Description

`islinphase(h)` determines if the filter object `h` is linear phase, and returns `1` if true and `0` if false. `adapfilt`, `dfilt`, and `mfilt` objects work with `islinphase`.

`islinphase(h,tolerance)` uses input argument `tolerance` to determine whether the filter coefficients are close enough in value to be considered symmetric or antisymmetric, returning `1` if true (the difference between the values is less than `tolerance`) and `0` if not.

The phase determination is based on the reference coefficients. A filter has linear phase if it is FIR and its transfer function coefficients are symmetric or antisymmetric. If it is IIR and it has poles on or outside the unit circle and both numerator and denominator are symmetric or antisymmetric, it is linear phase also.

Examples

This IIR filter has linear phase.

```
d = fdesign.lowpass('n,fc',10,0.55);
h = design(d,'window');
islinphase(h)
ans =

     1
```

Using the specification `nb,na,fp,fst` results in an IIR filter that is not linear phase in this design.

```
nb=15
na=10
d=fdesign.lowpass('nb,na,fp,fst',nb,na,0.45,0.55)

d =
```

```
                 Response: 'Lowpass'
            Specification: 'Nb,Na,Fp,Fst'
              Description: {4x1 cell}
        NormalizedFrequency: true
                  NumOrder: 15
                  DenOrder: 10
                     Fpass: 0.45
                     Fstop: 0.55


h=design(d) % Use the default design method iirlpnorm.

h =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
       sosMatrix: [8x6 double]
     ScaleValues: [-0.0051749857036492;1;1;1;1;1;1;1;1]
PersistentMemory: false

islinphase(h)

ans =

     0
```

See Also

isallpass, isfir, ismaxphase, isminphase, isreal, issos, isstable

ismaxphase

Purpose

Determine whether filter is maximum phase

Syntax

```
ismaxphase(h)
ismaxphase(h,tolerance)
```

Description

`ismaxphase(h)` determines if the filter object `h` is maximum phase, and returns `1` if true and `0` if false. `adapfilt`, `dfilt`, and `mfilt` objects work with `ismaxphase`.

`ismaxphase(h,tolerance)` uses input argument `tolerance` to determine whether the zeros of the filter transfer function have values that are close enough to 1 to be considered 1 or greater (on or outside the unit circle, returning `1` if true (the difference between the coefficient value and 1 is less than `tolerance`) and `0` if not.

The phase determination is based on the reference coefficients. A filter is maximum phase when the zeros of its transfer function are on or outside the unit circle, or when the numerator is a scalar.

Examples

Two examples show `ismaxphase` in use. The first is a discrete-time `dfilt` object and the second an adaptive filter.

```
fp = 100;
fst= 120;
fs = 800;
ap = 1;
ast= 80;
d = fdesign.lowpass('fp,fst,ap,ast',fp,fst,ap,ast,fs);
h = design(d,'equiripple','minphase',true);
isminphase(h)

ans =

     1
```

To make this a maximum phase filter, use `fliplr` to change the coefficient order. Reordering the coefficients this way changes the phase from minimum to maximum.

```
h.numerator=fliplr(h.numerator);
ismaxphase(h)

ans =

     1
```

returns `1` so this is a maximum phase filter. Compare to `isminphase`.

For the adaptive filter example, try the following code:

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 1;                % NLMS step size
offset = 50;           % NLMS offset
ha = adaptfilt.nlms(32,mu,1,offset);
[y,e] = filter(ha,x,d);
```

After adapting, `ha` is an FIR filter that does not exhibit maximum phase.

```
ismaxphase(ha)

ans =

     0
```

See Also

`isallpass`, `isfir`, `islinphase`, `isminphase`, `isreal`, `issos`, `isstable`

isminphase

Purpose

Determine whether filter is minimum phase

Syntax

```
isminphase(h)
isminphase(h,tolerance)
```

Description

`isminphase(h)` determines if the filter object `h` is maximum phase, and returns `1` if true and `0` if false. `adapfilt`, `dfilt`, and `mfilt` objects work with `isminphase`.

`isminphase(h,tolerance)` uses input argument `tolerance` to determine whether the values of the filter transfer function zeros are close enough to 1 to be considered to be on the unit circle, returning `1` if true (the difference between the transfer function zero values and 1 is less than `tolerance`) and `0` if not.

The determination is based on the reference coefficients. A filter is minimum phase when the zeros of its transfer function are on or inside the unit circle, or the numerator is a scalar.

Examples

This example creates a minimum-phase filter.

```
fp = 200;
fst= 230;
fs = 900;
ap = 1;
ast= 80;
d = fdesign.lowpass('fp,fst,ap,ast',fp,fst,ap,ast,fs);
h = design(d,'equiripple','minphase',true);
isminphase(h)

ans =

     1
```

When you make `h` a fixed-point filter, the quantization process results in the filter no longer being minimum phase.

```
h.arithmetic='fixed';
```

```
isminphase(h)

ans =

     0
```

See Also

isallpass, isfir, islinphase, ismaxphase, isreal, issos, isstable,

Purpose

Determine whether filter uses real coefficients

Syntax

```
isreal(hd)
```

Description

`isreal(hd)` returns `1` (or true) if all filter coefficients for the filter `hd` are real, and returns `0` (or false) otherwise.

`isreal(hd)` returns `1` if all filter coefficients in filter `hd` have zero imaginary part. Otherwise, `isreal(hd)` returns a `0` indicating that the filter is complex. Complex filters have one or more coefficients with nonzero imaginary parts.

Note Quantizing a filter cannot make a real filter into a complex filter.

Examples

To demonstrate the `isreal` test, this example creates a double-precision filter and fixed-point filter, and tests the coefficients of the fixed-point filter to see if they are strictly real.

```
d=fdesign.lowpass('n,fp,ap,ast',5,0.4,0.5,20);
hd=design(d,'ellip')

hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
       sosMatrix: [3x6 double]
ScaleValues: [0.362583368859661;0.918321077151039;
              0.496533475964919;1]
PersistentMemory: false

% Use d to design the fixed-point filter.
hq=design(d,'ellip');
% Convert to fixed-point arithmetic.
hq.arithmetic='fixed';
isreal(hq)
```

```
ans =

     1
```

See Also

isfir, islinphase, ismaxphase, isminphase, issos, isstable, isallpass

issos

Purpose

Determine whether filter is SOS form

Syntax

```
issos(hd)
```

Description

`issos(hd)` determines whether quantized filter `hq` consists of second-order sections. Returns `1` if all sections of quantized filter `hq` have order less than or equal to two, and `0` otherwise.

Examples

By default, fdesign and design return SOS filters when possible. This example designs a lowpass SOS filter that uses fixed-point arithmetic.

```
d=fdesign.lowpass('n,fp,ap,ast',40,0.55,0.1,60)

d =

               Response: 'Lowpass'
          Specification: 'N,Fp,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
            FilterOrder: 40
                  Fpass: 0.55
                  Apass: 0.1
                  Astop: 60

designmethods(d)


Design Methods for class fdesign.lowpass (N,Fp,Ap,Ast):


ellip
equiripple

hd=design(d,'ellip');
hd.arithmetic='fixed';

issos(hd)
```

```
ans =

     1
```

Fixed-point filter `hd` is in second-order section form, as is the double-precision version.

See Also

`isallpass`, `isfir`, `islinphase`, `ismaxphase`, `isminphase`, `isreal`, `isstable`

isstable

Purpose

Determine whether filter is stable

Syntax

```
isstable(hq)
```

Description

`isstable(hq)` tests quantized filter `hq` to determine whether its poles are inside the unit circle. If the poles lie on or outside the circle, `isstable` returns `0`. If the poles are inside the circle, `isstable` returns `1`.

To determine the filter stability, `isstable` checks the filter coefficients. When the poles lie on or inside the unit circle, the filter is stable. FIR filters are stable by design since the defining transfer functions do not have denominator polynomials, thus no feedback to cause instability.

Examples

Since filter stability is very important in your design process, use `isstable` to determine whether your double-precision or fixed-point IIR filter is stable.

```
d=fdesign.nyquist(2,'n,tw',24,0.1);
hd=design(d,'iirlinphase')

hd =

     FilterStructure: Cascade
            Stage(1): Scalar
            Stage(2): Parallel
                      Stage(1): Delay
                      Stage(2): Cascade
                            Stage(1): Delay
                            Stage(2): Cascade
    PersistentMemory: false

isstable(hd)

ans =

     1
```

```
hd2=design(d,'equiripple');
isstable(hd2)

ans =

     1
```

See Also

isallpass, isfir, islinphase, ismaxphase, isminphase, isreal, issos, zplane

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 `designopts` 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 halfband filter specification object.

```
d=fdesign.halfband('n,tw',100,0.004)

d =
```

```
                Response: 'Halfband with filter order and transition width'
           Specification: 'N,TW'
             Description: {2x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
             FilterOrder: 100
         TransitionWidth: 0.0040
designopts(d,'kaiserwin')

ans =

    FilterStructure: 'dffir'

hd= design(d,'kaiserwin','filterstructure','dffir')
Warning: Filter order is too low. Design may be inaccurate.

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'double'
               Numerator: [1x101 double]
    ResetBeforeFiltering: 'on'
                  States: [100x1 double]
```

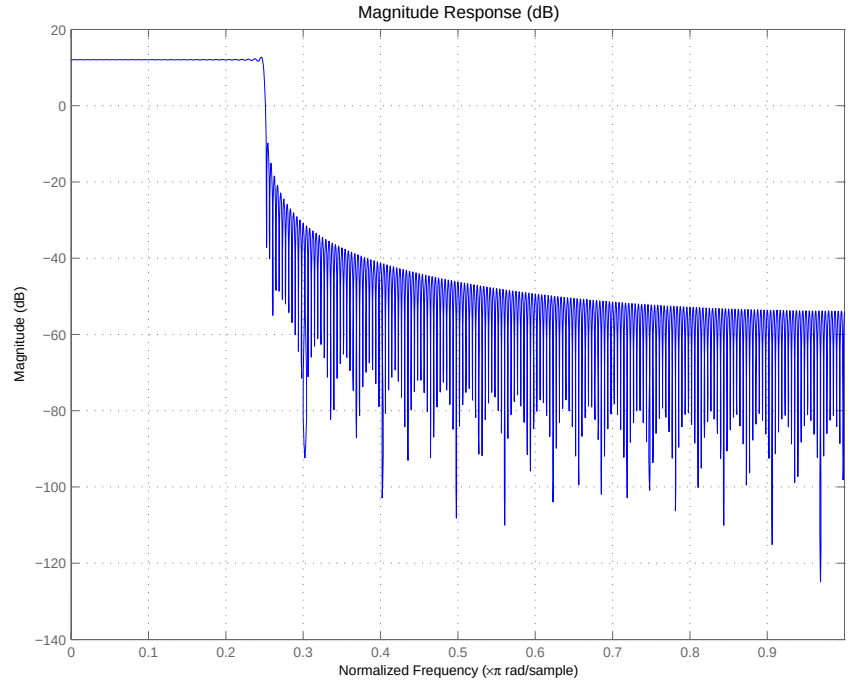
In this example, `kaiserwin` uses an interpolating filter specification object to implement a multirate filter.

```
d=fdesign.interp(4,'pl,tw',120,0.004)

d =

                Response: [1x46 char]
           Specification: 'PL,TW'
             Description: {2x1 cell}
    InterpolationFactor: 4
    NormalizedFrequency: true
                     Fs: 'Normalized'
```

```
        PolyphaseLength: 120
        TransitionWidth: 0.0040


hm = design(d,'kaiserwin')

hm =

         FilterStructure: 'Direct-Form FIR Polyphase
                           Interpolator'
               Numerator: [1x480 double]
     InterpolationFactor: 4
    ResetBeforeFiltering: 'on'
                  States: [119x1 double]
```

With the polyphase length of 120 you do not see the warning about the filter accuracy. Increasing the transition width `tw` can also reduce the possible inaccuracies.

FVTool shows clearly the multirate filter `hm`.



See Also

`equiripple`, `firls`

lagrange

Purpose

Fractional delay filter from `fdesign.fracdelay` specification object

Syntax

```
hd = lagrange(d)
hd = design(d,'lagrange')
hd = design(d,'lagrange',FilterStructure,structure)
```

Description

`hd = lagrange(d)` returns a fractional delay filter based on the Lagrange design method. By default, the filter provides the fractional delay filter design structure `fd`. Provide the fractional delay value in samples, between 0 and 1. This is the only available structure.

`hd = design(d,'lagrange')` is identical to `hd = lagrange(d)`.

`hd = design(d,'lagrange',FilterStructure,structure)` specifies the Lagrange design method and the *`structure`* filter structure for `hd`. The sole valid filter structure string for `structure` is `fd`, describing the fractional delay structure.

Examples

This example uses a fractional delay of 0.30 samples. The `help` and `designopts` commands provide the details about designing fractional delay filters.

```
d=fdesign.fracdelay(.30)

d =

               Response: 'Fractional Delay'
          Specification: 'N'
            Description: {'Filter Order'}
              FracDelay: 0.3
    NormalizedFrequency: true
            FilterOrder: 3

designmethods(d)


Design Methods for class fdesign.fracdelay (N):
```

```
lagrange

help(d,'lagrange')

 DESIGN Design a Lagrange fractional delay filter.
 HD = DESIGN(D, 'lagrange') designs a Lagrange filter
specified by the FDESIGN object D.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns
a filter with the structure STRUCTURE.  STRUCTURE is 'fd'
by default and can be any of the following:

    'fd'

% Example #1 - Design a linear Lagrange fractional
% delay filter of 0.2 samples.
h = fdesign.fracdelay(0.2,'N',2);
Hd = design(h, 'lagrange', 'FilterStructure', 'fd')

% Example #2 - Design a cubic Lagrange fractional
% delay filter.
Fs = 8000;      % Sampling frequency of 8kHz
fdelay = 50e-6; % Fractional delay of 50 microseconds.
h  = fdesign.fracdelay(fdelay,'N',3,Fs);
Hd = design(h, 'lagrange', 'FilterStructure', 'fd');
```

This example designs a linear Lagrange fractional delay filter where you set the delay to 0.2 seconds and the filter order `N` to 2.

```
h = fdesign.fracdelay(0.2,'N',2); .
hd = design(h,'lagrange','FilterStructure','fd')
```

Design a cubic Lagrange fractional delay filter with filter order equal to 3..

```
Fs = 8000;      % Sampling frequency of 8 kHz.
```

```
fdelay = 50e-6; % Fractional delay of 50 microseconds.
h  = fdesign.fracdelay(fdelay,'N',3,Fs);
hd = design(h,'lagrange','FilterStructure','fd');
```

Reference

Laakso, T. I., V. Välimäki, M. Karjalainen, and Unto K. Laine, “Splitting the Unit Delay - Tools for Fractional Delay Filter Design,” *IEEE® Signal Processing Magazine*, Vol. 13, No. 1, pp. 30-60, January 1996.

See Also

`design`, `designmethods`, `designopts`, `fdesign`, `fdesign.fracdelay`

Purpose

Response of single-rate, fixed-point IIR filter

Syntax

```
report = limitcycle(hd)
report = limitcycle(hd,ntrials,inputlengthfactor,stopcriterion)
```

Description

`report = limitcycle(hd)` returns the structure `report` that contains information about how filter `hd` responds to a zero-valued input vector. By default, the input vector has length equal to twice the impulse response length of the filter.

`limitcycle` returns a structure whose elements contain the details about the limit cycle testing. As shown in this table, the report includes the following details.

Output Object Property	Description
`LimitCycleType`	Contains one of the following results: • `Granular` — indicates that a granular overflow occurred. • `Overflow` — indicates that an overflow limit cycle occurred. • `None` — indicates that the test did not find any limit cycles.
`Zi`	Contains the initial condition value(s) that caused the detected limit cycle to occur.
`Output`	Contains the output of the filter in the steady state.
`Trial`	Returns the number of the Monte Carlo trial on which the limit cycle testing stopped. For example, `Trial = 10` indicates that testing stopped on the tenth Monte Carlo trial.

Using an input vector longer than the filter impulse response ensures that the filter is in steady-state operation during the limit cycle testing. `limitcycle` ignores output that occurs before the filter reaches the steady state. For example, if the filter impulse length is 500 samples, `limitcycle` ignores the filter output from the first 500 input samples.

To perform limit cycle testing on your IIR filter, you must set the filter `Arithmetic` property to `fixed` and `hd` must be a fixed-point IIR filter of one of the following forms:

- `df1` — direct-form I
- `df1t` — direct-form I transposed
- `df1sos` — direct-form I with second-order sections
- `df1tsos` — direct-form I transposed with second-order sections
- `df2` — direct-form II
- `df2t` — direct-form II transposed
- `df2sos` — direct-form II with second-order sections
- `df2tsos` — direct-form II transposed with second-order sections

When you use `limitcycle` without optional input arguments, the default settings are

- Run 20 Monte Carlo trials
- Use an input vector twice the length of the filter impulse response
- Stop testing if the simulation process encounters either a granular or overflow limit cycle

To determine the length of the filter impulse response, use `impzlength`:

```
impzlength(hd)
```

During limit cycle testing, if the simulation runs reveal both overflow and granular limit cycles, the overflow limit cycle takes precedence and is the limit cycle that appears in the report.

Each time you run `limitcycle`, it uses a different sequence of random initial conditions, so the results can differ from run to run.

Each Monte Carlo trial uses a new set of randomly determined initial states for the filter. Test processing stops when `limitcycle` detects a zero-input limit cycle in filter `hd`. `report = limitcycle(hd,ntrials,inputlengthfactor,stopcriterion)` lets you set the following optional input arguments:

- `ntrials` — Number of Monte Carlo trials (default is 20).
- `inputlengthfactor` — integer factor used to calculate the length of the input vector. The length of the input vector comes from `(impzlength(hd) * inputlengthfactor)`, where `inputlengthfactor = 2` is the default value.
- `stopcriterion` — the criterion for stopping the Monte Carlo trial processing. `stopcriterion` can be set to **`either`** (the default), **`granular`**, **`overflow`**. This table describes the results of each stop criterion.

stopcriterion Setting	**Description**
`either`	Stop the Monte Carlo trials when `limitcycle` detects either a granular or overflow limit cycle.
`granular`	Stop the Monte Carlo trials when `limitcycle` detects a granular limit cycle.
`overflow`	Stop the Monte Carlo trials when `limitcycle` detects an overflow limit cycle.

Note An important feature is that if you specify a specific limit cycle stop criterion, such as `overflow`, the Monte Carlo trials do not stop when testing encounters a granular limit cycle. You receive a warning that no `overflow` limit cycle occurred, but consider that a `granular` limit cycle might have occurred.

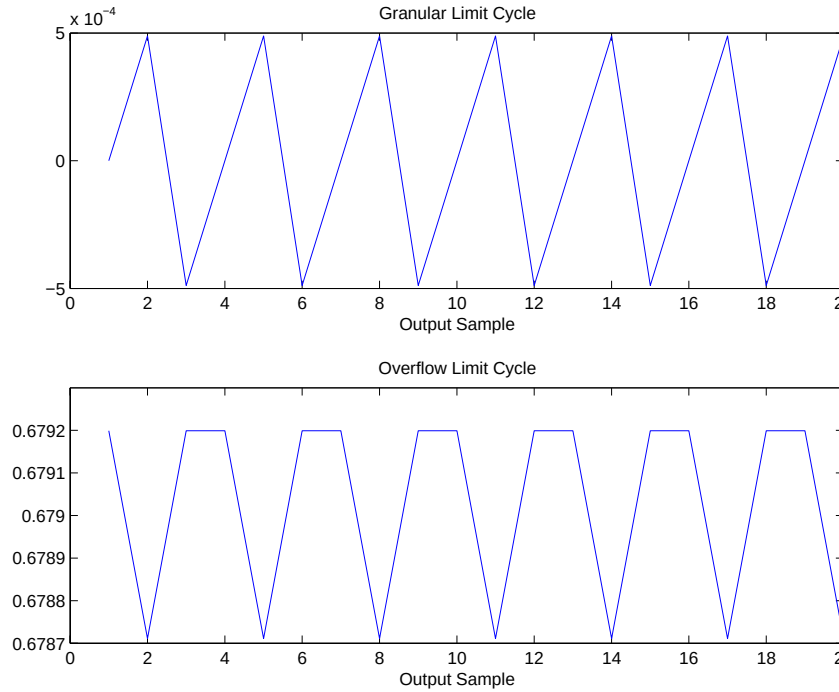
Examples

In this example, there is a region of initial conditions in which no limit cycles occur and a region where they do. If no limit cycles are detected before the Monte Carlo trials are over, the state sequence converges to zero. When a limit cycle is found, the states do not end at zero. Each time you run this example, it uses a different sequence of random initial conditions, so the plot you get can differ from the one displayed in the following figure.

```
s = [1 0 0 1 0.9606 0.9849];
hd = dfilt.df2sos(s);
hd.arithmetic = 'fixed';
greport = limitcycle(hd,20,2,'granular')
oreport = limitcycle(hd,20,2,'overflow')
figure,
subplot(211),plot(greport.Output(1:20)), title('Granular Limit Cycle');
subplot(212),plot(oreport.Output(1:20)), title('Overflow Limit Cycle');

greport =

    LimitCycle: 'granular'
            Zi: [2x1 double]
        Output: [1303x1 embedded.fi]
         Trial: 1

oreport =

    LimitCycle: 'overflow'
            Zi: [2x1 double]
        Output: [1303x1 embedded.fi]
```

```
Trial: 2
```

The plots shown in this figure present both limit cycle types — the first displays the small amplitude granular limit cycle, the second the larger amplitude overflow limit cycle.



As you see from the plots, and as is generally true, overflow limit cycles are much greater magnitude than granular limit cycles. This is why `limitcycle` favors overflow limit cycle detection and reporting.

See Also

`freqz`, `noisepsd`

Purpose

Maximum step size for adaptive filter convergence

Syntax

```
mumax = maxstep(ha,x)
[mumax,mumaxmse] = maxstep(ha,x)
```

Description

`mumax = maxstep(ha,x)` predicts a bound on the step size to provide convergence of the mean values of the adaptive filter coefficients. The columns of the matrix `x` contain individual input signal sequences. The signal set is assumed to have zero mean or nearly so.

`[mumax,mumaxmse] = maxstep(ha,x)` predicts a bound on the adaptive filter step size to provide convergence of the LMS adaptive filter coefficients in the mean-square sense. `maxstep` issues a warning when `ha.stepsize` is outside of the range 0 < `ha.stepsize` < `mumaxmse/2`.

`maxstep` is available for the following adaptive filter objects:

- `adaptfilt.blms`
- `adaptfilt.blmsfft`
- `adaptfilt.lms`
- `adaptfilt.nlms` (uses a different syntax. Refer to the text below.)
- `adaptfilt.se`

Note With `adaptfilt.nlms` filter objects, `maxstep` uses the following slightly different syntax:

```
mumax = maxstep(ha)
[mumax,mumaxmse] = maxstep(ha)
```

The maximum step size for convergence is fully defined by the filter object `ha`. Matrix `x` is not necessary. If you include an `x` input matrix, MATLAB returns an error.

Examples

Analyze and simulate a 32-coefficient (31st-order) LMS adaptive filter object. To demonstrate the adaptation process, run 2000 iterations and 50 trials.

```
% Specify [numiterations,numexamples] = size(x);
x = zeros(2000,50);
d = x;
obj = fdesign.lowpass('n,fc',31,0.5);
hd = design(obj,'window'); % FIR filter to identified.
coef = cell2mat(hd.coefficients); % Convert cell array to matrix.

for k=1:size(x,2);   % Create input and desired response signal
                     % matrices.
% Set the (k)th input to the filter.
  x(:,k) = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x,1),1)));
  n = 0.1*randn(size(x,1),1);  % (k)th observation noise signal.
  d(:,k) = filter(coef,1,x(:,k))+n; % (k)th desired signal end.
end
mu = 0.1;                       % LMS step size.
ha = adaptfilt.lms(32,mu);
[mumax,mumaxmse] = maxstep(ha,x);

Warning: Step size is not in the range 0 < mu < mumaxmse/2:
Erratic behavior might result.

mumax

mumax =

    0.0623

mumaxmse

mumaxmse =

    0.0530
```

See Also

msepred, msesim, filter

Purpose

Measure filter magnitude response

Syntax

```
measure(hd)
measure(hm)
```

Description

`measure(hd)` returns measured values for specific points in the magnitude response curve for filter object `hd`. When you use a design object `d` to create a filter (by using `fdesign.`*`type`* to create `d`), you specify one or more values that define your desired filter response. `measure(hd)` tests the filter to determine the actual values in the magnitude response of the filter, such as the stopband attenuation or the passband ripple. Comparing the results returned by `measure` to the specifications you provided in the design object helps you assess whether the filter meets your design criteria.

Note To use `measure`, `hd` or `hm` must result from using a filter design method with a filter specifications object. `measure` works with multirate filters and discrete-time filters. It does not support adaptive filters because you cannot use `fdesign.`*`type`* to construct adaptive filter specifications objects.

`measure(hd)` returns specifications determined by the response type of the design object you use to create the filter. For example, for single-rate lowpass filters made from design objects, `measure(hd)` returns the following filter specifications.

Lowpass Filter Specification	Description
`Sampling Frequency`	Filter sampling frequency.
`Passband Edge`	Location of the edge of the passband as it enters transition.
`3-dB Point`	Location of the -3 dB point on the response curve.

Lowpass Filter Specification	Description
`6-dB Point`	Location of the -6 dB point on the response curve.
`Stopband Edge`	Location of the edge of the transition band as it enters the stopband.
`Passband Ripple`	Ripple in the passband.
`Seopband Atten.`	Attenuation in the stopband.
`Transition Width`	Width of the transition between the passband and stopband, in normalized frequency or absolute frequency. Measured between `Fpass` and `Fstop`.

In contrast, when you use a bandstop design object, `measure(hd)` returns these specifications for the resulting bandstop filter.

Bandstop Filter Specification	Description
`Sampling Frequency`	Filter sampling frequency.
`First Passband Edge`	Location of the edge of the first passband.
`First 3-dB Point`	Location of the edge of the -3 dB point in the first transition band.
`First 6-dB Point`	Location of the edge of the -6 dB point in the first transition band.
`First Stopband Edge`	Location of the start of the stopband.
`Second Stopband Edge`	Location of the end of the stopband.
`Second 6-dB Point`	Location of the edge of the -6 dB point in the second transition band.

Bandstop Filter Specification	Description
`Second 3-dB Point`	Location of the edge of the -3 dB point in the second transition band.
`Second Passband Edge`	Location of the start of the second passband.
`First Passband Ripple`	Ripple in the first passband.
`Stopband Atten.`	Attenuation in the stopband.
`Second Passband Edge`	Ripple in the second passband.
`First Transition Width`	Width of the first transition region. Measured between the -3 and -6 dB points.
`Second Transition Width`	Width of the second transition region. Measured between the -6 and -3 dB points.

Filters from different filter responses return their designated sets of specifications. Also, whether the filter is single-rate or multirate changes the list of specifications that `measure` tests.

`measure(hm)` is the same as `measure(hd)`, where `hm` is a multirate filter object. For multirate filters, the set of filter specifications that `measure` returns might be different from the discrete-filter set.

The set of response measurements that `measure` returns depends on the response you use to design the filter. When `hm` is an FIR lowpass interpolator (response is lowpass), for example, `measure(hm)` returns this set of measurements.

Interpolator Filter Specification	Description
`First Passband Edge`	Location of the edge of the passband as it enters transition.

Interpolator Filter Specification	Description
`3-dB Point`	Location of the -3 dB point on the response curve.
`6-dB Point`	Location of the -6 dB point on the response curve.
`Stopband Edge`	Location of the edge of the transition band as it enters the stopband.
`Passband Ripple`	Ripple in the passband.
`Stopband Atten.`	Attenuation in the stopband.
`Transition Width`	Width of the transition between the passband and stopband, in normalized frequency or absolute frequency. Measured between `Fpass` and `Fstop`.

For reference, this is the specification object `d` that created the interpolator specifications shown in the preceding table.

```
d=fdesign.interpolator(6,'lowpass')

d =

           MultirateType: 'Interpolator'
    InterpolationFactor: 6
                Response: 'Lowpass'
           Specification: 'Fp,Fst,Ap,Ast'
             Description: {4x1 cell}
    NormalizedFrequency: true
                   Fpass: 0.133333333333333
                   Fstop: 0.166666666666667
                   Apass: 1
                   Astop: 60
```

Examples

For the first example, create a lowpass filter and check whether the actual filter meets the specifications. For this case, use normalized frequency for Fs, the default setting.

```
d2=fdesign.lowpass('Fp,Fst,Ap,Ast',0.45,0.55,0.1,80)

d2 =

               Response: 'Lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.45
                  Fstop: 0.55
                  Apass: 0.1
                  Astop: 80

designmethods(d2)


Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage

hd2=design(d2) % Use the default equiripple design method.

hd2 =

     FilterStructure: 'Direct-Form FIR'
```

```
          Arithmetic: 'double'
           Numerator: [1x68 double]
    PersistentMemory: false

measure(hd2)

ans =

Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.45
3-dB Point         : 0.47794
6-dB Point         : 0.48909
Stopband Edge      : 0.55
Passband Ripple    : 0.09615 dB
Stopband Atten.    : 80.2907 dB
Transition Width   : 0.1
```

`Stopband Edge`, `Passband Edge`, `Passband Ripple`, and `Stopband Atten.` all meet the specifications.

Now, using `Fs` in linear frequency, create a bandpass filter and measure the magnitude response characteristics.

```
d=fdesign.bandpass

d =

              Response: 'Bandpass'
         Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
           Description: {7x1 cell}
    NormalizedFrequency: true
                 Fstop1: 0.35
                 Fpass1: 0.45
                 Fpass2: 0.55
                 Fstop2: 0.65
                 Astop1: 60
                  Apass: 1
                 Astop2: 60
```

```
normalizefreq(d,false,1.5e3) % Convert to linear freq.

hd=design(d,'cheby2');

measure(hd)

ans =

Sampling Frequency      : 1.5 kHz
First Stopband Edge     : 0.2625 kHz
First 6-dB Point        : 0.31996 kHz
First 3-dB Point        : 0.32497 kHz
First Passband Edge     : 0.3375 kHz
Second Passband Edge    : 0.4125 kHz
Second 3-dB Point       : 0.42503 kHz
Second 6-dB Point       : 0.43004 kHz
Second Stopband Edge    : 0.4875 kHz
First Stopband Atten.   : 60 dB
Passband Ripple         : 0.17985 dB
Second Stopband Atten.  : 60 dB
First Transition Width  : 0.075 kHz
Second Transition Width : 0.075 kHz
```

`measure(hd)` returns the actual response values, in the units you chose. In this example, all frequencies appear in Hz because the sampling frequency is Hz.

See Also

`design`, `fdesign`, `normalizefreq`

Purpose

Multirate filter

Syntax

```
hm = mfilt.structure(input1,input2,...)
```

Description

`hm = mfilt.structure(input1,input2,...)` returns the object `hm` of type *`structure`*. As with `dfilt` and `adaptfilt` objects, you must include the *`structure`* string to construct a multirate filter object. You can, however, construct a default multirate filter object of a given structure by not including input arguments in your calling syntax.

Multirate filters include decimators and interpolators, and fractional decimators and fractional interpolators where the resulting interpolation or decimation factor is not an integer.

Structures

Each of the following multirate filter structures has a reference page of its own.

Filter Structure String	Description of Resulting Multirate Filter
`mfilt.cascade`	Cascade multirate filters to form another filter
`mfilt.cicdecim`	Cascaded integrator-comb decimator
`mfilt.cicinterp`	Cascaded integrator-comb interpolator
`mfilt.farrowsrc`	Multirate Farrow filter
`mfilt.fftfirinterp`	Overlap-add FIR polyphase interpolator
`mfilt.firdecim`	Direct-form FIR polyphase decimator
`mfilt.firfracdecim`	Direct-form FIR polyphase fractional decimator
`mfilt.firfracinterp`	Direct-form FIR polyphase fractional interpolator
`mfilt.firinterp`	Direct-form FIR polyphase interpolator

Filter Structure String	Description of Resulting Multirate Filter
`mfilt.firsrc`	Direct-form FIR polyphase sample rate converter
`mfilt.firtdecim`	Direct-form transposed FIR polyphase decimator
`mfilt.holdinterp`	FIR hold interpolator
`mfilt.iirdecim`	IIR decimator
`mfilt.iirinterp`	IIR interpolator
`mfilt.linearinterp`	FIR Linear interpolator
`mfilt.iirwdfdecim`	IIR wave digital filter decimator
`mfilt.iirwdfinterp`	IIR wave digital filter interpolator

Copying mfilt Objects

To create a copy of an `mfilt` object, use the `copy` method.

```
h2 = copy(hd)
```

Note The syntax `hd2 = hd` copies only the object handle. It does not create a new object. `hd2` and `hd` are not independent. If you change the property value for one of the two, such as `hd2`, you are changing the property for both.

Examples

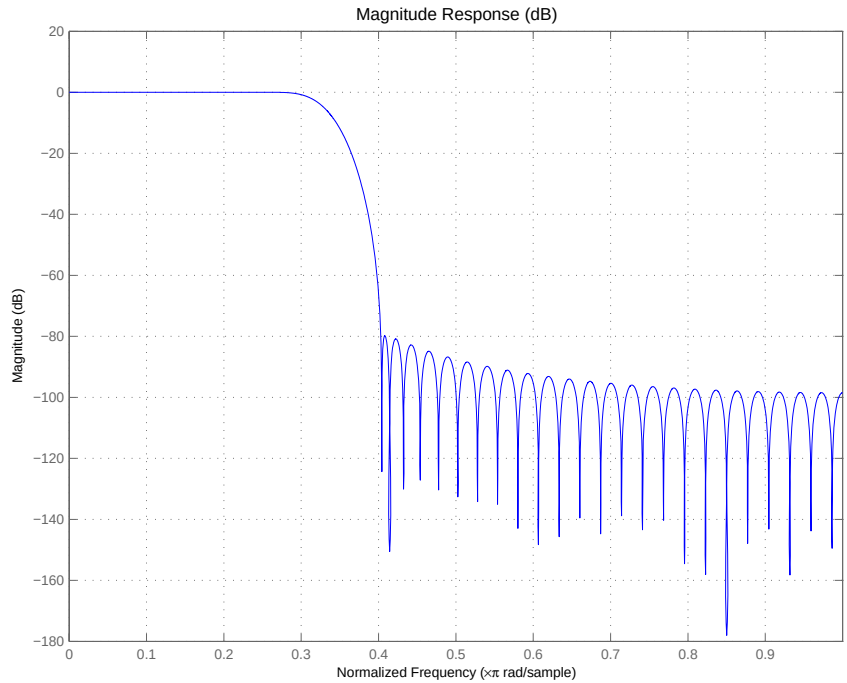
Create an FIR decimator that uses a decimation factor equal to three. In this case, the only input argument needed is m, the decimation factor. Other input arguments are available — refer to the reference page for the structure that interests you for more information.

```
m=3;
hm=mfilt.firdecim(m)
```

```
hm =

            FilterStructure: 'Direct-Form FIR Polyphase Decimator'
                  Numerator: [1x73 double]
           DecimationFactor: 3
    NumberOfSamplesProcessed: 0
                ResetStates: 'on'
                     States: [72x1 double]
```

To demonstrate a few of the methods that apply to multirate filters, here are two examples of using `hm`, your FIR decimator.

Use the Filter Visualization tool to review the magnitude response of your decimator.



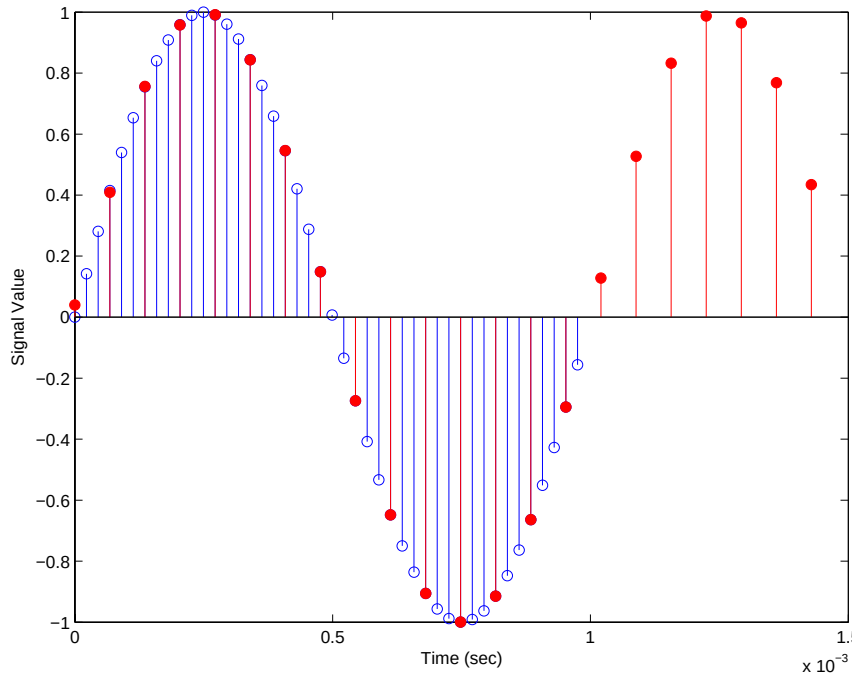
Now check to see if your filter is stable.

```
isstable(hm)

ans =

     1
```

Finally, pass a signal through the filter to see if it indeed decimates by three.

```
m = 3;                          % Decimation factor
hm = mfilt.firdecim(m);         % We use the default filter
fs = 44.1e3;                    % Original sample freq: 44.1kHz.
n = 0:10239;                    % 10240 samples, 0.232 second long
                                % signal
x  = sin(2*pi*1e3/fs*n);        % Original signal, sinusoid at 1 kHz
y = filter(hm,x);              % 5120 samples, still 0.232 seconds
stem(n(1:44)/fs,x(1:44))       % Plot original sampled at 44.1kHz
hold on                         % Plot decimated signal (22.05kHz) in red
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```

Here is the stem plot that shows the result of the decimation process.




```
hm =
        FilterStructure: 'Direct-Form FIR Polyphase
                         Decimator'
              Numerator: [1x73 double]
        DecimationFactor: 3
        PersistentMemory: 'on'
                  States: [72x1 double]
```

The filter processes 10239 samples with 1 unprocessed sample whose value is 0.8963. One nonprocessed sample results from dividing the number of samples, 10240, by the decimation factor, 3, to get 3413 output samples and one left over.

Note Multirate filters can also have complex coefficients. For example, you can specify complex coefficients in the argument `num` passed to the filter structure. This works for all multirate filter structures.

```
m = 2;
num = [0.5 0.5+0.2*i];
Hm = mfilt.firdecim(m, num);
y = filter(Hm, [1:10]);
```

See Also

`mfilt.firfracdecim`, `mfilt.firfracinterp`, `mfilt.firinterp`, `mfilt.firsrc`, `mfilt.firtdecim`

mfilt.cascade

Purpose

Cascade filter objects

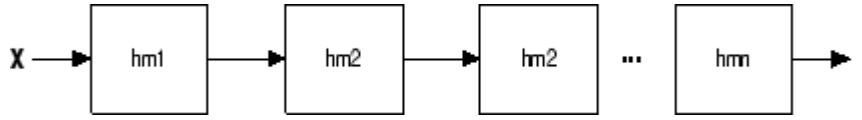
Syntax

```
hm = cascade(hm1,hm2,...,hmn)
```

Description

`hm = cascade(hm1,hm2,...,hmn)` creates filter object `hm` by cascading (connecting in series) the individual filter objects `hm1`, `hm2`, and so on to `hmn`.

In block diagram form, the cascade looks like this, with x as the input to the filter `hm` and y the output from the cascade filter `hm`:



`mfilt.cascade` accepts any combination of `mfilt` and `dfilt` objects (discrete time filters) to cascade, as well as Farrow filter objects.

Examples

Create a variety of `mfilt` objects and cascade them together.

```
hm(1) = mfilt.firdecim(12);
hm(2) = mfilt.firdecim(4);
h1 = mfilt.cascade(hm(1),hm(2));

hm(3) = mfilt.firinterp(4);
hm(4) = mfilt.firinterp(12);
h2 = mfilt.cascade(hm(3),hm(4));
```

Now cascade `h1` and `h2` together to get another multirate filter.

```
h3 = mfilt.cascade(h1,h2,9600);
```

See Also

`dfilt.cascade` in Signal Processing Toolbox™ documentation

Purpose

Fixed-point CIC decimator

Syntax

```
hm = mfilt.cicdecim(r,m,n,iwl,owl,wlps)
```

Description

`hm = mfilt.cicdecim(r,m,n,iwl,owl,wlps)` returns a cascaded integrator-comb (CIC) decimation filter object. All of the input arguments are optional.

All of the input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The following table describes the input arguments for creating `hm`.

Input Arguments	Description
`r`	Decimation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. Default value is 2.
`m`	Differential delay. Changes both the shape and number of nulls in the filter response. Also affects the null locations. Increasing `m` increases the number and sharpness of the nulls and response between nulls. Generally, one or two work best as values for `m`. Default is 1.
`n`	Number of sections. Deepens the nulls in the response curve. Note that this is the number of either comb or integrator sections, not the total section count. 2 is the default value.
`iwl`	Word length of the input signal. Use any integer number of bits. The default value is 16 bits.

Input Arguments	Description
`owl`	Word length of the output signal. It can be any positive integer number of bits. By default, `owl` is 16 bits.
`wlps`	Defines the number of bits per word in each filter section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter `wlps` as a scalar or vector of length 2*`n`, where `n` is the number of sections. When `wlps` is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator. When you elect to specify `wlps` as an input argument, the `SectionWordLengthMode` property automatically switches from the default value of `MinWordLengths` to `SpecifyWordLengths`.

Constraints and Word Length Considerations

CIC decimators have the following constraint — the word lengths of the filter section must be monotonically decreasing. The word length of each filter section must be the same size as, or smaller than, the word length of the previous filter section.

The formula for B_{max}, the most significant bit at the filter output, is given in the Hogenauer paper in the References below.

$$B_{max} = (N\log_2 RM + B_{in} - 1)$$

where B_{in} is the number of bits of the input.

The cast operations shown in the diagram in "Algorithm" on page 2-1017 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints above, the constructor returns an error.

When you specify the word lengths correctly, the most significant bit B_{max} stays the same throughout the filter, while the word length of each section either decreases or stays the same. This can cause the fraction length to change throughout the filter as least significant bits are truncated to decrease the word length, as shown in "Algorithm" on page 2-1017.

Properties of the Object

Objects have properties that control the way the object behaves. This table lists all the properties for the filter, with a description of each.

Name	Values	Default	Description
`Arithmetic`	`fixed`	`fixed`	Reports the kind of arithmetic the filter uses. CIC decimators are always fixed-point filters.
`DecimationFactor`	Any positive integer	2	Amount to reduce the input sampling rate.
`DifferentialDelay`	Any integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.
`FilterStructure`	`mfilt` structure string	None	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` objects.
`FilterInternals`	`FullPrecision`, `MinWordLengths`, `SpecifyPrecision`, `SpecifyWordLengths`	`FullPrecision`	Set the usage mode for the filter. Refer to "Usage Modes" on page 2-1008 below for details.

Name	Values	Default	Description
`InputFracLength`	Any positive integer	15	The number of bits applied to the fraction length to interpret the input data to the filter.
`InputOffset`	0 -> `r`.	0	Indicates the length of the output signal given the length of the input signal. `InputOffset` starts at zero and cycles through the phases as follows for each input sample: 0->`r`->(`r`-1)->(`r`-2)->(`r`-`p`)->0 where `p` = `r`-1.
`InputWordLength`	Any positive integer	16	The number of bits applied to the word length to interpret the input data to the filter.
`NumberOfSections`	Any positive integer	2	Number of sections used in the decimator. Generally called `n`. Reflects either the number of decimator or comb sections, not the total number of sections in the filter.
`OutputFracLength`	Any positive integer	15	The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.

Name	Values	Default	Description
`OutputWordLength`	Any positive integer	16	The number of bits applied to the word length to interpret the output data from the filter.
`PersistentMemory`	`false` or `true`	`false`	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When `PersistentMemory` is `false`, you cannot access the filter states. Setting `PersistentMemory` to `true` reveals the `States` property so you can modify the filter states.

Name	Values	Default	Description
`SectionWord Lengths`	Any integer or a vector of length 2*n.	16	Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter `SectionWordLengths` as a scalar or vector of length 2*n, where `n` is the number of sections. When `SectionWordLengths` is a scalar, the scalar value is applied to each filter section. When `SectionWordLengths` is a vector of values, the values apply to the sections in order. The default is 16 for each section in the decimator. Available when `SectionWord-LengthMode` is `SpecifyWordLengths`.

Name	Values	Default	Description
`SectionWordLengthMode`	`MinWordLengths` or `SpecifyWordLengths`	`MinWordLength`	Determines whether the filter object sets the section word lengths or you provide the word lengths explicitly. By default, the filter uses the input and output word lengths in the command to determine the optimal word lengths for each section, according to the information in [1]. When you choose `SpecifyWordLengths`, you provide the word length for each section. In addition, choosing `SpecifyWordLengths` exposes the `SectionWordLengths` property for you to modify as needed.

Name	Values	Default	Description
States	filtstates.cic object	m+1-by-n matrix of zeros, after you call function int.	Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. m is the differential delay of the comb section and n is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when PersistentMemory is true. Refer to the filtstates object in Signal Processing Toolbox™ documentation for more general information about the filtstates object.

Usage Modes

There are four modes of usage for this which are set using the FilterInternals property

- FullPrecision — All word and fraction lengths set to B_{max} + 1, called B_{accum} by fred harris in [3]. Full Precision is the default setting.
- MinWordLengths — Automatically set the sections for minimum word lengths.

- `SpecifyWordLengths` — Specify the word lengths for each section.
- `SpecifyPrecision` — Specify precision by providing values for the word and fraction lengths for each section.

Full Precision

In full precision mode, the word lengths of all sections and the output are set to B_{accum} as defined by $B_{accum} = ceil(N_{secs}(Log_2(D \times M)) + InputWordLength)$ where N_{secs} is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display looks for this mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

Minimum Wordlengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than B_{accum}, roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [1]) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than B_{accum} as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits computed using eq. 21 in Hogenauer's paper from B_{accum} to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the fraction length of that section by one bit compared to the input fraction length. Setting the output wordlength for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'MinWordLengths'

OutputWordLength: 16
```

Specify word lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length 2*(`NumberOfSections`), where each vector element represents the word length for a section. If you specify a scalar, such as B_{accum}, the

full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display looks like for the `SpecifyWordLengths` mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyWordLengths'

SectionWordLengths: [19 18 18 17]

OutputWordLength: 16
```

Specify precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the `SpecifyPrecision` mode, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the word length for each section. When you enter a scalar such as B_{accum},

mfilt.cicdecim expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length 2*(NumberOfSections) with elements that represent the fraction length for each section as well. When you enter a scalar such as B_{accum}, mfilt.cicdecim applies scalar expansion as done for the word lengths.

Here is the SpecifyPrecision display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11
```

About the States of the Filter

In the states property you find the states for both the integrator and comb portions of the filter. states is a matrix of dimensions m + 1-by-n, with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.

- States for the comb portion fill the remaining rows in the state matrix..

To review the states of a CIC filter, use `int` to assign the states to a variable in MATLAB. As an example, here are the states for a CIC decimator `hm` before and after filtering a data set.

```
x = fi(ones(1,10),true,16,0); % Fixed-point input data.
hm = mfilt.cicdecim(2,1,2,16,16,16);
sts=int(hm.states)

sts =

      0      0
      0      0

set(hm,'InputFracLength',0); % Integer input specified.
y=filter(hm,x)

sts=int(hm.states)

sts =

     10     45
     28     13
```

STS is an integer matrix that `int` returns from the contents of the `filtstates.cic` object in `hm`.

Design Considerations

When you design your CIC decimation filter, remember the following general points:

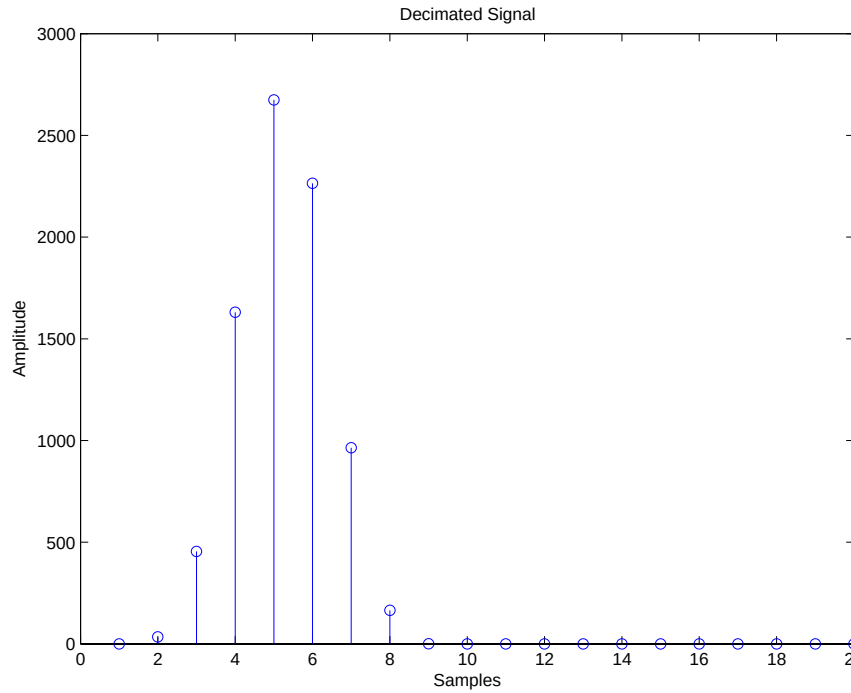
- The filter output spectrum has nulls at $\omega = k * 2\pi/rm$ radians, $k = 1,2,3....$
- Aliasing and imaging occur in the vicinity of the nulls.

- `n`, the number of sections in the filter, determines the passband attenuation. Increasing `n` improves the filter ability to reject aliasing and imaging, but it also increases the droop (or rolloff) in the filter passband. Using an appropriate FIR filter in series after the CIC decimation filter can help you compensate for the induced droop.
- The DC gain for the filter is a function of the decimation factor. Raising the decimation factor increases the DC gain.

Examples

This example applies a decimation factor `r` equal to `8` to a 160-point impulse signal. The signal output from the filter has 160/`r`, or 20, points or samples. Choosing 10 bits for the word length represents a fairly common setting for analog to digital converters. The plot shown after the code presents the stem plot of the decimated signal, with 20 samples remaining after decimation:

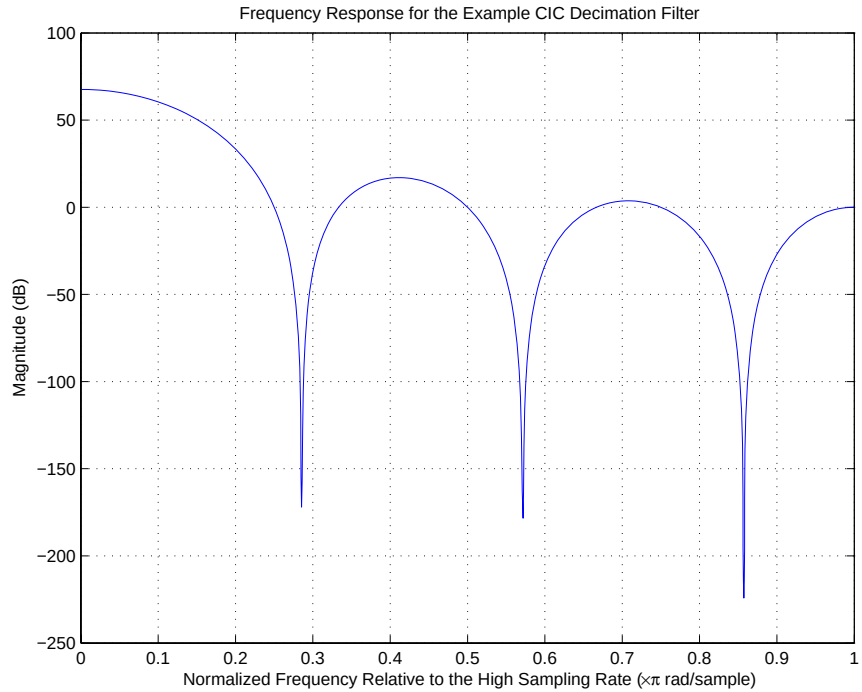
```
m = 2;  % Differential delays in the filter.
n = 4;  % Filter sections
r = 8   % Decimation factor
x = int16(zeros(160,1)); x(1) = 1;    % Create a 160-point
                                      % impulse signal.
hm = mfilt.cicdecim(r,m,n); % Expects 16-bit input
                            % by default.
y = filter(hm,x);
stem(double(y));    % Plot output as a stem plot.
xlabel('Samples'); ylabel('Amplitude');
title('Decimated Signal');
```



The next example demonstrates one way to compute the filter frequency response, using a 4-section decimation filter with the decimation factor set to 7:

```
hm = mfilt.cicdecim(7,1,4);
fvtool(hm)
```

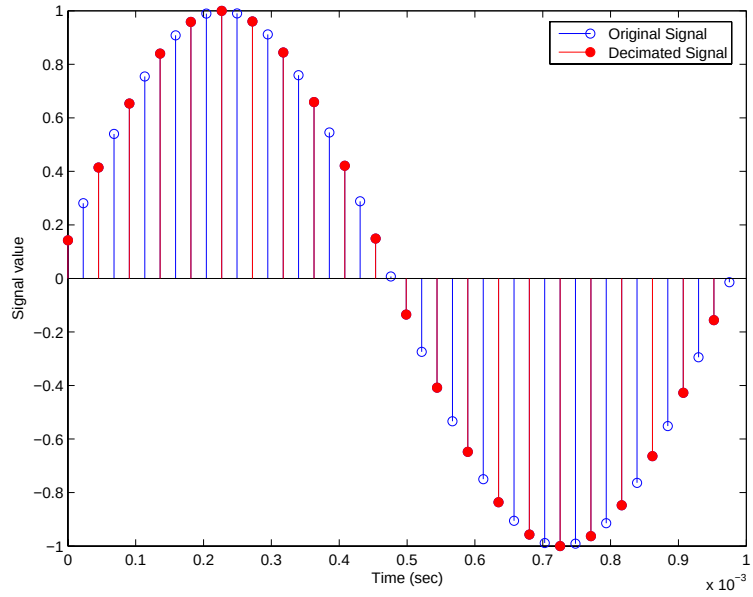
FVTool provides ways for you to change the title and x labels to match the figure shown. Here's the frequency response plot for the filter. For details about the transfer function used to produce the frequency response, refer to [1] in the References section.



This final example demonstrates the decimator for converting from 44.1 kHz audio to 22.05 kHz — decimation by two. To overlay the before and after signals, scale the output and plot the signals on a stem plot.

```
r = 2;                  % Decimation factor.
hm = mfilt.cicdecim(r); % Use default NumberOfSections &
                        % DifferentialDelay property values.
fs = 44.1e3;            % Original sampling frequency: 44.1kHz.
n = 0:10239;            % 10240 samples, 0.232 second long signal.
x  = sin(2*pi*1e3/fs*n);% Original signal, sinusoid at 1kHz.

y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% Scale the output to overlay the stem plots.
```

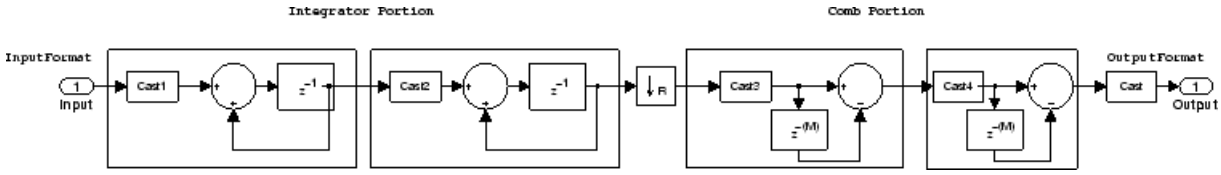
```
x = double(x);
y = double(y_fi);
y = y/max(abs(y));
stem(n(1:44)/fs,x(2:45)); hold on;  % Plot original signal
                                    % sampled at 44.1kHz.
stem(n(1:22)/(fs/r),y(3:24),'r','filled'); % Plot decimated
                                           % signal (22.05kHz)
                                           % in red.
xlabel('Time (seconds)');ylabel('Signal Value');
```



Algorithm

To show how the CIC decimation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC decimation filter (`n = 2`). *fs* is the high sampling rate, the input to the decimation process.

For details about the bits that are removed in the Comb section, refer to [1] in References.



`mfilt.cicdecim` calculates the fraction length at each section of the decimator to avoid overflows at the output of the filter.

See Also

`mfilt`, `mfilt.cicinterp`

References

[1] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE® Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(2): pp. 155-162, 1981

[2] Meyer-Baese, Uwe, "Hogenauer CIC Filters," in *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, pp. 155-172

[3] harris, fredric j, *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004 , pp. 343

Purpose

Fixed-point CIC interpolator

Syntax

```
hm = mfilt.cicinterp(r,m,n,ilw,owl,wlps)
hm = mfilt.cicinterp
hm = mfilt.cicinterp(r,...)
```

Description

`hm = mfilt.cicinterp(r,m,n,ilw,owl,wlps)` constructs a cascaded integrator-comb (CIC) interpolation filter object that uses fixed-point arithmetic.

All of the input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The following table describes the input arguments for creating `hm`.

Input Arguments	Description
`r`	Interpolation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. 2 is the default value.
`m`	Differential delay. Changes both the shape and number of nulls in the filter response. Also affects the null locations. Increasing `m` increases the number and sharpness of the nulls and response between nulls. Generally, one or two work as values for `m`. 1 is the default.
`n`	Number of sections. Deepens the nulls in the response curve. Note that this is the number of either comb or integrator sections, not the total section count. By default, the filter has two sections.
`iwl`	Word length of the input signal. Use any integer number of bits. The default value is 16 bits.

Input Arguments	Description
`owl`	Word length of the output signal. It can be any positive integer number of bits. By default, `owl` is 16 bits.
`wlps`	Defines the number of bits per word in each filter section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter `wlps` as a scalar or vector of length 2*`n`, where `n` is the number of sections. When `wlps` is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the integrator. When you elect to specify `wlps` as an input argument, the `SectionWordLengthMode` property automatically switches from the default value of `MinWordLengths` to `SpecifyWordLengths`.

`hm = mfilt.cicinterp` constructs the CIC interpolator using the default values for the optional input arguments.

`hm = mfilt.cicinterp(r,...)` constructs the CIC interpolator applying the values you provide for `r` and any other values you specify as input arguments.

Constraints and Conversions

In Hogenauer [1], the author describes the constraints on CIC interpolator filters. `mfilt.cicinterp` enforces a constraint—the word lengths of the filter sections must be non-decreasing. That is, the word length of each filter section must be the same size as, or greater than, the word length of the previous filter section.

The formula for W_j, the minimum register width, is derived in [1]. The formula for W_j is given by

$$W_j = \text{ceil}(B_{in} + \log_2 G_j)$$

where G_j, the maximum register growth up to the jth section, is given by

$$G_j = \begin{cases} 2^j, & j = 1, 2, \ldots, N \\ \dfrac{2^{2N-j}(RM)^{j-N}}{R}, & j = N+1, \ldots, 2N \end{cases}$$

When the differential delay, M, is 1, there is also a special condition for the register width of the last comb, W_N, that is given by

$$W_N = B_{in} + N - 1 \qquad \text{if } M = 1$$

The conversions denoted by the cast blocks in the integrator diagrams in "Algorithm" on page 2-1034 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints described in this section, `mfilt.cicinterp` returns an error.

The fraction lengths and scalings of the filter sections do not change. At each section the word length is either staying the same or increasing. The signal scaling can change at the output after the final filter section if you choose the output word length to be less than the word length of the final filter section.

Properties of the Object

Objects have properties that control the way the object behaves. This table lists all the properties for the filter, with a description of each.

Name	Values	Default	Description
`Arithmetic`	`fixed`	`fixed`	Reports the kind of arithmetic the filter uses. CIC interpolators are always fixed-point filters.
`InterpolationFactor`	Any positive integer	2	Amount to increase the input sampling rate.
`DifferentialDelay`	Any integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.
`FilterStructure`	`mfilt` structure string	None	Reports the type of filter object, such as a interpolator or fractional integrator. You cannot set this property — it is always read only and results from your choice of `mfilt` objects.
`FilterInternals`	`FullPrecision`, `MinWordLengths`, `Specify Precision`, `SpecifyWord Lengths`	`FullPrecision`	Set the usage mode for the filter. Refer to "Usage Modes" on page 2-1027 below for details.
`InputFracLength`	Any positive integer	16	The number of bits applied as the fraction length to interpret the input data to the filter.

Name	Values	Default	Description
`InputWordLength`	Any positive integer	16	The number of bits applied to the word length to interpret the input data to the filter.
`NumberOfSections`	Any positive integer	2	Number of sections used in the interpolator. Generally called `n`. Reflects either the number of interpolator or comb sections, not the total number of sections in the filter.
`OutputFracLength`	Any positive integer	15	The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.

Name	Values	Default	Description
`OutputWordLength`	Any positive integer	16	The number of bits applied to the word length to interpret the output data from the filter.
`PersistentMemory`	`false` or `true`	`false`	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When `PersistentMemory` is `false`, you cannot access the filter states. Setting `PersistentMemory` to `true` reveals the `States` property so you can modify the filter states.

Name	Values	Default	Description
`SectionWord Lengths`	Any integer or a vector of length 2*n.	16	Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter `SectionWordLengths` as a scalar or vector of length 2*n, where `n` is the number of sections. When `SectionWordLengths` is a scalar, the scalar value is applied to each filter section. When `SectionWordLengths` is a vector of values, the values apply to the sections in order. The default is 16 for each section in the interpolator. Available when `SectionWordLengthMode` is `SpecifyWordLengths`.

Name	Values	Default	Description
`SectionWord LengthMode`	`MinWord Lengths` `SpecifyWord Lengths`	`MinWordLength`	Determines whether the filter object sets the section word lengths or you provide the word lengths explicitly. By default, the filter uses the input and output word lengths in the command to determine the proper word lengths for each section, according to the information in [1]. When you choose `SpecifyWordLengths`, you provide the word length for each section. In addition, choosing `SpecifyWordLengths` exposes the `SectionWordLengths` property for you to modify as needed.

Name	Values	Default	Description
States	filtstates.cic object	m+1-by-n matrix of zeros, after you call function int.	Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. m is the differential delay of the comb section and n is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when PersistentMemory is true. Refer to the filtstates object in Signal Processing Toolbox™ documentation for more general information about the filtstates object.

Usage Modes

There are four modes of usage for this which are set using the FilterInternals property

- FullPrecision — All word and fraction lengths set to B_{max} + 1, called B_{accum} by fred harris in [3]. Full Precision is the default setting.
- MinWordLengths — Automatically set the sections for minimum word lengths.
- SpecifyWordLengths — Specify the word lengths for each section.

- `SpecifyPrecision` — Specify precision by providing values for the word and fraction lengths for each section.

Full Precision

In full precision mode, the word lengths of all sections and the output are set to B_{accum} as defined by

$$B_{accum} = ceil(N_{secs}(Log_2(D \times M)) + InputWordLength)$$

where N_{secs} is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display looks for this mode.

```
FilterStructure: 'Cascaded Integrator-Comb Interpolator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
InterpolationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

Minimum Wordlengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than B_{accum}, roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [1]) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than B_{accum} as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number

of bits computed using eq. 21 in Hogenauer's paper from B_{accum} to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore, each bit thrown out at a particular section decrements the frection length of that section by one bit compared to the input fraction length. Setting the output wordlength for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Interpolator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
InterpolationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'MinWordLengths'

OutputWordLength: 16
```

Specify Wordlengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length 2*(`NumberOfSections`), where each vector element represents the word length for a section. If you specify a scalar, such as B_{accum}, the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicinterp;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display looks like for the `SpecifyWordLengths` mode.

```
FilterStructure: 'Cascaded Integrator-Comb Interpolator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
InterpolationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyWordLengths'

SectionWordLengths: [19 18 18 17]

OutputWordLength: 16
```

Specify Precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the `SpecifyPrecision` mode, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the word length for each section. When you enter a scalar such as B_{accum}, `mfilt.cicinterp` expands that scalar to a vector of the appropriate size

and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the fraction length for each section as well. When you enter a scalar such as B_{accum}, `mfilt.cicinterp` applies scalar expansion as done for the word lengths.

Here is the `SpecifyPrecision` display.

```
FilterStructure: 'Cascaded Integrator-Comb Interpolator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11
```

About the States of the Filter

In the `states` property you find the states for both the integrator and comb portions of the filter. `states` is a matrix of dimensions `m+1-by-n`, with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.
- States for the comb portion fill the remaining rows in the state matrix.

To review the states of a CIC filter, or any filter object states, use `int` to assign the states to a variable in MATLAB. As an example, here are the states for a CIC interpolator `hm` before and after filtering a data set.

```
x = fi(ones(1,10),true,16,0); % Fixed-point input data.
hm = mfilt.cicinterp(2,1,2,16,16,16);
sts=int(hm.states)

sts =

      0      0
      0      0

set(hm,'InputFracLength',0); % Integer input specified.
y=filter(hm,x)

sts=int(hm.states)

sts =

     10     45
     28     13
```

Design Considerations

When you design your CIC interpolation filter, remember the following general points:

- The filter output spectrum has nulls at $\omega = k * 2\pi/rm$ radians, $k = 1,2,3....$
- Aliasing and imaging occur in the vicinity of the nulls.
- `n`, the number of sections in the filter, determines the passband attenuation. Increasing `n` improves the filter ability to reject aliasing and imaging, but it also increases the droop or rolloff in the filter passband. Using an appropriate FIR filter in series after the CIC interpolation filter can help you compensate for the induced droop.

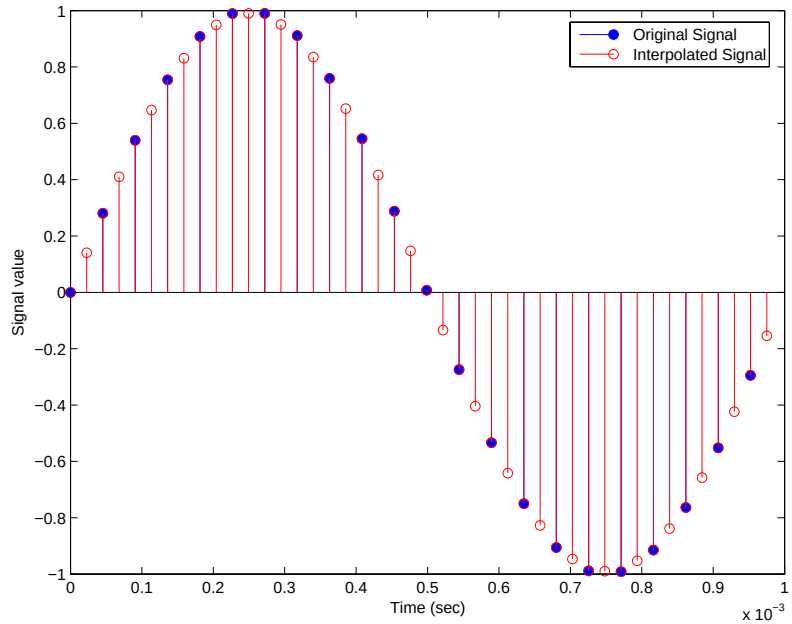
- The DC gain for the filter is a function of the interpolation factor. Raising the interpolation factor increases the DC gain.

Examples

Demonstrate interpolation by a factor of two, in this case from 22.05 kHz to 44.1 kHz. Note the scaling required to see the results in the stem plot and to use the full range of the `int16` data type.

```
R = 2;                      % Interpolation factor.
hm = mfilt.cicinterp(R);    % Use default NumberOfSections and
                            % DifferentialDelay property values.
fs = 22.05e3;               % Original sample frequency:22.05 kHz.
n = 0:5119;                % 5120 samples, .232 second long signal.
x = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz.

y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% Scale the output to overlay stem plots correctly.
x = double(x);
y = double(y_fi);
y = y/max(abs(y));
stem(n(1:22)/fs,x(1:22),'filled'); % Plot original signal sampled
                                   % at 22.05 kHz.
hold on;
stem(n(1:44)/(fs*R),y(4:47),'r');  % Plot interpolated signal
                                   % (44.1 kHz) in red.
xlabel('Time (sec)');ylabel('Signal Value');
```

As you expect, the plot shows that the interpolated signal matches the input sine shape, with additional samples between each original sample.



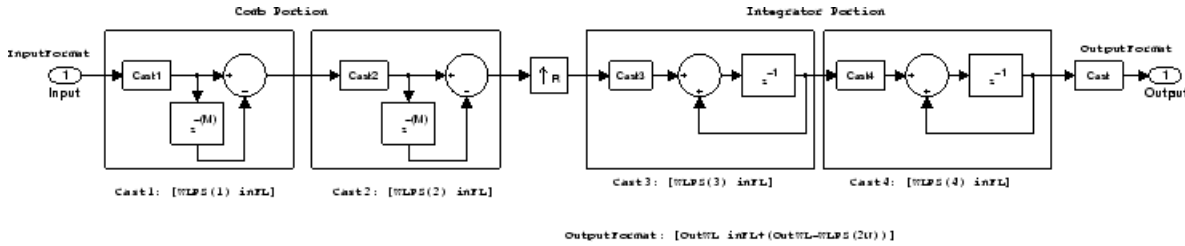
Use the filter visualization tool (FVTool) to plot the response of the interpolator object. For example, to plot the response of an interpolator with an interpolation factor of 7, 4 sections, and 1 differential delay, do something like the following:

```
hm = mfilt.cicinterp(7,1,4)
fvtool(hm)
```

Algorithm

To show how the CIC interpolation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC interpolation filter (`n = 2`). *fs* is the high sampling rate, the output from the interpolation process.

For details about the bits that are removed in the integrator section, refer to [1] in References.



When you select `MinWordLengths`, the filter section word lengths are automatically set to the minimum number of bits possible in a valid CIC interpolator. `mfilt.cicinterp` computes the wordlength for each section so the roundoff noise introduced by all sections is less than the roundoff noise introduced by the quantization at the output.

References

[1] Hogenauer, E. B., “An Economical Class of Digital Filters for Decimation and Interpolation,” IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-29(2): pp. 155-162, 1981

[2] Meyer-Baese, Uwe, “Hogenauer CIC Filters,” in Digital Signal Processing with Field Programmable Gate Arrays, Springer, 2001, pp. 155-172

[3] harris, fredric j, *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004 , pp. 343

Purpose

Sample rate converter with arbitrary conversion factor

Syntax

```
hm = mfilt.farrowsrc(L,M,C)
hm = mfilt.farrowsrc
hm = mfilt.farrowsrc(l,...)
```

Description

`hm = mfilt.farrowsrc(L,M,C)` returns a filter object that is a natural extension of dfilt.farrowfd with a time-varying fractional delay. It provides a economical implementation of a sample rate converter with an arbitrary conversion factor. This filter works well in the interpolation case, but may exhibit poor anti-aliasing properties in the decimation case.

Note You can use the `realizemdl` method to create a Simulink block of a filter created using `mfilt.farrowsrc`.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. The default value of `l` is 3.
`m`	Decimation factor for the filter. `m` specifies the amount to decrease the input sampling rate. The default value for `m` is 2.
`c`	Coefficients for the filter. When no input arguments are specified, the default coefficients are `[-1 1; 1, 0]`

`hm = mfilt.farrowsrc` constructs the filter using the default values for `l`, `m`, and `c`.

`hm = mfilt.farrowsrc(l,...)` constructs the filter using the input arguments you provide and defaults for the argument you omit.

mfilt.farrowsrc Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.farrowsrc` objects. The next table describes each property for an `mfilt.farrowsrc` filter object.

Name	Values	Description
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`Arithmetic`	String	Reports the arithmtetic precision used by the filter.
`Coefficients`	Vector	Vector containing the coefficients of the FIR lowpass filter
`InterpolationFactor`	Integer	Interpolation factor for the filter. It specifies the amount to increase the input sampling rate.

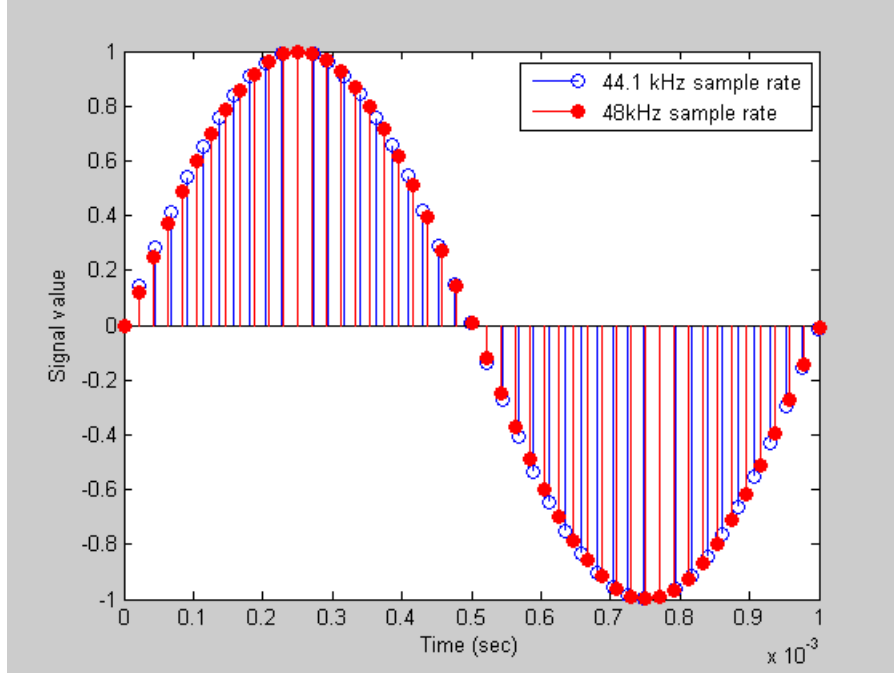
Name	Values	Description
DecimationFactor	Integer	Decimation factor for the filter. It specifies the amount to increase the input sampling rate.
PersistentMemory	false or true	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.

Example

Interpolation by a factor of 8. This object removes the spectral replicas in the signal after interpolation.

```
[L,M] = rat(48/44.1);
Hm = mfilt.farrowsrc(L,M);          % We use the default filter
Fs = 44.1e3;                        % Original sampling frequency
n = 0:9407;                         % 9408 samples, 0.213 seconds long
x  = sin(2*pi*1e3/Fs*n);            % Original signal, sinusoid at 1kHz
y = filter(Hm,x);                   % 10241 samples, still 0.213 seconds
stem(n(1:45)/Fs,x(1:45))            % Plot original sampled at 44.1kHz
hold on
% Plot fractionally interpolated signal (48kHz) in red
stem((n(2:50)-1)/(Fs*L/M),y(2:50),'r','filled')
xlabel('Time (sec)');ylabel('Signal value')
legend('44.1 kHz sample rate','48kHz sample rate')
```

The results of the example are shown in the following figure:


44.1 kHz sample rate
48kHz sample rate
Signal value
Time (sec)
x 10^-3
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

Purpose

Overlap-add FIR polyphase interpolator

Syntax

```
hm = mfilt.fftfirinterp(l,num,bl)
hm = mfilt.fftfirinterp
hm = mfilt.fftfirinterp(l,...)
```

Description

`hm = mfilt.fftfirinterp(l,num,bl)` returns a discrete-time FIR filter object that uses the overlap-add method for filtering input data.

The input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The number of FFT points is given by `[bl+ceil(length(num)/l)-1]`. It is to your advantage to choose `bl` such that the number of FFT points is a power of two—using powers of two can improve the efficiency of the FFT and the associated interpolation process.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 2.
`num`	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When `num` is not provided as an input, `fftfirinterp` uses a lowpass Nyquist filter with gain equal to `l` and cutoff frequency equal to `π/l` by default.
`bl`	Length of each block of input data used in the filtering. `bl` must be an integer. When you omit input `bl`, it defaults to 100

`hm = mfilt.fftfirinterp` constructs the filter using the default values for `l`, `num`, and `bl`.

`hm = mfilt.fftfirinterp(l,...)` constructs the filter using the input arguments you provide and defaults for the argument you omit.

mfilt.fftfirinterp Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.fftfirinterp` objects. The next table describes each property for an `mfilt.fftfirinterp` filter object.

Name	Values	Description
`FilterStructure`		Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`Numerator`		Vector containing the coefficients of the FIR lowpass filter used for interpolation.
`InterpolationFactor`		Interpolation factor for the filter. It specifies the amount to increase the input sampling rate. It must be an integer.
`BlockLength`		Length of each block of input data used in the filtering.

Name	Values	Description
`PersistentMemory`	`false` or `true`	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.
`States`		Stored conditions for the filter, including values for the interpolator states.

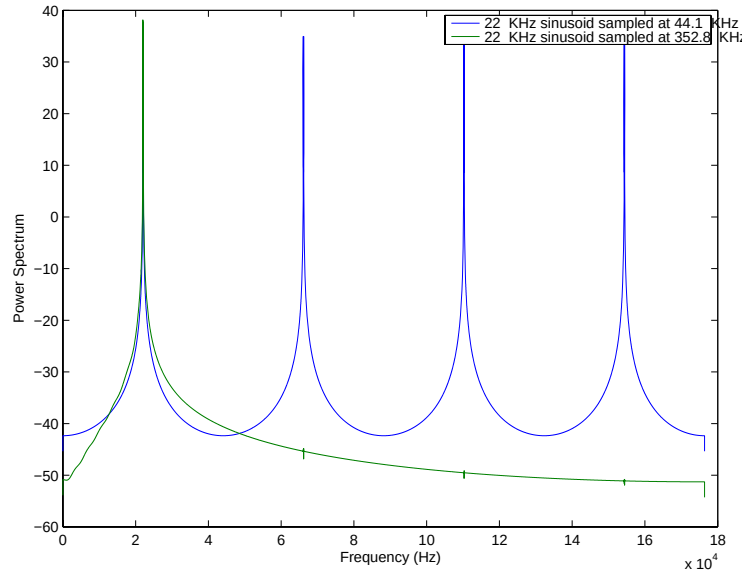
Examples

Interpolation by a factor of 8. This object removes the spectral replicas in the signal after interpolation.

```
l = 8;                          % Interpolation factor
hm = mfilt.fftfirinterp(l);     % We use the default filter
n = 8192;                       % Number of points
hm.blocklength = n;             % Set block length to number of points
fs = 44.1e3;                    % Original sample freq: 44.1 kHz.
n = 0:n-1;                      % 0.1858 secs of data
x = sin(2*pi*n*22e3/fs);        % Original signal, sinusoid at 22 kHz
y = filter(hm,x);               % Interpolated sinusoid
xu = l*upsample(x,8);           % Upsample to compare--the spectrum
                                % does not change
[px,f]=periodogram(xu,[],65536,l*fs);% Power spectrum of original
                                     % signal
[py,f]=periodogram(y,[],65536,l*fs); % Power spectrum of
                                     % interpolated signal
```

```
plot(f,10*log10(([fs*px,l*fs*py])))
legend('22  kHz sinusoid sampled at 44.1  kHz',...
'22  kHz sinusoid sampled at 352.8  kHz')
xlabel('Frequency (Hz)'); ylabel('Power Spectrum');
```

To see the results of the example, look at this figure.



See Also

`mfilt.firinterp`, `mfilt.holdinterp`, `mfilt.linearinterp`, `mfilt.firfracinterp`, `mfilt.cicinterp`

mfilt.firdecim

Purpose

Direct-form FIR polyphase decimator

Syntax

```
hm = mfilt.firdecim(m)
hm = mfilt.firdecim(m,num)
```

Description

`hm = mfilt.firdecim(m)` returns a direct-form FIR polyphase decimator object `hm` with a decimation factor of `m`. A lowpass Nyquist filter of gain 1 and cutoff frequency of `π/m` is designed by default. This filter allows some aliasing in the transition band but it very efficient because the first polyphase component is a pure delay.

`hm = mfilt.firdecim(m,num)` uses the coefficients specified by `num` for the decimation filter. This lets you specify more completely the FIR filter to use for the decimator.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`m`	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for `m` it defaults to 2.
`num`	Vector containing the coefficients of the FIR lowpass filter used for decimation. When `num` is not provided as an input, `mfilt.firdecim` constructs a lowpass Nyquist filter with gain of 1 and cutoff frequency equal to `π/m` by default. The default length for the Nyquist filter is 24*`m`. Therefore, each polyphase filter component has length 24.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firdecim` objects. The next table describes each property for an `mfilt.firdecim` filter object.

Name	Values	Description
`Arithmetic`	`Double`, `single`, `fixed`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operation mode for your filter.

Name	Values	Description
`DecimationFactor`	Integer	Decimation factor for the filter. `m` specifies the amount to reduce the sampling rate of the input signal. It must be an integer.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Describes the signal flow for the filter object.
`InputOffset`	Integers	Contains a value derived from the number of input samples and the decimation factor — `InputOffset = mod(length(nx),m)` where `nx` is the number of input samples that have been processed so far and `m` is the decimation factor.
`Numerator`	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.

Name	Values	Description
`PersistentMemory`	`false`, `true`	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` set to `false` returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to `true` allows you to modify the `States`, `InputOffset`, and `PolyphaseAccum` properties.
`PolyphaseAccum`	0 in `double`, `single`, or `fixed` for the different filter arithmetic settings.	Differentiates between the adders in the filter that work in full precision at all times (`PolyphaseAccum`) and the adders in the filter that the user controls and that may introduce quantization effects when `FilterInternals` is set to `SpecifyPrecision`.
`States`	`Double`, `single`, or `fi` matching the filter arithmetic setting.	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Double is the default setting for floating-point filters in double arithmetic.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the filter. You see one or more of these properties when you set Arithmetic to fixed. Some of the properties have different default values when they refer fixed point filters. One example is the property `PolyphaseAccum` which stores data as doubles when you use your filter in double-precision mode, but stores a `fi` object in fixed-point mode.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use `info(hm)` where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties”.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters.
`AccumWordLength`	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.

Name	Values	Description
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits[16]	Specifies the word length applied to interpret input data.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilterInternals` to `SpecifyPrecision`.

Name	Values	Description
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

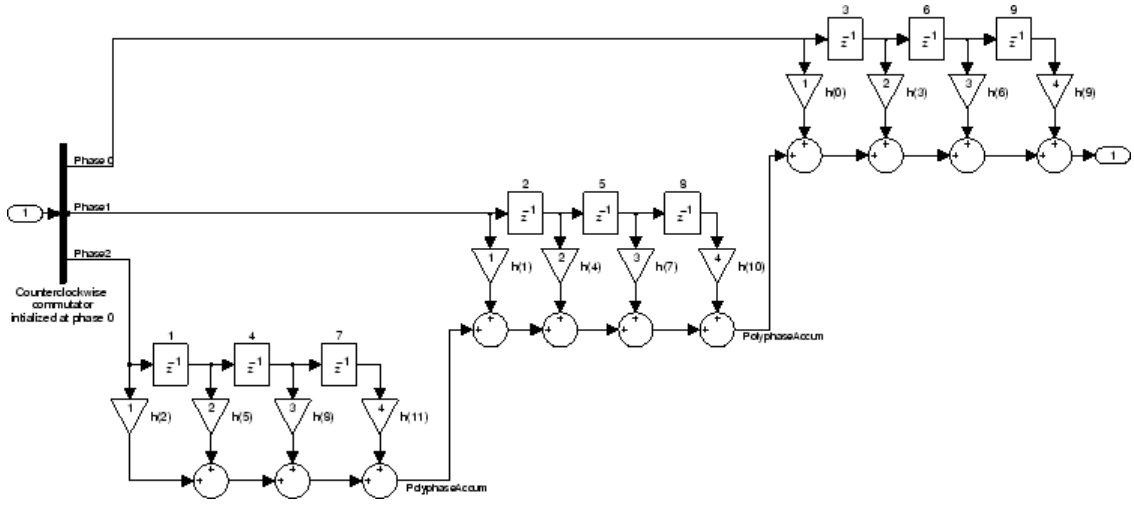
Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system. For information about the ordering of the states, refer to the filter structure section.

Filter Structure

To provide decimation, `mfilt.firdecim` uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firdecim`.



Notice the order of the states in the filter flow diagram. States 1 through 9 appear in the diagram above each delay element. State 1 applies to the first delay element in phase 2. State 2 applies to the first delay element in phase 1. State 3 applies to the first delay element in phase 0. State 4 applies to the second delay in phase 2, and so on. When you provide the states for the filter as a vector to the `States` property, the above description explains how the filter assigns the states you specify.

In property value form, the states for a filter `hm` are

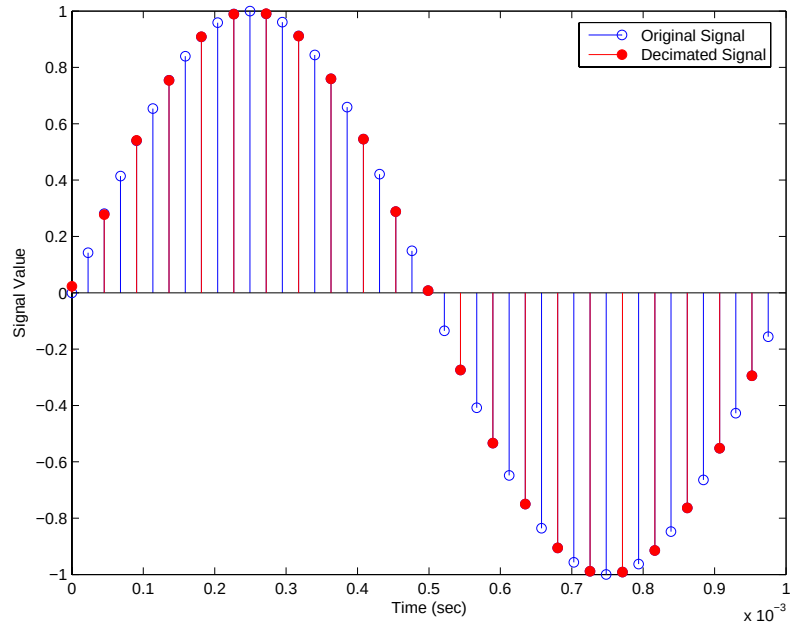
```
hm.states=[1:9];
```

Examples

Convert an input signal from 44.1 kHz to 22.05 kHz using decimation by a factor of 2. In the figure that appears after the example code, you see the results of the decimation.

```
m = 2;                      % Decimation factor.
hm = mfilt.firdecim(m);     % Use the default filter.
fs = 44.1e3;                % Original sample freq: 44.1kHz.
n = 0:10239;                % 10240 samples, 0.232 second long
                            % signal.
x  = sin(2*pi*1e3/fs*n);    % Original signal--sinusoid at 1kHz.
```

```
y = filter(hm,x);              % 5120 samples, 0.232 seconds.
stem(n(1:44)/fs,x(1:44))       % Plot original sampled at 44.1 kHz.
hold on                        % Plot decimated signal (22.05 kHz)
                               % in red.
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```



See Also

`mfilt.firtdecim`, `mfilt.firfracdecim`, `mfilt.cicdecim`

Purpose

Direct-form FIR polyphase fractional decimator

Syntax

```
hm = mfilt.firfracdecim(l,m,num)
```

Description

`hm = mfilt.firfracdecim(l,m,num)` returns a direct-form FIR polyphase fractional decimator. Input argument `l` is the interpolation factor. `l` must be an integer. When you omit `l` in the calling syntax, it defaults to 2. `m` is the decimation factor. It must be an integer. If not specified, it defaults to l+1.

`num` is a vector containing the coefficients of the FIR lowpass filter used for decimation. If you omit `num`, a lowpass Nyquist filter of gain l and cutoff frequency of π/`max(l,m)` is used by default.

By specifying both a decimation factor and an interpolation factor, you can decimate your input signal by noninteger amounts. The fractional decimator first interpolates the input, then decimates to result in an output signal whose sample rate is `l/m` of the input rate. By default, the resulting decimation factor is 2/3 when you do not provide `l` and `m` in the calling syntax. Specify `l` smaller than `m` for proper decimation.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. It must be an integer. When you do not specify a value for `l` it defaults to 2.

Input Argument	Description
num	Vector containing the coefficients of the FIR lowpass filter used for decimation. When num is not provided as an input, firfracdecim uses a lowpass Nyquist filter with gain equal to l and cutoff frequency equal to π/max(l,m) by default.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to l + 1.

mfilt.firfracdecim Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firfracdecim objects. The next table describes each property for an mfilt.firfracdecim filter object.

Name	Values	Description
FilterStructure	String	Reports the type of filter object, such as a decimator or fractional decimator. You cannot set this property — it is always read only and results from your choice of mfilt object.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for interpolation.
RateChangeFactors	[l,m]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal.

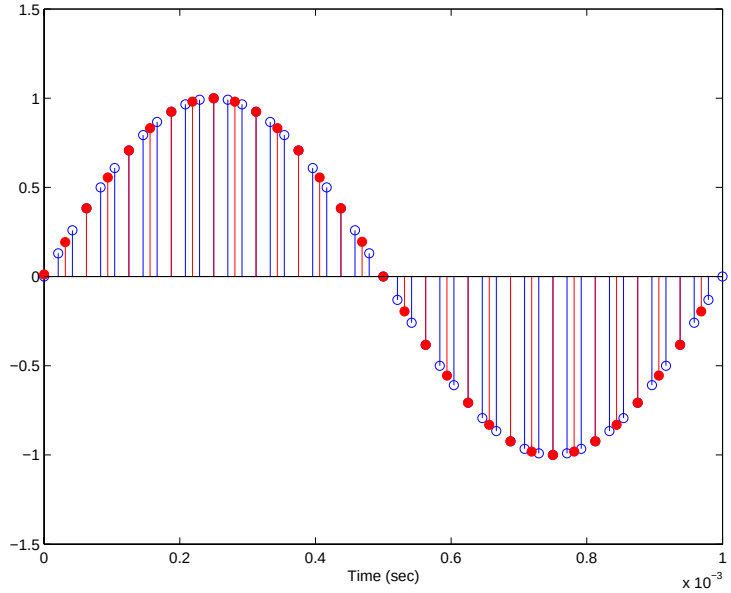
Name	Values	Description
`PersistentMemory`	`false` or `true`	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.
`States`	Matrix	Stored conditions for the delays between each interpolator phase, the filter states, and the states at the output of each phase in the filter. The number of states is `(lh-1)*m+(l-1)*(lo+mo)` where `lh` is the length of each subfilter, and `l` and `m` are the interpolation and decimation factors. `lo` and `mo`, the input and output delays between each interpolation phase, are integers from Euclid's theorem such that `lo*l-mo*m = -1` (refer to the reference for more details). Use `euclidfactors` to get `lo` and `mo` for an `mfilt.firfracdecim` object

Example

To demonstrate `firfracdecim`, perform a fractional decimation by a factor of 2/3. This is one way to downsample a 48 kHz signal to 32 kHz, commonly done in audio processing.

```
l  = 2; m = 3;                % Interpolation/decimation factors.
hm = mfilt.firfracdecim(l,m); % We use the default
fs = 48e3;                    % Original sample freq: 48 kHz.
n  = 0:10239;                 % 10240 samples, 0.213 second long
                              % signal
x  = sin(2*pi*1e3/fs*n);      % Original signal, sinusoid at 1 kHz
y  = filter(hm,x);            % 9408 samples, still 0.213 seconds
stem(n(1:49)/fs,x(1:49)); hold on; % Plot original signal sampled
                                   % at 48 kHz
stem(n(1:32)/(fs*l/m),y(13:44),'r','filled') % Plot decimated
                                             % signal at 32 kHz
xlabel('Time (sec)');
```

As shown, the plot clearly demonstrates the reduced sampling frequency of 32 kHz.



See Also

`mfilt.firsrc`, `mfilt.firfracinterp`, `mfilt.firinterp`, `mfilt.firdecim`

References

Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley & Sons, Ltd., 1994

mfilt.firfracinterp

Purpose

Direct-form FIR polyphase fractional interpolator

Syntax

```
hm = mfilt.firfracinterp(l,m,num)
```

Description

`hm = mfilt.firfracinterp(l,m,num)` returns a direct-form FIR polyphase fractional interpolator mfilt object. `l` is the interpolation factor. It must be an integer. If not specified, `l` defaults to 3.

`m` is the decimation factor. Like `l`, it must be an integer. If you do not specify `m` in the calling syntax, it defaults to 1. If you also do not specify a value for `l`, `m` defaults to 2.

`num` is a vector containing the coefficients of the FIR lowpass filter used for interpolation. If omitted, a lowpass Nyquist filter of gain `l` and cutoff frequency of π/`max(l,m)` is used by default.

By specifying both a decimation factor and an interpolation factor, you can interpolate your input signal by noninteger amounts. The fractional interpolator first interpolates the input, then decimates to result in an output signal whose sample rate is `l/m` of the input rate. For proper interpolation, you specify `l` to be greater than `m`. By default, the resulting interpolation factor is 3/2 when you do not provide `l` and `m` in the calling syntax.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 3.

Input Argument	Description
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, firfracinterp uses a lowpass Nyquist filter with gain equal to l and cutoff frequency equal to π/max(l,m) by default.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to 1. When you do not specify l as well, m defaults to 2.

mfilt.firfracinterp Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firfracinterp objects. The next table describes each property for an mfilt.firfracinterp filter object.

Name	Values	Description
FilterStructure		Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object.
Numerator		Vector containing the coefficients of the FIR lowpass filter used for interpolation.

Name	Values	Description
`RateChangeFactors`	`[l,m]`	Reports the decimation (`m`) and interpolation (`l`) factors for the filter object. Combining these factors results in the final rate change for the signal.
`PersistentMemory`	`false` or `true`	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to the default values any state that the filter changes during processing. States that the filter does not change are not affected.
`States`	Matrix	Stored conditions for the filter, including values for the interpolator and comb states.

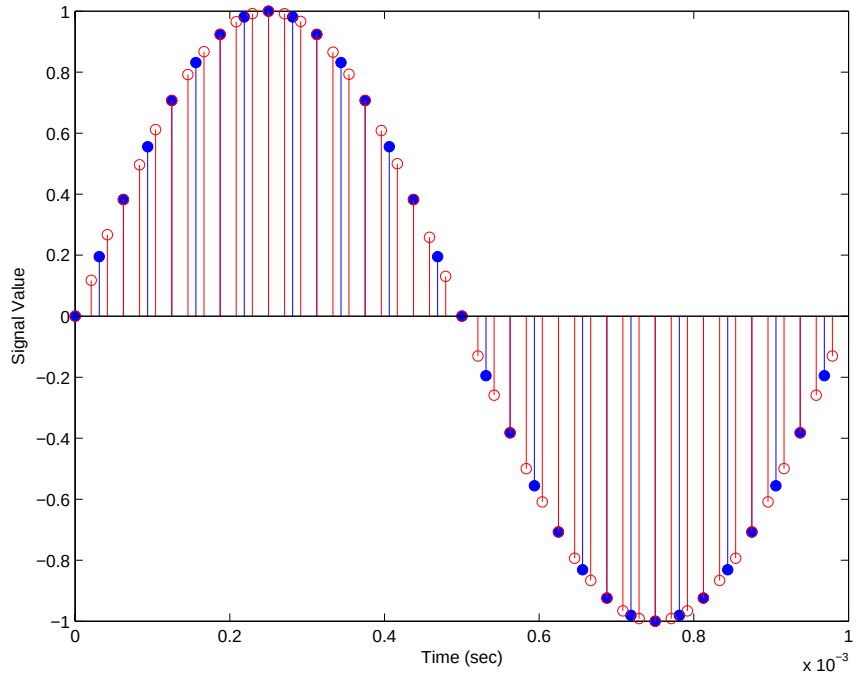
Examples

To convert a signal from 32 kHz to 48 kHz requires fractional interpolation. This example uses the `mfilt.firfracinterp` object to upsample an input signal. Setting `l = 3` and `m = 2` returns the same `mfilt` object as the default `mfilt.firfracinterp` object.

```
l  = 3; m = 2;              % Interpolation/decimation factors.
hm = mfilt.firfracinterp(l,m);    % We use the default filter
fs = 32e3;                  % Original sample freq: 32 kHz.
n = 0:6799;                 % 6800 samples, 0.212 second long signal
x  = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz
```

```
y = filter(hm,x);            % 10200 samples, still 0.212 seconds
stem(n(1:32)/fs,x(1:32),'filled') % Plot original sampled at
                                  % 32 kHz
hold on;
% Plot fractionally interpolated signal (48 kHz) in red
stem(n(1:48)/(fs*l/m),y(20:67),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```

The ability to interpolate by fractional amounts lets you raise the sampling rate from 32 to 48 kHz, something you cannot do with integral interpolators. Both signals appear in the following figure.



See Also

`mfilt.firsrc`, `mfilt.firfracdecim`, `mfilt.firinterp`, `mfilt.firdecim`

mfilt.firinterp

Purpose

FIR filter-based interpolator

Syntax

```
hm = mfilt.firinterp(l)
hm = mfilt.firinterp(l,num)
```

Description

`hm = mfilt.firinterp(l)` returns an FIR-based interpolator object `hm` with an interpolation factor of `l`. A lowpass Nyquist filter of gain `l` and cutoff frequency of π/`l` is the default if you do not include `l` as an input.

`hm = mfilt.firinterp(l,num)` uses the coefficients specified by `num` for the numerator coefficients of the interpolation filter.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 2.
`num`	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When `num` is not provided as an input, `firinterp` uses a lowpass Nyquist filter with gain equal to `l` and cutoff frequency equal to `π/l` by default. The default length for the Nyquist filter is 24*`l`. Therefore, each polyphase filter component has length 24.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firinterp` objects. The next table describes each property for an `mfilt.firinterp` filter object.

Name	Values	Description
`Arithmetic`	`Double`, `single`, `fixed`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operation mode for your filter.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Describes the signal flow for the filter object.

Name	Values	Description
`InterpolationFactor`	Integer	Interpolation factor for the filter. `1` specifies the amount to increase the sampling rate of the input signal. It must be an integer.
`Numerator`	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
`PersistentMemory`	`[false]`, `true`	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` set to `false` returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to `true` allows you to modify the `States` property.
`States`	`Double`, `single`, matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.firinterp` filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to "Multirate Filter Properties".

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumWordLength`	Any integer number of bits[39]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.

Name	Values	Description
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilterInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.

Name	Values	Description
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

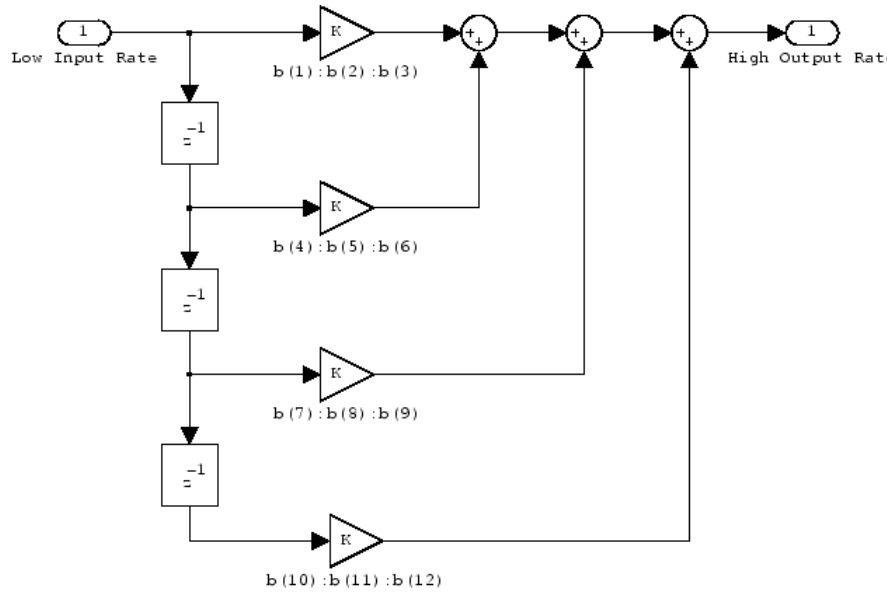
Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object to match the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system.

Filter Structure

To provide interpolation, `mfilt.firinterp` uses the following structure.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firinterp`. In the figure, the delay line updates happen at the lower input rate. The remainder of the filter — the sums and coefficients — operate at the higher output rate.

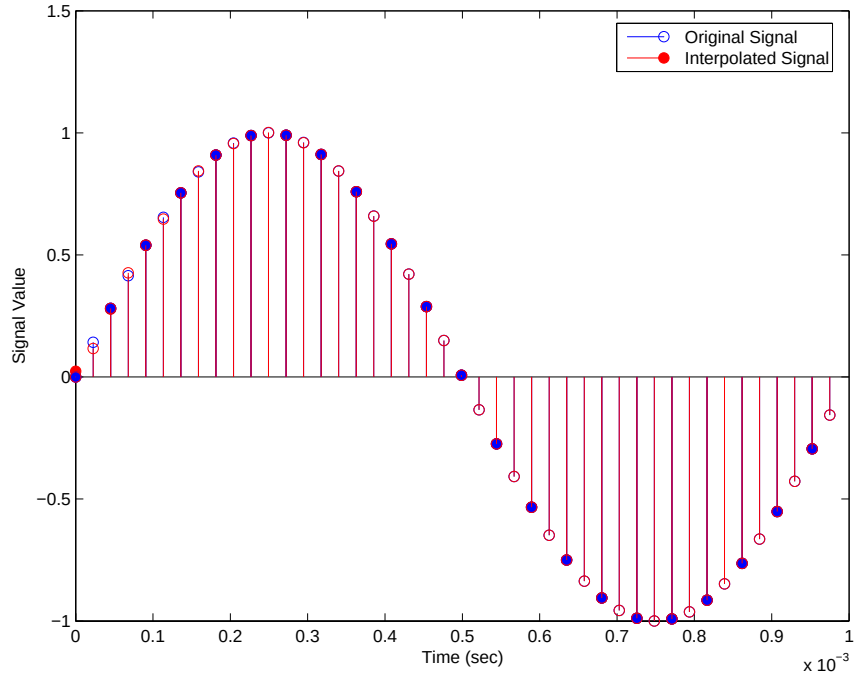


Examples

This example uses `mfilt.firinterp` to double the sample rate of a 22.05 kHz input signal. The output signal ends up at 44.1 kHz. Although `l` is set explicitly to 2, this represents the default interpolation value for `mfilt.firinterp` objects.

```
l = 2;                      % Interpolation factor.
hm = mfilt.firinterp(l);    % Use the default filter.
fs = 22.05e3;                % Original sample freq: 22.05 kHz.
n = 0:5119;                   % 5120 samples, 0.232s long signal.
x  = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz.
y = filter(hm,x);           % 10240 samples, still 0.232s.
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
                                  % 22.05 kHz.
hold on;

% Plot interpolated signal (44.1 kHz) in red
stem(n(1:44)/(fs*l),y(25:68),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```

With interpolation by 2, the resulting signal perfectly matches the original, but with twice as many samples — one between each original sample, as shown in the following figure.



See Also

`mfilt.holdinterp`, `mfilt.linearinterp`, `mfilt.fftfirinterp`, `mfilt.firfracinterp`, `mfilt.cicinterp`

Purpose

Direct-form FIR polyphase sample rate converter

Syntax

```
hm = mfilt.firsrc(l,m,num)
```

Description

`hm = mfilt.firsrc(l,m,num)` returns a direct-form FIR polyphase sample rate converter. `l` specifies the interpolation factor. It must be an integer and when omitted in the calling syntax, it defaults to 2.

`m` is the decimation factor. It must be an integer. If not specified, `m` defaults to 1. If `l` is also not specified, `m` defaults to 3 and the overall rate change factor is 2/3.

You specify the coefficients of the FIR lowpass filter used for sample rate conversion in `num`. If omitted, a lowpass Nyquist filter with gain `l` and cutoff frequency of π/`max(l,m)` is the default.

Combining an interpolation factor and a decimation factor lets you use `mfilt.firsrc` to perform fractional interpolation or decimation on an input signal. Using an `mfilt.firsrc` object applies a rate change factor defined by `l/m` to the input signal. For proper rate changing to occur, `l` and `m` must be relatively prime — meaning the ratio `l/m` cannot be reduced to a ratio of smaller integers.

When you are doing sample-rate conversion with large values of `l` or `m`, such as `l` or `m` greater than 20, using the `mfilt.firsrc` structure is the most effective approach. Other possible fractional rate change structures, such as `mfilt.firfracinterp` (where `l > m`) or `mfilt.firfracdecim` (where `l < m`) may have prohibitively large memory requirements for applications that require large rate changes.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Note You can use the `realizemdl` method to create a Simulink block of a filter created using `mfilt.firsrc`.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l`, it defaults to 2.
`num`	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When `num` is not provided as an input, `mfilt.firsrc` uses a lowpass Nyquist filter with gain equal to `l` and cutoff frequency equal to `π/max(l,m)` by default. The default length for the Nyquist filter is 24*`m`. Therefore, each polyphase filter component has length 24.
`m`	Decimation factor for the filter. `m` specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for `m`, it defaults to 1. When `l` is unspecified as well, `m` defaults to 3.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also

input arguments for creating `mfilt.firsrc` objects. The next table describes each property for an `mfilt.firsrc` filter object.

Name	Values	Description
`Arithmetic`	`[Double]`, `single`, `fixed`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operation mode for your filter.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Describes the signal flow for the filter object.
`InputOffset`	Integers	Contains a value derived from the number of input samples and the decimation factor — `InputOffset = mod(length(nx),m)` where `nx` is the number of input samples and `m` is the decimation factor.
`Numerator`	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
`PersistentMemory`	`false`, `true`	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` set to `false` returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to `true` allows you to modify the `States`, `InputOffset`, and `PolyphaseAccum` properties.

Name	Values	Description
RateChangeFactors	Positive integers. [2 3]	Specifies the interpolation and decimation factors [l m] (the rate change factors) for changing the input sample rate by nonintegral amounts.
States	Double, single, matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.firsrc filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to “Multirate Filter Properties”.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters.

Name	Values	Description
`AccumWordLength`	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.

Name	Values	Description
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilterInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`RateChangeFactors`	Positive integers [2 3]	Specifies the interpolation and decimation factors `[l m]` (the rate change factors) for changing the input sample rate by nonintegral amounts.

Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system. For information about the ordering of the states, refer to the filter structure section.

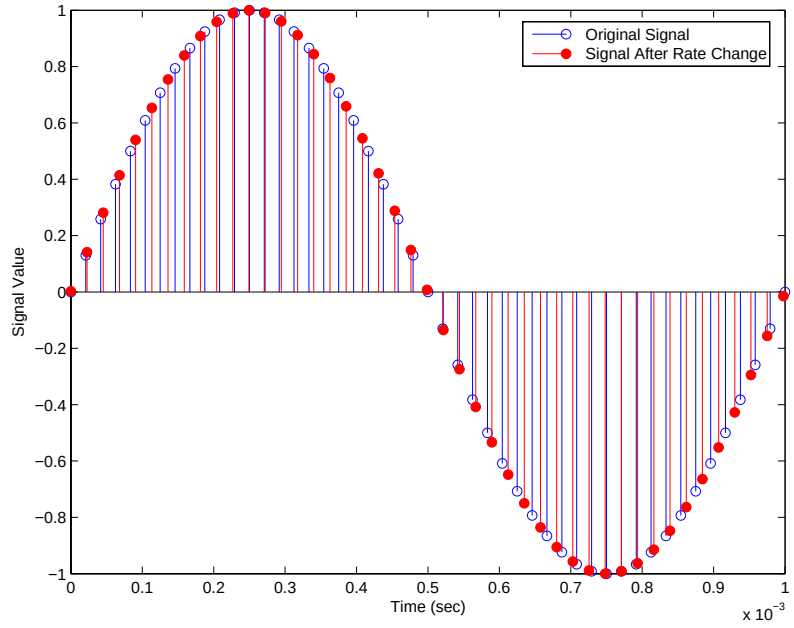
Examples

This is an example of a common audio rate change process — changing the sample rate of a high end audio (48 kHz) signal to the compact disc sample rate (44.1 kHz). This conversion requires a rate change factor of 0.91875, or `l` = 147 and `m` = 160.

```
l  = 147; m = 160;            % Interpolation/decimation factors.
hm = mfilt.firsrc(l,m);       % Use the default FIR filter.
fs = 48e3;                    % Original sample freq: 48 kHz.
n = 0:10239;                  % 10240 samples, 0.213 seconds long.
x  = sin(2*pi*1e3/fs*n);      % Original signal, sinusoid at 1 kHz.
y = filter(hm,x);             % 9408 samples, still 0.213 seconds.
stem(n(1:49)/fs,x(1:49))      % Plot original sampled at 48 kHz.
hold on

% Plot fractionally decimated signal (44.1 kHz) in red
stem(n(1:45)/(fs*l/m),y(13:57),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```

Fractional decimation provides you the flexibility to pick and choose the sample rates you want by carefully selecting `l` and `m`, the interpolation and decimation factors, that result in the final fractional decimation.

The following figure shows the signal after applying the rate change filter `hm` to the original signal.



See Also

`mfilt.firfracinterp`, `mfilt.firfracdecim`, `mfilt.firinterp`, `mfilt.firdecim`

Purpose

Direct-form transposed FIR filter

Syntax

```
hm = mfilt.firtdecim(m)
hm = mfilt.firtdecim(m,num)
```

Description

`hm = mfilt.firtdecim(m)` returns a polyphase decimator `mfilt` object `hm` based on a direct-form transposed FIR structure with a decimation factor of `m`. A lowpass Nyquist filter of gain 1 and cutoff frequency of π/`m` is the default.

`hm = mfilt.firtdecim(m,num)` uses the coefficients specified by `num` for the decimation filter. `num` is a vector containing the coefficients of the transposed FIR lowpass filter used for decimation. If omitted, a lowpass Nyquist filter with gain of 1 and cutoff frequency of π/`m` is the default.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, firtdecim uses a lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to π/m by default. The default length for the Nyquist filter is 24*m. Therefore, each polyphase filter component has length 24.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firtdecim objects. The next table describes each property for an mfilt.firtdecim filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Specifies the arithmetic the filter uses to process data while filtering.
DecimationFactor	Integer	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer.

Name	Values	Description
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Also describes the signal flow for the filter object.
`InputOffset`	Integers	Contains a value derived from the number of input samples and the decimation factor — `InputOffset = mod(length(nx),m)` where `nx` is the number of input samples that have been processed so far and `m` is the decimation factor.
`Numerator`	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
`PersistentMemory`	`[false]`, `true`	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` set to `false` returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to `true` allows you to modify the `States`, `InputOffset`, and `PolyphaseAccum` properties.

Name	Values	Description
PolyphaseAccum	Double, single [0]	The idea behind having both PolyphaseAccum and Accum is to differentiate between the adders in the filter that work in full precision at all times (PolyphaseAccum) from the adders in the filter that the user controls and that may introduce quantization effects when FilterInternals is set to SpecifyPrecision.
States	Double, single matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.firtdecim filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to “Multirate Filter Properties”.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumWordLength`	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilterInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.

Name	Values	Description
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`PolyphaseAccum`	`fi` object with zeros to start	Differentiates between the adders in the filter that work in full precision at all times (`PolyphaseAccum`) and the adders in the filter that the user controls and that may introduce quantization effects when `FilterInternals` is set to `SpecifyPrecision`.

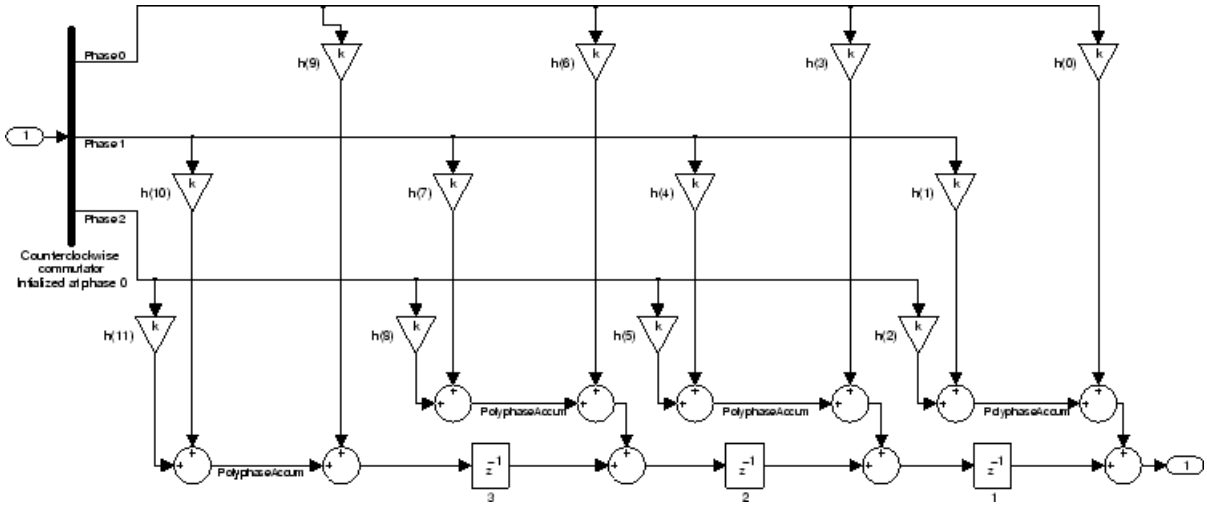
Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system. For information about the ordering of the states, refer to the filter structure section.

Filter Structure

To provide sample rate changes, `mfilt.firtdecim` uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter. To keep track of the position of the commutator, the `mfilt` object uses the property `InputOffset` which reports the current position of the commutator in the filter.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firtdecim`.



Notice the order of the states in the filter flow diagram. States 1 through 3 appear in the following diagram at each delay element. State 1 applies to the third delay element in phase 2. State 2 applies to the second delay element in phase 2. State 3 applies to the first delay element in phase 2. When you provide the states for the filter as a vector to the `States` property, the above description explains how the filter assigns the states you specify.

In property value form, the states for a filter `hm` are

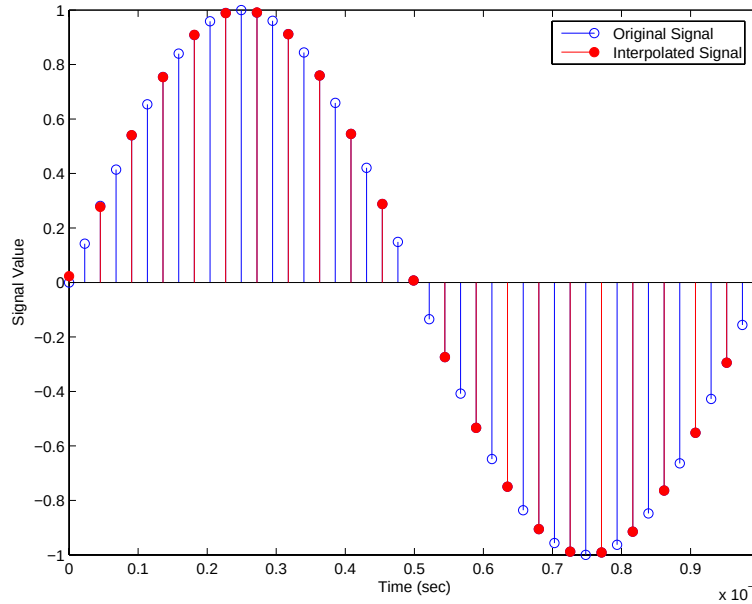
```
hm.states=[1:3];
```

Examples

Demonstrate decimating an input signal by a factor of 2, in this case converting from 44.1 kHz down to 22.05 kHz. In the figure shown following the code, you see the results of decimating the signal.

```
m = 2;                  % Decimation factor.
hm = mfilt.firtdecim(m); % Use the default filter coeffs.
fs = 44.1e3;            % Original sample freq: 44.1 kHz.
n = 0:10239;            % 10240 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n); % Original signal--sinusoid at 1 kHz.
y = filter(hm,x);       % 5120 samples, 0.232 seconds.
```

```
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1 kHz.
hold on                  % Plot decimated signal (22.05 kHz) in red
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```



See Also

`mfilt.firdecim`, `mfilt.firfracdecim`, `mfilt.cicdecim`

mfilt.holdinterp

Purpose

FIR hold interpolator

Syntax

```
hm = mfilt.holdinterp(l)
```

Description

`hm = mfilt.holdinterp(l)` returns the object `hm` that represents a hold interpolator with the interpolation factor `l`. To work, `l` must be an integer. When you do not include `l` in the calling syntax, it defaults to 2. To perform interpolation by noninteger amounts, use one of the fractional interpolator objects, such as `mfilt.firsrc` or `mfilt.firfracinterp`.

When you use this hold interpolator, each sample added to the input signal between existing samples has the value of the most recent sample from the original signal. Thus you see something like a staircase profile where the interpolated samples form a plateau between the previous and next original samples. The example demonstrates this profile clearly. Compare this to the interpolation process for other interpolators in the toolbox, such as `mfilt.linearinterp`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.holdinterp` objects. The next table describes each property for an `mfilt.interp` filter object.

Name	Values	Description
`Arithmetic`	`Double`, `single`, `fixed`	Specifies the arithmetic the filter uses to process data while filtering.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`Interpolation-Factor`	Integer	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer.

Name	Values	Description
PersistentMemory	'false' or 'true'	Determines whether the filter states are restored to zero for each filtering operation.
States	Double or single array	Filter states. states defaults to a vector of zeros that has length equal to nstates(hm). Always available, but visible in the display only when PersistentMemory is true.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.holdinterp filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to "Multirate Filter Properties".

Name	Values	Description
`Arithmetic`	`Double`, `single`, `fixed`	Specifies the arithmetic the filter uses to process data while filtering.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`Interpolation-Factor`	Integer	Interpolation factor for the filter. `1` specifies the amount to increase the input sampling rate. It must be an integer.

Name	Values	Description
`PersistentMemory`	`'false'` or `'true'`	Determine whether the filter states get restored to zero for each filtering operation
`States`	`fi` object	Contains the filter states before, during, and after filter operations. For hold interpolators, the states are always empty — hold interpolators do not have states. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system.

Filter Structure

Hold interpolators do not have structures or filter coefficients.

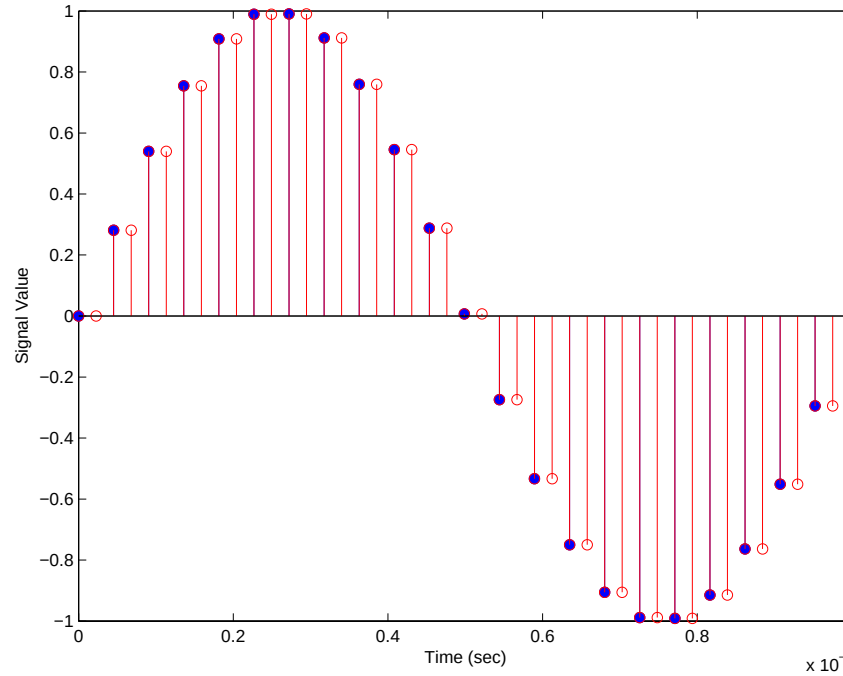
Examples

To see the effects of hold-based interpolation, interpolate an input sine wave from 22.05 to 44.1 kHz. Note that each added sample retains the value of the most recent original sample.

```
l = 2;                      % Interpolation factor
hm = mfilt.holdinterp(l);
fs = 22.05e3;               % Original sample freq: 22.05 kHz.
n = 0:5119;                 % 5120 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz
y = filter(hm,x);           % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
                                  % 22.05 kHz
hold on                     % Plot interpolated signal (44.1 kHz)
```

```
in red
stem(n(1:44)/(fs*l),y(1:44),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```

The following figure shows clearly the step nature of the signal that comes from interpolating the signal using the hold algorithm approach. Compare the output to the linear interpolation used in `mfilt.linearinterp`.



See Also

`mfilt.linearinterp`, `mfilt.firinterp`, `mfilt.firfracinterp`, `mfilt.cicinterp`

Purpose

IIR decimator

Syntax

```
hm = mfilt.iirdecim(c1,c2,...)
```

Description

`hm = mfilt.iirdecim(c1,c2,...)` constructs an IIR decimator filter given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The resulting IIR decimator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR decimators.

Usually you do not construct IIR decimators explicitly. Instead, you obtain an IIR decimator filter as a result of designing a halfband decimator. The first example in the following section illustrates this case.

Examples

Design an elliptic halfband decimator with a decimation factor of 2. The example specifies the optional sampling frequency argument.

```
tw = 100;   % Transition width of filter.
ast = 80;   % Stopband attenuation of filter.
fs = 2000;  % Sampling frequency of signal to filter.
m = 2;      % Decimation factor.
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

`d` contains the specifications for a decimator defined by `tw`, `ast`, `m`, and `fs`.

Use the specification object `d` to perform an actual filter design. `hm` is an `mfilt.iirdecim` filter object.

```
hm = design(d,'ellip','filterstructure','iirdecim');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

Designing a linear phase decimator is similar to the previous example. In this case, design a halfband linear phase decimator with decimation factor of 2.

```
tw  = 100;  % Transition width of filter.
ast = 60;   % Stopband attenuation of filter.
fs  = 2000; % Sampling frequency of signal to filter.
m = 2;      % Decimation factor.
```

Create a specification object for the decimator.

```
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Finally, design the actual filter `hm`. As designed, `hm` is an `mfilt.iirdecim` filter object.

```
hm = design(d,'iirlinphase','filterstructure','iirdecim');
% Note that realizemdl requires Simulink
realizemdl(hm)  % Build model of the filter.
```

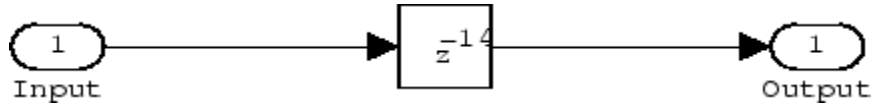
The filter implementation appears in this model, generated by `realizemdl` and Simulink®.

Given the design specifications shown here

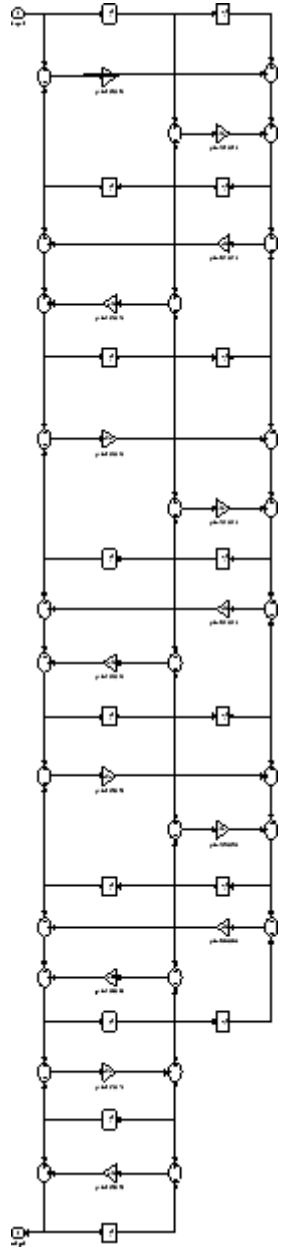
```
hm =

     FilterStructure: 'IIR Polyphase Decimator'

          Polyphase: Phase1: Section1: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
```

```
                    Phase2: Section1: [1.14740498857167 0.409481636102326]

                            Section2: [0.751016281415127 0.36048597074495]
                            Section3: [0.272921271612044 0.343931116911137]
                            Section4: [-0.244601181956782 0.33691092991289]
                            Section5: [-0.711317191438094 0.333590883744604]
                            Section6: [-1.03562723857273 0.332039064718955]
                            Section7: 0.893704991634848
                            Section8: -0.575824830892574
          DecimationFactor: 2
        PersistentMemory: false
```

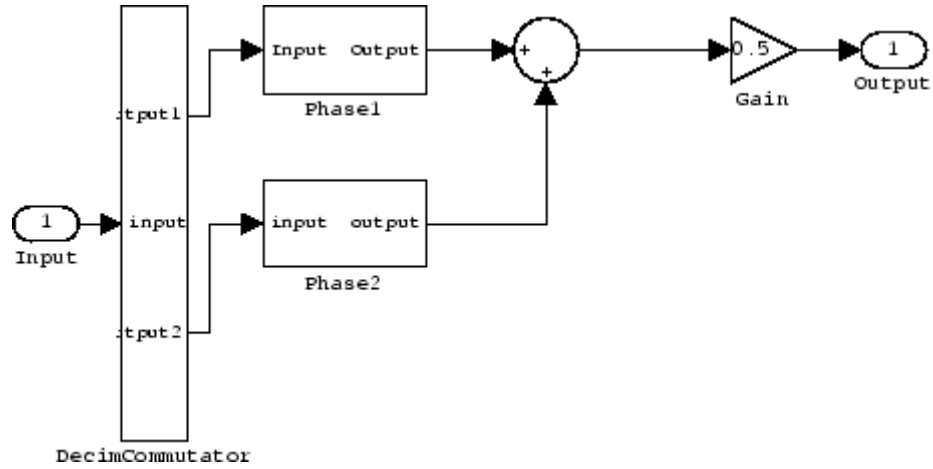
the first phase is a delay section with 0s and a 1 for coefficients and the second phase is a linear phase decimator, shown in the next models.



Phase 1 model



Phase 2 model



Overall model

See Also

`dfilt.cascadeallpass`, `mfilt`, `mfilt.iirinterp`, `mfilt.iirwdfdecim`

Purpose

IIR interpolator

Syntax

```
hm = mfilt.iirinterp(c1,c2,...)
```

Description

`hm = mfilt.iirinterp(c1,c2,...)` constructs an IIR interpolator filter given the coefficients specified in the cell arrays C1, C2, etc.

The IIR interpolator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array—if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and a unique element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR interpolators.

Usually you do not construct IIR interpolators explicitly. Instead, you obtain an IIR interpolator filter as a result of designing a halfband interpolator. The first example in the following section illustrates this case.

Examples

Design an elliptic halfband interpolator with a interpolation factor of 2.

```
tw = 100;  % Transition width of filter.
ast = 80;   % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l  = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

Specification object `d` stores the interpolator design specifics. With the details in `d`, design the filter, returning `hm`, an `mfilt.iirinterp` object. Use `hm` to realize the filter if you have Simulink® installed.

```
hm = design(d,'ellip','filterstructure','iirinterp');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

Designing a linear phase halfband interpolator follows the same pattern.

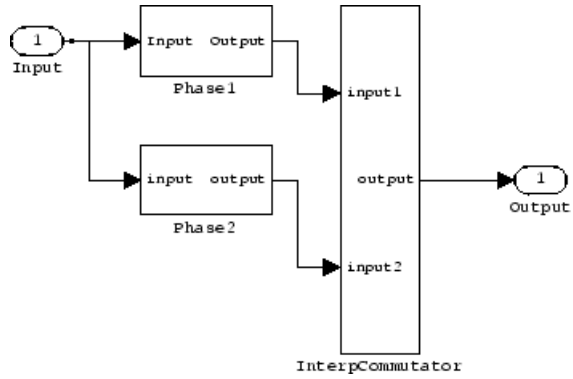
```
tw = 100;  % Transition width of filter.
ast= 60;   % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l = 2;     % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

`fdesign.interpolator` returns a specification object that stores the design features for an interpolator.

Now perform the actual design that results in an `mfilt.iirinterp` filter, `hm`.

```
hm = design(d,'iirlinphase','filterstructure','iirinterp');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

The toolbox creates a Simulink model for `hm`, shown here. Phase1, Phase2, and InterpCommutator are all subsystem blocks.



See Also dfilt.cascadeallpass, mfilt, mfilt.iirdecim, mfilt.iirwdfinterp

mfilt.iirwdfdecim

Purpose

IIR wave digital filter decimator

Syntax

```
hm = mfilt.iirwdfdecim(c1,c2,...)
```

Description

`hm = mfilt.iirwdfdecim(c1,c2,...)` constructs an IIR wave digital decimator given the coefficients specified in the cell arrays c1, c2, and so on. The IIR decimator `hm` is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirwdfdecim` interprets them same way as `mfilt.cascadewdfallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadewdfallpass` section. This exception occurs with quasi-linear phase IIR decimators.

Usually you do not construct IIR wave digital filter decimators explicitly. Instead, you obtain an IIR wave digital filter decimator as a result of designing a halfband decimator. The first example in the following section illustrates this case.

Examples

Design an elliptic halfband decimator with a decimation factor equal to 2. Both examples use the `iirwdfdecim` filter structure (an input argument to the `design` method) to design the final decimator.

The first portion of this example generates a filter specification object `d` that stores the specifications for the decimator.

```
tw  = 100;  % Transition width of filter to design, 100 Hz.
ast = 80;   % Stopband attenuation of filter 80 dB.
fs  = 2000; % Sampling frequency of the input signal.
m   = 2;    % Decimation factor.
```

```
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Now perform the actual design using `d`. Filter object `hm` is an `mfilt.iirwdfdecim` filter.

```
Hm = design(d,'ellip','FilterStructure','iirwdfdecim');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

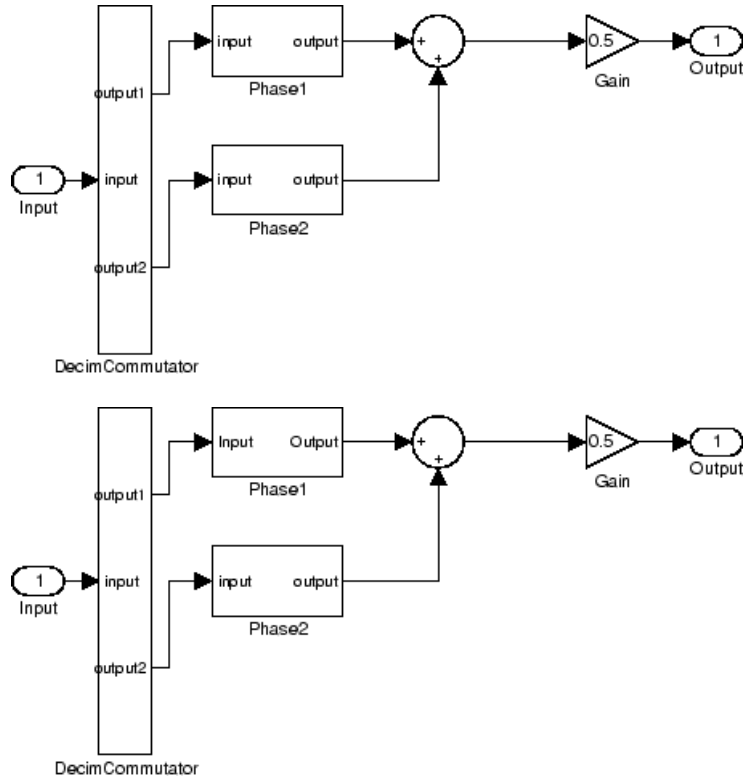
Design a linear phase halfband decimator for decimating a signal by a factor of 2.

```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 60;   % Filter stopband attenuation = 80 dB
fs  = 2000; % Input signal sampling frequency.
m   = 2;    % Decimation factor.
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

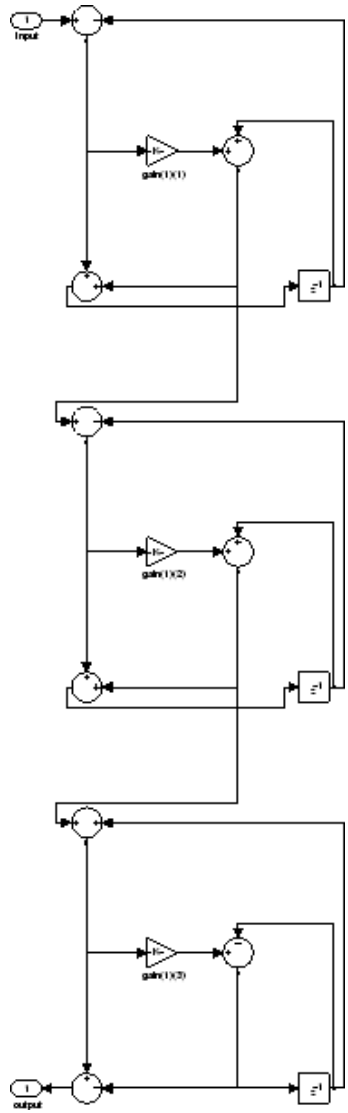
Use `d` to design the final filter `hm`, an `mfilt.iirwdfdecim` object.

```
hm = design(d,'iirlinphase','filterstructure',...
'iirwdfdecim');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

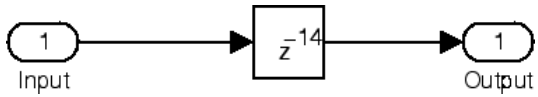
The models that `realizemdl` returns for each example appear below. At this level, the realizations of the filters are identical. The differences appear in the subsystem blocks Phase1 and Phase2.



This is the Phase1 subsystem from the halfband model.



Phase1 subsystem from the linear phase model is less revealing—an allpass filter.



See Also

`dfilt.cascadewdfallpass`, `mfilt`, `mfilt.iirdecim`, `mfilt.iirwdfinterp`

Purpose

IIR wave digital filter interpolator

Syntax

```
hm = mfilt.iirwdfinterp(c1,c2,...)
```

Description

`hm = mfilt.iirwdfinterp(c1,c2,...)` constructs an IIR wave digital interpolator given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The IIR interpolator `hm` is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirwdfinterp` interprets them same way as `mfilt.cascadewdfallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadewdfallpass` section. This exception occurs with quasi-linear phase IIR interpolators.

Usually you do not construct IIR wave digital filter interpolators explicitly. Rather, you obtain an IIR wave digital interpolator as a result of designing a halfband interpolator. The first example in the following section illustrates this case.

Examples

Design an elliptic halfband interpolator with interpolation factor equal to 2. At the end of the design process, hm is an IIR wave digital filter interpolator.

```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 80;   % Stopband attenuation of filter, 80 dB.
fs  = 2000; % Sampling frequency after interpolation.
l   = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

The specification object `d` stores the interpolator design requirements. Now use `d` to design the actual filter `hm`.

```
hm = design(d,'ellip','filterstructure','iirwdfinterp');
```

If you have Simulink® installed, you can realize your filter as a model built from blocks in Signal Processing Blockset.

```
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

For variety, design a linear phase halfband interpolator with an interpolation factor of 2.

```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 80;   % Stopband attenuation of filter, 80 dB.
fs  = 2000; % Sampling frequency after interpolation.
l   = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

Now perform the actual design process with `d`. Filter `hm` is an IIR wave digital filter interpolator. As in the previous example, `realizemdl` returns a Simulink model of the filter if you have Simulink installed.

```
hm = design(d,'iirlinphase','filterstructure',...
'iirwdfinterp');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

See Also

`dfilt.cascadewdfallpass`, `mfilt.iirinterp`, `mfilt.iirwdfdecim`

Purpose

Linear interpolator

Syntax

```
hm = mfilt.linearinterp(l)
```

Description

`hm = mfilt.linearinterp(l)` returns an FIR linear interpolator `hm` with an integer interpolation factor `l`. Provide `l` as a positive integer. The default value for the interpolation factor is 2 when you do not include the input argument `l`.

When you use this linear interpolator, the samples added to the input signal have values between the values of adjacent samples in the original signal. Thus you see something like a smooth profile where the interpolated samples continue a line between the previous and next original samples. The example demonstrates this smooth profile clearly. Compare this to the interpolation process for `mfilt.holdinterp`, which creates a stairstep profile.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input argument for `mfilt.linearinterp`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.linearinterp` objects. The next table describes each property for an `mfilt.linearinterp` filter object.

Name	Values	Description
`Arithmetic`	`Double`, `single`, `fixed`	Specifies the arithmetic the filter uses to process data while filtering.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`InterpolationFactor`	Integer	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer.
`PersistentMemory`	'false' or 'true'	Determine whether the filter states get restored to zero for each filtering operation
`States`	Double or single array	Filter states. `states` defaults to a vector of zeros that has length equal to `nstates(hm)`. Always available, but visible in the display only when `PersistentMemory` is `true`.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.holdinterp` filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to "Multirate Filter Properties".

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits. Depends on L. [29 when L=2]	Specifies the fraction length used to interpret data output by the accumulator.
`AccumWordLength`	Any integer number of bits [33]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.

Name	Values	Description
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.

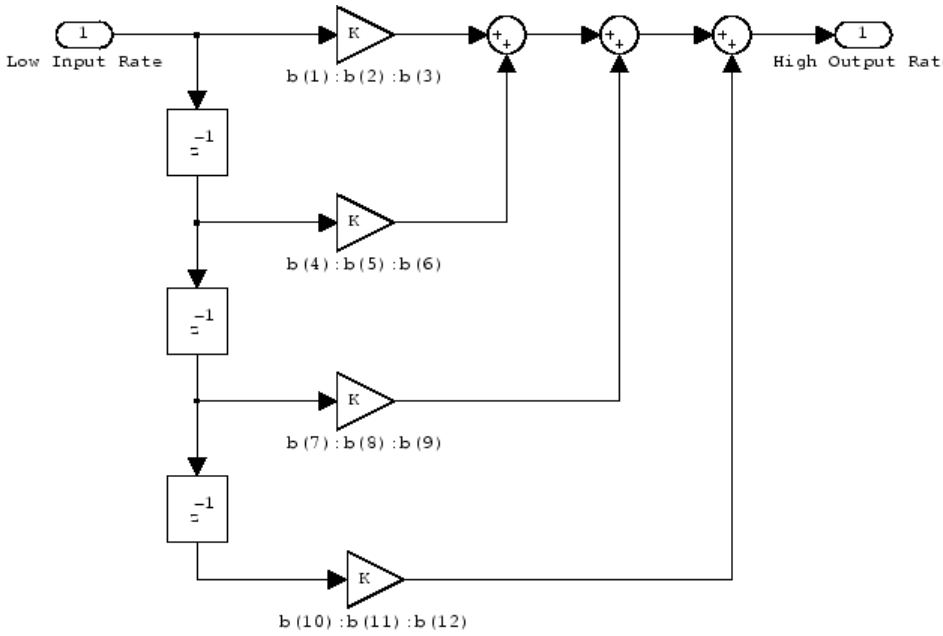
Name	Values	Description
`OutputFracLength`	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilterInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`RoundMode`	[convergent], `ceil`,`fix`,`floor`, `round`	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). • `convergent` — Round up to the next allowable quantized value. • `ceil` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `1`. • `fix` — Round negative numbers up and positive numbers down to the next allowable quantized value. • `floor` — Round down to the next allowable quantized value. • `round` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in \&tm_fixedpointtoolbox; Toolbox documentation or in the online Help system. For information about the ordering of the states, refer to the filter structure in the following section.

Filter Structure

Linear interpolator structures depend on the FIR filter you use to implement the filter. By default, the structure is direct-form FIR.

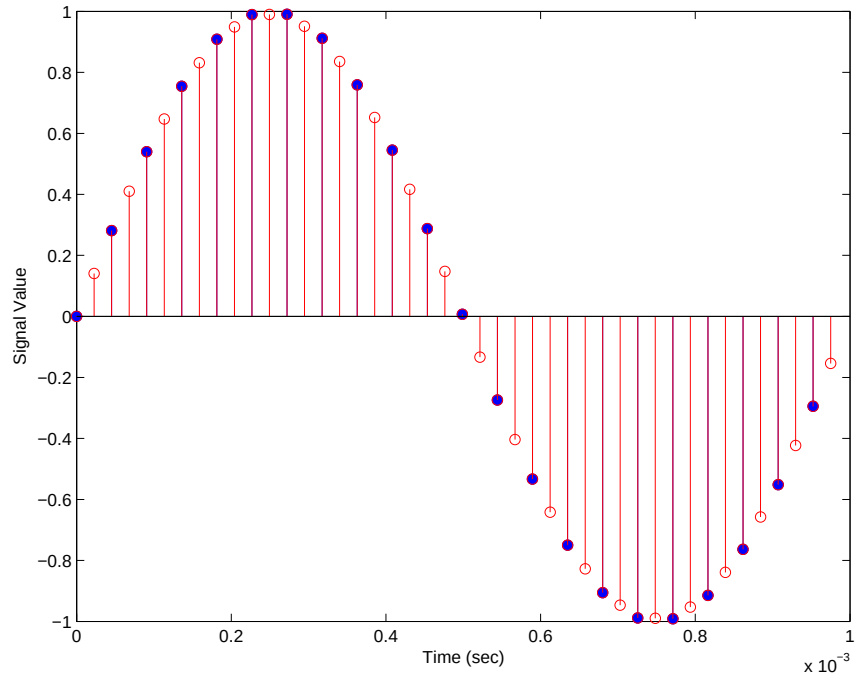


Examples

Interpolation by a factor of 2 (used to convert the input signal sampling rate from 22.05 kHz to 44.1 kHz).

```
l = 2;                      % Interpolation factor
hm = mfilt.linearinterp(l);
fs = 22.05e3;               % Original sample freq: 22.05 kHz.
n = 0:5119;                 % 5120 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz
y = filter(hm,x);           % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
                                  % 22.05 kHz
hold on                     % Plot interpolated signal (44.1
                            % kHz) in red
stem(n(1:44)/(fs*l),y(2:45),'r')
xlabel('Time (s)');ylabel('Signal Value')
```

Using linear interpolation, as compared to the hold approach of `mfilt.holdinterp`, provides greater fidelity to the original signal.



See Also

`mfilt.holdinterp`, `mfilt.firinterp`, `mfilt.firfracinterp`, `mfilt.cicinterp`

msepred

Purpose

Predicted mean-squared error for adaptive filter

Syntax

```
[mmse,emse] = msepred(ha,x,d)
[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d)
[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d,m)
```

Description

`[mmse,emse] = msepred(ha,x,d)` predicts the steady-state values at convergence of the minimum mean-squared error (`mmse`) and the excess mean-squared error (`emse`) given the input and desired response signal sequences in `x` and `d` and the property values in the adaptfilt object `ha`.

`[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d)` calculates three sequences corresponding to the analytical behavior of the LMS adaptive filter defined by `ha`:

- `meanw` — contains the sequence of coefficient vector means. The columns of matrix `meanw` contain predictions of the mean values of the LMS adaptive filter coefficients at each time instant. The dimensions of `meanw` are (`size(x,1)`)-by-(`ha.length`).
- `mse` — contains the sequence of mean-square errors. This column vector contains predictions of the mean-square error of the LMS adaptive filter at each time instant. The length of `mse` is equal to `size(x,1)`.
- `tracek` — contains the sequence of total coefficient error powers. This column vector contains predictions of the total coefficient error power of the LMS adaptive filter at each time instant. The length of `tracek` is equal to `size(x,1)`.

`[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d,m)` specifies an optional input argument `m` that is the decimation factor for computing `meanw`, `mse`, and `tracek`. When `m` > 1, `msepred` saves every `m`th predicted value of each of these sequences. When you omit the optional argument `m`, it defaults to one.

Note `msepred` is available for the following adaptive filters only: — `adaptfilt.blms` — `adaptfilt.blmsfft` — `adaptfilt.lms` — `adaptfilt.nlms` — `adaptfilt.se` Using `msepred` is the same for any `adaptfilt` object constructed by the supported filters.

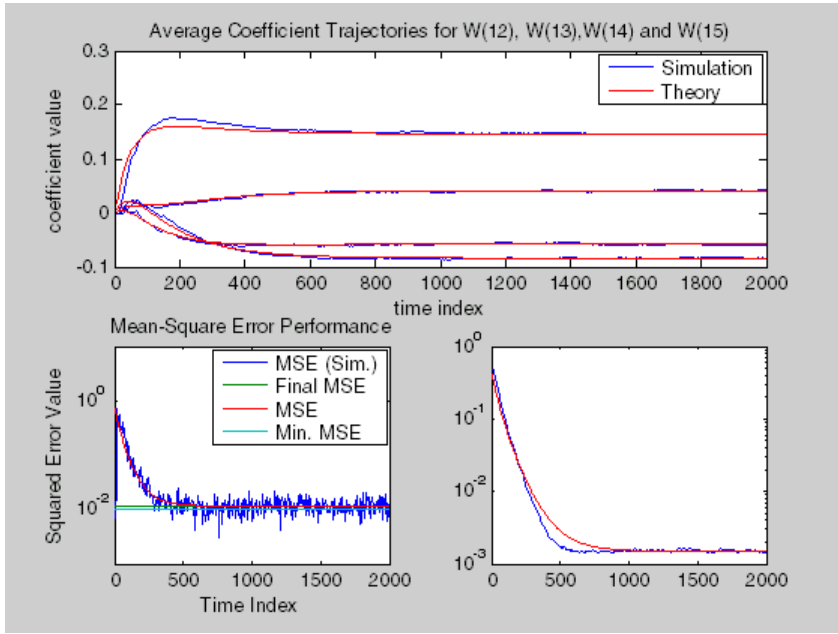
Examples

Analyze and simulate a 32-coefficient adaptive filter using 25 trials of 2000 iterations each.

```
x = zeros(2000,25); d = x;     % Initialize variables
ha = fir1(31,0.5);             % FIR system to be identified
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));        % observation noise signal
d = filter(ha,1,x)+n;          % desired signal
l = 32;                        % Filter length
mu = 0.008;                    % LMS step size.
m  = 5;                        % Decimation factor for analysis
                               % and simulation results
ha = adaptfilt.lms(l,mu);
[mmse,emse,meanW,mse,traceK] = msepred(ha,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r');
title('Average Coefficient Trajectories for W(12), W(13),...
 W(14) and W(15)');
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
(emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse]);
title('Mean-Square Error Performance');
axis([0 size(x,1) 0.001 10]);
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
```

```
subplot(2,2,4);
semilogy(nn,traceKsim,nn,traceK,'r');
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)...
0.0001 1]);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Squared Error Value');
```

Viewing the plots in this figure you see the various error values plotted in both simulation and theory. Each subplot reveals more information about the results as the simulation converges with the theoretical performance.



See Also

`filter`, `maxstep`, `msesim`

Purpose

Measured mean-squared error for adaptive filter

Syntax

```
mse = msesim(ha,x,d)
[mse,meanw,w,tracek] = msesim(ha,x,d)
[mse,meanw,w,tracek] = msesim(ha,x,d,m)
```

Description

`mse = msesim(ha,x,d)` returns the sequence of mean-square errors in column vector `mse`. The vector contains estimates of the mean-square error of the adaptive filter at each time instant during adaptation. The length of `mse` is equal to `size(x,1)`. The columns of matrix `x` contain individual input signal sequences, and the columns of the matrix `d` contain corresponding desired response signal sequences.

`[mse,meanw,w,tracek] = msesim(ha,x,d)` calculates three parameters that correspond to the simulated behavior of the adaptive filter defined by `ha`:

- `meanw` — sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the LMS adaptive filter coefficients at each time instant. The dimensions of `meanw` are (`size(x,1)`)-by-(`ha.length`).
- `w` — estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `ha`.
- `tracek` — sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the LMS adaptive filter at each time instant. The length of `tracek` is equal to `size(X,1)`.

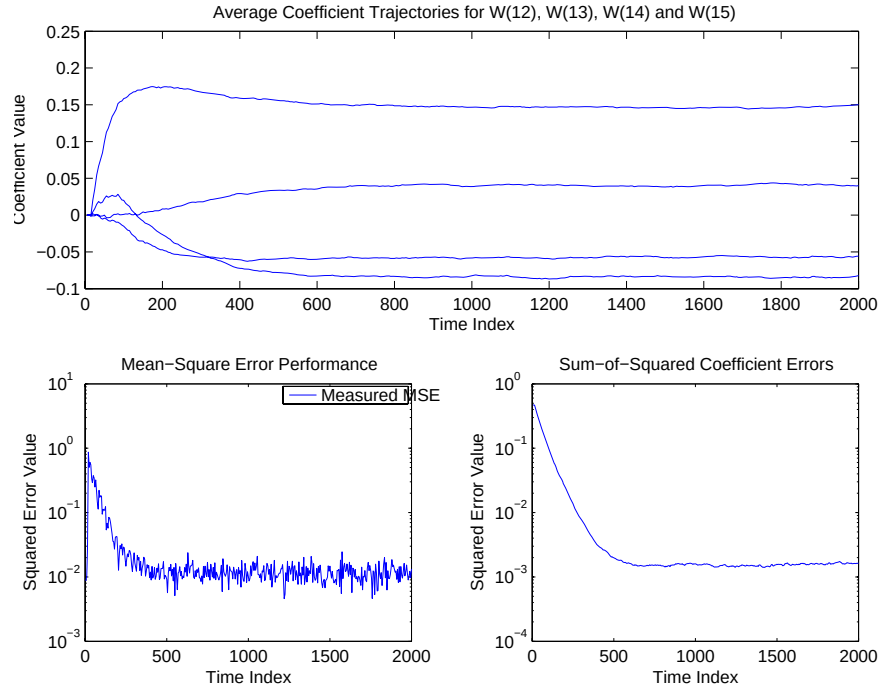
`[mse,meanw,w,tracek] = msesim(ha,x,d,m)` specifies an optional input argument `m` that is the decimation factor for computing `meanw`, `mse`, and `tracek`. When `m` > 1, `msepsim` saves every `m`th predicted value of each of these sequences. When you omit the optional argument `m`, it defaults to one.

Examples

Simulation of a 32-coefficient FIR filter using 25 trials, each trial having 2000 iterations of the adaptation process.

```
x = zeros(2000,25); d = x;          % Initialize variables
ha = fir1(31,0.5);                  % FIR system to be identified
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));             % Observation noise signal
d = filter(ha,1,x)+n;               % Desired signal
l = 32;                             % Filter length
mu = 0.008;                         % LMS Step size.
m  = 5;                             % Decimation factor for analysis
                                    % and simulation results
ha = adaptfilt.lms(l,mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanWsim(:,13:15),'b');
title('Average Coefficient Trajectories for W(12), W(13),...
W(14) and W(15)');
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse);
title('Mean-Square Error Performance'); axis([0 size(x,1) 0.001...
10]);
legend('Measured MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim);
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)...
0.0001 1]);
xlabel('Time Index'); ylabel('Squared Error Value');
```

Calculating the mean squared error for an adaptive filter is one measure of the performance of the adapting algorithm. In this figure, you see a variety of measures of the filter, including the error values.



See Also

`filter`, `msepred`

Purpose

Multistage filter from specification object

Syntax

```
hd = design(d,'multistage')
hd = design(...,'filterstructure',structure)
hd = design(...,'nstages',nstages)
hd = design(...,'usehalfbands',hb)
```

Description

`hd = design(d,'multistage')` designs a multistage filter whose response you specified by the filter specification object `d`.

`hd = design(...,'filterstructure',structure)` returns a filter with the structure specified by `structure`. Input argument `structure` is `dffir` by default and can also be one of the following strings.

structure String	Valid with These Responses
`firdecim`	Lowpass or Nyquist response
`firtdecim`	Lowpass or Nyquist response
`firinterp`	Lowpass or Nyquist response
`lowpass`	Default lowpass only

Multistage design applies to the default lowpass filter specification object and to decimators and interpolators that use either lowpass or Nyquist responses.

`hd = design(...,'nstages',nstages)` specifies `nstages`, the number of stages to be used in the design. `nstages` must be an integer or the string `auto`. To allow the design algorithm to use the optimal number of stages while minimizing the cost of using the resulting filter, `nstages` is `auto` by default. When you specify an integer for `nstages`, the design algorithm minimizes the cost for the number of stages you specify.

`hd = design(...,'usehalfbands',hb)` uses halfband filters when you set `hb` to `true`. The default value for `hb` is `false`.

Note To see a list of the design methods available for your filter, use `designmethods(hd)`.

Examples

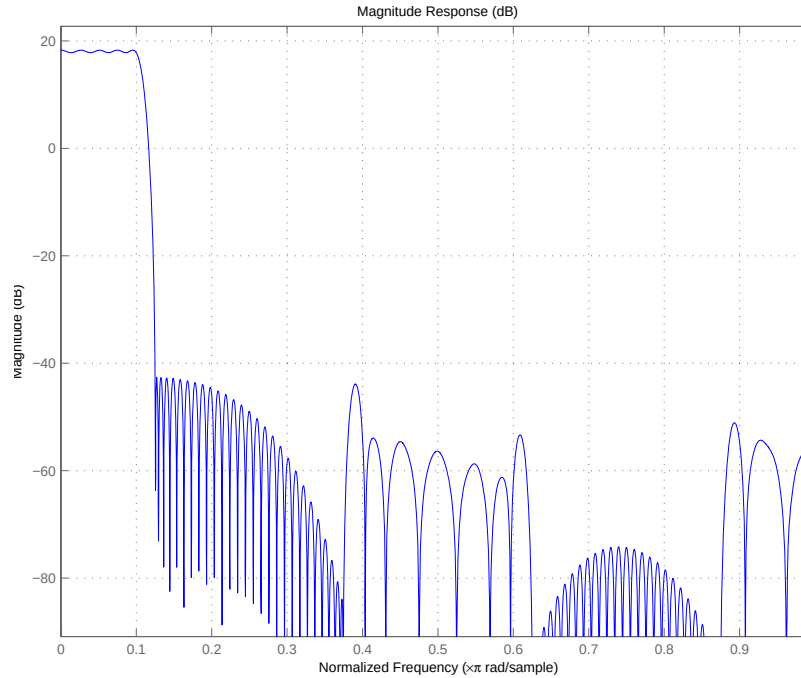
Design a minimum-order, multistage Nyquist interpolator. Use the `FilterStructure` property to specify the Nyquist response.

```
l = 15;   % Interpolation factor. Also the Nyquist band.
tw = 0.05; % Normalized transition width
ast = 40;   % Minimum stopband attenuation in dB
d = fdesign.interpolator(l,'filterstructure','nyquist',l,tw,ast);
hm = design(d,'multistage');
fvtool(hm);
```

Design a multistage lowpass interpolator with an interpolation factor of 8.

```
m = 8; % Interpolation factor;
d = fdesign.interpolator(m,'lowpass');
hm = design(d,'multistage','Usehalfbands',true); % Use halfband filters
                                                 % if possible.
fvtool(hm);
```

This figure shows the response for `hm`.



See Also

`design`, `designopts`

Purpose

Power spectral density of filter output

Syntax

```
hpsd = noisepsd(hd,l)
hpsd = noisepsd(hd,l,p1,v1,p2,v2,...)
noisepsd(hd,l,opts)
```

Description

`hpsd = noisepsd(hd,l)` computes the power spectral density (PSD) at the output of filter `hd` due to roundoff noise produced by quantization errors within the filter. `l` is the number of trials used to compute the average. The PSD is computed from the average over the `l` trials. The more trials you specify, the better the estimate, but at the expense of longer computation time. When you do not explicitly set `l`, it defaults to 10 trials.

`hpsd` is a `psd` data object. To extract the PSD vector (the data from the PSD) from `hpsd`, enter

```
get(hpsd,'data')
```

at the prompt. Plot the PSD data with `plot(hpsd)`. The average power of the output noise (the integral of the PSD) can be computed with `avgpower`, a method of `dspdata` objects:

```
avgpwr = avgpower(hpsd).
```

`hpsd = noisepsd(hd,l,p1,v1,p2,v2,...)` specifies optional parameters via propertyname/propertyvalue pairs. The properties of the `psd` object, and the valid entries are:

Property Name	Default Value	Description and Valid Entries
`Nfft`	512	Specifies the number of FFT points to use to calculate the PSD.

Property Name	Default Value	Description and Valid Entries
`NormalizedFrequency`	`true`	Determines whether to use normalized frequency. Enter one of the logical `true` or `false`. Note that you do not use single quotations around this property value because it is a logical, not a string.
`Fs`	`normalized`	Specifies the sampling frequency to use when you set `NormalizedFrequency` to `false`. Any integer value greater than 1 works. Enter the value in Hz.

Property Name	Default Value	Description and Valid Entries
`SpectrumType`	`onesided`	Tells `noisepsd` whether to generate a one-sided PSD or two-sided. Options are `onesided` or `twosided`. If you choose a two-sided computation, you can also choose `centerdc = true`. Otherwise, `centerdc` must be `false`. • `onesided` converts the spectrum to a spectrum calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically. • `twosided` converts the spectrum to a spectrum calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.
`CenterDC`	`false`	Shifts the zero-frequency component to the center of a two-sided spectrum. • When you set `SpectrumType` to `onesided`, it is changed to `twosided` and the data is converted to a two-sided spectrum. • Setting `CenterDC` to `false` shifts the data and the frequency values in the object so that DC is in the left edge of the spectrum. This operation does not effect the `SpectrumType` property setting.

Note If the spectrum data you specify is calculated over half the Nyquist interval and you do not specify a corresponding frequency vector, the default frequency vector assumes that the number of points in the whole FFT was even. Also, the plot option to convert to a whole or two-sided spectrum assumes the original whole FFT length was even.

`noisepsd(hd,l,opts)` uses an options object `opts` to specify the optional input arguments instead of specifying property-value pairs in the command. Use `opts = noisepsdopts(hd)` to create the object. `opts` then has the `noisepsd` settings from `hd`. After creating `opts`, you change the property values before calling `noisepsd`:

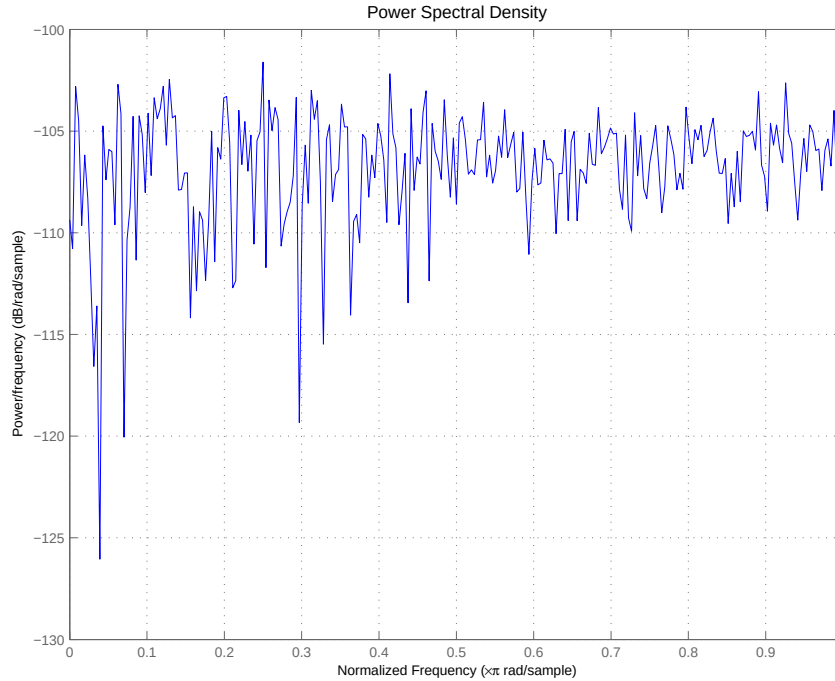
```
set(opts,'fs',48e3); % Set Fs to 48 kHz.
```

Examples

Compute the PSD of the output noise caused by the quantization processes in a fixed-point, direct form FIR filter.

```
b = firgr(27,[0 .4 .6 1],[1 1 0 0]);
h = dfilt.dffir(b); % Create the filter object.
% Quantize the filter to fixed-point.
h.arithmetic = 'fixed';
hpsd = noisepsd(h);
plot(hpsd)
```

`hpsd` looks similar to the following figure—the data resulting from the noise PSD calculation. You can review the data in `hpsd.data'`.



Here is the specification for hpsd.

```
hpsd =

                    Name: 'Power Spectral Density'
                    Data: [257x1 double]
           SpectrumType: 'Onesided'
             Frequencies: [257x1 double]
    NormalizedFrequency: true
                     Fs: 'Normalized'
```

See Also

`filter`, `noisepsdopts`, `norm`, `reorder`, `scale`

`spectrum.welch` in Signal Processing Toolbox™ documentation

References

McClellan, et al., *Computer-Based Exercises for Signal Processing Using MATLAB 5*, Prentice-Hall, 1998.

Purpose

Options for running filter output noise PSD

Syntax

opts = noisepsdopts(hd)

Description

`opts = noisepsdopts(hd)` uses the current settings in the filter `hd` to create an options object `opts` that contains specified options for computing the output noise PSD for a filter `hd`. You can pass `opts` to the `scale` method as an input argument to apply scaling settings to a second-order filter.

Within `opts`, the `noisepsd` options object returned by `noisepsdopts`, you can set the following properties:

Property Name	Default Value	Description and Valid Entries
`Nfft`	512	Specifies the number of FFT points to use to calculate the PSD.
`NormalizedFrequency`	`true`	Determines whether to use normalized frequency. Enter one of the logical `true` or `false`. Note that you do not use single quotations around this property value because it is a logical value, not a string.
`Fs`	`normalized`	Specifies the sampling frequency to use when you set `NormalizedFrequency` to `false`. Any integer value greater than 1 works. Enter the value in Hz.

Property Name	Default Value	Description and Valid Entries
`SpectrumType`	`onesided`	Tells `noisepsd` whether to generate a one-sided PSD or two-sided. Options are `onesided` or `twosided`. If you choose a two-sided computation, you can also choose `centerdc = true`. Otherwise, `centerdc` must be `false`. • `onesided` converts the spectrum to a spectrum calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically. • `twosided` converts the spectrum to a spectrum calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.
`CenterDC`	`false`	Shifts the zero-frequency component to the center of a two-sided spectrum. • When you set `SpectrumType` to `onesided`, it is changed to `twosided` and the data is converted to a two-sided spectrum. • Setting `CenterDC` to `false` shifts the data and the frequency values in the object so that DC is in the left edge of the spectrum. This operation does not effect the `SpectrumType` property setting.

See Also

noisepsd

Purpose

P-norm of filter

Syntax

```
l = norm(ha)
l = norm(ha,pnorm)
l = norm(hd)
l = norm(hd,pnorm)
l = norm(hm)
l = norm(hm,pnorm)
```

Description

All of the variants of `norm` return the filter p-norm for the object in the syntax, either an adaptive filter, a digital filter, or a multirate filter. When you omit the `pnorm` argument, `norm` returns the L2-norm for the object.

Note that by Parseval's theorem, the L2-norm of a filter is equal to the l2 norm. This equality is not true for the other norm variants.

For adaptfilt Objects

`l = norm(ha)` returns the L2-norm of an adaptive filter.`l = norm(ha,pnorm)` adds the input argument `pnorm` to let you specify the norm returned. `pnorm` can be either

- Frequency-domain norms specified by one of `L1`, `L2`, or `Linf`
- Discrete-time domain norms specified by one of `l1`, `l2`, or `linf`

For dfilt Objects

`l = norm(hd)` returns the L2-norm of a discrete-time filter.

`l = norm(hd,pnorm)` includes input argument `pnorm` that lets you specify the norm returned. `pnorm` can be either

- Frequency-domain norms specified by one of `L1`, `L2`, or `Linf`
- Discrete-time domain norms specified by one of `l1`, `l2`, or `linf`

By Parseval's theorem, the L2-norm of a filter is equal to the l2 norm. This equality is not true for the other norm variants.

IIR filters respond slightly differently to `norm`. When you compute the l2, linf, L1, and L2 norms for an IIR filter, `norm(...,L2,tol)` lets you specify the tolerance for the accuracy in the computation. For l1, l2, L2, and linf, `norm` uses the tolerance to truncate the infinite impulse response that it uses to calculate the norm. For L1, `norm` passes the tolerance to the numerical integration algorithm. Refer to Examples to see this in use. You cannot specify `Linf` for the norm and include the `tol` option.

For mfilt Objects

`l = norm(hm)` returns the L2-norm of a multirate filter.

`l = norm(hm,pnorm)` includes argument `pnorm` to let you specify the norm returned. `pnorm` can be either

- Frequency-domain norms specified by one of `L1`, `L2`, or `Linf`
- Discrete-time domain norms specified by one of `l1`, `l2`, or `linf`

Note that, by Parseval's theorem, the L2-norm of a filter is equal to the l2 norm. This equality is not true for the other norm variants.

Examples

Adaptfilt Objects

For the adaptive filter example, compute the 2-norm of an `adaptfilt` object, here an LMS-based adaptive filter.

```
ha = adaptfilt.lms; % norm(ha) is zero because all coeffs are zero
% Create some data to filter to generate filter coeffs
x = randn(100,1);
d = x + randn(100,1);
[y,e] = filter(ha,x,d);
l2 = norm(ha); % Now norm(ha) is nonzero
l2 =

    1.1231
```

Dfilt Objects

To demonstrate the tolerance option used with an IIR filter (`dfilt` object), compute the 2-norm of filter `hd` with a tolerance of 1e-10.

```
d=fdesign.lowpass('n,fc',5,0.4)

d =

                Response: 'Lowpass with cutoff'
           Specification: 'N,Fc'
             Description: {2x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
             FilterOrder: 5
                 Fcutoff: 0.4000


hd = butter(d);
l2=norm(hd,'l2',1e-10)

l2 =

    0.6336
```

Mfilt Objects

In this example, compute the infinity norm of an FIR interpolator, which is an `mfilt` object.

```
hm = mfilt.firinterp;
linf = norm(hm,inf);
linf =

    2.0002
```

See Also

`reorder`, `scale`, `scalecheck`

Purpose

Normalize filter numerator or feed-forward coefficients

Syntax

```
normalize(hq)
g = normalize(hd)
```

Description

`normalize(hq)` normalizes the filter numerator coefficients for a quantized filter to have values between -1 and 1. The coefficients of `hq` change — `normalize` does not copy `hq` and return the copy. To restore the coefficients of `hq` to the original values, use `denormalize`.

Note that for lattice filters, the feed-forward coefficients stored in the property `lattice` are normalized.

`g = normalize(hd)` normalizes the numerator coefficients for the filter `hq` to between -1 and 1 and returns the gain `g` due to the normalization operation. Calling `normalize` again does not change the coefficients. `g` always returns the gain returned by the first call to normalize the filter.

Examples

Create a direct form II quantized filter that uses second-order sections. Then use `normalize` to maximize the use of the range of representable coefficients.

```
d=fdesign.lowpass('n,fp,ap,ast',8,.5,2,40);

hd=ellip(d);

hd =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [4x6 double]
             ScaleValues: [5x1 double]
    PersistentMemory: 'on'
                  States: [2x4 double]

hd.arithmetic='fixed'

hd =
```

```
     FilterStructure: 'Direct-Form II, Second-Order Sections'
          Arithmetic: 'fixed'
           sosMatrix: [4x6 double]
         ScaleValues: [5x1 double]
 PersistentMemory: 'on'
              States: [1x1 embedded.fi]

     CoeffWordLength: 16
      CoeffAutoScale: true
              Signed: true

     InputWordLength: 16
     InputFracLength: 15

 StageInputWordLength: 16
  StageInputAutoScale: true

StageOutputWordLength: 16
 StageOutputAutoScale: true

    OutputWordLength: 16
          OutputMode: 'AvoidOverflow'

     StateWordLength: 16
     StateFracLength: 15

         ProductMode: 'FullPrecision'

           AccumMode: 'KeepMSB'
     AccumWordLength: 40
       CastBeforeSum: true

           RoundMode: 'convergent'
        OverflowMode: 'wrap'

     InheritSettings: false
```

Check the filter coefficients to see that some of them are greater than 1.

```
hd.sosMatrix

ans =

    1.0000    1.5132    1.0000    1.0000   -0.9207    0.4373
    1.0000    0.3867    1.0000    1.0000   -0.2779    0.8242
    1.0000    0.0929    1.0000    1.0000   -0.0514    0.9610
    1.0000    0.0339    1.0000    1.0000   -0.0020    0.9934
```

Use `normalize` to modify the coefficients into the range between -1 and 1. A quick check of the SOS matrix shows all of the numerator coefficients now within the limits. You see that `g` contains the gains applied to each section of the SOS filter.

```
g = normalize(hd)

g =

    1.5132
    1.0000
    1.0000
    1.0000

hd.sosMatrix

ans =

    0.6608    1.0000    0.6608    1.0000   -0.9207    0.4373
    1.0000    0.3867    1.0000    1.0000   -0.2779    0.8242
    1.0000    0.0929    1.0000    1.0000   -0.0514    0.9610
    1.0000    0.0339    1.0000    1.0000   -0.0020    0.9934
```

None of the numerator coefficients exceed -1 or 1.

See Also

`denormalize`

normalizefreq

Purpose

Switch filter specification between normalized frequency and absolute frequency

Syntax

```
normalizefreq(d)
normalizefreq(d,flag)
normalizefreq(d,false,fs)
```

Description

`normalizefreq(d)` normalizes the frequency specifications in filter specifications object `d`. By default, the `NormalizedFrequency` property is set to `true` when you create a design object. You provide the design specifications in normalized frequency units. `normalizefreq` does not affect filters that already use normalized frequency.

If you use this syntax when `d` does not use normalized frequency specifications, all of the frequency specifications are normalized by `fs`/2 so they lie between 0 and 1, where `fs` is specified in the object. Included in the normalization are the filter properties that define the filter pass and stopband edge locations by frequency:

- `F3 dB` — Used by IIR filter specifications objects to describe the passband cutoff frequency
- `Fcutoff` — Used by FIR filter specifications objects to describe the passband cutoff frequency
- `Fpass` — Describes the passband edges
- `Fstop` — Describes the stopband edges

In this syntax, `normalizefreq(d)` assumes you specified `fs` when you created `d` or changed `d` to use absolute frequency specifications.

`normalizefreq(d,flag)` where `flag` is either **`true`** or **`false`**, specifies whether the `NormalizedFrequency` property value is `true` or `false` and therefore whether the filter normalizes the sampling frequency `fs` and other related frequency specifications. `fs` defaults to 1 for this syntax.

When you do not provide the input argument `flag`, it defaults to `true`. If you set `flag` to `false`, affected frequency specifications are multiplied by `fs`/2 to remove the normalization. Use this syntax to switch your

filter between using normalized frequency specifications and not using normalized frequency specifications.

`normalizefreq(d,false,fs)` lets you specify a new sampling frequency `fs` when you set the `NormalizedFrequency` property to `false`.

Examples

These examples demonstrate using `normalizefreq` in both of the major syntax applications—setting the design object frequency specifications to use absolute frequency (normalizefreq(hd,**false**,fs)) and resetting a design object to using normalized frequencies (`normalizefreq(d)`).

Construct a highpass filter specifications object by specifying the passband and stopband edges and the desired attenuations in the bands. By default, provide the frequency specifications in normalized values between 0 and 1.

```
d=fdesign.highpass(0.35, 0.45, 60, 40)

d =

                Response: 'Highpass'
           Specification: 'Fst,Fp,Ast,Ap'
             Description: {4x1 cell}
    NormalizedFrequency: true
                   Fstop: 0.35
                   Fpass: 0.45
                   Astop: 60
                   Apass: 40
```

`Fstop` and `Fpass` are in normalized form, and the property `NormalizedFrequency` is `true`.

Now use `normalizedfreq` to convert to absolute frequency specifications, with a sampling frequency of 1000 Hz.

```
normalizefreq(d,false,1e3)
d

d =
```

```
                Response: 'Highpass'
           Specification: 'Fst,Fp,Ast,Ap'
             Description: {4x1 cell}
       NormalizedFrequency: false
                      Fs: 1000
                   Fstop: 175
                   Fpass: 225
                   Astop: 60
                   Apass: 40
```

Both of the attenuation specifications remain the same. The passband and stopband edge definitions now appear in Hz, where the new value represents the normalized values multiplied by `Fs`/2, or 500 Hz.

Converting to using normalized frequencies consists of using `normalizefreq` with the design object `d`.

```
normalizefreq(d)
d

d =

                Response: 'Highpass'
           Specification: 'Fst,Fp,Ast,Ap'
             Description: {4x1 cell}
       NormalizedFrequency: true
                   Fstop: 0.35
                   Fpass: 0.45
                   Astop: 60
                   Apass: 40
```

For bandstop, bandpass, and multiple band filter specifications objects, `normalizefreq` works the same way for all band edge definitions. When you do not provide the sampling frequency `Fs` as an input argument and you are converting to absolute frequency specifications, `normalizefreq` sets `Fs` to 1, as shown in this example.

```
d=fdesign.bandstop(0.25,0.35,0.55,0.65,50,60)

d =

               Response: 'Bandstop'
          Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
            Description: {7x1 cell}
    NormalizedFrequency: true
                  Fpass1: 0.25
                  Fstop1: 0.35
                  Fstop2: 0.55
                  Fpass2: 0.65
                  Apass1: 50
                   Astop: 60
                  Apass2: 50

normalizefreq(d,false)
d

d =

               Response: 'Bandstop'
          Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
            Description: {7x1 cell}
    NormalizedFrequency: false
                     Fs: 1
                  Fpass1: 0.125
                  Fstop1: 0.175
                  Fstop2: 0.275
                  Fpass2: 0.325
                  Apass1: 50
                   Astop: 60
                  Apass2: 50
```

See Also

fdesign.lowpass, fdesign.halfband, fdesign.highpass, fdesign.interpolator

Purpose

Number of filter states

Syntax

```
n = nstates(hd)
n = nstates(hm)
```

Description

Discrete-Time Filters

`n = nstates(hd)` returns the number of states `n` in the discrete-time filter `hd`. The number of states depends on the filter structure and the coefficients.

Multirate Filters

`n = nstates(hm)` returns the number of states `n` in the multirate filter `hm`. The number of states depends on the filter structure and the coefficients.

Examples

Check the number of states for two different filters, one a direct form FIR filter, the other a multirate filter.

```
h=firls(30,[0 .1 .2 .5]*2,[1 1 0 0])

hd=dfilt.dffir(h)

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'double'
               Numerator: [1x31 double]
    PersistentMemory: 'on'
                  States: [30x1 double]

n=nstates(hd)

n =

    30
```

```
hm=mfilt.firfracdecim(2,3)

hm =

         FilterStructure: [1x46 char]
               Numerator: [1x72 double]
       RateChangeFactors: [2 3]
        PersistentMemory: false
                  States: [35x1 double]

n=nstates(hm)

n =

    35
```

See Also

mfilt

order

Purpose

Order of fixed-point filter

Syntax

```
n = order(hq)
```

Description

`n = order(hq)` returns the order `n` of the quantized filter `hq`. When `hq` is a single-section filter, `n` is the number of delays required for a minimum realization of the filter.

When `hq` has more than one section, `n` is the number of delays required for a minimum realization of the overall filter.

Examples

Create a discrete-time filter. Quantize the filter and convert to second-order section form. Then use `order` to check the order of the filter.

```
[b,a] = ellip(4,3,20,.6); % Create the reference filter.
hq = dfilt.df2(b,a);
% Quantize the filter and convert to second-order sections.
set(hq,'arithmetic','fixed');

n=order(hq) % Check the order of the overall filter.
n = 4
```

Purpose

Multirate parallel filter structure

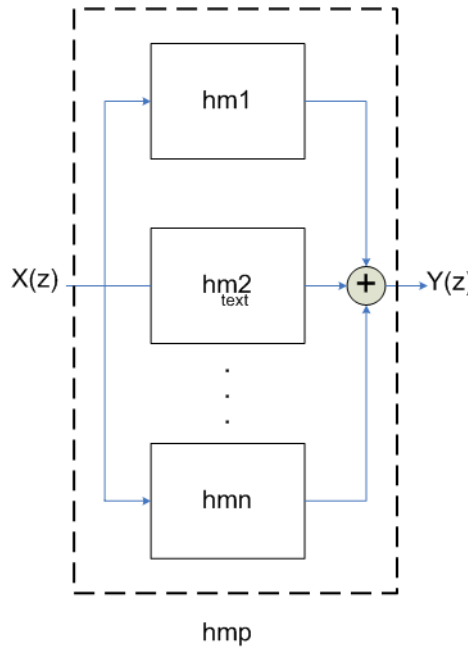
Syntax

```
hmp = parallel(hm1,hm2,...,hmn)
```

Description

`hmp = parallel(hm1,hm2,...,hmn)` returns a multirate filter `hmp` that is two or more `mfilt` objects `hm1`, `hm2`, and so on connected in a parallel structure. Each filter in the structure is one stage and all stages must have the same rate change factor.

Access the individual filters in the parallel structure by



See Also

`dfilt.parallel`, `mfilt`

phasedelay

Purpose

Phase delay of filter

Syntax

```
phasedelay(hd)
[phi,w]=phasedelay(hd,n)
[phi,w]=phasedelay(...,f)
phasedelay(hm)
[phi,w] = phasedelay(hm,n)
[phi,w] = phasedelay(...,f)
[phi,w] = phasedelay(...,fs)
```

Description

The following sections describe `phasedelay` operation for discrete-time filters and multirate filters. For more information about optional input arguments for `phasedelay`, refer to `phasez` in Signal Processing Toolbox™ documentation.

Discrete-Time Filters

`phasedelay(hd)` displays the phase delay response of `hd` in the Filter Visualization Tool (FVTool).

`[phi,w]=phasedelay(hd,n)` returns vectors `phi` and `w` containing the instantaneous phase delay response of the adaptive filter `hd`, and the frequencies in radians at which it is evaluated. The response is evaluated at `n` points equally spaced around the upper half of the unit circle. When you do not specify `n`, it defaults to 8192.

If `hd` is a vector of filter objects, `phasedelay` returns `phi` as a matrix. Each column of `phi` corresponds to one filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `phi` corresponds to each filter in the vector. You can provide `fs`, the sampling frequency, as an input as well. `phasedelay` uses `fs` to calculate the delay response and plots the response to `fs`/2.

Multirate Filters

`phasedelay(hm)` displays the phase response of `hm` in the Filter Visualization Tool (FVTool).

`[phi,w]=phasedelay(hm,n)` returns vectors `phi` and `w` containing the instantaneous phase delay response of the adaptive filter `hm`, and

the frequencies in radians at which it is evaluated. The response is evaluated at `n` points equally spaced around the upper half of the unit circle. When you do not specify `n`, it defaults to 8192.

If `hm` is a vector of filter objects, `phasedelay` returns `phi` as a matrix. Each column of `phi` corresponds to one filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `phi` corresponds to each filter in the vector.

Note that the multirate filter delay response is computed relative to the rate at which the filter is running. When you specify `fs` (the sampling rate) as an input argument, `phasedelay` assumes the filter is running at that rate.

For multistage cascades, `phasedelay` forms a single-stage multirate filter that is equivalent to the cascade and computes the response relative to the rate at which the equivalent filter is running. `phasedelay` does not support all multistage cascades. Only cascades for which it is possible to derive an equivalent single-stage filter are allowed for analysis.

As an example, consider a 2-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. An equivalent single-stage filter with an overall interpolation factor of 8 can be found. `phasedelay` uses the equivalent filter for the analysis. If a sampling frequency `fs` is specified as an input argument to `phasedelay`, the function interprets `fs` as the rate at which the equivalent filter is running.

See Also

`freqz`, `grpdelay`, `phasez`, `zerophase`, `zplane`

`freqz`, `fvtool`, `phasez`, `zerophase` in Signal Processing Toolbox documentation

phasez

Purpose

Unwrapped phase response for filter

Syntax

```
phasez(ha)
[phi,w] = phasez(ha,n)
[phi,w] = phasez(...,f)
phasez(hd)
[phi,w] = phasez(hd,n)
[phi,w] = phasez(...,f)phasez(hm)
[phi,w] = phasez(hm,n)
[phi,w] = phasez(...,f)
[phi,w] = phasez(...,fs)
```

Description

The following sections describe `phasez` operation for adaptive filters, discrete-time filters, and multirate filters. For more information about optional input arguments for `phasez`, refer to `phasez` in Signal Processing Toolbox™ documentation.

Adaptive Filters

For adaptive filters, `phasez` returns the instantaneous unwrapped phase response based on the current filter coefficients.

`phasez(ha)` displays the phase response of `ha` in the Filter Visualization Tool (FVTool).

`[phi,w]=phasez(ha,n)` returns vectors `phi` and `w` containing the instantaneous phase response of the adaptive filter `ha`, and the frequencies in radians at which it is evaluated. The phase response is evaluated at `n` points equally spaced around the upper half of the unit circle. When you do not specify `n`, it defaults to 8192.

If `ha` is a vector of filter objects, `phasez` returns `phi` as a matrix. Each column of `phi` corresponds to one filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `phi` corresponds to each filter in the vector.

Discrete-Time Filters

`phasez(hd)` displays the phase response of `hd` in the Filter Visualization Tool (FVTool).

`[phi,w]=phasez(hd,n)` returns vectors `phi` and `w` containing the instantaneous phase response of the adaptive filter `hd`, and the frequencies in radians at which it is evaluated. The phase response is evaluated at `n` points equally spaced around the upper half of the unit circle. When you do not specify `n`, it defaults to 8192.

If `hd` is a vector of filter objects, `phasez` returns `phi` as a matrix. Each column of `phi` corresponds to one filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `phi` corresponds to each filter in the vector.

Multirate Filters

`phasez(hm)` displays the phase response of `hm` in the Filter Visualization Tool (FVTool).

`[phi,w]=phasez(hm,n)` returns vectors `phi` and `w` containing the instantaneous phase response of the adaptive filter `hm`, and the frequencies in radians at which it is evaluated. The phase response is evaluated at `n` points equally spaced around the upper half of the unit circle. When you do not specify `n`, it defaults to 8192.

If `hm` is a vector of filter objects, `phasez` returns `phi` as a matrix. Each column of `phi` corresponds to one filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `phi` corresponds to each filter in the vector.

Note that the multirate filter response is computed relative to the rate at which the filter is running. When you specify `fs` (the sampling rate) as an input argument, `phasez` assumes the filter is running at that rate.

For multistage cascades, `phasez` forms a single-stage multirate filter that is equivalent to the cascade and computes the response relative to the rate at which the equivalent filter is running. `phasez` does not support all multistage cascades. Only cascades for which it is possible to derive an equivalent single-stage filter are allowed for analysis.

As an example, consider a 2-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. An equivalent single-stage filter with an overall interpolation factor of 8 can be found. `phasez` uses the

equivalent filter for the analysis. If a sampling frequency `fs` is specified as an input argument to `phasez`, the function interprets `fs` as the rate at which the equivalent filter is running.

See Also

`freqz`, `grpdelay`, `phasedelay`, `zerophase`, `zplane`

`freqz`, `fvtool`, `phasez` in Signal Processing Toolbox documentation

Purpose

Polyphase decomposition of multirate filter

Syntax

```
p = polyphase(hm)
polyphase(hm)
```

Description

`p = polyphase(hm)` returns the polyphase matrix `p` of the multirate filter `hm`. Each row in the matrix represents one subfilter of the multirate filter. The first row of matrix `p` represents the first subfilter, the second row the second subfilter, and so on to the last subfilter.

`polyphase(hm)` called with no output argument launches the Filter Visualization Tool (FVTool) with all the polyphase subfilters to allow you to analyze each component subfilter individually.

Examples

When you create a multirate filter that uses polyphase decomposition, polyphase lets you analyze the component filters individually by returning the components as rows in a matrix.

This example creates an interpolate by eight filter.

```
hm=mfilt.firinterp(8)

hm =

         FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
               Numerator: [1x192 double]
     InterpolationFactor: 8
        PersistentMemory: false
                  States: [23x1 double]
```

In this syntax, the matrix `p` contains all of the subfilters for `hm`, one filter per matrix row.

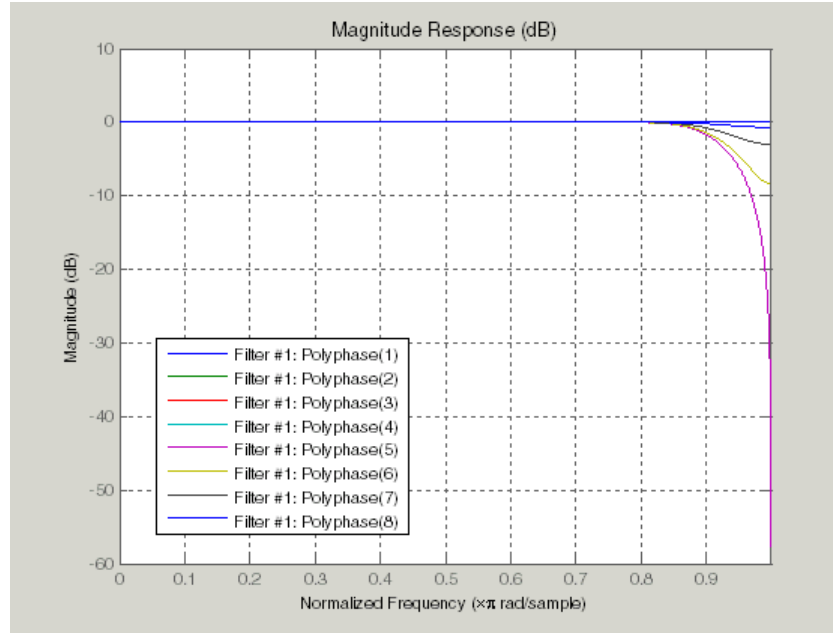
```
p=polyphase(hm)

p =

  Columns 1 through 8
```

```
         0         0         0         0         0         0         0         0
   -0.0000    0.0002   -0.0006    0.0013   -0.0026    0.0048   -0.0081    0.0133
   -0.0001    0.0004   -0.0012    0.0026   -0.0052    0.0094   -0.0160    0.0261
   -0.0001    0.0006   -0.0017    0.0038   -0.0074    0.0132   -0.0223    0.0361
   -0.0002    0.0008   -0.0020    0.0045   -0.0086    0.0153   -0.0257    0.0415
   -0.0002    0.0008   -0.0021    0.0045   -0.0086    0.0151   -0.0252    0.0406
   -0.0002    0.0007   -0.0018    0.0038   -0.0071    0.0124   -0.0205    0.0330
   -0.0001    0.0004   -0.0011    0.0022   -0.0041    0.0072   -0.0118    0.0189

Columns 9 through 16

         0         0         0         0    1.0000         0         0         0
   -0.0212    0.0342   -0.0594    0.1365    0.9741   -0.1048    0.0511   -0.0303
   -0.0416    0.0673   -0.1189    0.2958    0.8989   -0.1730    0.0878   -0.0527
   -0.0576    0.0938   -0.1691    0.4659    0.7814   -0.2038    0.1071   -0.0648
   -0.0661    0.1084   -0.2003    0.6326    0.6326   -0.2003    0.1084   -0.0661
   -0.0648    0.1071   -0.2038    0.7814    0.4659   -0.1691    0.0938   -0.0576
   -0.0527    0.0878   -0.1730    0.8989    0.2958   -0.1189    0.0673   -0.0416
   -0.0303    0.0511   -0.1048    0.9741    0.1365   -0.0594    0.0342   -0.0212

Columns 17 through 24

         0         0         0         0         0         0         0         0
    0.0189   -0.0118    0.0072   -0.0041    0.0022   -0.0011    0.0004   -0.0001
    0.0330   -0.0205    0.0124   -0.0071    0.0038   -0.0018    0.0007   -0.0002
    0.0406   -0.0252    0.0151   -0.0086    0.0045   -0.0021    0.0008   -0.0002
    0.0415   -0.0257    0.0153   -0.0086    0.0045   -0.0020    0.0008   -0.0002
    0.0361   -0.0223    0.0132   -0.0074    0.0038   -0.0017    0.0006   -0.0001
    0.0261   -0.0160    0.0094   -0.0052    0.0026   -0.0012    0.0004   -0.0001
    0.0133   -0.0081    0.0048   -0.0026    0.0013   -0.0006    0.0002   -0.0000
```

Finally, using `polyphase` without an output argument opens the Filter Visualization Tool, ready for you to use the analysis capabilities of the tool to investigate the interpolator `hm`.

```
polyphase(hm)
```

In the following figure, FVTool shows the magnitude responses for the subfilters.



See Also

`mfilt`

Purpose

Most recent fixed-point filtering operation report

Syntax

```
rlog = qreport(h)
```

Description

`rlog = qreport(h)` returns the logging report stored in the filter object `h` in the object `rlog`. The ability to log features of the filtering operation is integrated in the fixed-point filter object and the `filter` method.

Each time you filter a signal with `h`, new log data overwrites the results in the filter from the previous filtering operation. To save the log from a filtering simulation, change the name of the output argument for the operation before subsequent filtering runs.

Note `qreport` requires \&tm_fixedpointtoolbox; software and that filter `h` is a fixed-point filter. Data logging for `fi` operations is a preference you set for each MATLAB session. To learn more about logging, `LoggingMode`, and `fi` object preferences, refer to `fipref` in the documentation for \&tm_fixedpointtoolbox; software in the online Help system.

Also, you cannot use `qreport` to log the filtering operations from a fixed-point Farrow filter.

Enable logging during filtering by setting `LoggingMode` to `on` for `fi` objects for your MATLAB session. Trigger logging by setting the `Arithmetic` property for `h` to `fixed`, making `h` a fixed-point filter and filtering an input signal.

Using Fixed-Point Filtering Logging

Filter operation logging with `qreport` requires some preparation in MATLAB. Complete these steps before you use `qreport`.

1. Set the fixed-point object preference for `LoggingMode` to `on` for your MATLAB session. This setting enables data logging.

```
fipref('LoggingMode','on')
```

2. Create your fixed-point filter.
3. Filter a signal with the filter.
4. Use `qreport` to return the filtering information stored in the filter object.

`qreport` provides a way to instrument your fixed-point filters and the resulting data log offers insight into how the filter responds to a particular input data signal.

Report object `rlog` contains a filter-structure-specific list of internal signals for the filter. Each signal contains

- Minimum and maximum values that were recorded during the last simulation. Minimum and maximum values correspond to values before quantization.
- Representable numerical range of the word length and fraction length format
- Number of overflows during filtering for that signal.

Examples

`qreport` depends on the `LoggingMode` preference for fixed-point objects. This example demonstrates the process for enabling and using `qreport` to log the results of filtering with a fixed-point filter. `hd` is a fixed-point direct-form FIR filter.

```
f = fipref('loggingmode','on');
hd = design(fdesign.lowpass,'equiripple');
hd.arithmetic = 'fixed';
fs = 1000;          % Input sampling frequency.
t = 0:1/fs:1.5;     % Signal length = 1501 samples.
x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.
y = filter(hd,x);
rlog = qreport(hd)
```

```
rlog =

                                    Fixed-Point Report
    ---------------------------------------------------------------------------------------------
                     Min              Max       |              Range              |      Number of Overflows
    ---------------------------------------------------------------------------------------------
           Input:             -1       0.99996948 |             -1       0.99996948 |             15/1501 (1%)
          Output:     -1.0232311        1.0232163 |             -2                2 |              0/1501 (0%)
         Product:    -0.48538208       0.48536727 |           -0.5              0.5 |             0/64543 (0%)
     Accumulator:     -1.0852132        1.0851984 |             -2                2 |             0/63042 (0%)
```

View the logging report of a direct-form II, second-order sections IIR filter the same way. While this example sets `loggingmode` to `on`, you do that only once for a MATLAB session, unless you reset the mode to `off` during the session.

```
fipref('loggingmode','on');
hd = design(fdesign.lowpass,'ellip');
hd.arithmetic = 'fixed';
rand('state',0);
y = filter(hd,rand(100,1));
rlog = qreport(hd)
```

See Also

`dfilt`, `mfilt`

Purpose

Simulink® subsystem block for filter

Syntax

```
realizemdl(hq)
realizemdl(hq,propertyname1,propertyvalue1,...)
```

Description

`realizemdl(hq)` generates a model of filter `hq` in a Simulink subsystem block using sum, gain, and delay blocks from Simulink. The properties and values of `hq` define the resulting subsystem block parameters.

`realizemdl` requires Simulink. To accurately realize models of quantized filters, use Simulink Fixed-Point.

`realizemdl(hq,propertyname1,propertyvalue1,...)` generates the model or `hq` with the associated `propertyname`/`propertyvalue` pairs, and any other values you set in `hq`.

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`*'.
`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.

Property Name	Property Values	Description
`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.

Examples

To demonstrate how `realizemdl` works to create models, these two examples show the default and optional syntaxes in use. Both examples begin from a quantized filter designed by `butter` in Signal Processing Toolbox™ documentation.

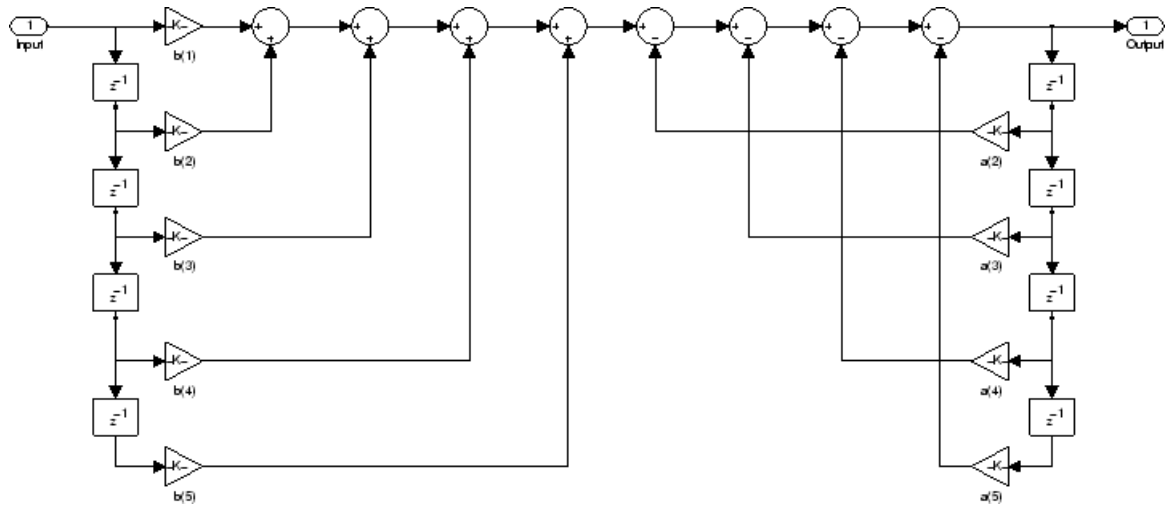
```
[b,a] = butter(4,.5);
hq = dfilt.df1(b,a);
```

Example 1

Using the default syntax to realize a model of your quantized filter `hq`. When you use this syntax, `realizemdl` uses blocks from Simulink and Simulink Fixed-Point to realize the subsystem in your current Simulink model.

```
realizemdl(hq);
```

Look at the figure to see the model as realized by `realizemdl`.



Example 2

Using propertyname/propertyvalue pairs to specify the features of the subsystem block model created by `realizemdl`.

First, convert the filter to fixed-point arithmetic to ensure a few zero valued coefficients:

```
hq.arithmetic = 'fixed';
```

Your filter has two zero value denominators, a(2) and a(4):

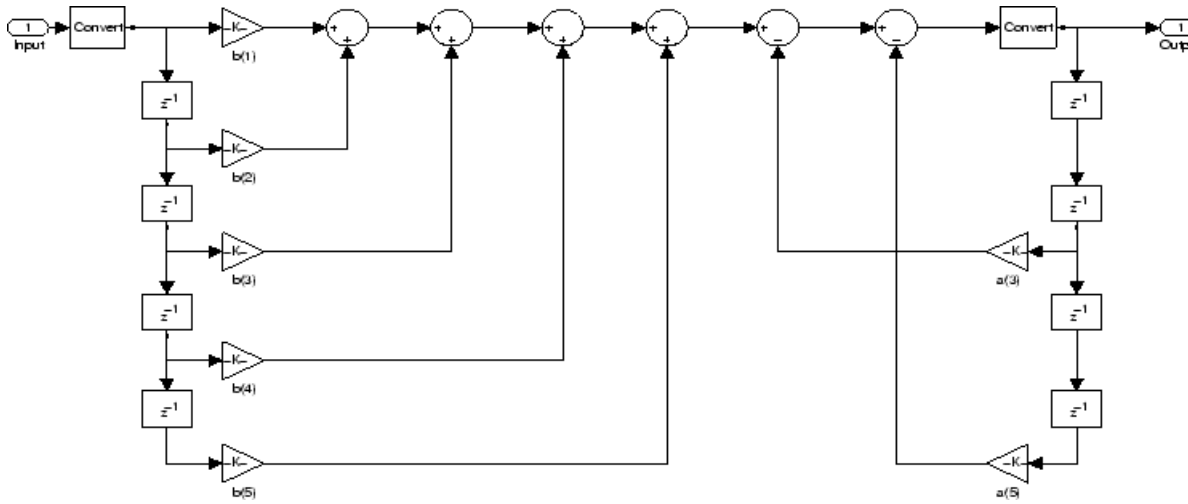
```
 FilterStructure: 'Direct-Form I'
      Arithmetic: 'fixed'
        Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]
    Denominator: [1 0 0.4860 0 0.0176]
PersistentMemory: false
          States: Numerator:  [4x1 fi]
                  Denominator:[4x1 fi]
```

Now realize the model implementation.

```
realizemdl(hq,'optimizezeros','on',...
```

```
'blockname','newfiltermodel');
```

Since this example uses the optional property name `optimizezeros`, set to 'on', the resulting block subsystem is slightly different — the zero-gain blocks for coefficients `a(2)` and `a(4)` are not included in the subsystem.



See Also

`realizemdl` under the methods for `dfilt` in Signal Processing Toolbox documentation

Purpose

Reference filter for fixed-point or single-precision filter

Syntax

```
href = reffilter(hd)
```

Description

`href = reffilter(hd)` returns a new filter `href` that has the same structure as `hd`, but uses the reference coefficients and has its arithmetic property set to `double`. Note that `hd` can be either a fixed-point filter (arithmetic property set to '`fixed`', or a single-precision floating-point filter whose arithmetic property is '`single`').

`reffilter(hd)` differs from `double(hd)` in that

- the filter `href` returned by `reffilter` has the reference coefficients of `hd`.
- `double(hd)` returns the quantized coefficients of `hd` represented in double-precision.

To check the performance of your fixed-point filter, use `href = reffilter(hd)` to quickly have the floating-point, double-precision version of `hd` available for comparison.

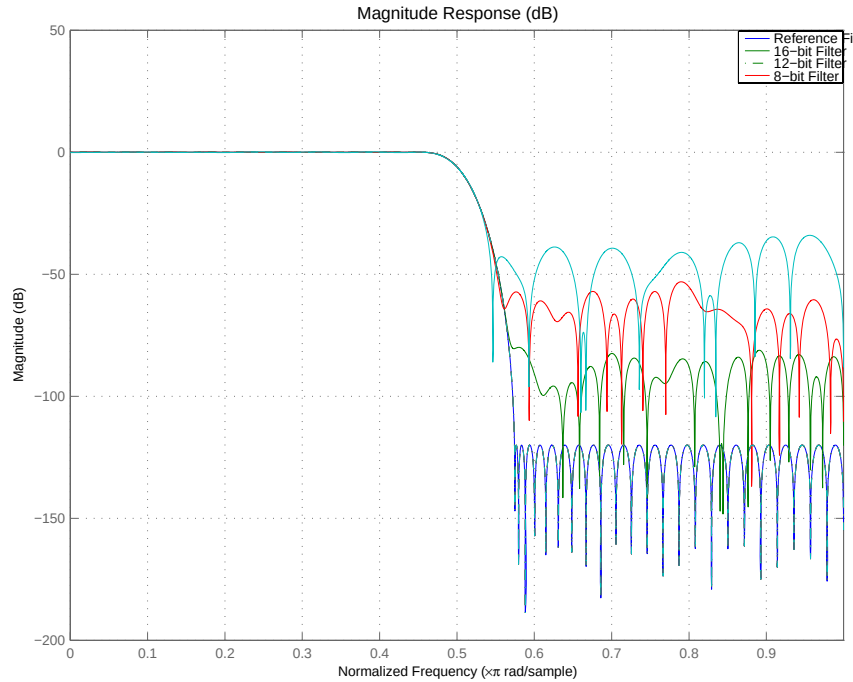
Examples

Compare several fixed-point quantizations of a filter with the same double-precision floating-point version of the filter.

```
h = dfilt.dffir(firceqrip(87,.5,[1e-3,1e-6])); % Lowpass filter.
h1 = copy(h); h2 = copy(h); % Create copies of h.
h.arithmetic = 'fixed';   % Set h to filter using fixed-point...
                          % arithmetic.
h1.arithmetic = 'fixed';  % Same for h1.
h2.arithmetic = 'fixed';  % Same for h2.
h.CoeffWordLength  = 16;  % Use 16 bits to represent the...
                          % coefficients.
h1.CoeffWordLength = 12;  % Use 12 bits to represent the...
                          % coefficients.
h2.CoeffWordLength =  8;  % Use 8 bits to represent the...
                          % coefficients.
href = reffilter(h);
```

```
hfvt = fvtool(href,h,h1,h2);
set(hfvt,'ShowReference','off'); % Reference displayed once
                                 % already.
legend(hfvt,'Reference filter','16-bits','12-bits','8-bits');
```

The following plot, taken from FVTool, shows `href`, the reference filter, and the effects of using three different word lengths to represent the coefficients.



As expected, the fidelity of the fixed-point filters suffers as you change the representation of the coefficients. With `href` available, it is easy to see just how the fixed-point filter compares to the ideal.

See Also

`double`

reorder

Purpose

Rearrange sections in SOS filter

Syntax

```
reorder(hd,order)
reorder(hd,numorder,denorder)
reorder(hd,numorder,denorder,svorder)
reorder(hd,filter_type)
reorder(hd,dir_flag)
reorder(hd,dir_flag,sv)
```

Description

`reorder(hd,order)` rearranges the sections of filter `hd` using the vector of indices provided in `order`.

`order` does not need to contain all of the indices of the filter. Omitting one or more filter section indices removes the omitted sections from the filter. You can use a logical array to remove sections from the filter, but not to reorder it (refer to the Examples to see this done).

`reorder(hd,numorder,denorder)` reorders the numerator and denominator separately using the vectors of indices in `numorder` and `denorder`. These two vectors must be the same length.

`reorder(hd,numorder,denorder,svorder)` the scale values can be independently reordered. When `svorder` is not specified, the scale values are reordered with the numerator. The output scale value always remains on the end when you use the argument `numorder` to reorder the scale values.

`reorder(hd,filter_type)` where `filter_type` is one of `auto`, `lowpass`, `highpass`, `bandpass`, or `bandstop`, reorders `hd` in a way suitable for the filter type you specify by `filter_type`. This reordering mode can be especially helpful for fixed-point implementations where the order of the filter sections can significantly affect your filter performance.

The `auto` option and automatic ordering only apply to filters that you used `fdesign` to create. With the `auto` option as an input argument, `reorder` automatically rearranges the filter sections depending on the specification response type of the design, such as lowpass, or bandstop. This technique appears in the first example.

`reorder(hd,dir_flag)` if `dir_flag` is `up`, the first filter section contains the poles closest to the origin, and the last section contains the poles closest to the unit circle. When `dir_flag` is `down`, the sections are ordered in the opposite direction. `reorder` always pairs zeros with the poles closest to them.

`reorder(hd,dir_flag,sv)` `sv` is either the string `poles` or `zeros` and describes how to reorder the scale values. By default the scale values are not reordered when you use the `dir_flag` option.

Examples

Being able to rearrange the order of the sections in a filter can be a powerful tool for controlling the filter process. This example uses `reorder` to change the sections of a `df2sos` filter. Let `reorder` do the reordering automatically in the first example. In the second, use `reorder` to specify the new order for the sections.

First use the automatic reordering option on a lowpass filter.

```
d = fdesign.lowpass('n,f3db',15,0.75)
hd = design(d,'butter');
d =

               Response: 'Lowpass'
          Specification: 'N,F3dB'
            Description: {'Filter Order';'3dB Frequency'}
    NormalizedFrequency: true
            FilterOrder: 15
                   F3dB: 0.75

reorder(hd,'auto')
hd

hd =

         FilterStructure: 'Direct-Form II,
                           Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [8x6 double]
```

```
            ScaleValues: [9x1 double]
      PersistentMemory: false
```

The SOS matrices show the reordering.

```
hd.sosMatrix

ans =
  1.0000   2.0000   1.0000    1.0000   1.3169   0.8623
  1.0000   2.0000   1.0000    1.0000   1.1606   0.6414
  1.0000   2.0000   1.0000    1.0000   1.0448   0.4776
  1.0000   2.0000   1.0000    1.0000   0.9600   0.3576
  1.0000   2.0000   1.0000    1.0000   0.8996   0.2722
  1.0000   2.0000   1.0000    1.0000   0.8592   0.2151
  1.0000   2.0000   1.0000    1.0000   0.8360   0.1823
  1.0000   1.0000        0    1.0000   0.4142        0

hdreorder.sosMatrix

ans =

  1.0000   2.0000   1.0000   1.0000   1.0448   0.4776
  1.0000   2.0000   1.0000   1.0000   0.8360   0.1823
  1.0000   2.0000   1.0000   1.0000   0.8996   0.2722
  1.0000   2.0000   1.0000   1.0000   1.3169   0.8623
  1.0000   2.0000   1.0000   1.0000   0.9600   0.3576
  1.0000   1.0000        0   1.0000   0.4142        0
  1.0000   2.0000   1.0000   1.0000   0.8592   0.2151
  1.0000   2.0000   1.0000   1.0000   1.1606   0.6414
```

For another example of using reorder, create an SOS filter in the direct form II implementation.

```
[z,p,k] = butter(15,.5);
[sos, g] = zp2sos(z,p,k);
hd = dfilt.df2sos(sos,g);
```

Reorder the sections by moving the second section to be between the seventh and eighth sections.

```
reorder(hd, [1 3:7 2 8]);
hfvt = fvtool(hd, 'analysis', 'coefficients');
```

Remove the third, fourth and seventh sections.

```
hd1 = copy(hd);
reorder(hd1, logical([1 1 0 0 1 1 0 1]));
setfilter(hfvt, hd1);
```

Move the first filter to the end and remove the eighth section

```
hd2 = copy(hd);
reorder(hd2, [2:7 1]);
setfilter(hfvt, hd2);
```

Move the numerator and denominator independently.

```
hd3 = copy(hd);
reorder(hd3, [1 3:8 2], [1:8]);
setfilter(hfvt, hd3);
```

See Also

`cumsec`, `scale`, `scaleopts`

References

Schlichthärle, Dietrich, *Digital Filters Basics and Design*, Springer-Verlag Berlin Heidelberg, 2000.

Purpose

Reset filter properties to initial conditions

Syntax

```
reset(ha)
reset(hd)
reset(hm)
```

Description

`reset(ha)` resets all the properties of the adaptive filter `ha` that are updated when filtering to the value specified at construction. If you do not specify a value for any particular property when you construct an adaptive filter, the property value for that property is reset to the default value for the property.

`reset(hd)` resets all the properties of the discrete-time filter `hd` to their factory values that are modified when you run the filter. In particular, the `States` property is reset to zero.

`reset(hm)` resets all the properties of the multirate filter `hm` to their factory value that are modified when the filter is run. In particular, the `States` property is reset to zero when `hm` is a decimator. Additionally, the filter internal properties are also reset to their factory values.

Examples

Denoise a sinusoid and reset the filter after filtering with it.

```
h = adaptfilt.lms(5,.05,1,[0.5,0.5,0.5,0.5,0.5]);
n = filter(1,[1 1/2 1/3],.2*randn(1,2000));
d = sin((0:1999)*2*pi*0.005) + n; % Noisy sinusoid
x = n;
[y,e]= filter(h,x,d);             % e has denoised signal
disp(h)
reset(h); % Reset the coefficients and states.
disp(h)
```

See Also

`quantizer`, `set`

Purpose

Scale sections of SOS filter

Syntax

```
scale(hd)
scale(hd,pnorm)
scale(hd,pnorm,p1,v1,p2,v2,...)
scale(hd,pnorm,opts)
```

Description

`scale(hd)` scales the second-order section filter `hd` using peak magnitude response scaling (L-infinity, Linf), to reduce the possibility of overflows when your filter `hd` operates in fixed-point arithmetic mode.

`scale(hd,pnorm)` specifies the norm used to scale the filter. `pnorm` can be either a discrete-time-domain norm or a frequency-domain norm.

Valid time-domain norm values for `pnorm` are `l1`, `l2`, and `linf`. Valid frequency-domain norm values are `L1`, `L2`, and `Linf`. Note that `L2` norm is equal to `l2` norm (by Parseval's theorem) but this is not true for other norms — `l1` is not the same as `L1` and `Linf` is not the same as `linf`.

Filter norms can be ordered in terms of how stringent they are, as follows from most stringent to least:

```
l1 >= Linf >= L2 = l2 >= L1 >= linf
```

Using `l1`, the most stringent scaling, produces a filter that is least likely to overflow, but has the worst signal-to-noise ratio performance. `Linf` scaling, the least stringent, and the default scaling, is the most commonly used scaling norm.

`scale(hd,pnorm,p1,v1,p2,v2,...)` uses parameter name/parameter value pair input arguments to specify optional scaling parameters. Valid parameter names and options values appear in the table.

Parameter	Default	Description and Valid Value
`MaxNumerator`	2	Maximum allowed value for numerator coefficients.

Parameter	Default	Description and Valid Value
`MaxScaleValue`	`Not Used`	Maximum allowed scale values. The filter applies the `MaxScaleValue` limit only when you set `ScaleValueConstraint` to a value other than `unit` (the default setting). Setting `MaxScaleValue` to any numerical value automatically changes the `ScaleValueConstraint` setting to `none`.
`NumeratorConstraint`	`none`	Specifies whether and how to constrain numerator coefficient values. Options are `none`, `normalize`, `po2`, and `unit`
`OverflowMode`	`wrap`	Sets the way the filter handles arithmetic overflow situations during scaling. Choose from `wrap`, `saturate` or `satall`.

Parameter	Default	Description and Valid Value
`ScaleValueConstraint`	`unit`	Specify whether to constrain the filter scale values, and how to constrain them. Valid options are `none`, `po2`, and `unit`. Choosing `unit` for the constraint disables the `MaxScaleValue` property setting. `po2` constrains the scale values to be powers of 2, while `none` removes any constraint on the scale values.
`sosReorder`	auto	Reorder filter sections prior to applying scaling. Select one of `auto`, `none`, `up`, or `down`.

If your device does not have guard bits available and you are using saturation arithmetic for filtering, use the `satall` setting for `OverFlowMode` instead of `saturate`.

With the `Arithmetic` property of `hd` set to `double` or `single`, the filter uses the default values for all options that you do not specify explicitly. When you set `Arithmetic` to `fixed`, the values used for the scaling options are set according to the settings in filter `hd`. However, if you specify a scaling option different from the settings in `hd`, the filter uses your explicit option selection for scaling purposes, but does not change the property setting in `hd`.

`scale(hd,pnorm,opts)` uses an input scale options object `opts` to specify the optional scaling parameters in lieu of specifying parameter-value pairs. You can create the `opts` object using

```
opts = scaleopts(hd)
```

For more information about scaling objects, refer to `scaleopts` in the Help system.

Examples

Demonstrate the Linf-norm scaling of a lowpass elliptic filter with second-order sections. Start by creating a lowpass elliptical filter in zero, pole, gain (z,p,k) form.

```
[z,p,k] = ellip(5,1,50,.3);
[sos,g] = zp2sos(z,p,k);
hd = dfilt.df2sos(sos,g);
scale(hd,'linf','scalevalueconstraint','none','maxscalevalue',2)
```

See Also

`cumsec`, `norm`, `reorder`, `scalecheck`, `scaleopts`

Purpose

Check scaling of SOS filter

Syntax

```
s = scalecheck(hd,pnorm)
```

Description

For df1sos and df2tsos Filters

`s = scalecheck(hd,pnorm)` returns a row vector `s` that reports the p-norm of the filter computed from the filter input to the output of each second-order section. Therefore, the number of elements in `s` is one less than the number of sections in the filter. Note that this p-norm computation does not include the trailing scale value of the filter (which you can find by entering

```
hd.scalevalue(end)
```

at the MATLAB prompt.

`pnorm` can be either frequency-domain norms specified by `L1`, `L2`, or `Linf` or discrete-time-domain norms — `l1`, `l2`, `linf`. Note that the L2-norm of a filter is equal to the l2-norm (Parseval's theorem). This is not true for other norms.

For df2sos and df1tsos Filters

`s = scalecheck(hd,pnorm)` returns `s`, a row vector whose elements contain the p-norm from the filter input to the input of the recursive part of each second-order section. This computation of the p-norm corresponds to the input to the multipliers in these filter structures, and are the locations in the signal flow where overflow should be avoided.

When `hd` has nontrivial scale values, that is, if any scale values are not equal to one, `s` is a two-row matrix, rather than a vector. The first row elements of `s` report the p-norm of the filter computed from the filter input to the output of each second-order section. The elements of the second row of `s` contain the p-norm computed from the input of the filter to the input of each scale value between the sections. Note that for `df2sos` and `df1tsos` filter structures, the last numerator and the trailing scale value for the filter are not included when `scalecheck` checks the scale.

For a given p-norm, an optimally scaled filter has partial norms equal to one, so matrix s contain all ones.

Examples

Check the Linf-norm scaling of a filter.

```
% Create filter design specifications
hs = fdesign.lowpass;
object.
hd = ellip(hs);       % Design an elliptic sos filter
scale(hd,'Linf');
s = scalecheck(hd,'Linf')
```

Or, in another form:

```
[b,a]=ellip(10,.5,20,0.5);
[s,g]=tf2sos(b,a);
hd=dfilt.df1sos(s,g)

hd =

         FilterStructure: 'Direct-Form I, Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [5x6 double]
             ScaleValues: [6x1 double]
        PersistentMemory: false
                  States: [1x1 filtstates.dfiir]

1x1 struct array with no fields.

scalecheck(hd,'Linf')

ans =

    0.7631    0.9627    0.9952    0.9994    1.0000
```

See Also

norm, reorder, scale, scaleopts

Purpose

Options for scaling SOS filter

Syntax

```
opts = scaleopts(hd)
```

Description

`opts = scaleopts(hd)` uses the current settings in the filter `hd` to create an options object `opts` that contains specified scaling options for second-order section scaling. You can pass `opts` to the `scale` method as an input argument to apply scaling settings to a second-order filter.

Within `opts`, the scaling options object returned by `scaleopts`, you can set the following properties:

Parameter	Default	Description and Valid Value
`MaxNumerator`	2	Maximum allowed value for numerator coefficients.
`MaxScaleValue`	No default value	Maximum allowed scale values. The filter applies the `MaxScaleValue` limit only when you set `ScaleValueConstraint` to a value other than `unit`. Setting `MaxScaleValue` to a numerical value automatically changes the `ScaleValueConstraint` setting to `none`.
`NumeratorConstraint`	`none`	Specifies whether and how to constrain numerator coefficient values. Options are `none`, `normalize`, `po2`, and `unit`,

Parameter	Default	Description and Valid Value
OverflowMode	wrap	Sets the way the filter handles arithmetic overflow situations during scaling. Choose one ofwrap or saturate or satall.
ScaleValueConstraint	unit	Specify whether to constrain the filter scale values, and how to constrain them. Valid options are none, po2, and unit

When you set the properties of opts and then use opts as an input argument to scale(hd,opts), scale applies the settings in opts to scale hd.

Examples

From a filter hd, you can create an options scaling object that contains the scaling options settings you require.

```
[b,a]=ellip(10,.5,20,0.5);
[s,g]=tf2sos(b,a);
hd=dfilt.df1sos(s,g)
opts=scaleopts(hd)

opts =

             MaxNumerator: 2
     NumeratorConstraint: 'none'
            OverflowMode: 'wrap'
    ScaleValueConstraint: 'unit'
           MaxScaleValue: 'Not used'
```

See Also

cumsec, norm, reorder, scale, scalecheck

Purpose

Configure filter for integer filtering

Syntax

```
set2int(h)
set2int(h,coeffwl)
set2int(...,inwl)
g = set2int(...)
```

Description

These sections apply to both discrete-time (`dfilt`) and multirate (`mfilt`) filters.

`set2int(h)` scales the filter coefficients to integer values and sets the filter coefficient and input fraction lengths to zero.

`set2int(h,coeffwl)` uses the number of bits specified by `coeffwl` as the word length it uses to represent the filter coefficients.

`set2int(...,inwl)` uses the number of bits specified by `coeffwl` as the word length it uses to represent the filter coefficients and the number of bits specified by `inwl` as the word length to represent the input data.

`g = set2int(...)` returns the gain `g` introduced into the filter by scaling the filter coefficients to integers. `g` is always calculated to be a power of 2.

Note `set2int` does not work with CIC decimators or interpolators because they do not have coefficients.

Examples

These examples demonstrate some uses and ideas behind `set2int`.

The second parts of both examples depend on the following — after you filter a set of data, the input data and output data cover the same range of values, unless the filter process introduces gain in the output. Converting your filter object to integer form, and then filtering a set of data, does introduce gain into the system. When the examples refer to resetting the output to the same range as the input, the examples are accounting for this added gain feature.

Discrete-Time Filter Example

Two parts comprise this example. Part 1 compares the step response of an FIR filter in both the fractional and integer filter modes. Fractional mode filtering is essentially the opposite of integer mode. Integer mode uses a filter which has coefficients represented by integers. Fractional mode filters have coefficients represented in fractional form (nonzero fraction length).

```
b = firrcos(100,.25,.25,2,'rolloff','sqrt');
hd = dfilt.dffir(b);
hd.Arithmetic = 'fixed';
hd.InputFracLength = 0; % Integer inputs.
x = ones(100,1);
yfrac = filter(hd,x); % Fractional mode output.
g = set2int(hd);      % Convert to integer coefficients.
yint = filter(hd,x);  % Integer mode output.
```

Note that `yint` and `yfrac` are `fi` objects. Later in this example, you use the `fi` object properties `WordLength` and `FractionLength` to work with the output data.

Now use the gain `g` to rescale the output from the integer mode filter operation.

```
yints = double(yint)/g;
```

Verify that the scaled integer output is equal to the fractional output.

```
 max(abs(yints-double(yfrac)))
```

In part 2 , the example reinterprets the output binary data, putting the input and the output on the same scale by weighting the most significant bits in the input and output data equally.

```
WL = yint.WordLength;
FL = yint.Fractionlength + log2(g);
yints2 = fi(zeros(size(yint)),true,WL,FL);
yints2.bin = yint.bin;
```

```
max(abs(double(yints2)-double(yfrac)))
```

Multirate Filter Example

This two-part example starts by comparing the step response of a multirate filter in both fractional and integer modes. Fractional mode filtering is essentially the opposite of integer mode. Integer mode uses a filter which has coefficients represented by integers. Fractional mode filters have coefficients in fractional form with nonzero fraction lengths.

```
hm = mfilt.firinterp;
hm.Arithmetic = 'fixed';
hm.InputFracLength = 0; % Integer inputs.
x = ones(100,1);
yfrac = filter(hm,x);  % Fractional mode output.
g = set2int(hm);       %Convert to integer coefficients.
yint = filter(hm,x);   % Integer mode output.
```

Note that `yint` and `yfrac` are `fi` objects. In part 2 of this example, you use the `fi` object properties `WordLength` and `FractionLength` to work with the output data.

Now use the gain `g` to rescale the output from the integer mode filter operation.

```
yints = double(yint)/g;
```

Verify that the scaled integer output is equal to the fractional output.

```
max(abs(yints-double(yfrac)))
```

Part 2 demonstrates reinterpreting the output binary data by using the properties of `yint` to create a scaled version of `yint` named `yints2`. This process puts `yint` and `yints2` on the same scale by weighing the most significant bits of each object equally.

```
wl = yint.wordlength;
fl = yint.fractionlength + log2(g);
yints2 = fi(zeros(size(yint)),true,wl,fl);
yints2.bin = yint.bin;
```

```
max(abs(double(yints2)-double(yfrac)))
```

See Also

mfilt

Purpose

Specifications for filter specification object

Syntax

```
setspecs(d,specvalue1,specvalue2,...)
setspecs(d,Specification,specvalue1,specvalue2,...)
setspecs(...fs)
setspecs(...,inputunits)
```

Description

`setspecs(d,specvalue1,specvalue2,...)` sets the specifications in the order that they appear in the `Specification` property for the design object `d`.

`setspecs(d,Specification,specvalue1,specvalue2,...)` lets you change the specifications for the object and set values for the new specifiers. When you already have a filter specifications object, this syntax lets you change the `Specification` string and the associated specification values for the object, rather than recreating the object to change it.

`setspecs(...fs)` sets the `fs`. If you choose to specify the `fs`, it must be immediately after you provide all of the specifications for the current `Specification`. Refer to Examples to see this being used.

`setspecs(...,inputunits)` specifies the *`inputunits`* option allows you to specify your filter magnitude specification values in different units. *`inputunits`* can be either of these strings:

- **`linear`** — to indicate that your input specification values represent linear units, such as decimal values for the filter feature locations when you select normalized sampling frequency.
- **`squared`** — indicating that your input specification values represent squared magnitude values, usually decibels. This is the default value. When you omit the *`inputunits`* argument, `setspecs` assumes all specification values are in square magnitude form.

You are not required to provide `fs`, the sampling frequency, as an input when you use the `inputunits` option. As you see from the syntax options, the `inputunits` option must be the rightmost input argument in the syntax — `inputunits` must be passed as the final input.

Examples

To demonstrate using `setspecs`, the following examples show how to use various syntax forms to set the values in filter specifications objects.

Example 1

Create a lowpass design object `d` using filter order and a cutoff value for the location of the edge of the passband. Then change the cutoff and order specifications of `d`.

```
d = fdesign.lowpass('n,fc')

d =

           ResponseType: 'Lowpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
            FilterOrder: 10
                Fcutoff: 0.5000

setspecs(d, 20, .4);

d =

           ResponseType: 'Lowpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
            FilterOrder: 20
                Fcutoff: 0.4000
```

Example 2

Now specify a sampling frequency after you make `d`.

```
d = fdesign.lowpass('n,fc')
```

```
d =

           ResponseType: 'Lowpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
            FilterOrder: 10
                Fcutoff: 0.5000


setspecs(d, 20, 4, 20);
d

d =

           ResponseType: 'Lowpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: false
                     Fs: 20
            FilterOrder: 20
                Fcutoff: 4
```

Example 3

This example uses the `inputunits` argument to change from the default setting of square to linear unit. Start with the default lowpass design object that specifies the edge locations for the passband and stopband, and the desired attenuation in the passbands and stopbands.

```
d=fdesign.lowpass

d =

           ResponseType: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
```

```
    NormalizedFrequency: true
                     Fs: 'Normalized'
                  Fpass: 0.4500
                  Fstop: 0.5500
                  Apass: 1
                  Astop: 60
```

Convert to linear input values and reset the filter spec for d at the same time. With the linear argument included, the inputs for the response features now need to be in linear units.

```
setspecs(d,.4,.5,.1,.05,'linear')
d

d =

           ResponseType: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
                  Fpass: 0.4000
                  Fstop: 0.5000
                  Apass: 1.7430
                  Astop: 26.0206
```

Example 4

Finally, use `setspecs` to change the `Specification` string and apply new filter specifications to `d`.

```
d=fdesign.decim(3)

d =

          ResponseType: 'Minimum-order nyquist'
         Specification: 'TW,Ast'
           Description: {2x1 cell}
      DecimationFactor: 3
```

```
    NormalizedFrequency: true
                     Fs: 'Normalized'
        TransitionWidth: 0.1000
                  Astop: 80


setspecs(d,'n,ast',16,70)
d

d =

            ResponseType: 'Nyquist with filter order and stopband attenuation'
           Specification: 'N,Ast'
             Description: {2x1 cell}
        DecimationFactor: 3
    NormalizedFrequency: true
                     Fs: 'Normalized'
         PolyphaseLength: 16
                  Astop: 70
```

See Also

designmethods, fdesign.bandpass, fdesign.bandstop, fdesign.decimator, fdesign.halfband, fdesign.highpass, fdesign.interpolator, fdesign.lowpass, fdesign.nyquist, fdesign.rsrc

Purpose

Convert quantized filter to second–order sections (SOS) form

Syntax

```
Hq2 = sos(Hq)
Hq2 = sos(Hq, order)
Hq2 = sos(Hq, order, scale)
```

Description

`Hq2 = sos(Hq)` returns a quantized filter `Hq2` that has second-order sections and the `dft2` structure. Use the same optional arguments used in `tf2sos`.

`Hq2 = sos(Hq, order)` specifies the order of the sections in `Hq2`, where `order` is either of the following strings:

- `'down'` — to order the sections so the first section of `Hq2` contains the poles closest to the unit circle (L_∞ norm scaling)
- `'up'` — to order the sections so the first section of `Hq2` contains the poles farthest from the unit circle (L_2 norm scaling and the default)

`Hq2 = sos(Hq, order, scale)` also specifies the desired scaling of the gain and numerator coefficients of all second-order sections, where `scale` is one of the following strings:

- `'none'` — to apply no scaling (default)
- `'inf'` — to apply infinity-norm scaling
- `'two'` — to apply 2-norm scaling

Use infinity-norm scaling in conjunction with up-ordering to minimize the probability of overflow in the filter realization. Consider using 2-norm scaling in conjunction with down-ordering to minimize the peak round-off noise.

When `Hq` is a fixed-point filter, the filter coefficients are normalized so that the magnitude of the maximum coefficient in each section is 1. The gain of the filter is applied to the first scale value of `Hq2`.

`sos` uses the direct form II transposed (`dft2`) structure to implement second- order section filters.

Examples

```
[b,a]=butter(8,.5);
Hq = dfilt.df2t(b,a);
Hq.arithmetic = 'fixed';
Hq1 = sos(Hq)
```

See Also

`convert`, `dfilt`

`tf2sos` in Signal Processing Toolbox™ documentation

Purpose

Fixed-point scaling modes in direct-form FIR filter

Syntax

```
specifyall(hd)
specifyall(hd,false)
specifyall(hd,true)
```

Description

`specifyall` sets all of the `autoscale` property values of direct-form FIR filters to `false` and all `*modes` of the filters to `SpecifyPrecision`. In this table, you see the results of using `specifyall` with direct-form FIR filters.

Property Name	Default	Setting After Applying specifyall
`CoeffAutoScale`	`true`	`false`
`OutputMode`	`AvoidOverflow`	`SpecifyPrecision`
`ProductMode`	`FullPrecision`	`SpecifyPrecision`
`AccumMode`	`KeepMSB`	`SpecifyPrecision`
`RoundMode`	`convergent`	`convergent`
`OverflowMode`	`wrap`	`wrap`

`specifyall(hd)` gives you maximum control over all settings in a filter `hd` by setting all of the autoscale options that are `true` to `false`, turning off all autoscaling and resetting all modes — `OutputMode`, `ProductMode`, and `AccumMode` — to `SpecifyPrecision`. After you use `specifyall`, you must supply the property values for the mode- and scaling related properties.

`specifyall` provides an alternative to changing all these properties individually. Do note that `specifyall` changes all of the settings; to set some but not all of the modes, set each property as you require.

`specifyall(hd,false)` performs the opposite operation of `specifyall(hd)` by setting all of the autoscale options to `true`; all of the modes to their default values; and hiding the fraction length

properties in the display, meaning you cannot access them to set them or view them.

`specifyall(hd,true)` is equivalent to `specifyall(hd)`.

Examples

This examples demonstrates using `specifyall` to provide access to all of the fixed-point settings of an FIR filter implemented with the direct-form structure. Notice the displayed property values shown after you change the filter to fixed-point arithmetic, then after you use `specifyall` to disable all of the automatic filter scaling and reset the mode values.

```
b = fircband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],{'w''c'});
hd = dfilt.dffir(b);
hd.arithmetic = 'fixed'
hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [1x13 double]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: 'true'
                  Signed: 'on'

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

             ProductMode: 'FullPrecision'

               AccumMode: 'KeepMSB'
         AccumWordLength: 40
```

```
              CastBeforeSum: 'on'

                  RoundMode: 'convergent'
               OverflowMode: 'wrap'

             InheritSettings: 'off'

specifyall(hd)
hd

hd =

             FilterStructure: 'Direct-Form FIR'
                  Arithmetic: 'fixed'
                   Numerator: [1x13 double]
            PersistentMemory: false
                      States: [1x1 embedded.fi]

             CoeffWordLength: 16
              CoeffAutoScale: false
               NumFracLength: 16
                      Signed: true

             InputWordLength: 16
             InputFracLength: 15

            OutputWordLength: 16
                  OutputMode: 'SpecifyPrecision'
            OutputFracLength: 11

                 ProductMode: 'SpecifyPrecision'
           ProductWordLength: 32
           ProductFracLength: 31

                  AccumMode: 'SpecifyPrecision'
             AccumWordLength: 40
             AccumFracLength: 31
```

```
      CastBeforeSum: true

          RoundMode: 'convergent'
       OverflowMode: 'wrap'

     InheritSettings: false
```

The mode properties `InputMode`, `ProductMode`, and `AccumMode` now have the value `SpecifyPrecision` and the fraction length properties appear in the display. Now you use the properties (`InputFracLength`, `ProdFracLength`, `AccumFracLength`) to set the precision the filter applies to the input, product, and accumulator operations. `CoeffAutoScale` switches to `false`, meaning autoscaling of the filter coefficients will not be done to prevent overflows. None of the other filter properties change when you apply `specifyall`.

See Also

`double`, `reffilter`

`fi`, `fimath` in \&tm_fixedpointtoolbox; documentation

Purpose

Step response for filter

Syntax

```
[h,t] = stepz(ha)
stepz(ha)
[h,t] = stepz(hm)
stepz(hm)
```

Description

The next sections describe common `stepz` operation with adaptive and multirate filters. For more input options and for information about using `stepz` with discrete-time filters, refer to `stepz` in Signal Processing Toolbox™ documentation.

Adaptive Filters

For adaptive filters, `stepz` returns the instantaneous zero-phase response based on the current filter coefficients.

`[h,t] = stepz(ha)` returns the step response `h` of the multirate filter `ha`. The length of column vector `h` is the length of the impulse response of `ha`. Returned vector `t` contains the time samples at which `stepz` evaluated the step response. `stepz` returns `h` as a matrix when `ha` is a vector of filters. Each column of the matrix corresponds to one filter in the vector.

`stepz(ha)` displays the filter step response in the Filter Visualization Tool (FVTool).

Multirate Filters

`[h,t] = stepz(hm)` returns the step response `h` of the multirate filter `hm`. The length of column vector `h` is the length of the impulse response of `hm`. The vector `t` contains the time samples at which `stepz` evaluated the step response. `stepz` returns `h` as a matrix when `hm` is a vector of filters. Each column of the matrix corresponds to one filter in the vector.

`stepz(hm)` displays the step response in the Filter Visualization Tool (FVTool).

Note that the response is computed relative to the rate at which the filter is running. If a sampling frequency is specified, it is assumed that the filter is running at that rate.

Note that the multirate filter delay response is computed relative to the rate at which the filter is running. When you specify `fs` (the sampling rate) as an input argument, `stepz` assumes the filter is running at that rate.

For multistage cascades, `stepz` forms a single-stage multirate filter that is equivalent to the cascade and computes the response relative to the rate at which the equivalent filter is running. `stepz` does not support all multistage cascades. Only cascades for which it is possible to derive an equivalent single-stage filter are allowed for analysis.

As an example, consider a two-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. An equivalent single-stage filter with an overall interpolation factor of 8 can be found. `stepz` uses the equivalent filter for the analysis. If you specify a sampling frequency `fs` as an input argument to `stepz`, the function interprets `fs` as the rate at which the equivalent filter is running.

See Also

`freqz`, `impz`

Purpose

Transfer function to coupled allpass

Syntax

```
[d1,d2] = tf2ca(b,a)
[d1,d2] = tf2ca(b,a)
```

Description

`[d1,d2] = tf2ca(b,a)` where `b` is a real, symmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, returns real vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)* such that

$$H(z) = \frac{B(z)}{A(z)} = \frac{1}{2[H1(z) + H2(z)]}$$

representing a coupled allpass decomposition.

`[d1,d2] = tf2ca(b,a)` where `b` is a real, antisymmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, returns real vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)* such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

In some cases, the decomposition is not possible with real *H1(z)* and *H2(z)*. In those cases a generalized coupled allpass decomposition may be possible, whose syntax is

```
[d1,d2,beta] = tf2ca(b,a)
```

to return complex vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)*, and a complex scalar `beta`, satisfying `|beta| = 1`, such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[\bar{\beta} \bullet H1(z) + \beta \bullet H2(z)]$$

representing the generalized allpass decomposition.

In the above equations, *H1(z)* and *H2(z)* are real or complex allpass IIR filters given by

$$H1(z) = \frac{fliplr(\overline{(D1(z))})}{D1(z)}, H2(1)(z) = \frac{fliplr(\overline{(D2(1)(z))})}{D2(1)(z)}$$

where *D1(z)* and *D2(z)* are polynomials whose coefficients are given by `d1` and `d2`.

Note A coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to Signal Processing Toolbox™ User's Guide.

Examples

```
[b,a]=cheby1(9,.5,.4);
[d1,d2]=tf2ca(b,a);  % TF2CA returns denominators of the allpass.
num = 0.5*conv(fliplr(d1),d2)+0.5*conv(fliplr(d2),d1);
den = conv(d1,d2);  % Reconstruct numerator and denonimator.
max([max(b-num),max(a-den)]) % Compare original and reconstructed
       % numerator and denominators.
```

See Also

`ca2tf`, `cl2tf`, `iirpowcomp`, `latc2tf`, `tf2latc`

Purpose

Transfer function to coupled allpass lattice

Syntax

```
[k1,k2] = tf2cl(b,a)
[k1,k2] = tf2cl(b,a)
```

Description

`[k1,k2] = tf2cl(b,a)` where `b` is a real, symmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, will perform the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \frac{1}{2[H1(z) + H2(z)]}$$

of a stable IIR filter *H(z)* and convert the allpass transfer functions *H1(z)* and *H2(z)* to a coupled lattice allpass structure with coefficients given in vectors `k1` and `k2`.

`[k1,k2] = tf2cl(b,a)` where `b` is a real, antisymmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, performs the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

of a stable IIR filter *H(z)* and converts the allpass transfer functions *H1(z)* and *H2(z)* to a coupled lattice allpass structure with coefficients given in vectors `k1` and `k2`.

In some cases, the decomposition is not possible with real *H1(z)* and *H2(z)*. In those cases, a generalized coupled allpass decomposition may be possible, using the command syntax

```
[k1,k2,beta] = tf2cl(b,a)
```

to perform the generalized allpass decomposition of a stable IIR filter *H(z)* and convert the complex allpass transfer functions *H1(z)* and *H2(z)* to corresponding lattice allpass filters

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[\overline{\beta} \bullet H1(z) + \beta \bullet H2(z)]$$

where `beta` is a complex scalar of magnitude equal to 1.

Note Coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to Signal Processing Toolbox™ User's Guide.

Examples

```
[b,a]=cheby1(9,.5,.4);
[k1,k2]=tf2cl(b,a); % Get the reflection coeffs. for the lattices.
[num1,den1]=latc2tf(k1,'allpass'); % Convert each allpass lattice
[num2,den2]=latc2tf(k2,'allpass'); % back to transfer function.
num = 0.5*conv(num1,den2)+0.5*conv(num2,den1);
den = conv(den1,den2); % Reconstruct numerator and denonimator.
max([max(b-num),max(a-den)]) % Compare original and reconstructed
       % numerator and denominators.
```

See Also

`ca2tf`, `cl2tf`, `iirpowcomp`

`latc2tf`, `tf2ca`, `tf2latc` in Signal Processing Toolbox documentation

validstructures

Purpose

Structures for specification object with design method

Syntax

```
validstructures(d)
validstructures(d,'designmethod')
c = validstructures(d,'designmethod')
```

Description

`validstructures(d)` returns the list of structures for all design methods that are available for `d`.

`validstructures(d,'designmethod')` returns a list of the filter structures available for the specification object `d` and the design method in `designmethod`. Knowing which structures apply to your combination of design method and specification makes deciding on a filter structure to implement easier.

To determine the available structures, `validstructures` considers the filter response, such as lowpass or bandstop. It also considers the specifications you use to define the response, such as filter order or stopband attenuation, because changing the filter specifications often changes the available structures.

`c = validstructures(d,'designmethod')` returns the output cell array `c` that contains the filter structures as character strings.

Examples

These examples demonstrate some results of applying `validstructures` to a combination of a specification object and a design method.

Example 1

An interpolator that uses the Polyphase Length and Stopband Attenuation options to design the filter.

```
d=fdesign.interp(6,'PL,Ast',20,65)

d =

                Response: 'Lowpass interpolator'
           Specification: 'PL,Ast'
             Description: {'Polyphase Length';'Stopband Attenuation (dB)'}
```

```
    InterpolationFactor: 6
    NormalizedFrequency: true
        PolyphaseLength: 20
                  Astop: 65

designmethods(d)

FIR Design Methods for class fdesign.interp (PL,Ast):

kaiserwin

validstructures(d,'kaiserwin')

ans =

    'firinterp'    'fftfirinterp'
```

Now you can specify the filter structure when you design the filter `hm`.

```
hm=design(d,'kaiserwin','FilterStructure','firinterp')

hm =

FilterStructure: 'Direct-Form FIR Polyphase
                  Interpolator'
          Arithmetic: 'double'
           Numerator: [1x120 double]
InterpolationFactor: 6
       PersistentMemory: false
```

Example 2

A CIC decimator is used as a specification object. Because the object is a decimator and the structure is defined as CIC, the only valid structure is `cicdecim`.

```
d=fdesign.cicdecim(5)
```

```
d =

               Response: 'CIC Decimator'
          Specification: 'Fp,Ast'
            Description: {'Passband Frequency';'Aliasing
Attenuation(dB)'}
      DifferentialDelay: 5
    NormalizedFrequency: true
                  Fpass: 0.01
                  Astop: 60

designmethods(d)

FIR Design Methods for class fdesign.cicdecim (Fp,Ast):

multisection

c=validstructures(d,'multirate')

c =

    'cicdecim'
```

Example 3

This default highpass specification object has more design methods available, however, changing the design method changes the valid filter structures.

```
d=fdesign.highpass;
designmethods(d)

Design Methods for class fdesign.highpass (Fst,Fp,Ast,Ap):

butter
cheby1
cheby2
ellip
```

```
equiripple
ifir
kaiserwin

validstructures(d,'equiripple')

    'dffir'    'dffirt'    'dfsymfir'    'dfasymfir'    'fftfir'
```

Using the `cheby2` method results in both IIR filter structures and cascade allpass structure options..

```
c=validstructures(d,'cheby2')

c =

    'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass'
'cascadewdfallpass'
```

Example 4

Multirate filters support `validstructures`.

```
d=fdesign.rsrc(4,5);
designmethods(d)

FIR Design Methods for class fdesign.rsrc (TW,Ast):

equiripple
kaiserwin

validstructures(d,'kaiserwin')

     'firinterp'    'fftfirinterp'
```

See Also

`design`, `designmethods`, `designopts`, `fdesign`

Purpose

FIR filter using windowed impulse response

Syntax

```
h = window(d,fcnhndl,fcnarg)
h = window(d,win)
```

description

`h = window(d,fcnhndl,fcnarg)` 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. Example 2 shows this being done.

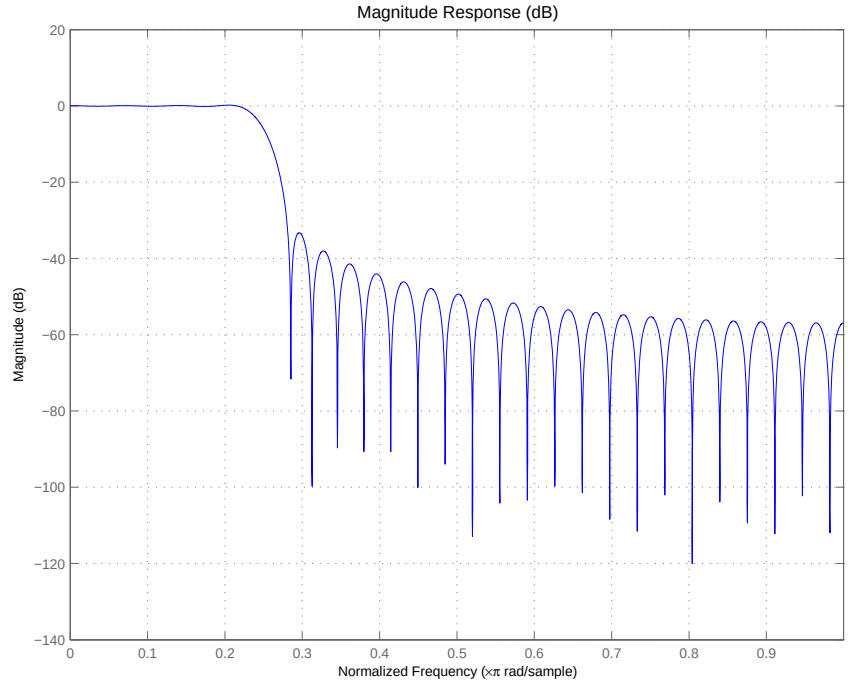
Examples

These examples design filters using the two design techniques of specifying a function handle or passing a window vector as an input argument.

Example 1

Use a function handle and optional input arguments to design a multirate filter. We use a function handle to the function Kaiser to provide the window. Since this example creates a decimating filter specifications object, `window` returns a multirate filter.

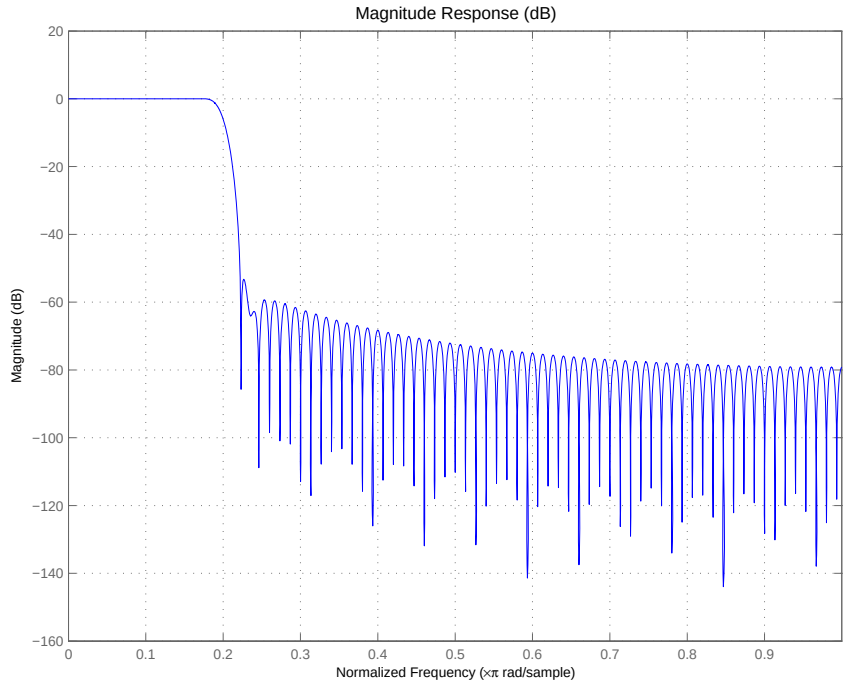
```
d = fdesign.decim(4,'pl',14);
hm = window(d,@kaiser,2.5);
fvtool(hm)
```



Example 2

Use a window vector provided by the `hamming` window design function. For this example, the design object is a Nyquist filter, thus window returns `hd` as a discrete-time filter.

```
d = fdesign.nyquist(5,'n',150);
hd = window(d,hamming(151));
fvtool(hd)
```



See Also

`firls`, `kaiserwin`

Purpose

Zero-phase response for filter

Syntax

```
zerophase(ha)
[hr,w] = zerophase(ha,n)
[hr,w] = zerophase(...,f)
zerophase(hd)
[hr,w] = zerophase(hd,n)
[hr,w] = zerophase(...,f)
zerophase(hm)
[hr,w] = zerophase(hm,n)
[hr,w] = zerophase(...,f)
[hr,w] = zerophase(...,fs)
```

Description

The next sections describe common `zerophase` operation with adaptive, discrete-time, and multirate filters. For more input options, refer to `zerophase` in Signal Processing Toolbox™ documentation.

Adaptive Filters

For adaptive filters, `zerophase` returns the instantaneous zero-phase response based on the current filter coefficients.

`zerophase(ha)` displays the zero-phase response of `ha` in the Filter Visualization Tool (FVTool).

`[hr,w] = zerophase(ha,n)` returns length `n` vectors `hr` and `w` containing the instantaneous zero-phase response of the adaptive filter `ha`, and the frequencies in radians at which `zerophase` evaluated the response. The zero-phase response is evaluated at `n` points equally spaced around the upper half of the unit circle. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs. If `n` is not specified, it defaults to 8192.

`[hr,w] = zerophase(ha)` returns a matrix `hr` if `ha` is a vector of filters. Each column of the matrix corresponds to each filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `hr` corresponds to one filter in the vector.

Discrete-Time Filters

`zerophase(hd)` displays the zero-phase response of `hd` in the Filter Visualization Tool (FVTool).

`[hr,w] = zerophase(hd,n)` returns length `n` vectors `hr` and `w` containing the instantaneous zero-phase response of the adaptive filter `hd`, and the frequencies in radians at which `zerophase` evaluated the response. The zero-phase response is evaluated at `n` points equally spaced around the upper half of the unit circle. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs. If `n` is not specified, it defaults to 8192.

`[hr,w] = zerophase(hd)` returns a matrix `hr` if `hd` is a vector of filters. Each column of the matrix corresponds to each filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `hr` corresponds to one filter in the vector.

Multirate Filters

`zerophase(hm)` displays the zero-phase response of `hd` in the Filter Visualization Tool (FVTool).

`[hr,w] = zerophase(hm,n)` returns length `n` vectors `hr` and `w` containing the instantaneous zero-phase response of the adaptive filter `hm`, and the frequencies in radians at which `zerophase` evaluated the response. The zero-phase response is evaluated at `n` points equally spaced around the upper half of the unit circle. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs. If `n` is not specified, it defaults to 8192.

`[hr,w] = zerophase(hm)` returns a matrix `hr` if `hm` is a vector of filters. Each column of the matrix corresponds to each filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `hr` corresponds to one filter in the vector.

Note that the response is computed relative to the rate at which the filter is running. If a sampling frequency is specified, it is assumed that the filter is running at that rate.

Note that the multirate filter delay response is computed relative to the rate at which the filter is running. When you specify `fs` (the sampling rate) as an input argument, `zerophase` assumes the filter is running at that rate.

For multistage cascades, `zerophase` forms a single-stage multirate filter that is equivalent to the cascade and computes the response relative to the rate at which the equivalent filter is running. `zerophase` does not support all multistage cascades. Only cascades for which it is possible to derive an equivalent single-stage filter are allowed for analysis.

As an example, consider a two-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. An equivalent single-stage filter with an overall interpolation factor of 8 can be found. `zerophase` uses the equivalent filter for the analysis. If a sampling frequency `fs` is specified as an input argument to `zerophase`, the function interprets `fs` as the rate at which the equivalent filter is running.

See Also

`freqz`, `fvtool`, `grpdelay`, `impz`, `mfilt`, `phasez`, `zerophase`, `zplane`

zpkbpc2bpc

Purpose

Zero-pole-gain complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations, W_{t1}, and W_{t2} respectively. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle; e.g., in the adaptive system.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

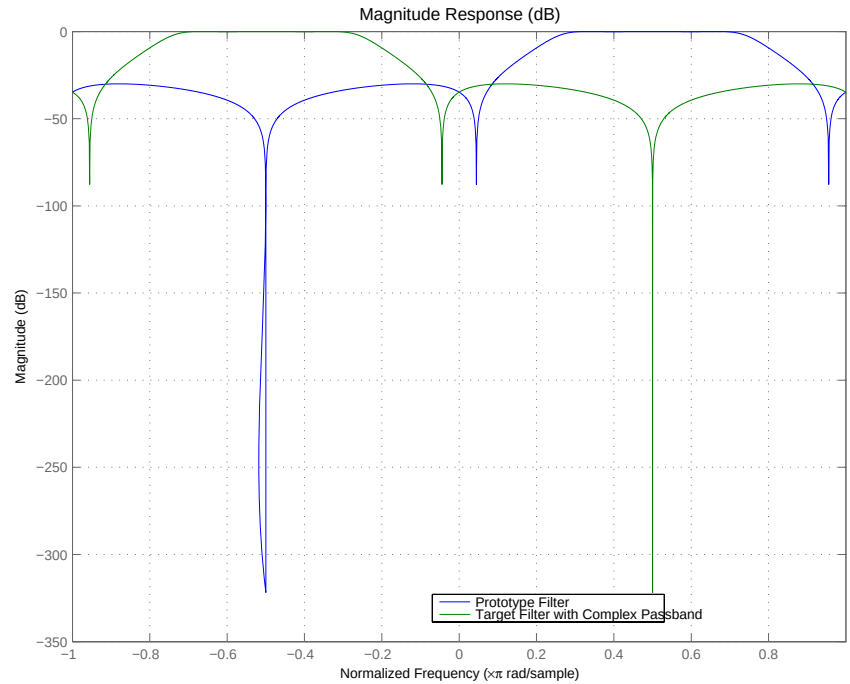
Create a complex passband from 0.25 to 0.75:

```
[b, a] = iirlp2bpc(b,a,0.5,[0.25,0.75]);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpkbpc2bpc(z,p,k,[0.25, 0.75],[-0.75, -0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Comparing the filters in FVTool shows the example results. Use the features in FVTool to check the filter coefficients, or other filter analyses.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpassbpc2bpc`, `iirbpc2bpc`

Purpose

Zero-pole-gain frequency transformation

Syntax

```
[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)
```

Description

`[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the transformed lowpass digital filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`. If `AllpassDen` is not specified it will default to 1. If neither `AllpassNum` nor `AllpassDen` is specified, then the function returns the input filter.

Examples

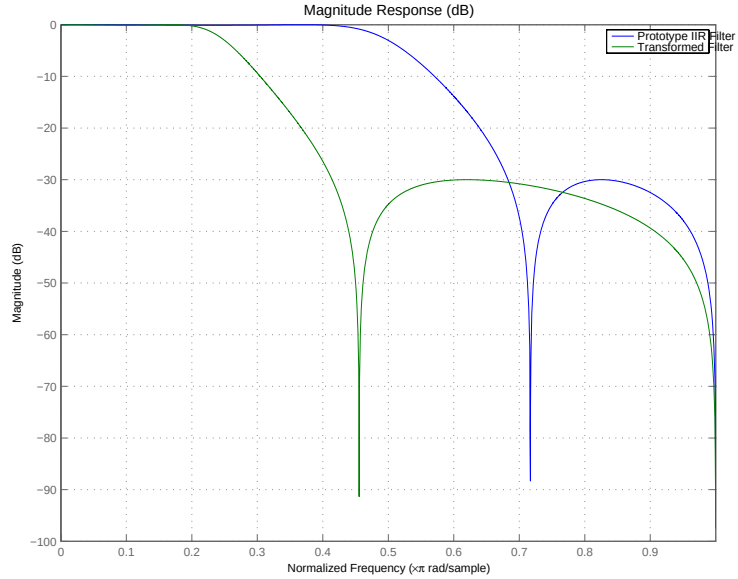
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[z2, p2, k2] = zpkftransf(roots(b),roots(a),b(1),AlpNum,AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

After transforming the filter, you get the response shown in the figure, where the passband has been shifted towards zero.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
FTFNum	Numerator of the mapping filter
FTFDen	Denominator of the mapping filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter

See Also

`iirftransf`

Purpose

Zero-pole-gain lowpass to bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Real lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be easily doubled and positioned at two distinct, desired frequencies.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bp(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2bp`, `iirlp2bp`

References

Constantinides, A.G., “Spectral transformations for digital filters,” *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, “Prototype reference transfer function parameters in the discrete-time frequency transformations,” *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., “Design of bandpass digital filters,” *IEEE® Proceedings*, vol. 1, pp. 1129-1231, June 1969.

zpklp2bpc

Purpose

Zero-pole-gain lowpass to complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bpc(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`zpkftransf`, `allpasslp2bpc`, `iirlp2bpc`

zpklp2bs

Purpose

Zero-pole-gain lowpass to bandstop frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_ts.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
```

```
k = b(1);
[z2,p2,k2] = zpklp2bs(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2bs`, `iirlp2bs`

References

Constantinides, A.G., “Spectral transformations for digital filters,” *IEEE® Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, “Prototype reference transfer function parameters in the discrete-time frequency transformations,”

Proceedings 33rd Midwest Symposium on Circuits and Systems, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose

Zero-pole-gain lowpass to complex bandstop frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

```
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bsc(z, p, k, 0.5, [0.2, 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`zpkftransf`, `allpasslp2bsc`, `iirlp2bsc`

Purpose

Zero-pole-gain lowpass to highpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real highpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency location, W_t, at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and the gain factor, `K`.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, or the deep minimum in the stopband, or other ones.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
```

```
k = b(1);
[z2,p2,k2] = zpklp2hp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2hp`, `iirlp2hp`

References

Constantinides, A.G., “Spectral transformations for digital filters,” *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, “Prototype reference transfer function parameters in the discrete-time frequency transformations,”

Proceedings 33rd Midwest Symposium on Circuits and Systems, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

zpklp2lp

Purpose

Zero-pole-gain lowpass to lowpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real lowpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency location, W_t.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to lowpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to lowpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

Examples

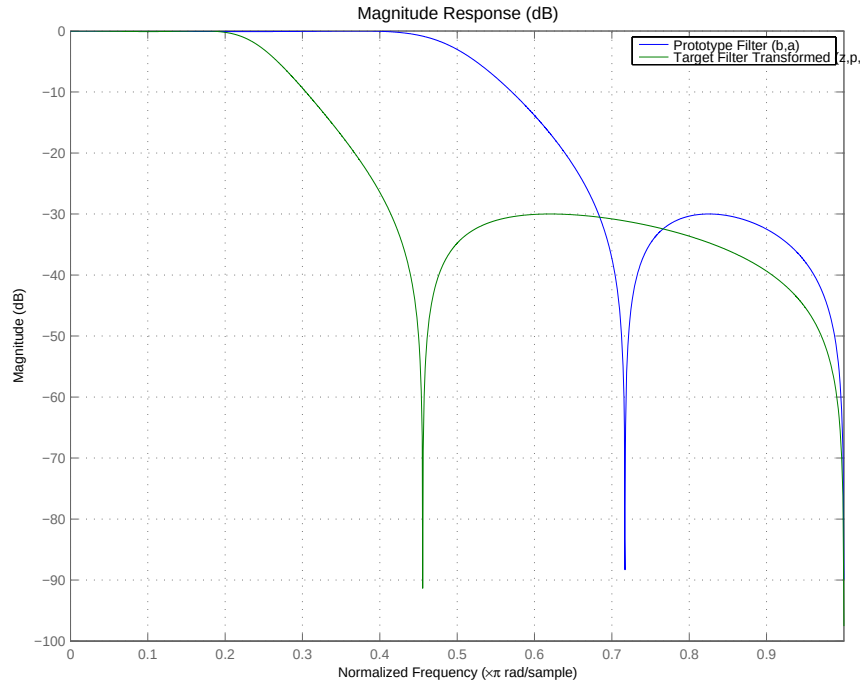
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2lp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Using `zpklp2lp` creates the desired half band IIR filter with the transformed features that you specify in the transformation function. This figure shows the results.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2lp`, `iirlp2lp`

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

Purpose

Zero-pole-gain lowpass to M-band frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `M`th-order real lowpass to real multibandpass frequency mapping. By default the DC feature is kept at its original location.

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the “DC Mobility” and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},\ldots,W_{tM}$.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a

number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

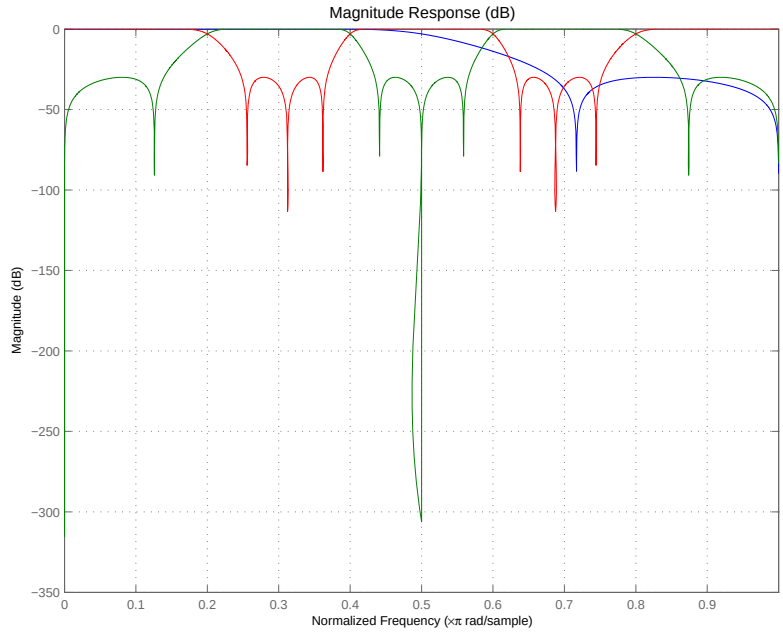
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'pass');
[z2,p2,k2] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

The resulting multiband filter that replicates features from the prototype appears in the figure shown. Note the accuracy of the replication process.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
Z2	Zeros of the target filter

Variable	Description
`P2`	Poles of the target filter
`K2`	Gain factor of the target filter
`AllpassNum`	Numerator of the mapping filter
`AllpassDen`	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2mb`, `iirlp2mb`

References

Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

Purpose

Zero-pole-gain lowpass to complex M-band frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `M`th-order real lowpass to complex multibandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature, for example, the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., to replicate notch filters and resonators at any required location.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

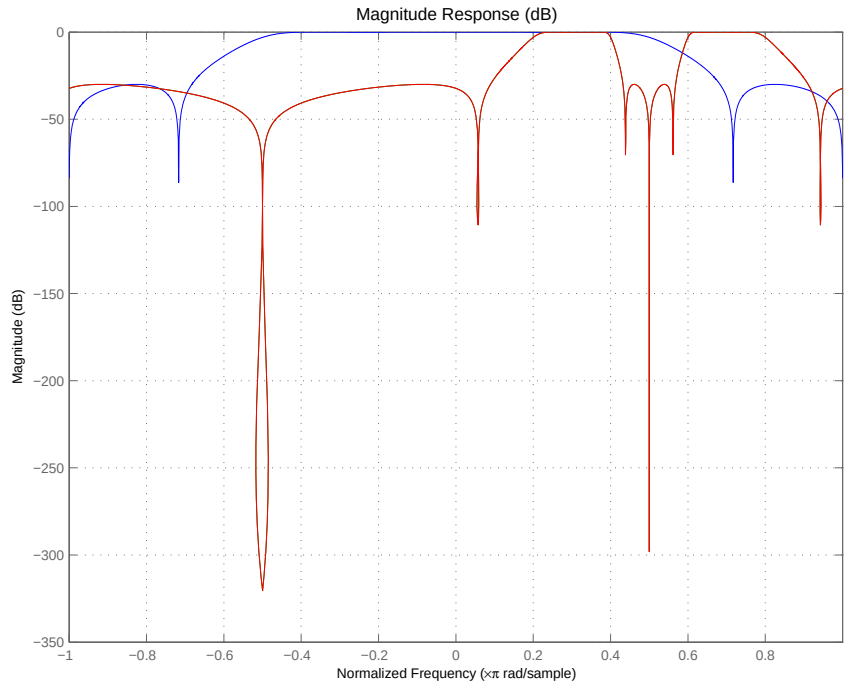
```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

```
[z1,p1,k1] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10);
[z2,p2,k2] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

You could review the coefficients to compare the filters, but the graphical comparison shown here is quicker and easier.



However, looking at the coefficients in FVTool shows the complex nature desired.

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`zpkftransf`, `allpasslp2mbc`, `iirlp2mbc`

Purpose

Zero-pole-gain lowpass to complex N-point frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to complex multipoint frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be

an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

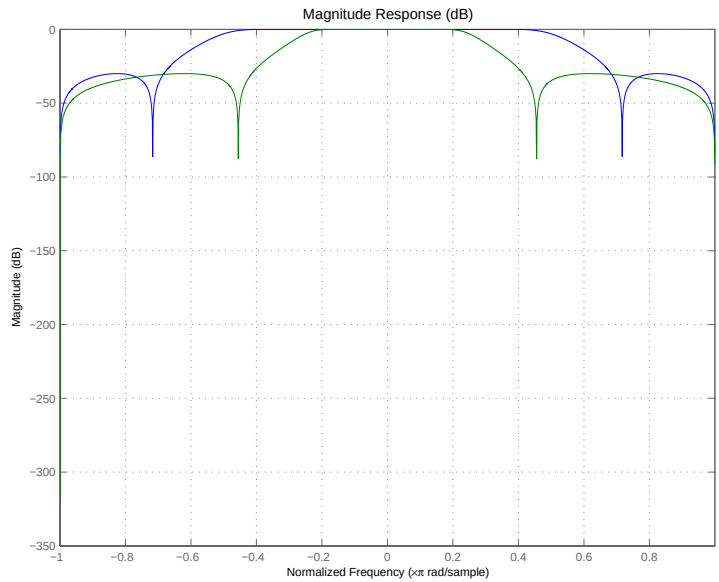
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xc(z, p, k, [-0.5 0.5], [-0.25 0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Plotting the filters on the same axes lets you compare the results graphically, shown here.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`zpkftransf`, `allpasslp2xc`, `iirlp2xc`

Purpose

Zero-pole-gain lowpass to N-point frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to real multipoint frequency transformation, where `N` is the number of features being mapped. By default the DC feature is kept at its original location.

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

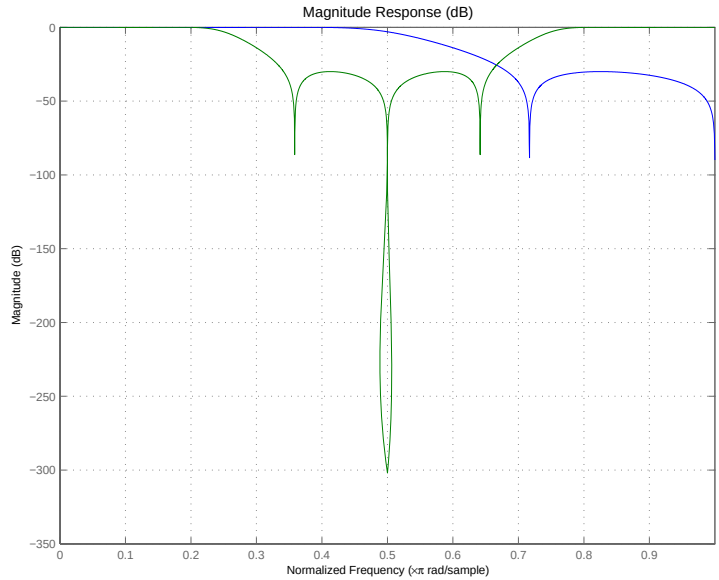
Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xn(z, p, k, [-0.5 0.5], [-0.25 0.25], 'pass');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```



As demonstrated by the figure, the target filter has the desired response shape and values replicated from the prototype.

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default

Variable	Description
`Z2`	Zeros of the target filter
`P2`	Poles of the target filter
`K2`	Gain factor of the target filter
`AllpassDen`	Numerator of the mapping filter
`AllpassDen`	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2xn`, `iirlp2xn`

References

Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

Purpose

Zero-pole-gain complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter being transformed from any prototype by applying an `N`th-order rateup frequency transformation, where `N` is the upsample ratio. Transformation creates `N` equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

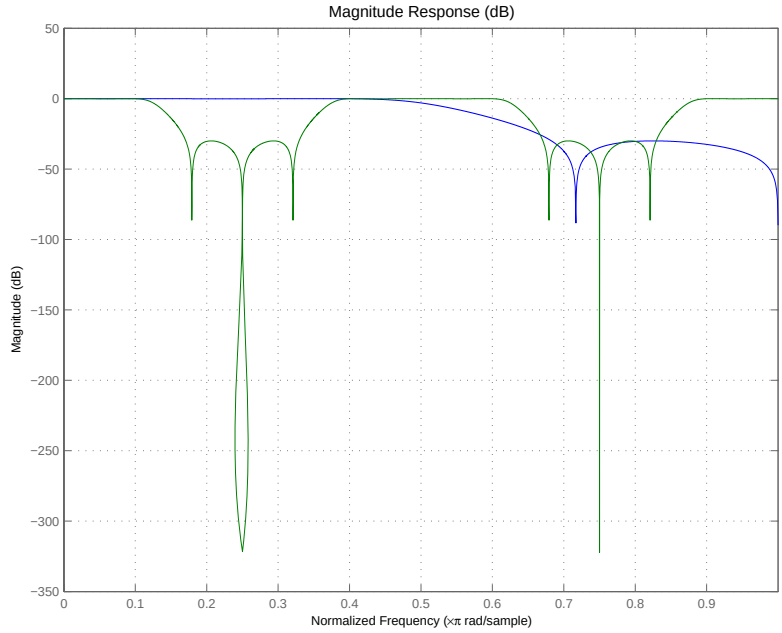
Upsample the prototype filter four times:

```
[z2,p2,k2] = zpkrateup(z, p, k, 4);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Applying the upsample process creates a bandpass filter, as shown here.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
N	Integer upsampling ratio
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkrateup`, `allpassrateup`, `iirrateup`

zpkshift

Purpose

Zero-pole-gain real shift frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real shift frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator of the allpass mapping filter, `AllpassDen`. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation places one selected feature of an original filter, located at frequency W_o, at the required target frequency location, W_t. This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without the need to design them again.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

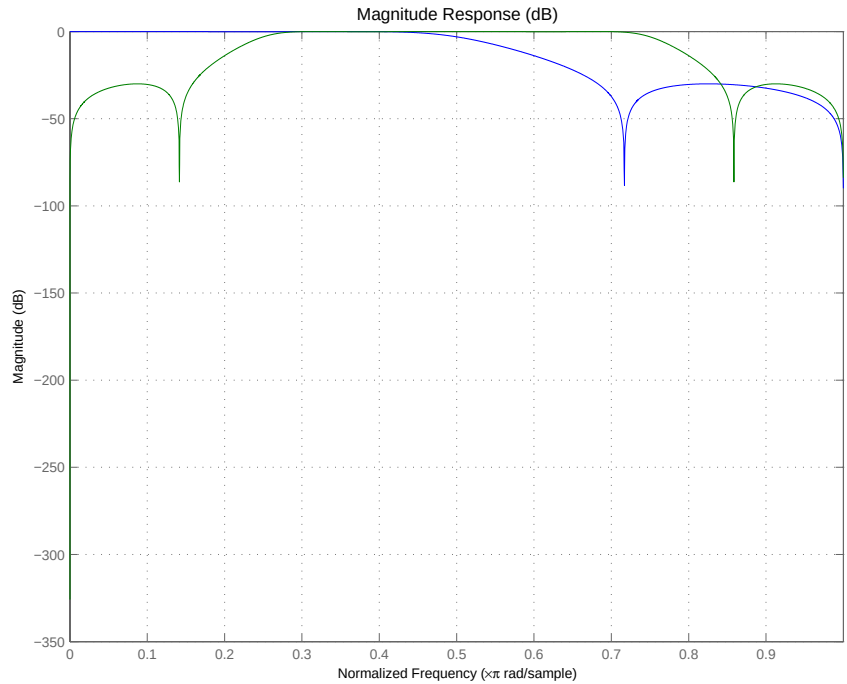
```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
```

```
p = roots(a);
k = b(1);
[z2,p2,k2] = zpkshift(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

It is clear from the following figure that the shift process has taken the response value at 0.5 in the prototype and replicated it in the target at 0.25, as specified.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpassshift`, `iirshift`

Purpose

Zero-pole-gain complex shift frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,-0.5)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at W_o, placed at W_t in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and the gain factor, `K`.

`[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,0.5)` performs the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,-0.5)` performs the inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

Example 1

Rotation by -0.25:

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0.5, 0.25);
```

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Example 2

Hilbert transform:

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0, 0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

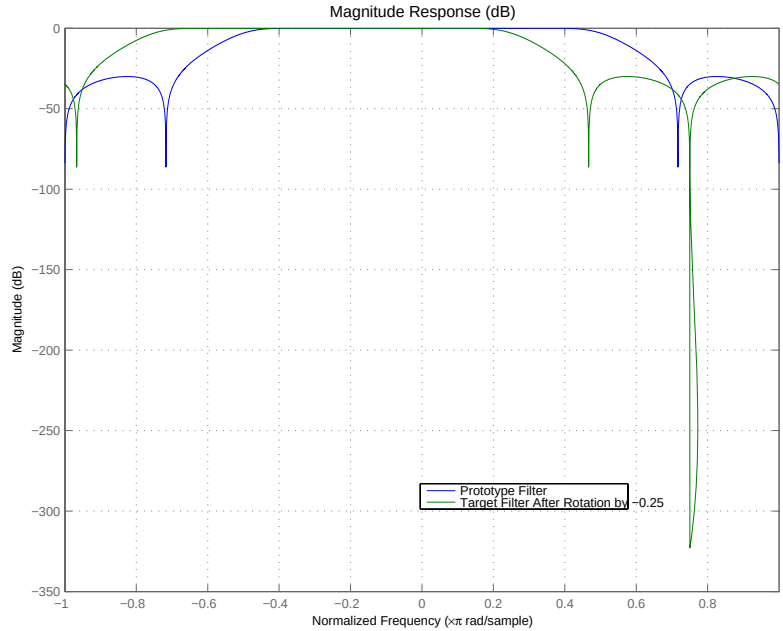
Example 3

Inverse Hilbert transform:

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0, -0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

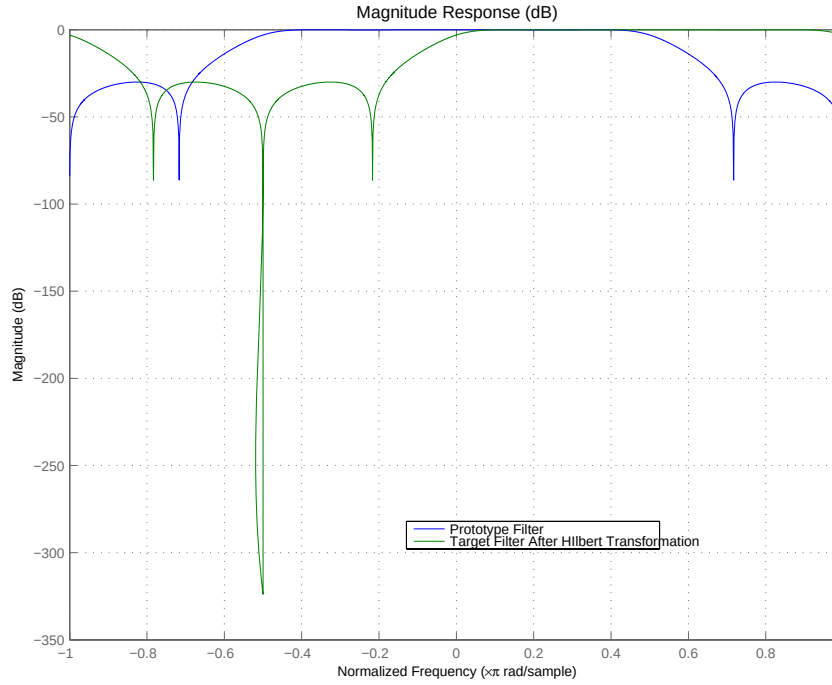
Result of Example 1

After performing the rotation, the resulting filter shows the features desired.



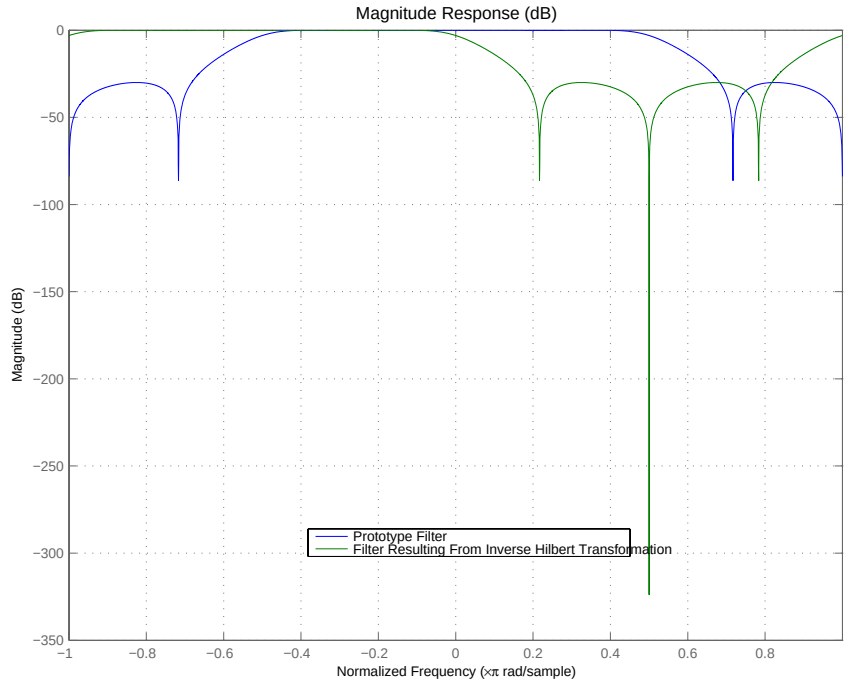
Result of Example 2

Similar to the first example, performing the Hilbert transformation generates the desired target filter, shown here.


Magnitude Response (dB)
Magnitude (dB)
Normalized Frequency (×π rad/sample)
Prototype Filter
Target Filter After HIlbert Transformation
0
−50
−100
−150
−200
−250
−300
−350
−1
−0.8
−0.6
−0.4
−0.2
0
0.2
0.4
0.6
0.8

Result of Example 3

Finally, using the inverse Hilbert transformation creates yet a third filter, as the figure shows.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter

Variable	Description
`Wt`	Desired frequency location in the transformed target filter
`Z2`	Zeros of the target filter
`P2`	Poles of the target filter
`K2`	Gain factor of the target filter
`AllpassDen`	Numerator of the mapping filter
`AllpassDen`	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpassshiftc`, `iirshiftc`

References

Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE® Transactions on Education*, vol. 32, pp. 314-318, August 1989.

Purpose

Zero-pole plot for filter

Syntax

```
zplane(Hq)
zplane(Hq,'plotoption')
zplane(Hq,'plotoption','plotoption2')
[zq,pq,kq] = zplane(Hq)
[zq,pq,kq,zr,pr,kr] = zplane(Hq)
```

Description

This function displays the poles and zeros of quantized filters, as well as the poles and zeros of the associated unquantized reference filter.

`zplane(Hq)` plots the zeros and poles of a quantized filter `Hq` in the current figure window. The poles and zeros of the quantized and unquantized filters are plotted by default. The symbol `o` represents a zero of the unquantized reference filter, and the symbol `x` represents a pole of that filter. The symbols □ and + are used to plot the zeros and poles of the quantized filter `Hq`. The plot includes the unit circle for reference.

`zplane(Hq,'plotoption')` plots the poles and zeros associated with the quantized filter `Hq` according to one specified plot option. The string '*plotoption*' can be either of the following reference filter display options:

- **`on`** to display the poles and zeros of both the quantized filter and the associated reference filter (default)
- **`off`** to display the poles and zeros of only the quantized filter

`zplane(Hq,'plotoption','plotoption2')` plots the poles and zeros associated with the quantized filter `Hq` according to two specified plot options. The string '*plotoption*' can be selected from the reference filter display options listed in the previous syntax. The string '*plotoption2*' can be selected from the section-by-section plotting style options described in the following list:

- **`individual`** to display the poles and zeros of each section of the filter in a separate figure window

- **overlay** to display the poles and zeros of all sections of the filter on the same plot
- **tile** to display the poles and zeros of each section of the filter in a separate plot in the same figure window

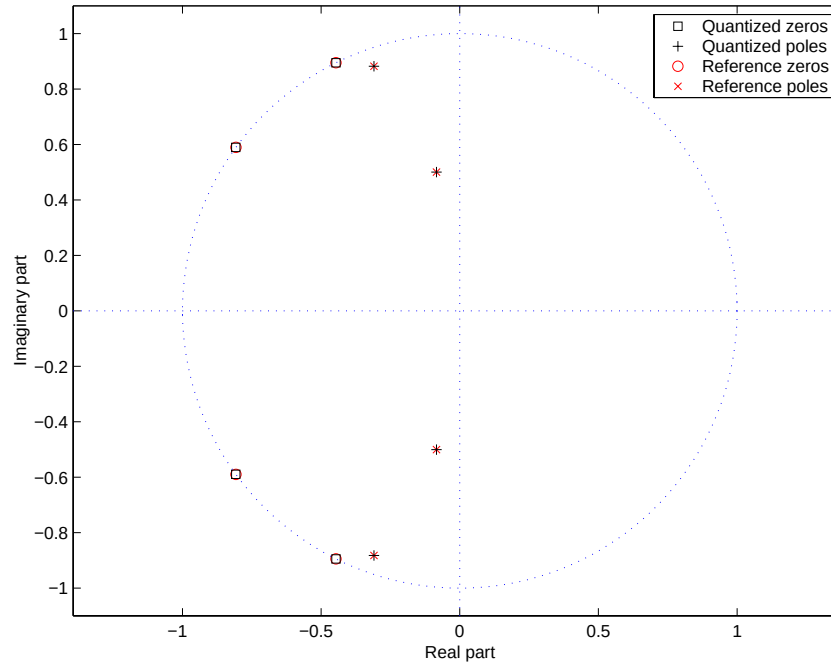
`[zq,pq,kq] = zplane(Hq)` returns the vectors of zeros `zq`, poles `pq`, and gains `kq`. If `Hq` has *n* sections, `zq`, `pq`, and `kq` are returned as 1-by-*n* cell arrays. If there are no zeros (or no poles), `zq` (or `pq`) is set to the empty matrix `[]`.

`[zq,pq,kq,zr,pr,kr] = zplane(Hq)` returns the vectors of zeros `zr`, poles `pr`, and gains `kr` of the reference filter associated with the quantized filter `Hq`, and returns the vectors of zeros `zq`, poles `pq`, and gains `kq` for the quantized filter `Hq`.

Examples

Create a quantized filter `Hq` from a fourth-order digital filter with cutoff frequency of 0.6. Scale the transfer function parameters to avoid overflows due to coefficient quantization. Plot the quantized and unquantized poles and zeros associated with this quantized filter.

```
[b,a] = ellip(4,.5,20,.6);
Hq = dfilt.df2(b/2 a/2);
Hq.arithmetic = 'fixed';
zplane(Hq);
```



See Also

`freqz`, `impz`

Index

T

Z