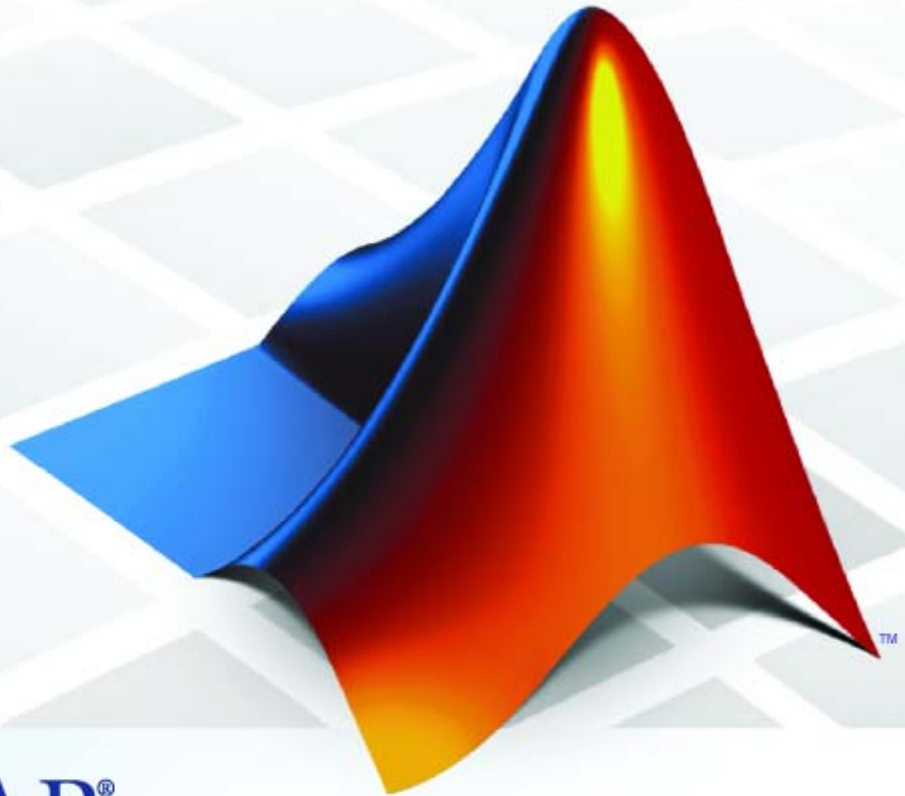


# Filter Design Toolbox™ 4

## Reference Guide



MATLAB®

## How to Contact The MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto: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*

© COPYRIGHT 2000–2010 by The MathWorks™, Inc.

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

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

### Trademarks

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

### Patents

The MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

**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)
October 2008	Online only	New for Version 4.4 (Release 2008b)
March 2009	Online only	New for Version 4.5 (Release 2009a)
September 2009	Online only	New for Version 4.6 (Release 2009b)
March 2010	Online only	New for Version 4.7 (Release 2010a)



## Function Reference

**1**

<b>Adaptive Filters</b> .....	1-2
Least Mean Squares .....	1-2
Recursive Least Squares .....	1-3
Affine Projection .....	1-3
Frequency Domain .....	1-4
Lattice .....	1-4
<b>Discrete-Time Filters</b> .....	1-5
<b>Filter Specifications — Response Types</b> .....	1-7
<b>Filter Specifications — Design Methods</b> .....	1-9
<b>Multirate Filters</b> .....	1-10
<b>GUIs</b> .....	1-11
<b>Filter Analysis</b> .....	1-12
<b>Fixed-Point Filters</b> .....	1-15
<b>Quantized Filter Analysis</b> .....	1-16
<b>SOS Conversion</b> .....	1-17
<b>Filter Design</b> .....	1-18
<b>Filter Conversion</b> .....	1-19

2

---

Reference for the Properties of Filter Objects

3

<b>Fixed-Point Filter Properties</b> .....	3-2
Overview of Fixed-Point Filters .....	3-2
Fixed-Point Objects and Filters .....	3-2
Summary — Fixed-Point Filter Properties .....	3-5
Property Details for Fixed-Point Filters .....	3-18
<b>Adaptive Filter Properties</b> .....	3-102
Property Summaries .....	3-102
Property Details for Adaptive Filter Properties .....	3-107
<b>References</b> .....	3-115
<b>Multirate Filter Properties</b> .....	3-116
Property Summaries .....	3-116
Property Details for Multirate Filter Properties .....	3-121
<b>References</b> .....	3-132

---

**Index**

# Function Reference

---

Adaptive Filters (p. 1-2)	Design adaptive filters
Discrete-Time Filters (p. 1-5)	Design FIR and IIR discrete-time filter objects
Filter Specifications — Response Types (p. 1-7)	Create objects that specify filter responses
Filter Specifications — Design Methods (p. 1-9)	Design filter objects from specification objects
Multirate Filters (p. 1-10)	Design multirate filter objects
GUIs (p. 1-11)	Use graphical user interface tools to design filters
Filter Analysis (p. 1-12)	Analyze filters and filter objects
Fixed-Point Filters (p. 1-15)	Create fixed-point filters
Quantized Filter Analysis (p. 1-16)	Analyze fixed-point filters
SOS Conversion (p. 1-17)	Work with second-order section filters
Filter Design (p. 1-18)	Design filters (not object-based)
Filter Conversion (p. 1-19)	Transform filters to other forms, or use features in filter to develop another filter

## Adaptive Filters

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

## Least Mean Squares

<code>adaptfilt.adjLMS</code>	FIR adaptive filter that uses adjoint LMS algorithm
<code>adaptfilt.blms</code>	FIR adaptive filter that uses BLMS
<code>adaptfilt.blmsfft</code>	FIR adaptive filter that uses FFT-based BLMS
<code>adaptfilt.dlms</code>	FIR adaptive filter that uses delayed LMS
<code>adaptfilt.filtxLMS</code>	FIR adaptive filter that uses filtered-x LMS
<code>adaptfilt.lms</code>	FIR adaptive filter that uses LMS
<code>adaptfilt.nlms</code>	FIR adaptive filter that uses NLMS
<code>adaptfilt.sd</code>	FIR adaptive filter that uses sign-data algorithm
<code>adaptfilt.se</code>	FIR adaptive filter that uses sign-error algorithm
<code>adaptfilt.ss</code>	FIR adaptive filter that uses sign-sign algorithm



## Recursive Least Squares

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

## Affine Projection

<code>adaptfilt.ap</code>	FIR adaptive filter that uses direct matrix inversion
<code>adaptfilt.apru</code>	FIR adaptive filter that uses recursive matrix updating
<code>adaptfilt.bap</code>	FIR adaptive filter that uses block affine projection

## Frequency Domain

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

## Lattice

<code>adaptfilt.gal</code>	FIR adaptive filter that uses gradient lattice
<code>adaptfilt.lsl</code>	Adaptive filter that uses LSL
<code>adaptfilt.qrdls1</code>	Adaptive filter that uses QR-decomposition-based LSL

## Discrete-Time Filters

<code>dfilt</code>	Discrete-time filter
<code>dfilt.allpass</code>	Allpass filter
<code>dfilt.calattice</code>	Coupled-allpass, lattice filter
<code>dfilt.calatticepc</code>	Coupled-allpass, power-complementary lattice filter
<code>dfilt.cascade</code>	Cascade of discrete-time filters
<code>dfilt.cascadeallpass</code>	Cascade of allpass discrete-time filters
<code>dfilt.cascadewdfallpass</code>	Cascade allpass WDF filters to construct allpass WDF
<code>dfilt.delay</code>	Delay filter
<code>dfilt.df1</code>	Discrete-time, direct-form I filter
<code>dfilt.df1sos</code>	Discrete-time, SOS direct-form I filter
<code>dfilt.df1t</code>	Discrete-time, direct-form I transposed filter
<code>dfilt.df1tsos</code>	Discrete-time, SOS direct-form I transposed filter
<code>dfilt.df2</code>	Discrete-time, direct-form II filter
<code>dfilt.df2sos</code>	Discrete-time, SOS, direct-form II filter
<code>dfilt.df2t</code>	Discrete-time, direct-form II transposed filter
<code>dfilt.df2tsos</code>	Discrete-time, SOS direct-form II transposed filter
<code>dfilt.dfasymfir</code>	Discrete-time, direct-form antisymmetric FIR filter
<code>dfilt.dffir</code>	Discrete-time, direct-form FIR filter

<code>dfilt.dffirt</code>	Discrete-time, direct-form FIR transposed filter
<code>dfilt.dfsymfir</code>	Discrete-time, direct-form symmetric FIR filter
<code>dfilt.farrowfd</code>	Fractional Delay Farrow filter
<code>dfilt.farrowlinearfd</code>	Farrow Linear Fractional Delay filter
<code>dfilt.ffsetfir</code>	Discrete-time, overlap-add, FIR filter
<code>dfilt.latticeallpass</code>	Discrete-time, lattice allpass filter
<code>dfilt.latticear</code>	Discrete-time, lattice, autoregressive filter
<code>dfilt.latticearma</code>	Discrete-time, lattice, autoregressive, moving-average filter
<code>dfilt.latticemamax</code>	Discrete-time, lattice, moving-average filter with maximum phase
<code>dfilt.latticemamin</code>	Discrete-time, lattice, moving-average filter with minimum phase
<code>dfilt.parallel</code>	Discrete-time, parallel structure filter
<code>dfilt.scalar</code>	Discrete-time, scalar filter
<code>dfilt.wdfallpass</code>	Wave digital allpass filter

## Filter Specifications — Response Types

<code>fdesign</code>	Filter specification object
<code>fdesign.arbmag</code>	Arbitrary response magnitude filter specification object
<code>fdesign.arbmagnphase</code>	Arbitrary response magnitude and phase filter specification object
<code>fdesign.audioweighting</code>	Audio weighting filter specification object
<code>fdesign.bandpass</code>	Bandpass filter specification object
<code>fdesign.bandstop</code>	Bandstop filter specification object
<code>fdesign.ciccomp</code>	CIC compensator filter specification object
<code>fdesign.comb</code>	IIR comb filter specification object
<code>fdesign.decimator</code>	Decimator filter specification object
<code>fdesign.differentiator</code>	Differentiator filter specification object
<code>fdesign.fracdelay</code>	Fractional delay filter specification object
<code>fdesign.halfband</code>	Halfband filter specification object
<code>fdesign.highpass</code>	Highpass filter specification object
<code>fdesign.hilbert</code>	Hilbert filter specification object
<code>fdesign.interpolator</code>	Interpolator filter specification
<code>fdesign.isinclp</code>	Inverse-sinc filter specification
<code>fdesign.lowpass</code>	Lowpass filter specification
<code>fdesign.notch</code>	Notch filter specification
<code>fdesign.nyquist</code>	Nyquist filter specification
<code>fdesign.octave</code>	Octave filter specification
<code>fdesign.parameq</code>	Parametric equalizer filter specification

<code>fdesign.peak</code>	Peak filter specification
<code>fdesign.polysrc</code>	Construct polynomial sample-rate converter (POLYSRC) filter designer
<code>fdesign.pulseshaping</code>	Pulse-shaping filter specification object
<code>fdesign.rsrc</code>	Rational-factor sample-rate converter specification

## Filter Specifications — Design Methods

<code>butter</code>	Butterworth IIR filter design using specification object
<code>cheby1</code>	Chebyshev Type I filter using specification object
<code>cheby2</code>	Chebyshev Type II filter using specification object
<code>designmethods</code>	Methods available for designing filter from specification object
<code>ellip</code>	Elliptic filter using specification object
<code>equiripple</code>	Equiripple single-rate or multirate FIR filter from specification object
<code>fircls</code>	FIR Constrained Least Squares filter
<code>ifir</code>	Interpolated FIR filter from filter specification
<code>iirlinphase</code>	Quasi-linear phase IIR filter from halfband filter specification
<code>kaiserwin</code>	Kaiser window filter from specification object
<code>maxflat</code>	Maxflat FIR filter
<code>multistage</code>	Multistage filter from specification object
<code>window</code>	FIR filter using windowed impulse response

## Multirate Filters

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



## GUIs

`fdatool`

Open Filter Design and Analysis  
Tool

`filterbuilder`

GUI-based filter design

## Filter Analysis

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

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

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

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

## Quantized Filter Analysis

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

## SOS Conversion

<code>cell2sos</code>	Convert a cell array to a second-order sections matrix
<code>sos</code>	Convert a quantized filter to second-order sections form, order, and scale
<code>sos2cell</code>	Convert a second-order sections matrix to a cell array

## Filter Design

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



## Filter Conversion

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



# 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.

- “Least Mean Squares (LMS) Based FIR Adaptive Filters” on page 2-3
- “Recursive Least Squares (RLS) Based FIR Adaptive Filters” on page 2-4
- “Affine Projection (AP) FIR Adaptive Filters” on page 2-4
- “FIR Adaptive Filters in the Frequency Domain (FD)” on page 2-5
- “Lattice Based (L) FIR Adaptive Filters” on page 2-5

**Least Mean Squares (LMS) Based FIR Adaptive Filters**

<b>adaptfilt.algorithm String</b>	<b>Algorithm Used to Generate Filter Coefficients</b>
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

<b>adaptfilt.algorithm String</b>	<b>Algorithm Used to Generate Filter Coefficients</b>
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

<b>adaptfilt.algorithm String</b>	<b>Algorithm Used to Generate Filter Coefficients</b>
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)**

<b>adaptfilt.algorithm String</b>	<b>Algorithm Used to Generate Filter Coefficients</b>
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**

<b>adaptfilt.algorithm String</b>	<b>Algorithm Used to Generate Filter Coefficients</b>
adaptfilt.gal	Use the gradient adaptive lattice filter adaptation algorithm
adaptfilt.lsl	Use the least squares lattice adaptation algorithm
adaptfilt.qrdsl	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
<code>adaptfilt/coefficients</code>	Return the instantaneous adaptive filter coefficients
<code>adaptfilt/filter</code>	Apply an <code>adaptfilt</code> object to your signal
<code>adaptfilt/freqz</code>	Plot the instantaneous adaptive filter frequency response
<code>adaptfilt/grpdelay</code>	Plot the instantaneous adaptive filter group delay
<code>adaptfilt/impz</code>	Plot the instantaneous adaptive filter impulse response.
<code>adaptfilt/info</code>	Return the adaptive filter information.
<code>adaptfilt/isfir</code>	Test whether an adaptive filter is an finite impulse response (FIR) filters.
<code>adaptfilt/islinphase</code>	Test whether an adaptive filter is linear phase
<code>adaptfilt/ismaxphase</code>	Test whether an adaptive filter is maximum phase
<code>adaptfilt/isminphase</code>	Test whether an adaptive filter is minimum phase
<code>adaptfilt/isreal</code>	True whether an adaptive filter has real coefficients
<code>adaptfilt/isstable</code>	Test whether an adaptive filter is stable
<code>adaptfilt/maxstep</code>	Return the maximum step size for an adaptive filter
<code>adaptfilt/msepred</code>	Return the predicted mean square error
<code>adaptfilt/msesim</code>	Return the measured mean square error via simulation.

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

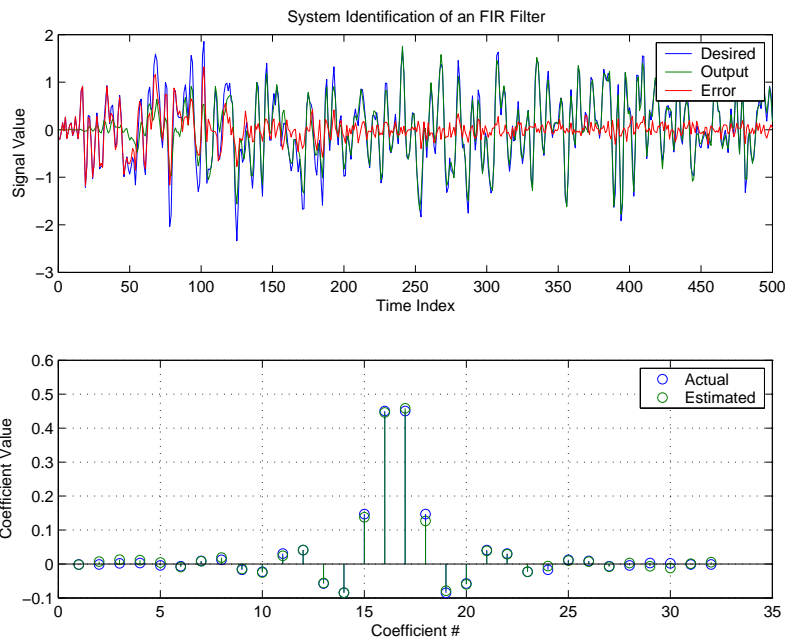
## Examples

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.adjlims

---

**Purpose** FIR adaptive filter that uses adjoint LMS algorithm

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

**Description** `ha = adaptfilt.adjlims(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 `adjlims` 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 <code>l</code> 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 <code>l</code> entries in <code>coefficients</code> . Updated filter coefficients are returned in <code>coefficients</code> when you use <code>s</code> 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	<code>pathcoeffs</code> 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 <code>length(errstates)</code>	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 <code>adaptfilt.adjlims</code> 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. <code>PersistentMemory</code> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to <code>false</code> .

## 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
```

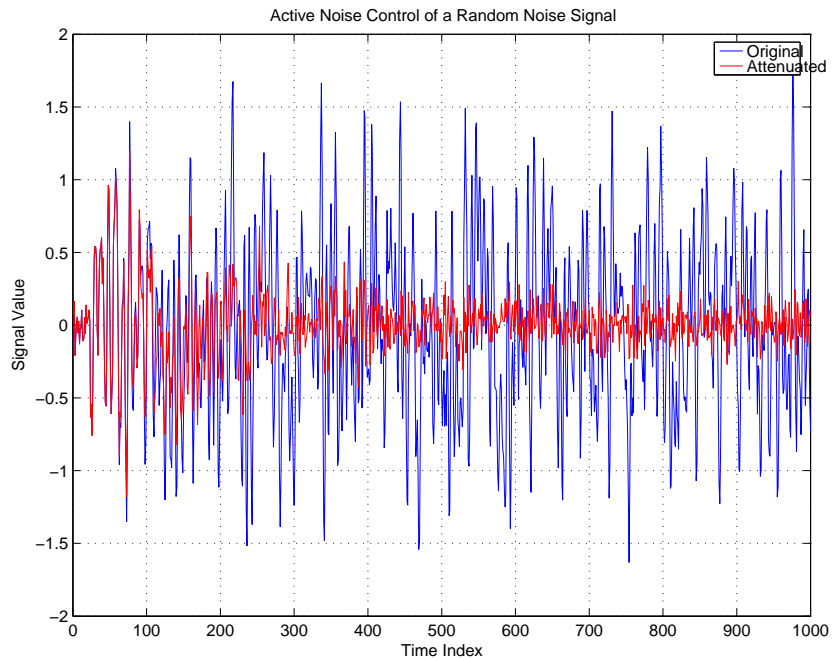


```

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 = adptfilt.adj1ms(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**

`adptfilt.d1ms`, `adptfilt.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(1,step,projectord,offset,coeffs,states,...  
errstates,epsstates)`

**Description**

`ha = adaptfilt.ap(1,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`.

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

<b>Input Argument</b>	<b>Description</b>
coeffs	Vector containing the initial filter coefficients. It must be a length 1 vector, the number of filter coefficients. coeffs defaults to length 1 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 $(1 + \text{projectord} - 2)$ .
errstates	Vector of the adaptive filter error states. errstates defaults to a zero vector with length equal to $(\text{projectord} - 1)$ .
epsstates	Vector of the epsilon values of the adaptive filter. epsstates defaults to a vector of zeros with $(\text{projectord} - 1)$ 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.

<b>Name</b>	<b>Range</b>	<b>Description</b>
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

<b>Name</b>	<b>Range</b>	<b>Description</b>
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 1 vector, the number of filter coefficients. coeffs defaults to length 1 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 $(1 + \text{projectord} - 2)$ .
ErrorStates	Vector of elements	Vector of the adaptive filter error states. errstates defaults to a zero vector with length equal to $(\text{projectord} - 1)$ .
EpsilonStates	Vector of elements	Vector of the epsilon values of the adaptive filter. epsstates defaults to a vector of zeros with $(\text{projectord} - 1)$ 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 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');

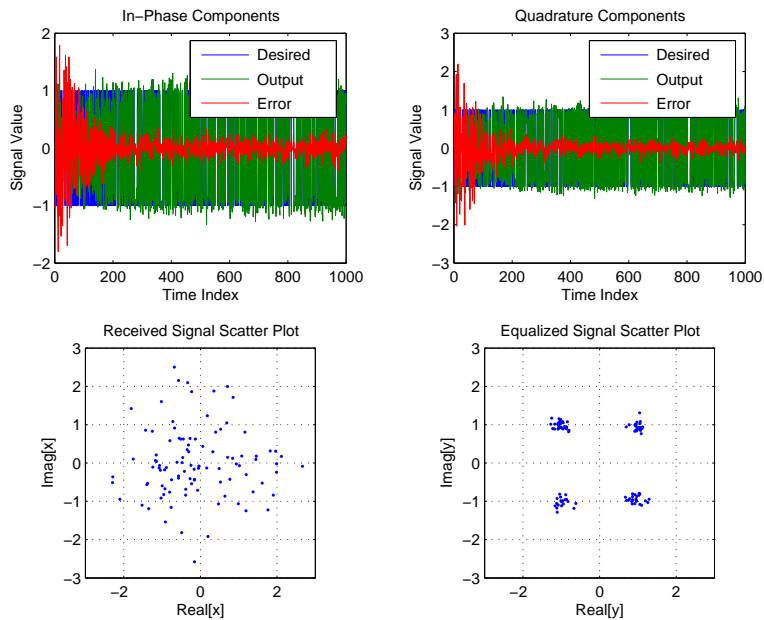
```

```

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**

`mssesim`

## 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(1,step,projectord,offset,coeffs,states, ...errstates,epsstates)`

**Description** `ha = adaptfilt.apru(1,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
<code>1</code>	Adaptive filter length (the number of coefficients or taps). It must be a positive integer. <code>1</code> defaults to 10.
<code>step</code>	Affine projection step size. This is a scalar that should be a value between zero and one. Setting <code>step</code> equal to one provides the fastest convergence during adaptation. <code>step</code> defaults to 1.
<code>projectord</code>	Projection order of the affine projection algorithm. <code>projectord</code> defines the size of the input signal covariance matrix and defaults to two.
<code>offset</code>	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. <code>offset</code> should be positive. <code>offset</code> defaults to one.
<code>coeffs</code>	Vector containing the initial filter coefficients. It must be a length <code>1</code> vector, the number of filter coefficients. <code>coeffs</code> defaults to length <code>1</code> vector of zeros when you do not provide the argument for input.

<b>Input Argument</b>	<b>Description</b>
states	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .
errstates	Vector of the adaptive filter error states. errstates defaults to a zero vector with length equal to $(\text{projectord} - 1)$ .
epsstates	Vector of the epsilon values of the adaptive filter. epsstates defaults to a vector of zeros with $(\text{projectord} - 1)$ 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.

<b>Name</b>	<b>Range</b>	<b>Description</b>
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. <code>ProjectionOrder</code> 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 1 vector, the number of filter coefficients. <code>coeffs</code> defaults to length 1 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. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .
ErrorStates	Vector of elements	Vector of the adaptive filter error states. <code>errstates</code> defaults to a zero vector with length equal to $(\text{projectord} - 1)$ .
EpsilonStates	Vector of elements	Vector of the epsilon values of the adaptive filter. <code>epsstates</code> defaults to a vector of zeros with $(\text{projectord} - 1)$ 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
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');

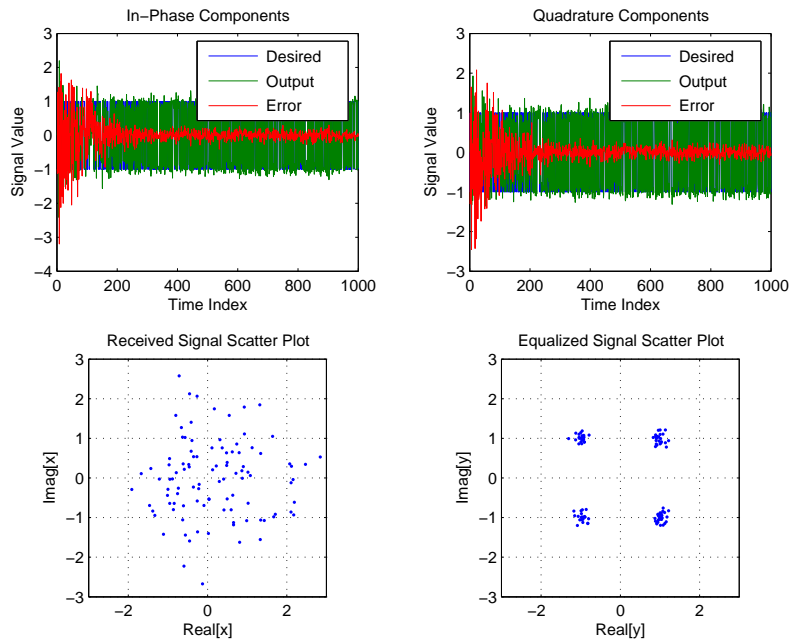
```

```

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(1,step,projectord,offset,coeffs,states)`

**Description** `ha = adaptfilt.bap(1,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
1	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. 1 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.

<b>Input Argument</b>	<b>Description</b>
coeffs	Vector containing the initial filter coefficients. It must be a length 1 vector, the number of filter coefficients. <code>coeffs</code> defaults to length 1 vector of zeros when you do not provide the argument for input.
states	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .

## 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.

<b>Name</b>	<b>Range</b>	<b>Description</b>
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. <code>ProjectionOrder</code> defines the size of the input signal covariance matrix and defaults to two.
OffsetCov	Matrix of values	Contains the offset covariance matrix



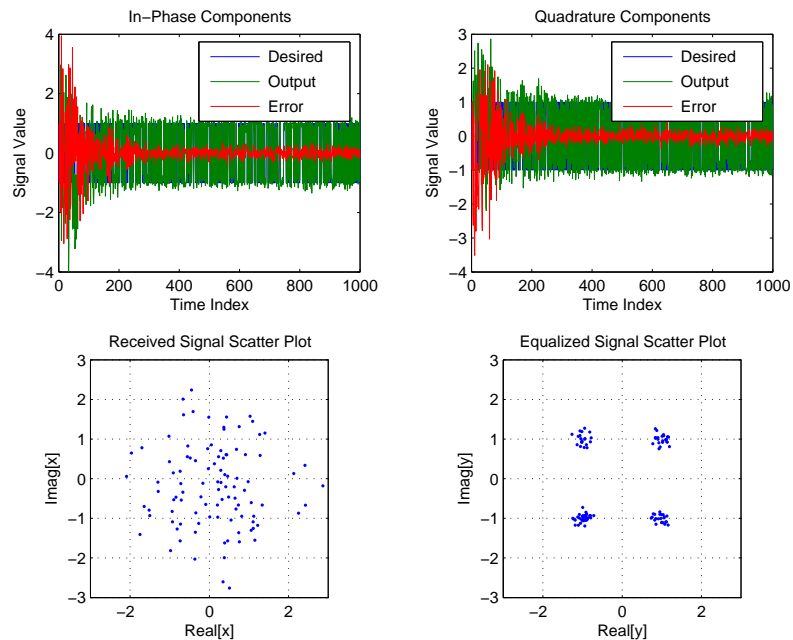
Name	Range	Description
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length 1 vector, the number of filter coefficients. <code>coeffs</code> defaults to length 1 vector of zeros when you do not provide the argument for input.
States	Vector of elements, data type double	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .
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. <code>PersistentMemory</code> 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; % delay
```

```
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 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

---

<b>Purpose</b>	FIR adaptive filter that uses BLMS
<b>Syntax</b>	<code>ha = adaptfilt.blms(l,step,leakage,blocklen,coeffs,states)</code>
<b>Description</b>	<p><code>ha = adaptfilt.blms(l,step,leakage,blocklen,coeffs,states)</code> constructs an FIR block LMS adaptive filter <code>ha</code>, where <code>l</code> is the adaptive filter length (the number of coefficients or taps) and must be a positive integer. <code>l</code> defaults to 10.</p> <p><code>step</code> is the block LMS step size. You must set <code>step</code> to a nonnegative scalar. You can use function <code>maxstep</code> to determine a reasonable range of step size values for the signals being processed. When unspecified, <code>step</code> defaults to 0.</p> <p><code>leakage</code> is the block LMS leakage factor. It must be a scalar between 0 and 1. If you set <code>leakage</code> to be less than one, you implement the leaky block LMS algorithm. <code>leakage</code> defaults to 1 specifying no leakage in the adapting algorithm.</p> <p><code>blocklen</code> is the block length used. It must be a positive integer and the signal vectors <code>d</code> and <code>x</code> should be divisible by <code>blocklen</code>. Larger block lengths result in faster per-sample execution times but with poor adaptation characteristics. When you choose <code>blocklen</code> such that <code>blocklen + length(coeffs)</code> is a power of 2, use <code>adaptfilt.blmsfft</code>. <code>blocklen</code> defaults to 1.</p> <p><code>coeffs</code> is a vector of initial filter coefficients. it must be a length <code>l</code> vector. <code>coeffs</code> defaults to length <code>l</code> vector of zeros.</p> <p><code>states</code> contains a vector of your initial filter states. It must be a length <code>l</code> vector and defaults to a length <code>l</code> vector of zeros when you do not include it in your calling function.</p>
<b>Properties</b>	<p>In the syntax for creating the <code>adaptfilt</code> 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.</p>

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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 1 vector where 1 is the number of filter coefficients. <code>coeffs</code> defaults to length 1 vector of zeros when you do not provide the argument for input.
States	Vector of elements	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros which has length equal to 1
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 1	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 <code>maxstep</code> 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. <code>PersistentMemory</code> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to <code>false</code> .

## 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.

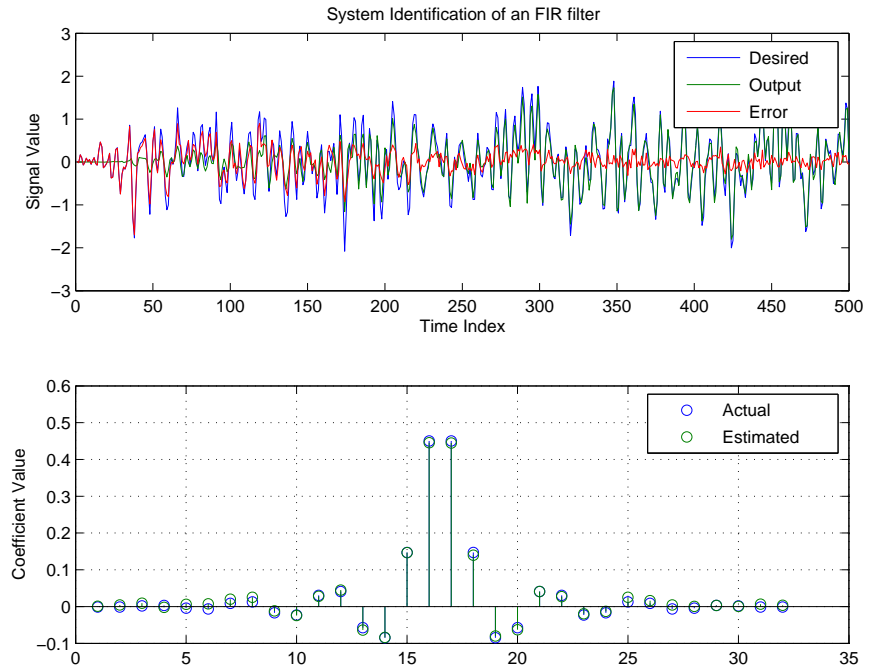
```
x = randn(1,500);           % Input to the filter
b = fir1(31,0.5);          % FIR system to be identified
no = 0.1*randn(1,500);     % Observation noise signal
d = filter(b,1,x)+no;      % Desired signal
mu = 0.008;                % Block LMS step size
n = 5;                      % Block length
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

$$\text{blocklen} + \text{length}(\text{coeffs})$$

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 1. 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.

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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 1 vector where 1 is the number of filter coefficients. <code>coefficients</code> defaults to length 1 vector of zeros when you do not provide the argument for input.
States	Vector of elements of length 1	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros which has length equal to 1
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 1	Size of the blocks of data processed in each iteration

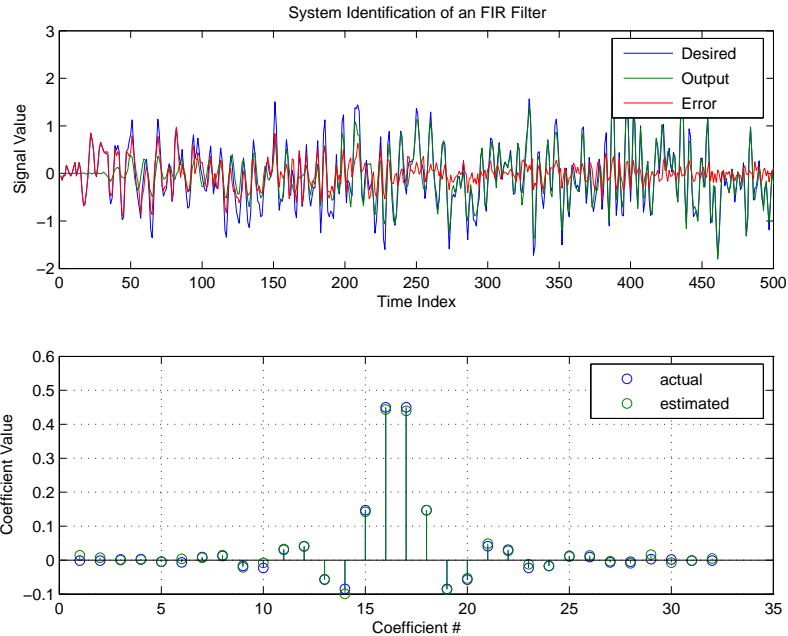
Property	Default Value	Description
StepSize	0.1	Sets the block LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution. Use <code>maxstep</code> to determine the maximum usable step size.
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. <code>PersistentMemory</code> 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'); grid on;  
xlabel('Coefficient #'); ylabel('Coefficient Value');
```



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(1,step,leakage,delay,errstates,coeffs, ...states)`

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

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

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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 <code>l</code> vector where <code>l</code> is the number of filter coefficients. <code>coeffs</code> defaults to length <code>l</code> 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 <code>l</code> entries in <code>coeffs</code> .
Delay	1	Specifies the update delay for the adaptive algorithm.

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .

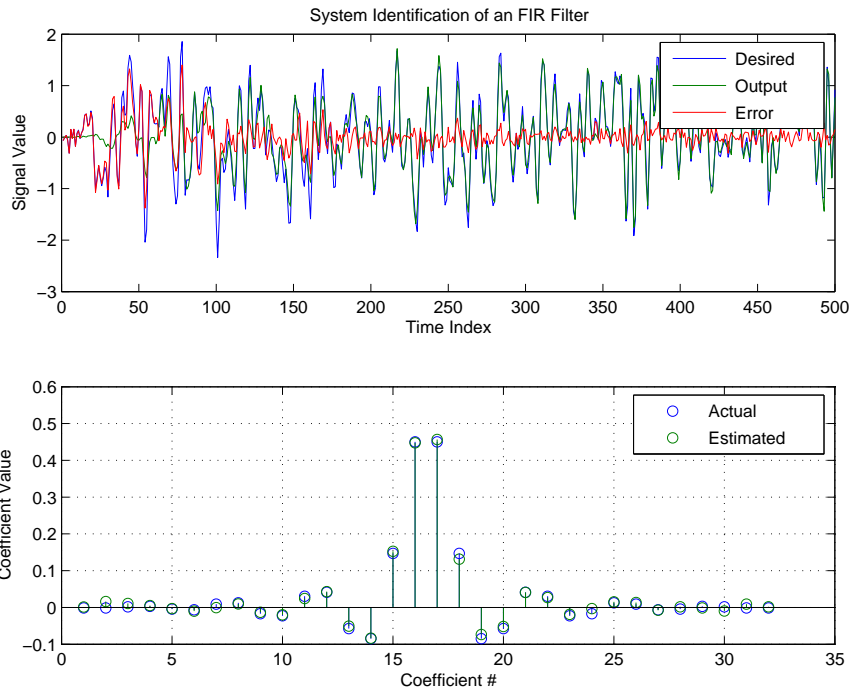
## Example

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'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

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.adjdlms`, `adaptfilt.filtxdlms`, `adaptfilt.dlms`

**References**

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

# adaptfilt.fdaf

---

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

<b>Input Argument</b>	<b>Description</b>
<code>blocklen</code>	Block length for the coefficient updates. This must be a positive integer. For faster execution, ( <code>blocklen + 1</code> ) should be a power of two. <code>blocklen</code> defaults to 1.
<code>offset</code>	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. <code>offset</code> defaults to zero.
<code>coeffs</code>	Initial time-domain coefficients of the adaptive filter. <code>coeff</code> should be a length <code>l</code> 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 <code>blocklen</code> .
<code>states</code>	The adaptive filter states. <code>states</code> defaults to a zero vector that has length equal to <code>l</code> .

## 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.

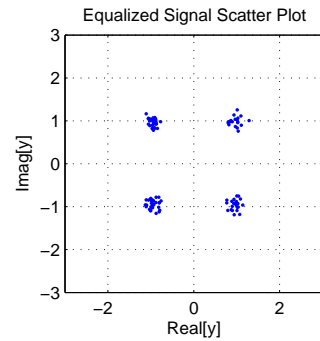
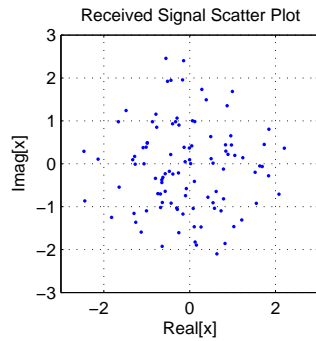
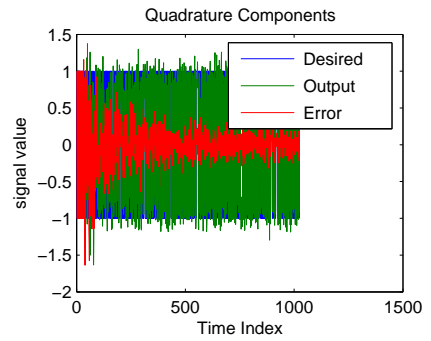
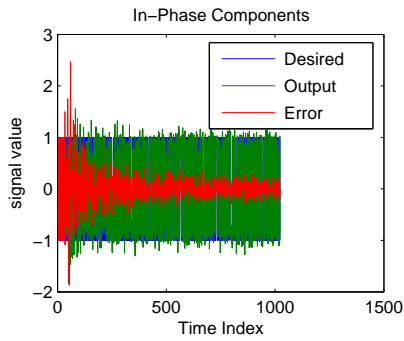
Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
AvgFactor	(0, 1]	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. Same as the input argument <code>lambda</code> .
BlockLength	Any integer	Block length for the coefficient updates. This must be a positive integer. For faster execution, $(\text{blocklen} + 1)$ should be a power of two. <code>blocklen</code> defaults to 1.
FFTCoefficients		Stores the discrete Fourier transform of the filter coefficients in <code>coeffs</code> .
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. <code>leakage</code> defaults to 1 — no leakage provided in the algorithm.

Name	Range	Description
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. <code>offset</code> 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. <code>PersistentMemory</code> 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*1 elements, each initialized with the value <code>delta</code> from the input arguments. As you filter data, <code>Power</code> 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)); %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

# adaptfilt.filtx1ms

---

**Purpose** FIR adaptive filter that uses filtered-x LMS

**Syntax** `ha = adaptfilt.filtx1ms(1,step,leakage,pathcoeffs,pathest,...errstates,pstates,coeffs,states)`

**Description** `ha = adaptfilt.filtx1ms(1,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.filtx1ms`.

Input Argument	Description
<code>1</code>	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. <code>1</code> defaults to 10.
<code>step</code>	Filtered LMS step size. it must be a nonnegative scalar. <code>step</code> defaults to 0.1.
<code>leakage</code>	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 <code>adaptfilt.filtx1ms</code> is implemented. <code>leakage</code> defaults to 1 (no leakage).
<code>pathcoeffs</code>	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.
<code>pathest</code>	is the estimate of the secondary path filter model. <code>pathest</code> defaults to the values in <code>pathcoeffs</code> .
<code>fstates</code>	is a vector of filtered input states of the adaptive filter. <code>fstates</code> defaults to a zero vector of length equal to $(1 - 1)$ .



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

## 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 1 vector where 1 is the number of filter coefficients. coeffs defaults to length 1 vector of zeros when you do not provide the argument for input.

Property	Default Value	Description
FilteredInputStates	1-1	Vector of filtered input states with length equal to 1 - 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. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$
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
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

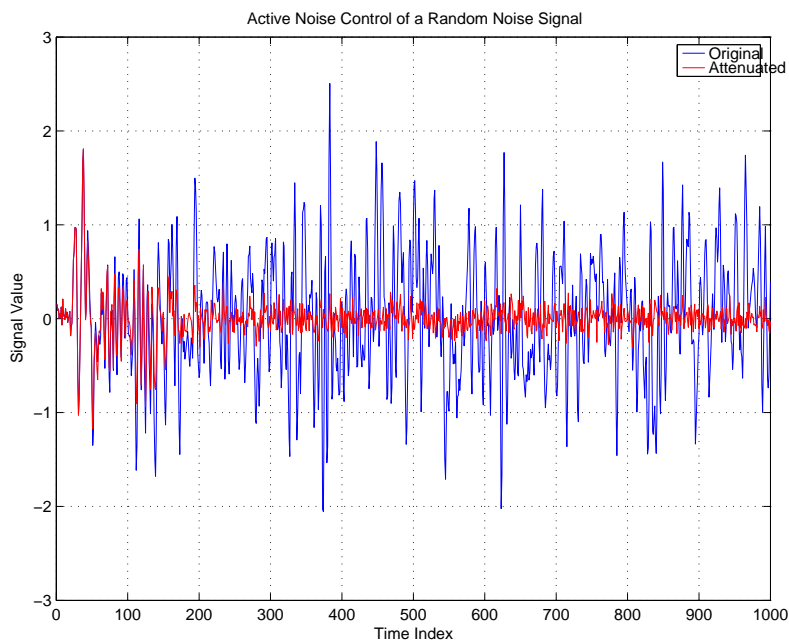
Property	Default Value	Description
		adaptive filter converges to the filter solution.

## 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
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

Kuo, S.M., and Morgan, D.R. *Active Noise Control Systems: Algorithms and DSP Implementations*, New York, N.Y: John Wiley & Sons, 1996.

Widrow, B., and Stearns, S.D. *Adaptive Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1985.

**Purpose** Fast transversal LMS adaptive filter

**Syntax** `ha = adaptfilt.ftf(1,lambda,delta,gamma,gstates,coeffs,states)`

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

## 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 <code>l</code> vector where <code>l</code> is the number of filter coefficients. <code>coeffs</code> defaults to length <code>l</code> vector of zeros when you do not provide the argument for input.
ConversionFactor		Conversion factor. Called <code>gamma</code> when it is an input argument, it defaults to the matrix <code>[1 -1]</code> 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 $(1-0.5/l, 1]$ . <code>lambda</code> 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. <code>delta</code> 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. <code>PersistentMemory</code> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to <code>false</code> .
States	Vector of elements, data type double	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .

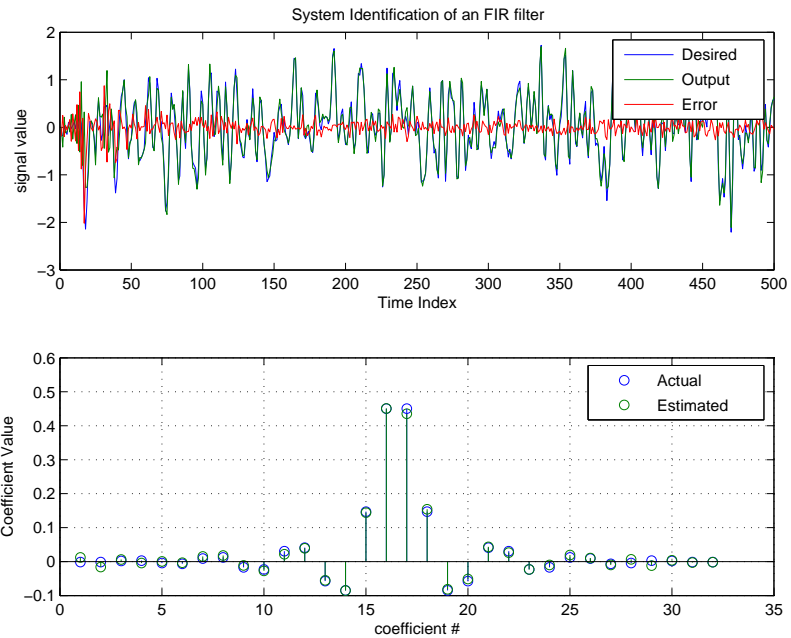
## Examples

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'); grid on;
xlabel('coefficient #'); ylabel('Coefficient Value');
```

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.

# adaptfilt.gal

---

**Purpose** FIR adaptive filter that uses gradient lattice

**Syntax** `ha = adaptfilt.gal(1,step,leakage,offset,rstep,delta,lambda,...rcoeffs,coeffs,states)`

**Description** `ha = adaptfilt.gal(1,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
<code>1</code>	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. <code>1</code> defaults to 10.
<code>step</code>	Joint process step size of the adaptive filter. This scalar should be a value between zero and one. <code>step</code> defaults to 0.
<code>leakage</code>	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. <code>leakage</code> defaults to 1 (no leakage).
<code>offset</code>	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 <code>offset</code> is useful to avoid divide-by-near-zero conditions when the input signal amplitude becomes very small. <code>offset</code> defaults to 1.

Input Argument	Description
<code>rstep</code>	Reflection process step size of the adaptive filter. This scalar should be a value between zero and one. <code>rstep</code> defaults to <code>step</code> .
<code>delta</code>	Initial common value of the forward and backward prediction error powers. It should be a positive value. 0.1 is the default value for <code>delta</code> .
<code>lambda</code>	Specifies the averaging factor used to compute the exponentially windowed forward and backward prediction error powers for the coefficient updates. <code>lambda</code> should lie in the range (0, 1]. <code>lambda</code> defaults to the value (1 - <code>step</code> ).
<code>rcoeffs</code>	Vector of initial reflection coefficients. It should be a length (l-1) vector. <code>rcoeffs</code> defaults to a zero vector of length (l-1).
<code>coeffs</code>	Vector of initial joint process filter coefficients. It must be a length l vector. <code>coeffs</code> defaults to a length l vector of zeros.
<code>states</code>	Vector of the backward prediction error states of the adaptive filter. <code>states</code> 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
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 <code>lambda</code> .
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 <code>1</code> vector where <code>1</code> is the number of filter coefficients. <code>coeffs</code> defaults to length <code>1</code> 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. <code>leakage</code> 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. <code>offset</code> 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. <code>PersistentMemory</code> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to <code>false</code> .
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. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .
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.

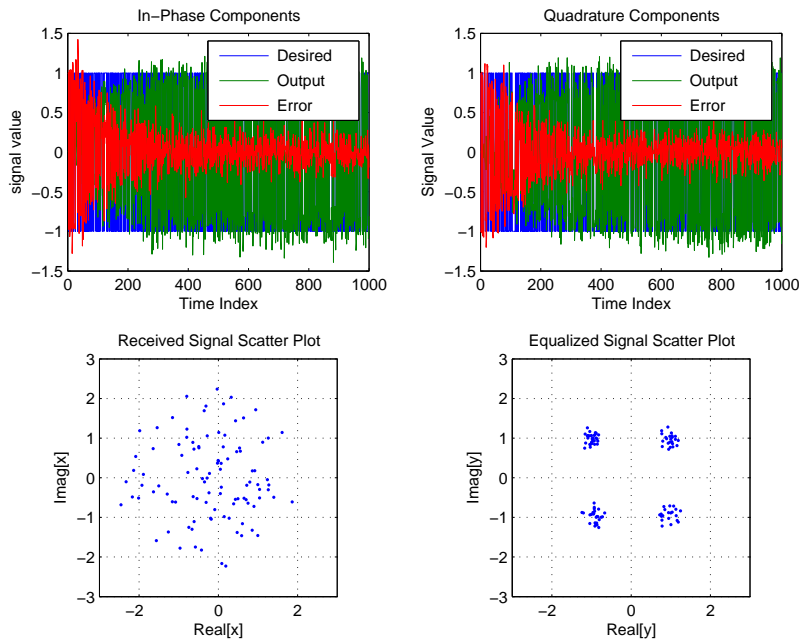
```

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)); % 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.tdafdfst

**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(1,lambda,sqrtinvcov,coeffs,states)`

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

<b>Name</b>	<b>Range</b>	<b>Description</b>
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 1 vector where 1 is the number of filter coefficients. <code>coeffs</code> defaults to length 1 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 <code>lambda</code> . 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.

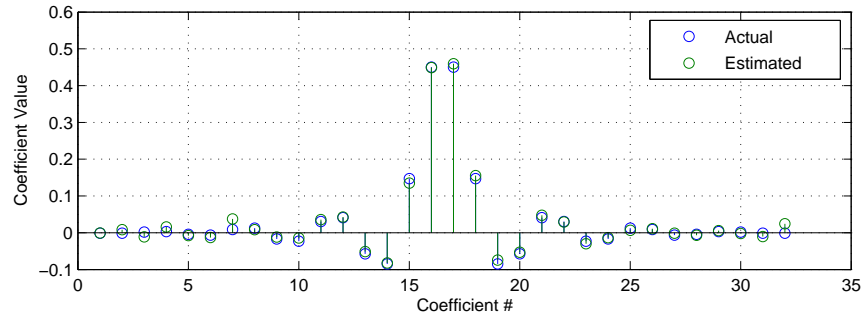
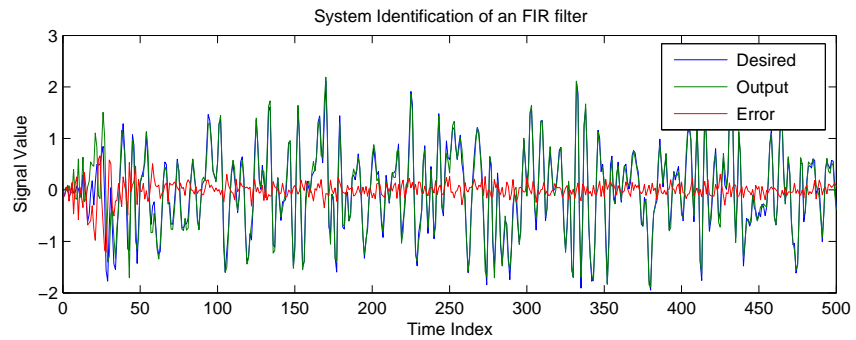
Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory 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'); grid on;  
xlabel('Coefficient #'); ylabel('Coefficient Value');
```



## See Also

[adaptfilt.hswrls](#), [adaptfilt.qrdrls](#), [adaptfilt.rls](#)

**Purpose** FIR adaptive filter that uses householder sliding window RLS

**Syntax** `ha = adaptfilt.hswrls(1,lambda,sqrtinvcov,swblocklen,dstates,coeffs,states)`

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

# adaptfilt.hswrls

---

Input Argument	Description
coeffs	Vector of initial filter coefficients. It must be a length 1 vector. coeffs defaults to being a length 1 vector of zeros.
states	Vector of initial filter states. It must be a length $(1 + \text{swblocklen} - 2)$ vector. states defaults to a length $(1 + \text{swblocklen} - 2)$ 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 1 vector where 1 is the number of filter coefficients. coeffs defaults to length 1 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 $(\text{swblocklen} - 1)$ .

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	(1,1) 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	1-by-1 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. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .
SwBlockLength	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. <code>swblocklen</code> defaults to 16.

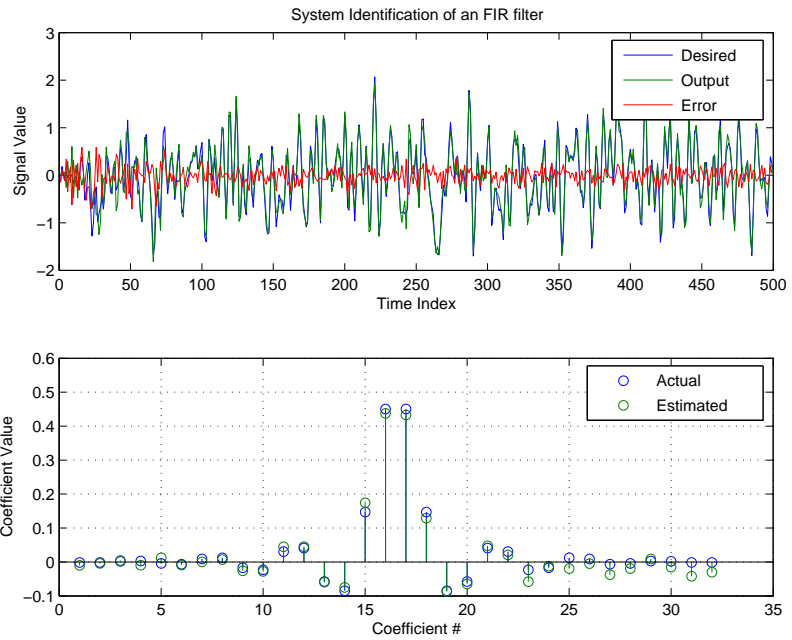
## Examples

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'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

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.hrls`, `adaptfilt.qdr1s`, `adaptfilt.rls`

# adaptfilt.lms

---

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

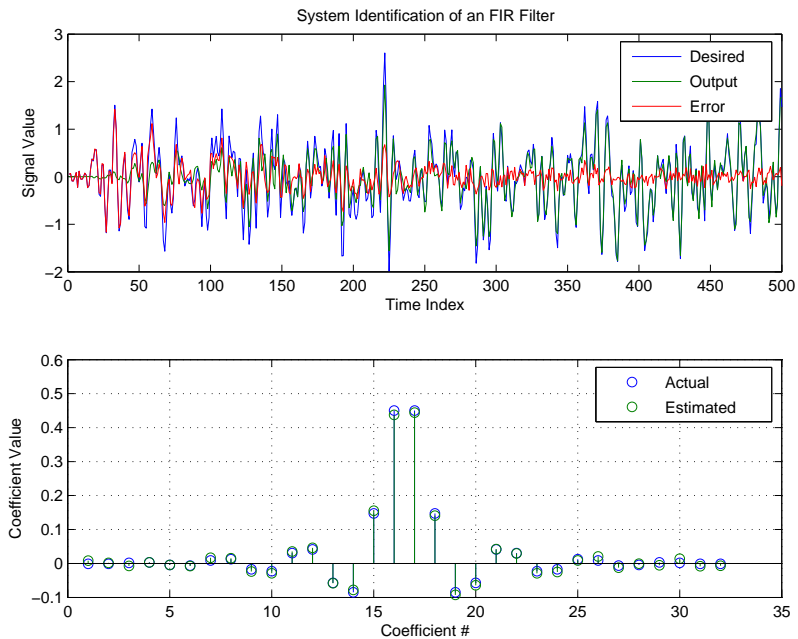
```
x = randn(1,500); % Input to the filter
b = fir1(31,0.5); % FIR system to be identified
n = 0.1*randn(1,500); % Observation noise signal
d = filter(b,1,x)+n; % Desired signal
mu = 0.008; % LMS step size.
ha = adaptfilt.lms(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
```

```

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(1,lambda,delta,coeffs,states)`

**Description** `ha = adaptfilt.lsl(1,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
<code>1</code>	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. <code>L</code> defaults to 10.
<code>lambda</code>	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. <code>lambda</code> defaults to 1. <code>lambda = 1</code> denotes infinite memory while adapting to find the new filter.
<code>delta</code>	Soft-constrained initialization factor in the least squares lattice algorithm. It should be positive. <code>delta</code> defaults to 1.
<code>coeffs</code>	Vector of initial joint process filter coefficients. It must be a length <code>1</code> vector. <code>coeffs</code> defaults to a length <code>1</code> vector of all zeros.
<code>states</code>	Vector of the backward prediction error states of the adaptive filter. <code>states</code> defaults to a length <code>1</code> 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

# adaptfilt.lsl

---

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 <code>l</code> vector where <code>l</code> is the number of filter coefficients. <code>coeffs</code> defaults to length <code>l</code> 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 <code>forgetting factor = 1</code> denotes infinite memory while adapting to find the new filter. Note that this is the <code>lambda</code> input argument.

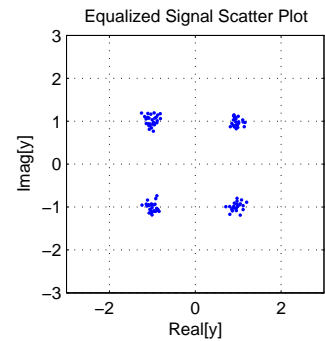
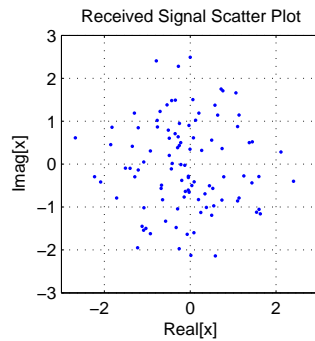
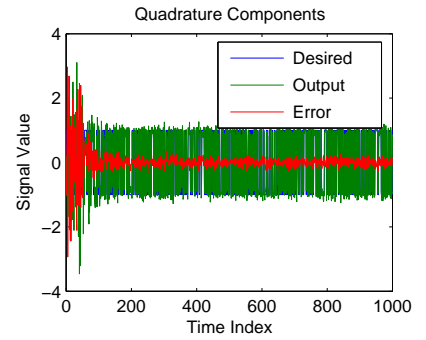
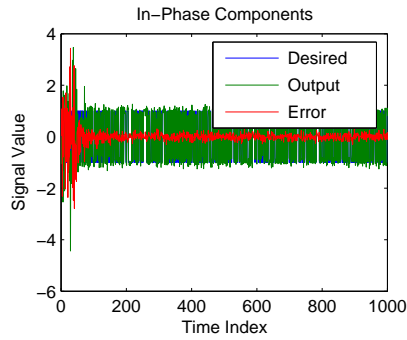


Name	Range	Description
FwdPrediction		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
InitFactor		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. <code>delta</code> 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. <code>PersistentMemory</code> 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. <code>states</code> defaults to a vector of zeros which has length equal to 1.

## 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)); % 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; % 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'); grid on;
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');
```



## See Also

`adaptfilt.qrdls1`, `adaptfilt.gal`, `adaptfilt.ftf`, `adaptfilt.rls`

## References

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

# adaptfilt.nlms

---

**Purpose** FIR adaptive filter that uses NLMS

**Syntax** `ha = adaptfilt.nlms(1,step,leakage,offset,coeffs,states)`

**Description** `ha = adaptfilt.nlms(1,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
<code>1</code>	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. <code>1</code> defaults to 10.
<code>step</code>	NLMS step size. It must be a scalar between 0 and 2. Setting this step size value to one provides the fastest convergence. <code>step</code> defaults to 1.
<code>leakage</code>	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. <code>leakage</code> defaults to 1 (no leakage).
<code>offset</code>	Specifies an optional offset for the denominator of the step size normalization term. You must specify <code>offset</code> 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, <code>offset</code> defaults to zero.

Input Argument	Description
coeffs	Vector composed of your initial filter coefficients. Enter a length 1 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 1-1. states defaults to a length 1-1 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 1 vector where 1 is the number of filter coefficients. coeffs defaults to length 1 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

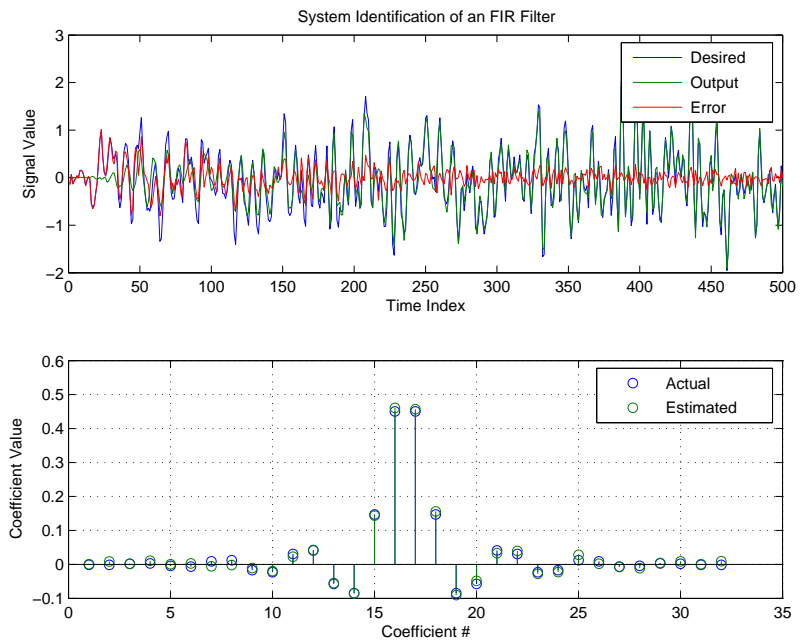
Property	Range	Property Description
Leakage	0 to 1	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. <code>leakage</code> defaults to 1 (no leakage).
Offset	0 or greater	Specifies an optional offset for the denominator of the step size normalization term. You must specify <code>offset</code> 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, <code>offset</code> 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. <code>PersistentMemory</code> 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 <code>false</code> .

Property	Range	Property Description
States	Vector of elements, data type double	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros which has length equal to $(1 - 1)$ .
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. <code>step</code> 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);
subplot(2,1,1);
plot(1:500,[d;y;e]);
legend('Desired','Output','Error');
title('System Identification of FIR Filter');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2);
stem([b', ha.coefficients']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```



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(1,step,leakage,delta,lambda,blocklen,offset,coeffs,states)`

**Description** `ha = adaptfilt.pbfdaf(1,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
<code>1</code>	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. <code>L</code> defaults to 10.
<code>step</code>	Step size of the adaptive filter. This is a scalar and should lie in the range $(0,1]$ . <code>step</code> defaults to 1.
<code>leakage</code>	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. <code>leakage</code> defaults to 1— no leakage.
<code>delta</code>	Initial common value of all of the FFT input signal powers. Its initial value should be positive. <code>delta</code> defaults to 1.
<code>lambda</code>	Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. <code>lambda</code> should lie in the range $(0,1]$ . <code>lambda</code> defaults to 0.9.

Input Argument	Description
<code>blocklen</code>	Block length for the coefficient updates. This must be a positive integer such that $(1/\text{blocklen})$ is also an integer. For faster execution, <code>blocklen</code> should be a power of two. <code>blocklen</code> defaults to two.
<code>offset</code>	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. <code>offset</code> defaults to zero.
<code>coeffs</code>	Initial time-domain coefficients of the adaptive filter. It should be a vector of length 1. The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
<code>states</code>	Specifies the filter initial conditions. <code>states</code> defaults to a zero vector of length 1.

## 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.

<b>Name</b>	<b>Range</b>	<b>Description</b>
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
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 (1/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.

# adaptfilt.pbfdaf

---

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, a leaky version of the PBFDAF algorithm is implemented. Leakage defaults to 1 — no leakage.
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.

Name	Range	Description
Power		A vector of 2*1 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.

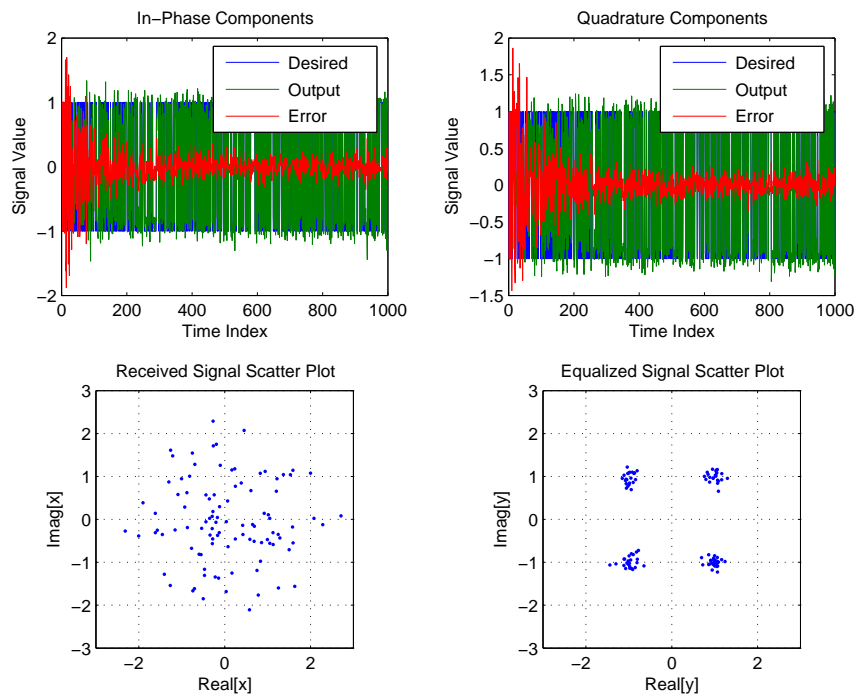
```

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 Borrillo, 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(1,step,leakage,delta,lambda,blocklen,...offset,coeffs,states)`

**Description** `ha = adaptfilt.pbufdaf(1,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
<code>l</code>	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. <code>L</code> defaults to 10.
<code>step</code>	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. <code>step</code> defaults to 1.
<code>leakage</code>	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. <code>leakage</code> defaults to 1 — no leakage.
<code>delta</code>	Initial common value of all of the FFT input signal powers. Its initial value should be positive. <code>delta</code> defaults to 1.
<code>lambda</code>	Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. <code>lambda</code> should lie in the range (0,1]. <code>lambda</code> defaults to 0.9.



Input Argument	Description
blocklen	Block length for the coefficient updates. This must be a positive integer such that $(1/\text{blocklen})$ is also an integer. For faster execution, <code>blocklen</code> should be a power of two. <code>blocklen</code> 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. <code>offset</code> defaults to zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a vector of length <code>l</code> . The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
states	Specifies the filter initial conditions. <code>states</code> defaults to a zero vector of length <code>l</code> .

## 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.

# adaptfilt.pbufdaf

---

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
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 (1/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.

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, a leaky version of the PBFDAF algorithm is implemented. <b>Leakage</b> defaults to 1 — no leakage.
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. <b>voffset</b> 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. <b>PersistentMemory</b> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to <b>false</b> .

Name	Range	Description
Power	2*1 element vector	A vector of 2*1 elements, each initialized with the value <code>delta</code> from the input arguments. As you filter data, <code>Power</code> 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]. <code>step</code> 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.

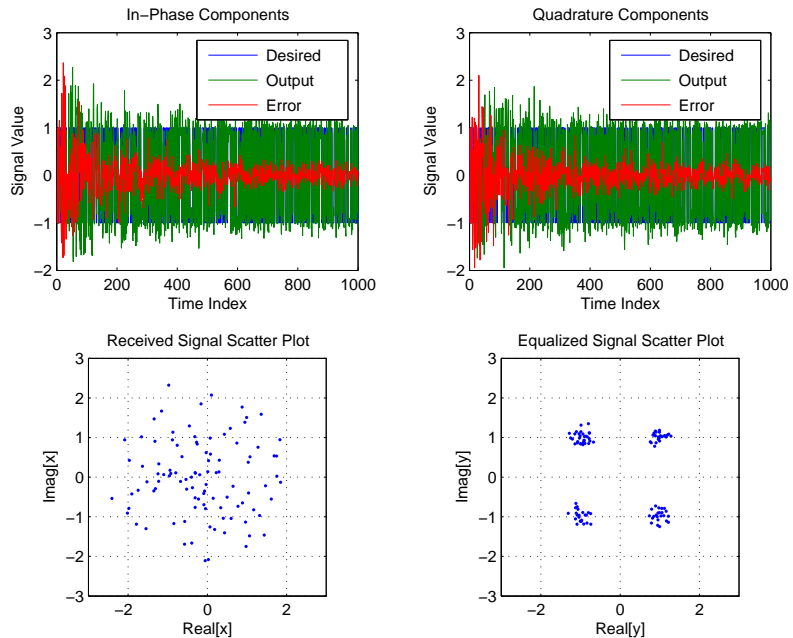
```
D = 16;          % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7];    % Denominator coefficients of channel
ntr= 1000;      % Number of iterations
s = sign(randn(1,ntr+D))+j*sign(randn(1,ntr+D)); % Baseband QPSK signal
n = 0.1*(randn(1,ntr+D) + 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 Borrillo, 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

**Purpose** Adaptive filter that uses QR-decomposition-based LSL

**Syntax** `ha = adaptfilt.qrdls1(1,lambda,delta,coeffs,states)`

**Description** `ha = adaptfilt.qrdls1(1,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.qrdls1`.

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

**Properties** Since your `adaptfilt.qrdls1` 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.qrdls1` 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. <code>coeffs</code> 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 <code>forgetting factor = 1</code> denotes infinite memory while adapting to find the new filter. Note that this is the <code>lambda</code> input argument.



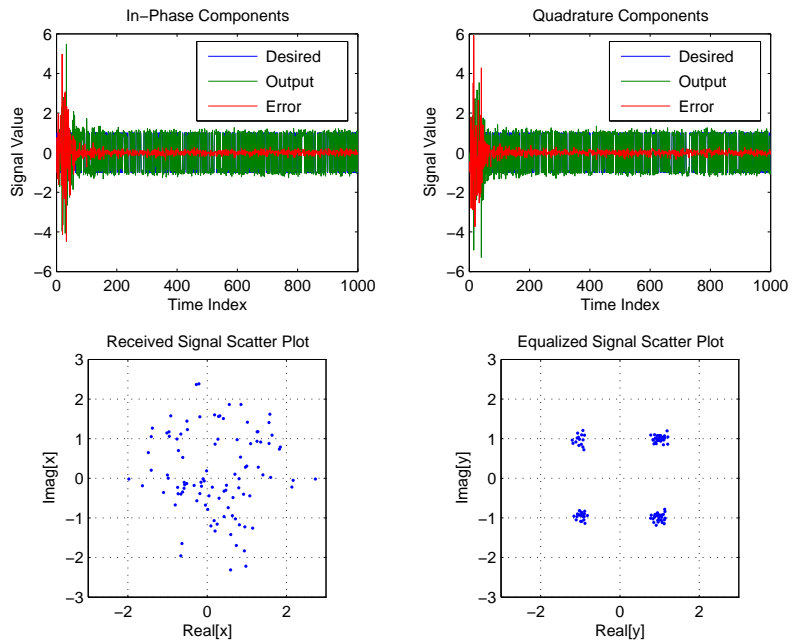
Name	Range	Description
FwdPrediction		Returns the predicted samples generated during adaptation in the forward direction. Refer to [2] in the bibliography for details about linear prediction.
InitFactor		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. <code>delta</code> 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. <code>PersistentMemory</code> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to <code>false</code> .
States	Vector of elements, data type double	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros which has length equal to $1 - 1$

### 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.qrdls1(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.qrdls`, `adaptfilt.gal`, `adaptfilt.ftf`, `adaptfilt.lsl`

## References

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

# adaptfilt.qdrpls

---

**Purpose** FIR adaptive filter that uses QR-decomposition-based RLS

**Syntax** `ha = adaptfilt.qdrpls(1,lambda,sqrtcov,coeffs,states)`

**Description** `ha = adaptfilt.qdrpls(1,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.qdrpls`.

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

## Properties

Since your `adaptfilt.qdrpls` 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.qdrpls` 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 1	Vector containing the initial filter coefficients. It must be a length 1 vector where 1 is the number of filter coefficients. <code>coeffs</code> defaults to length 1 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 <code>forgetting factor = 1</code> denotes infinite memory while adapting to find the new filter. Note that this is the <code>lambda</code> 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 1	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 (1 + 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

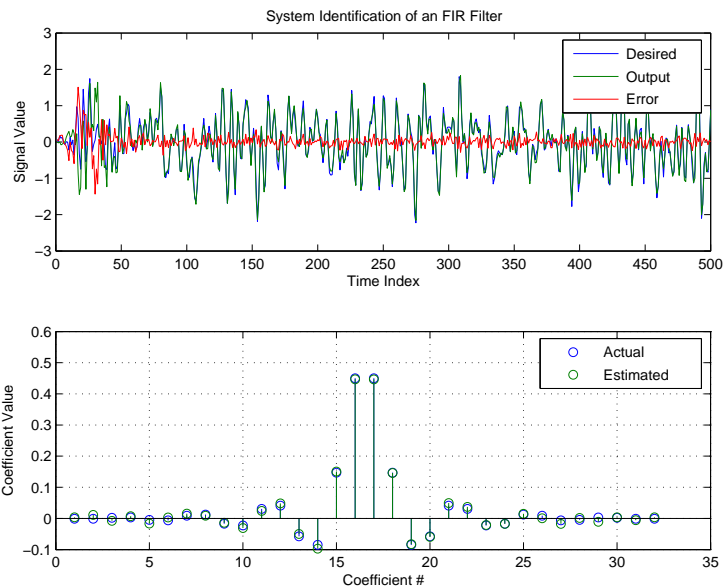
```

```

lam = 0.99;                % RLS forgetting factor
ha = adaptfilt.qrdrls(32,lam,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'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');

```

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](#)

# adaptfilt.rls

---

**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
<code>l</code>	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. <code>l</code> defaults to 10.
<code>lambda</code>	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. <code>lambda</code> defaults to 1.
<code>invcov</code>	Inverse of the input signal covariance matrix. For best performance, you should initialize this matrix to be a positive definite matrix.
<code>coeffs</code>	Vector of initial filter coefficients. it must be a length <code>l</code> vector. <code>coeffs</code> defaults to length <code>l</code> vector with elements equal to zero.
<code>states</code>	Vector of initial filter states for the adaptive filter. It must be a length <code>l-1</code> vector. <code>states</code> defaults to a length <code>l-1</code> 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 1 elements	Vector containing the initial filter coefficients. It must be a length 1 vector where 1 is the number of filter coefficients. <code>coeffs</code> defaults to length 1 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 <code>lambda</code> input argument.
InvCov	Matrix of size 1-by-1	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 (1,1)	Empty when you construct the object, this gets populated after you run the filter.

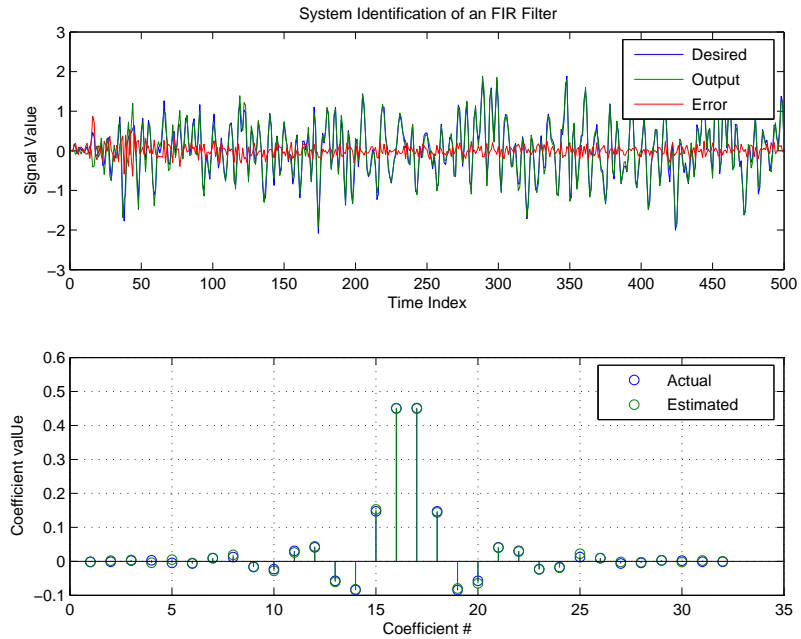
Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory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

## 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](#)

# adaptfilt.sd

---

**Purpose** FIR adaptive filter that uses sign-data algorithm

**Syntax** `ha = adaptfilt.sd(1,step,leakage,coeffs,states)`

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

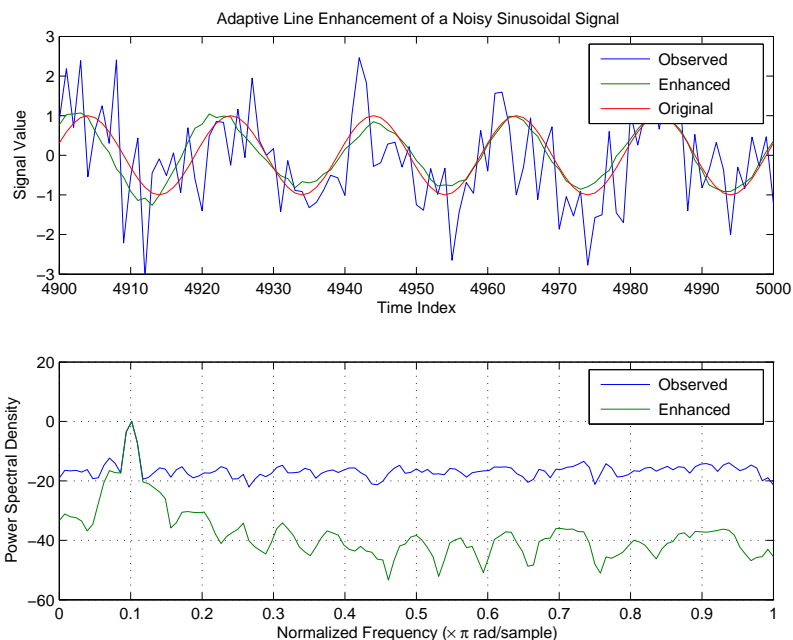
Adaptive line enhancement using a 32-coefficient FIR filter to perform the enhancement. This example runs for 5000 iterations, as you see in property iter.

```

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
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:end-1)]);
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(1,step,leakage,coeffs,states)`

**Description** `ha = adaptfilt.se(1,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
1	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. 1 defaults to 10.
step	SE step size. It must be a nonnegative scalar. You can use <code>maxstep</code> to determine a reasonable range of step size values for the signals being processed. <code>step</code> defaults to 0.1
leakage	Your SE leakage factor. It must be a scalar between 0 and 1. When <code>leakage</code> is less than one, <code>adaptfilt.se</code> implements a leaky SE algorithm. When you omit the <code>leakage</code> 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 1 vector. <code>coeffs</code> defaults to length 1 vector with elements equal to zero.
states	Vector of initial filter states for the adaptive filter. It must be a length 1-1 vector. <code>states</code> defaults to a length 1-1 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	<code>zeros(1,1)</code>	Vector containing the initial filter coefficients. It must be a length 1 vector where 1 is the number of filter coefficients. <code>coeffs</code> defaults to length 1 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(1-1,1)	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 -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.

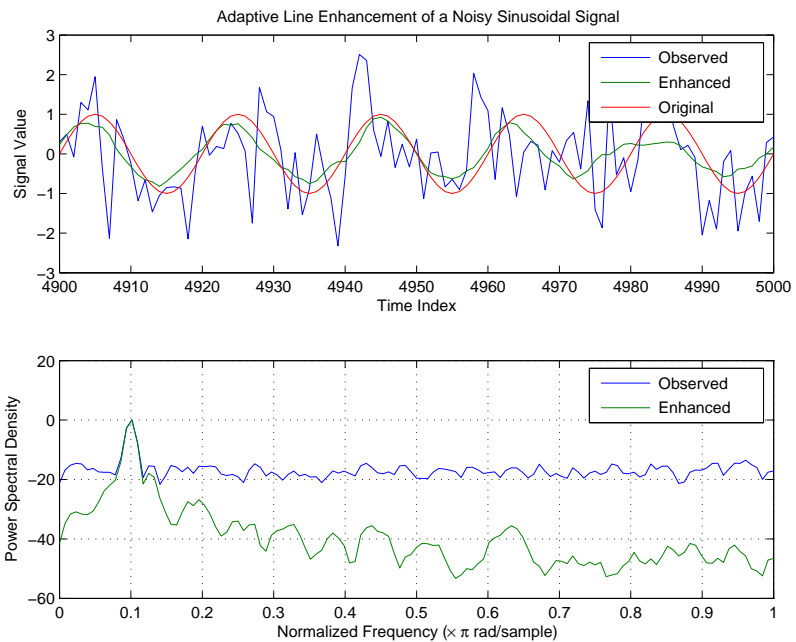
## Examples

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:end-1)]);
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)); ppy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2); plot(om/pi,10*log10([pxx/max(pxx),ppy/max(ppy)]));
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.

# adaptfilt.ss

**Purpose** FIR adaptive filter that uses sign-sign algorithm

**Syntax** `ha = adaptfilt.ss(1,step,leakage,coeffs,states)`

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

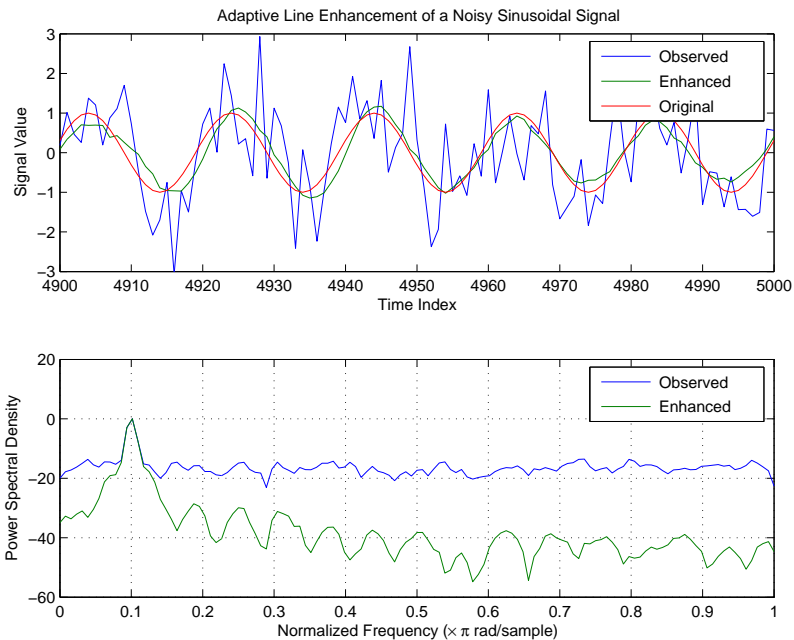
Demonstrating adaptive line enhancement using a 32-coefficient FIR filter provides a good introduction to the sign-sign algorithm.

```

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:end-1)]); 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)); ppy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),ppy/max(ppy)]));
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 least squares

**Syntax** `ha = adaptfilt.swftf(1,delta,blocklen,gamma,gstates, dstates,...coeffs,states)`

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

Input Argument	Description
coeffs	Vector of initial filter coefficients. It must be a length $l$ vector. <code>coeffs</code> defaults to length $l$ vector of all zeros.
states	Vector of initial filter states. <code>states</code> defaults to a zero vector of length equal to $(l + \text{blocklen} - 2)$ .

## Properties

Since your `adaptfilt.swtf` 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.swtf` 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. <code>coeffs</code> defaults to length $l$ vector of zeros when you do not provide the argument for input.

<b>Name</b>	<b>Range</b>	<b>Description</b>
ConversionFactor		Conversion factor. Called <code>gamma</code> when it is an input argument, it defaults to the matrix <code>[1 -1]</code> that specifies soft-constrained initialization.
DesiredSignalStates		Desired signal states of the adaptive filter. <code>dstates</code> defaults to a zero vector with length equal to <code>(blocklen - 1)</code> .
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. <code>delta</code> 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).

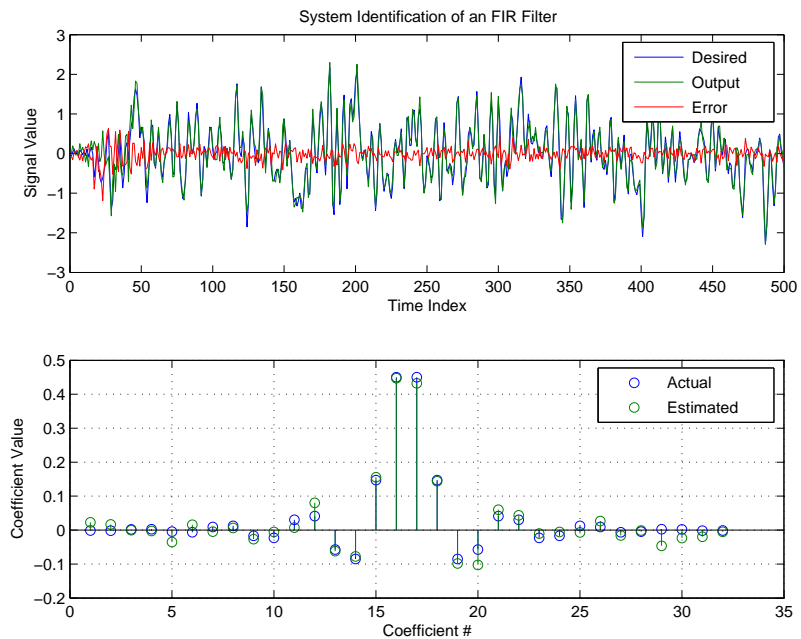
## Examples

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(1,lambda,invcov,swblocklen, dstates,...coeffs,states)`

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

## 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. <code>coeffs</code> 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. <code>dstates</code> defaults to a zero vector with length equal to $(\text{swblocklen} - 1)$ .
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

<b>Name</b>	<b>Range</b>	<b>Description</b>
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. <code>states</code> defaults to a vector of zeros which has length equal to $(1 + \text{swblocklen} - 2)$
SwBlockLength	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. <code>swblocklen</code> defaults to 16.

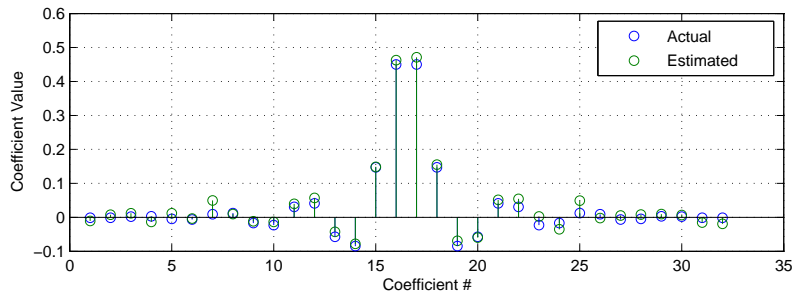
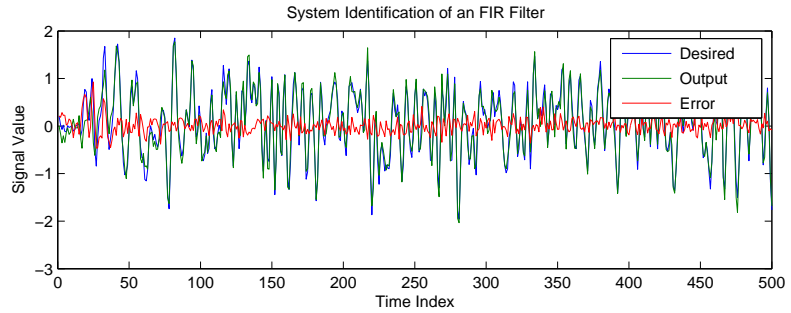
## Examples

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`

# adaptfilt.tdafdct

---

**Purpose** Adaptive filter that uses discrete cosine transform

**Syntax** `ha = adaptfilt.tdafdct(1, step, leakage, offset, delta, lambda, coeffs, states)`

**Description** `ha = adaptfilt.tdafdct(1, 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
<code>1</code>	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. <code>1</code> defaults to 10.
<code>step</code>	Adaptive filter step size. It must be a nonnegative scalar. You can use <code>maxstep</code> to determine a reasonable range of step size values for the signals being processed. <code>step</code> defaults to 0.
<code>leakage</code>	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. <code>leakage</code> defaults to 1 — no leakage.
<code>offset</code>	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. <code>offset</code> defaults to zero.

<b>Input Argument</b>	<b>Description</b>
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 1 vector. coeffs defaults to a zero vector of length 1.
states	Initial conditions of the adaptive filter. states defaults to a zero vector with length equal to (1 - 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.

<b>Name</b>	<b>Range</b>	<b>Description</b>
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
AvgFactor		Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. AvgFactor

Name	Range	Description
		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 1 vector where 1 is the number of filter coefficients. coeffs defaults to length 1 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.
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.



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	2*1 element vector	A vector of 2*1 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 (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.

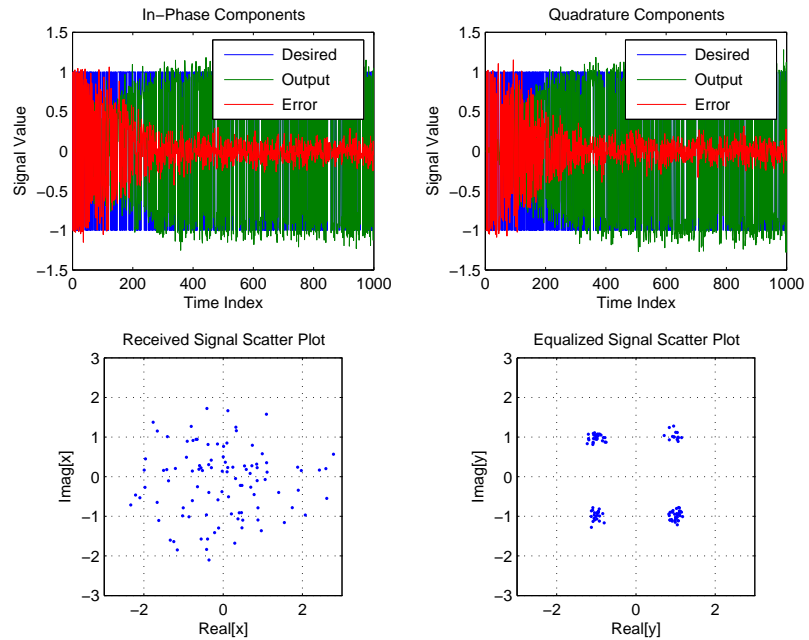
For checking the values of properties for an adaptive filter object, use `get(ha)` or enter the object name, without a trailing semicolon, at the MATLAB prompt.

## 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)); %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'); grid on;
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');
```

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(1,step,leakage,offset,  
delta,lambda,...coeffs,states)`

**Description** `ha = adaptfilt.tdafdft(1,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
<code>1</code>	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. <code>1</code> defaults to 10.
<code>step</code>	Adaptive filter step size. It must be a nonnegative scalar. You can use <code>maxstep</code> to determine a reasonable range of step size values for the signals being processed. <code>step</code> defaults to 0.
<code>leakage</code>	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. <code>leakage</code> defaults to 1 — no leakage.
<code>offset</code>	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. <code>offset</code> 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 1 vector. coeffs defaults to a zero vector of length 1.
states	Initial conditions of the adaptive filter. states defaults to a zero vector with length equal to (1 - 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
AvgFactor		Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. AvgFactor

Name	Range	Description
		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 1 vector where 1 is the number of filter coefficients. coeffs defaults to length 1 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.
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.

Name	Range	Description
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*1 element vector	A vector of 2*1 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 (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. step defaults to 0.

### Examples

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

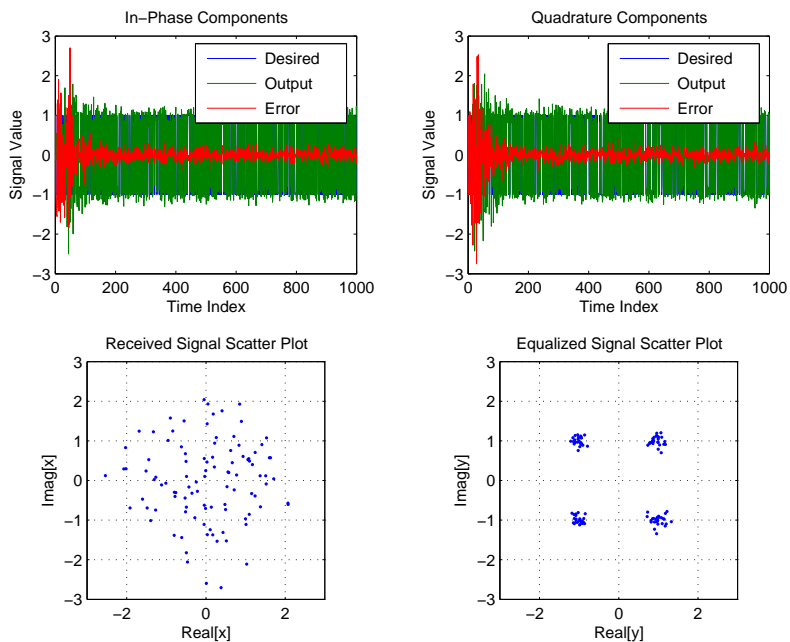
```
D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1000; % Number of iterations
```

```
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

# adaptfilt.ufdaf

---

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

**Syntax** `ha = adaptfilt.ufdaf(1,step,leakage,delta,lambda,blocklen,offset,coeffs,states)`

**Description** `ha = adaptfilt.ufdaf(1,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
1	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. 1 defaults to 10.
step	Adaptive filter step size. It must be a nonnegative scalar. <code>step</code> 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. <code>leakage</code> defaults to 1 — no leakage.
delta	Initial common value of all of the FFT input signal powers. the initial value of <code>delta</code> should be positive, and it defaults to 1.

<b>Input Argument</b>	<b>Description</b>
<code>lambda</code>	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. <code>lambda</code> should lie in the range (0,1]. For default UFDFAF filter objects, <code>lambda</code> defaults to 0.9.
<code>blocklen</code>	Block length for the coefficient updates. This must be a positive integer. For faster execution, ( <code>blocklen - 1</code> ) should be a power of two. <code>blocklen</code> defaults to 1.
<code>offset</code>	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.
<code>coeffs</code>	Initial time-domain coefficients of the adaptive filter. It should be a length 1 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 <code>blocklen</code> .
<code>states</code>	Adaptive filter states. <code>states</code> defaults to a zero vector with length equal to 1.

## 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.

<b>Name</b>	<b>Range</b>	<b>Description</b>
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 <code>coeffs</code> .
FFTStates		States for the FFT operation.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

<b>Name</b>	<b>Range</b>	<b>Description</b>
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*1 element vector	A vector of 2*1 elements, each initialized with the value <code>delta</code> 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 <code>maxstep</code> to determine a reasonable range of step size values for the signals being processed. <code>step</code> 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.

```

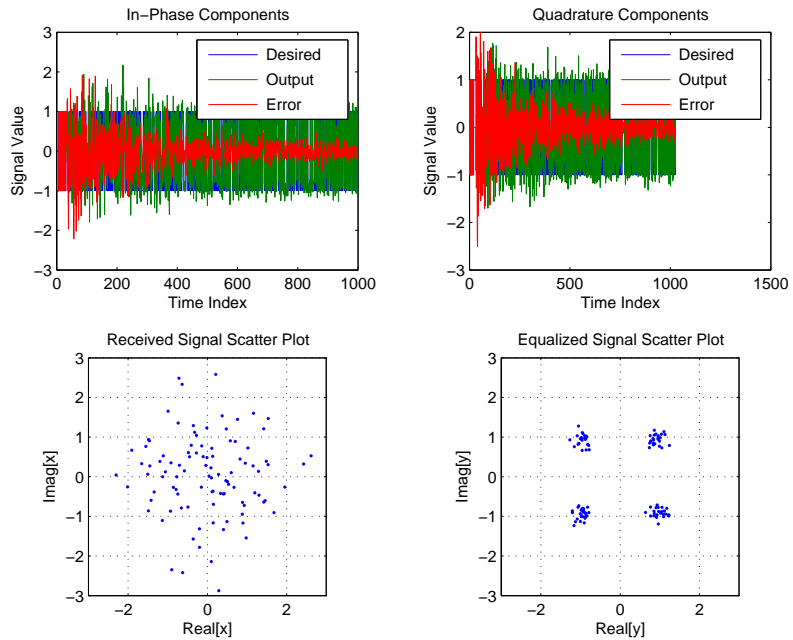
D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1024; % Number of iterations
s = sign(randn(1,ntr+D))+j*sign(randn(1,ntr+D)); % Baseband QPSK signal
n = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));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

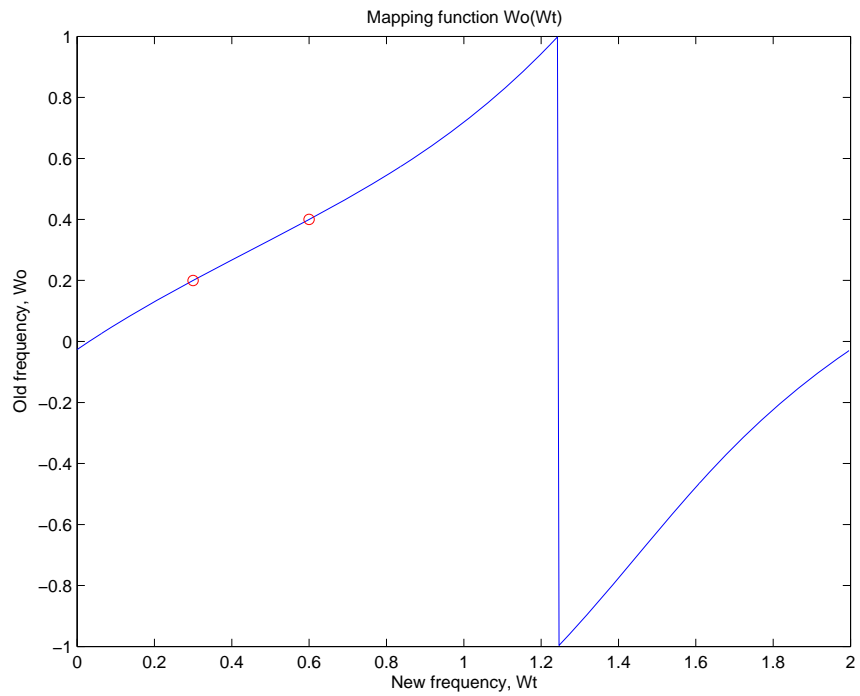
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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
<i>Wo</i>	Frequency values to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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);
```

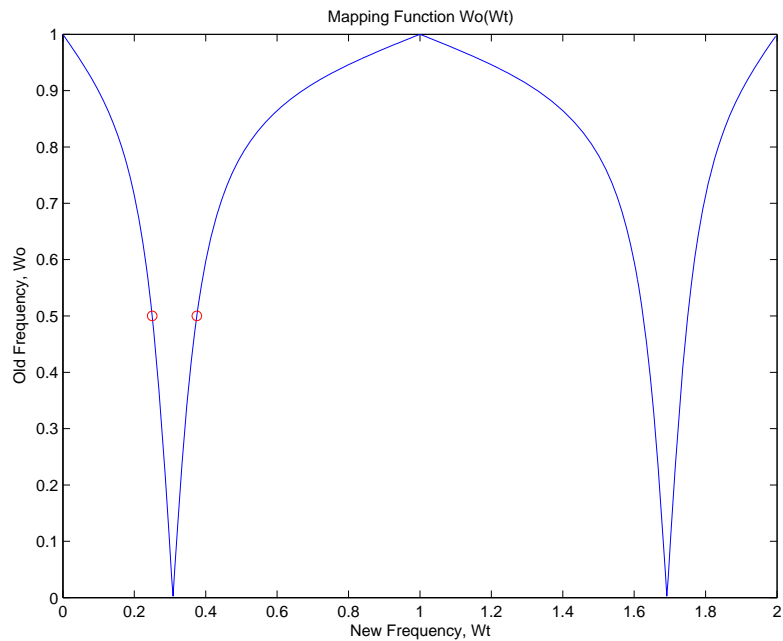
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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

iir1p2bp, zpk1p2bp

## 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);
```

# allpasslp2bpc

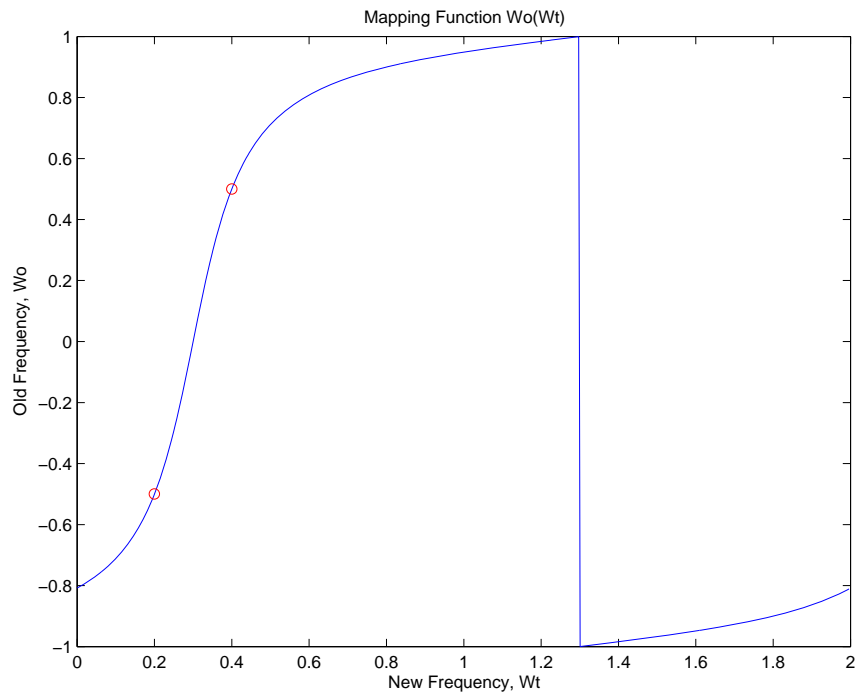
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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
<i>Wo</i>	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.
<i>Wt</i>	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.
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

## See Also

iir1p2bpc, zpk1p2bpc

# 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);
```

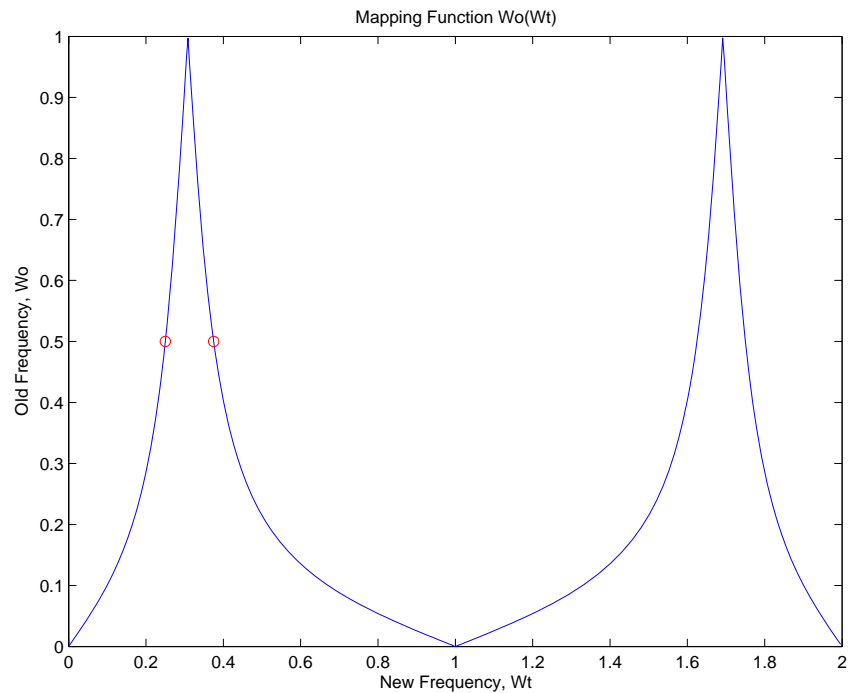
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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  $\pi$ , as mentioned earlier.



## Arguments

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

iir1p2bs, zpk1p2bs

## 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);
```

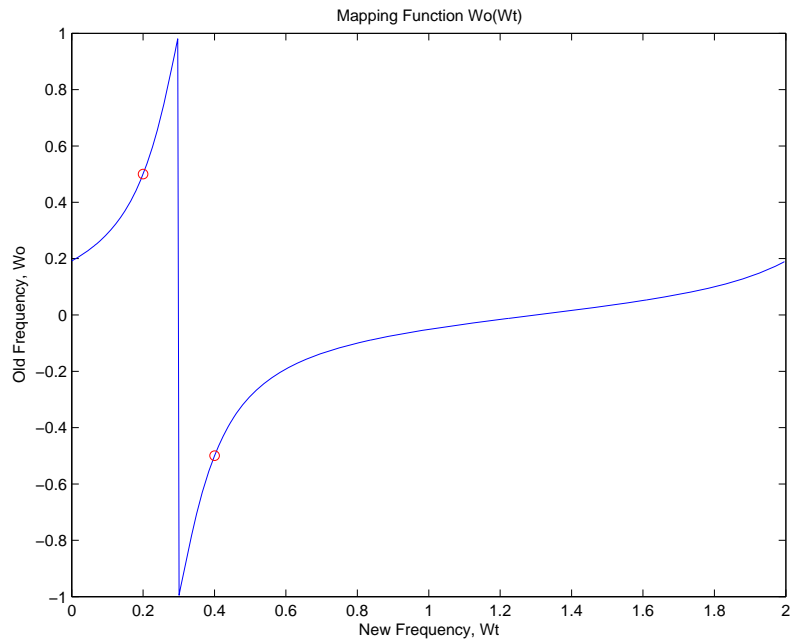
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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**

<b>Variable</b>	<b>Description</b>
<i>Wo</i>	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.
<i>Wt</i>	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.
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

**See Also**

iir1p2bsc, zpk1p2bsc

# 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:

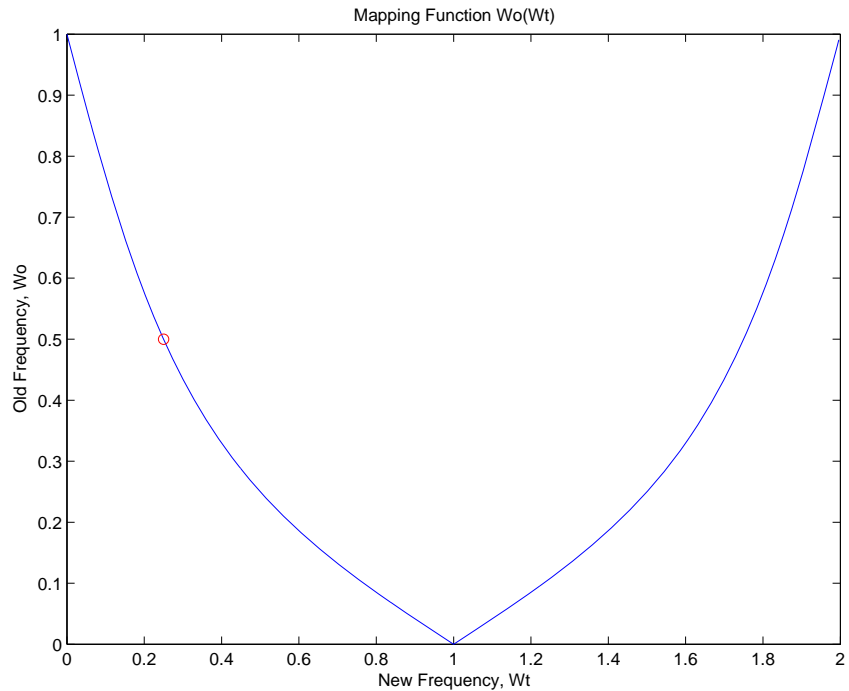
```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```



Plot the phase response normalized to  $\pi$ , 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'); m
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
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency location in the transformed target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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

iir1p2hp, zpk1p2hp

## 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  $W_t=0.25$ :

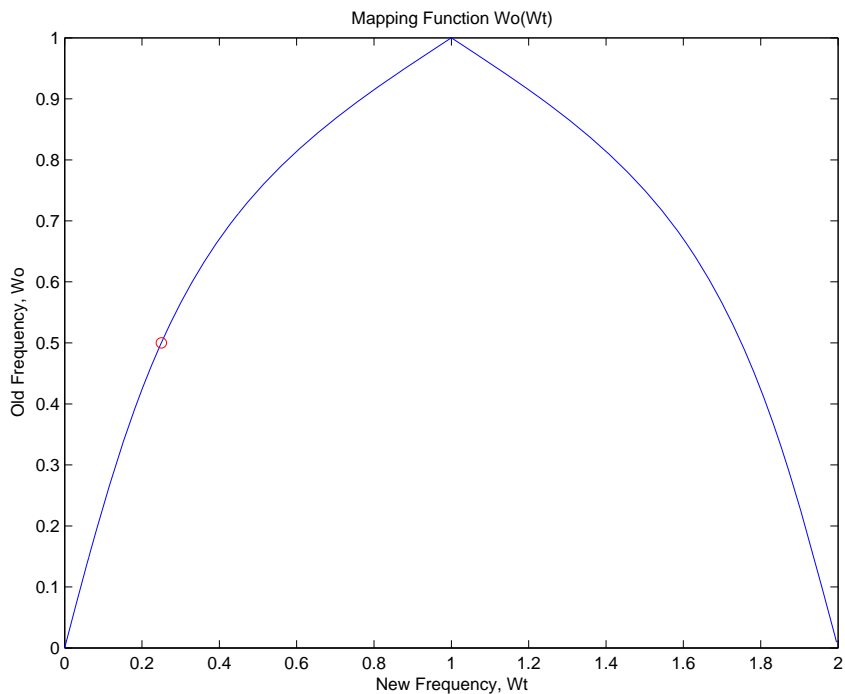
```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2lp(Wo, Wt);
```

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

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

Plot the phase response normalized to  $\pi$ , 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
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency location in the transformed target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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**

iir1p2lp, zpk1p2lp

**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 Mth-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,  $W_{t1}, \dots, W_{tM}$ . 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);
```

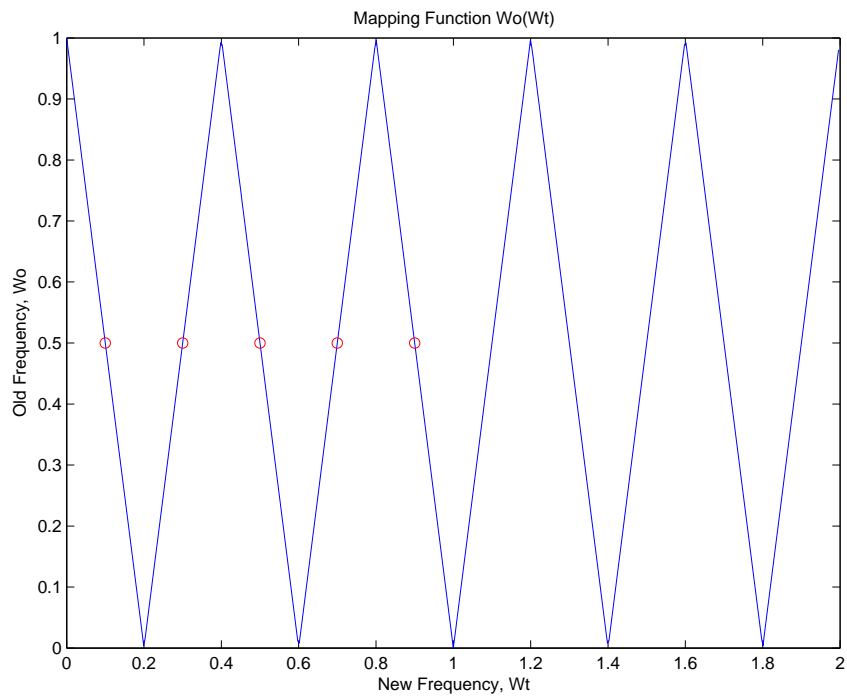
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>Pass</i>	Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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**

iir1p2mb, zpk1p2mb

**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 Mth-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,  $W_{t1}, \dots, W_{tM}$ .

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);
```

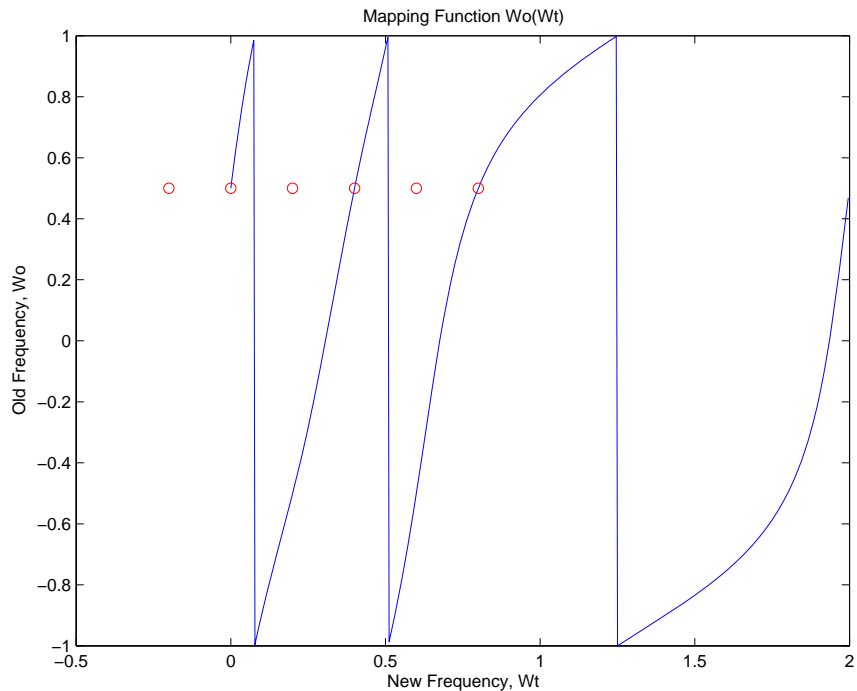
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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.



# allpasslp2mbc

---

## Arguments

Variable	Description
<i>Wo</i>	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.
<i>Wt</i>	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.
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

## See Also

iir1p2mbc, zpk1p2mbc

<b>Purpose</b>	Allpass filter for lowpass to complex N-point transformation
<b>Syntax</b>	[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)
<b>Description</b>	<p>[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt) returns the numerator, AllpassNum, and the denominator, AllpassDen, of the Nth-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 <math>W_{o1}, \dots, W_{oN}</math>, at the required target frequency locations, <math>W_{t1}, \dots, W_{tM}</math>.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation. For DC mobility feature <math>F_2</math> will precede <math>F_1</math> after the transformation.</p> <p>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.</p> <p>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.</p>
<b>Examples</b>	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:

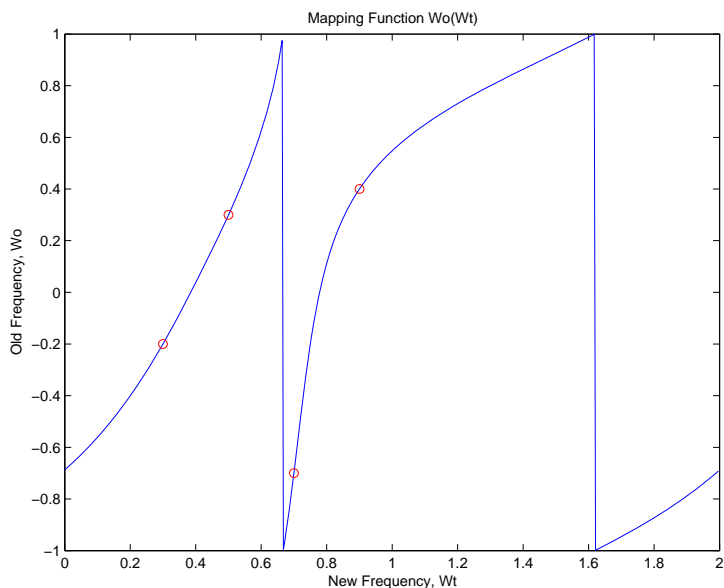
```
Wo = [-0.2, 0.3, -0.7, 0.4]; Wt = [0.3, 0.5, 0.7, 0.9];  
[AllpassNum, AllpassDen] = allpass1p2xc(Wo, Wt);
```

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

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

Plot the phase response normalized to  $\pi$ , 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**

<b>Variable</b>	<b>Description</b>
<i>Wo</i>	Frequency values to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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**

iir1p2xc, zpk1p2xc

**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 Nth-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  $W_{o1}, \dots, W_{oN}$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ . 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
<i>Wo</i>	Frequency values to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>Pass</i>	Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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

iir1p2xn, zpk1p2xn

## 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 Nth-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);
```

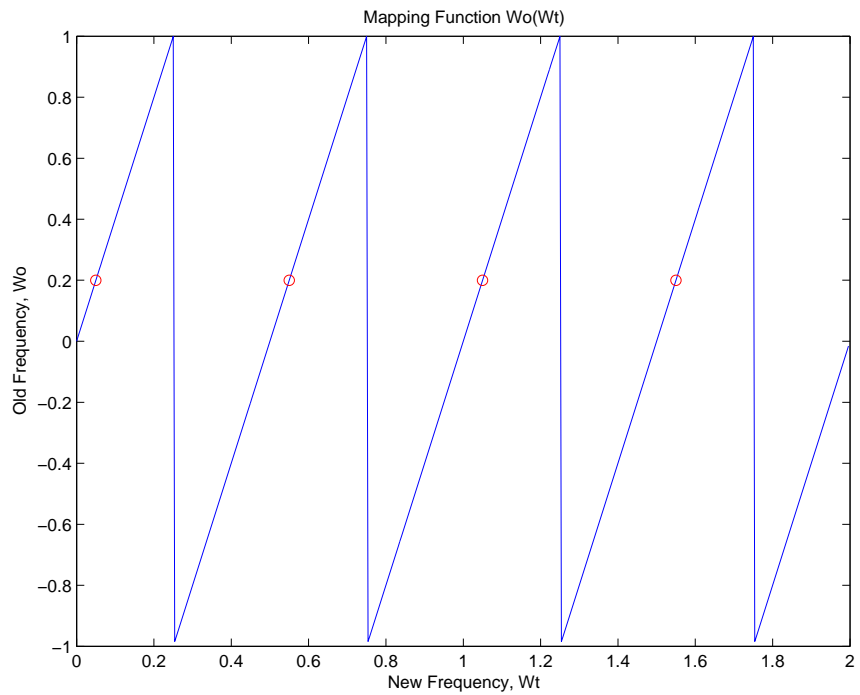
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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)
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

## See Also

iirateup, 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  $W_t=0.25$ :

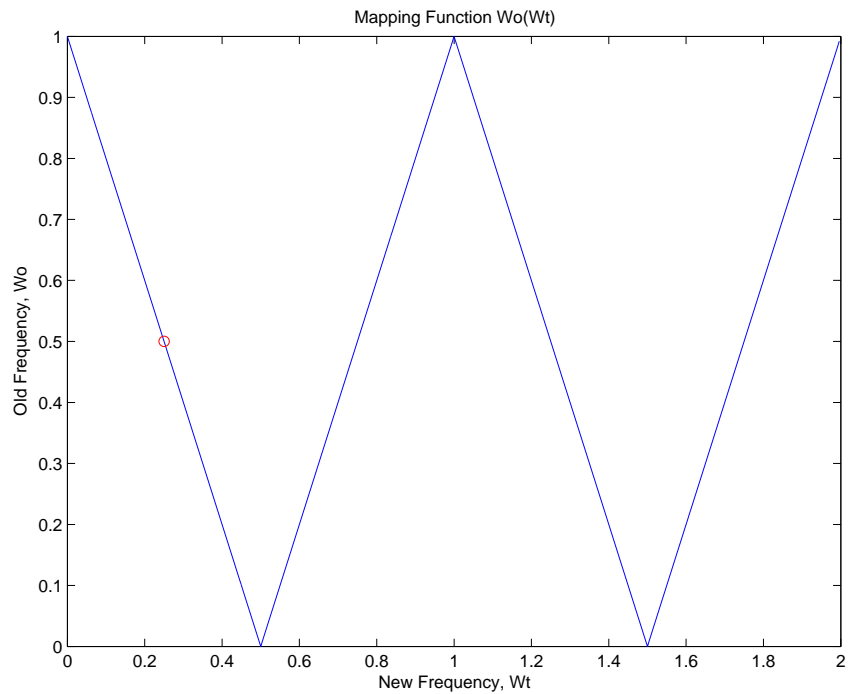
```
Wo = 0.5; Wt = 0.25;  
[AllpassNum, AllpassDen] = allpassshift(Wo, Wt);
```

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

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

Plot the phase response normalized to  $\pi$ , 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');
```



# allpassshift

---

## Arguments

Variable	Description
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency location in the transformed target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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, 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);
```

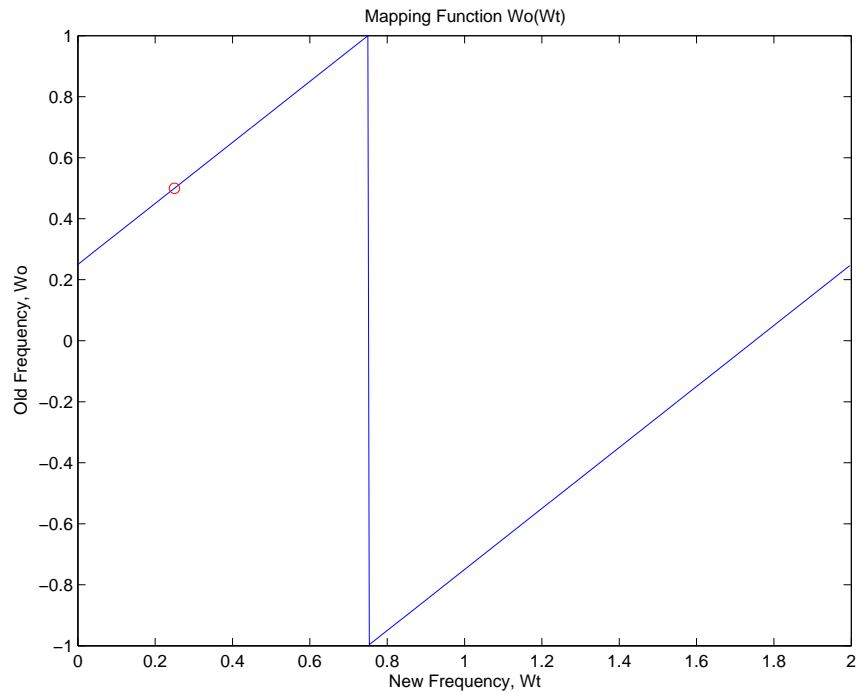
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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
$W_o$	Frequency value to be transformed from the prototype filter
$W_t$	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. Schaffer 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.

# autoscale

---

**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.

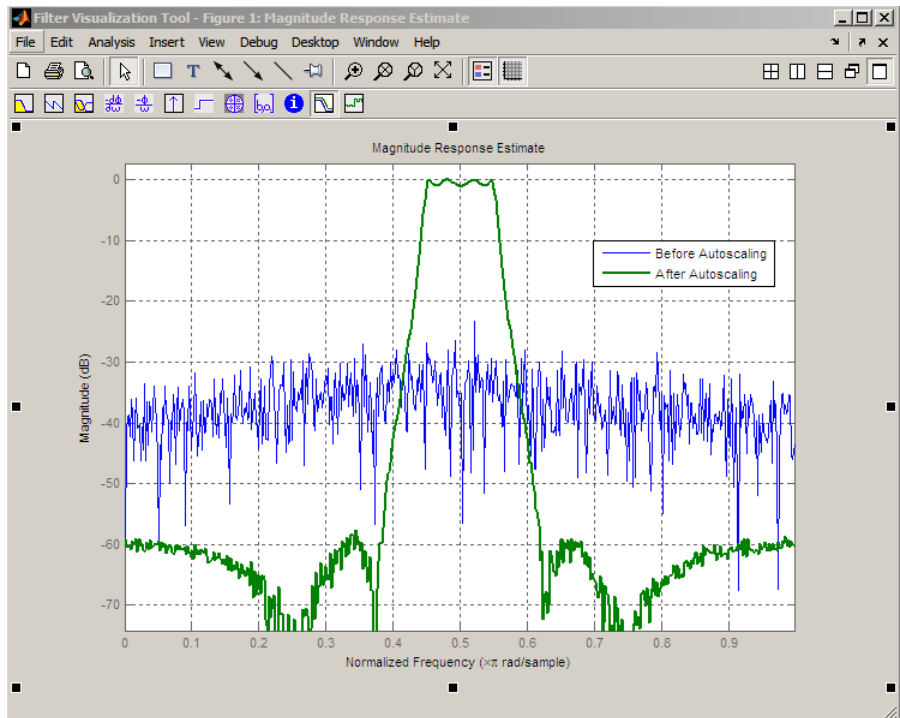
For introductory demonstrations of the automatic scale process, refer to the following demos in the toolbox:

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

## Examples

Demonstrate dynamic range scaling in a lattice ARMA filter:

```
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','mestimate','Showreference','off');
legend(hfvt, 'Before Autoscaling', 'After Autoscaling')
```



**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 <code>current</code> , <code>new</code> , or specify the name of an existing subsystem with <i>subsystemname</i> . Specifying <code>new</code> opens a new model and adds the block. <code>Current</code> adds the block to your current Simulink model. <code>Current</code> is the default setting. If you provide the name of a current subsystem in <i>subsystemname</i> , <code>block</code> 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 <code>filter</code> .

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

### Using `block` to Realize Fixed-Point Multirate Filters

When the source filter `hm` is fixed-point, the input word and fraction lengths for the block are derived from the block input signal. The realization process issues a warning and 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. 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.

```
hm = mfilt.firdecim(3);
```

Now use the default syntax to create a block.

```
block(hm);
```

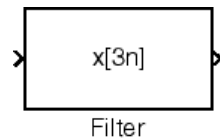
# block

---

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**

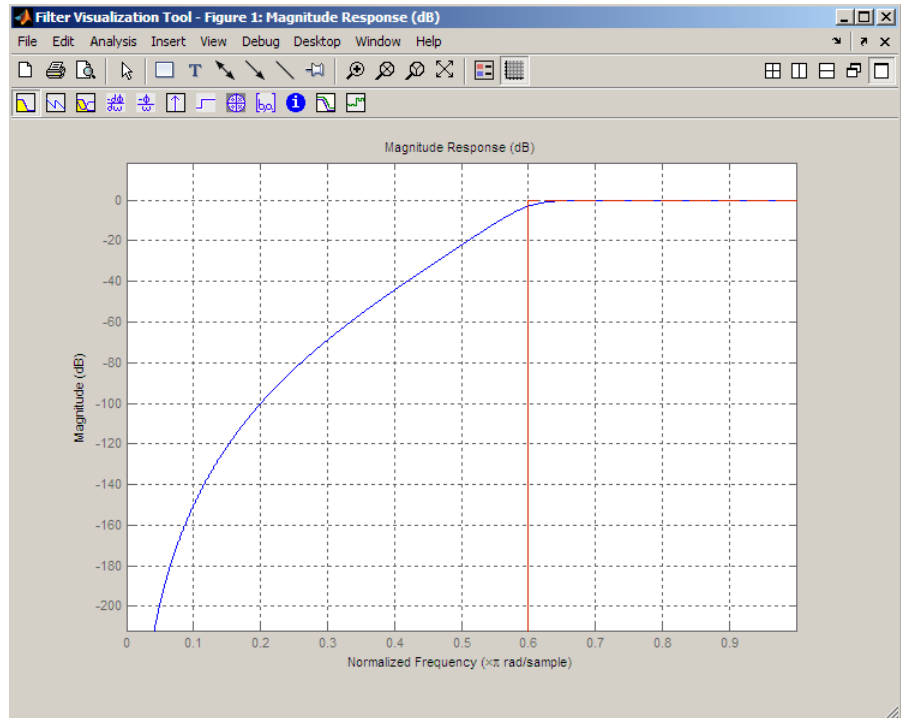
Construct a default lowpass filter specification object and design a Butterworth filter:

```
d = fdesign.lowpass; designopts(d, 'butter');  
hd = design(d, 'butter', 'matchexactly', 'stopband');
```

Construct a highpass filter specification object and design a Butterworth filter:

```
%Order 8 and 3 dB frequency 0.6*pi radians/sample  
d = fdesign.highpass('N,F3dB', 8, .6);  
design(d, 'butter');
```

# 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

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

[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)

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

## 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 type I Chebyshev IIR digital filter using the specifications supplied in the object `d`. Depending on the filter specification object, `cheby1` may or may not be a valid design. Use `designmethods` with the filter specification object to determine if a Chebyshev type I filter design is possible.

`hd = design(d,'cheby1',designoption,value,designoption,value,...)` returns a type I Chebyshev 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');  
hd = design(d,'cheby1','matchexactly','passband');
```

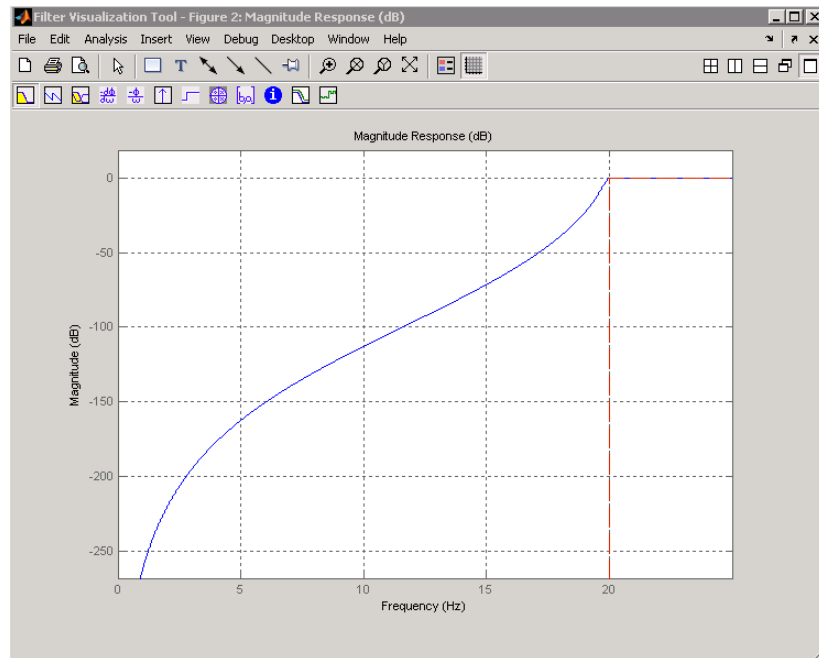
Specify the filter order, passband edge frequency, and the passband ripple to get the filter exactly as required.

# cheby1

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

Use fvtool to view the resulting filter.

```
fvtool(hd)
```



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

## See Also

`design`, `designmethods`, `fdesign`

**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');
```

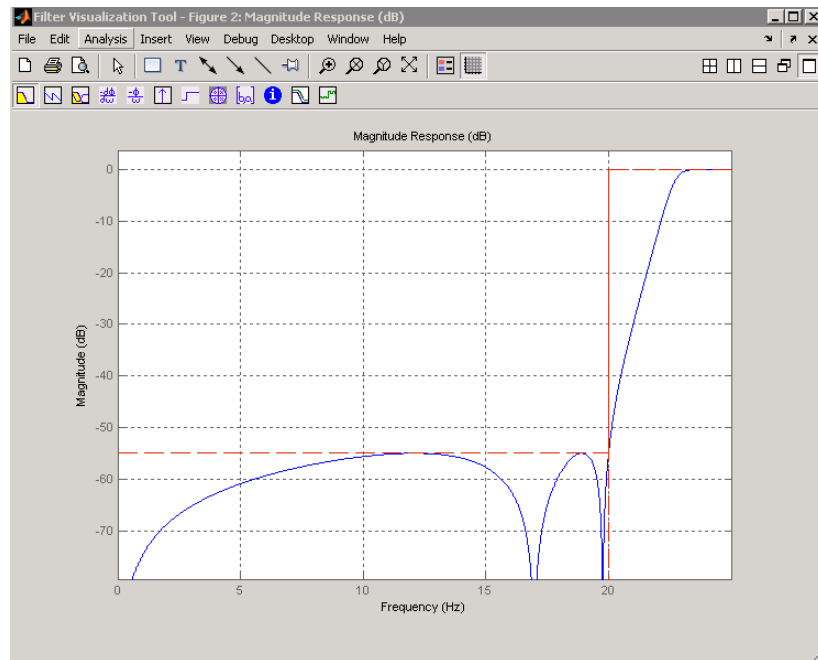
`cheby2` also designs highpass, bandpass, and bandstop filters. Here is a highpass filter where you specify the filter order, the stopband edge frequency, and the stopband attenuation to get the filter exactly as required.

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

```
hd=design(d,'cheby2');
```

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] = c12tf(k1,k2)
[b,a] = c12tf(k1,k2,beta)
[b,a,bp] = c12tf(k1,k2)
[b,a,bp] = c12tf(k1,k2,beta)
```

**Description**

[b,a] = c12tf(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] = c12tf(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

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

[b,a,bp] = c12tf(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] = c12tf(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  $G(z)$

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

## 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 filter
% and the power complementary one.
freqzplot([h hpc],w,s1);
```

## See Also

tf2cl, tf2ca, ca2tf, tf2latc, latc2tf, iirpowcomp



<b>Purpose</b>	Coefficients for filters
<b>Syntax</b>	<pre>s = coeffs(ha) s = coeffs(hd) s = coeffs(hm)</pre>
<b>Description</b>	<p>The next sections describe common <code>coeffs</code> operation with adaptive, discrete-time, and multirate filters.</p> <p><b>Adaptive Filters</b></p> <p><code>s = coeffs(ha)</code> returns a structure <code>s</code> containing the coefficients of adaptive filter <code>ha</code>. These are the instantaneous filter coefficients available at the time you use the function.</p> <p><b>Discrete-Time Filters</b></p> <p><code>s = coeffs(hd)</code> returns a structure <code>s</code> that contains the coefficients of discrete-time filter <code>hd</code>.</p> <p><b>Multirate Filters</b></p> <p><code>s = coeffs(hm)</code> returns <code>s</code>, a structure containing the coefficients of discrete-time filter <code>hm</code>. CIC-based filters do not have coefficients and this function does not work with constructors like <code>mfilt.cicdecim</code>.</p>
<b>Examples</b>	<p><code>coefficients</code> works the same way for all filters. This example uses a multirate filter <code>hm</code> to demonstrate the function.</p>

```
hm=mfilt.firdecim(3)

hm =

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

s=coeffs(hm)
```

## coeffs

---

```
s =
```

```
    Numerator: [1x72 double]
```

The filter coefficients may be extracted by typing `s.Numerator` at the command prompt.

### 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 = firceqip(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(R) 7.8 and the Filter Design Toolbox 4.5.
;
Radix = 10;
Coefficient_Width = 16;
CoefData =  -83,
-1702,
-732,
  615,
 1302,
   45,
-1746,
-1316,
 1498,
3008,...
```

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

# constraincoeffwl

---

**Purpose** Constrain coefficient wordlength

**Syntax**

```
Hq = constraincoeffwl(Hd,wordlength)
Hq = constraincoeffwl(Hd,wordlength,'Ntrials',N)
Hq = constraincoeffwl(Hd,wordlength,...,'NoiseShaping',
    NSFlag)
Hq = constraincoeffwl(Hd,wordlength,...,'Apasstol',Apasstol)
Hq = constraincoeffwl(Hd,wordlength,...,'Astoptol',Astoptol)
```

**Description**

Hq = constraincoeffwl(Hd,wordlength) returns a fixed-point filter Hq meeting the design specifications of the single-stage or multistage FIR filter object Hd with a wordlength of at most wordlength bits. For multistage filters, wordlength can either be a scalar or vector. If wordlength is a scalar, the same wordlength is used for all stages. If wordlength is a vector, each stage uses the corresponding element in the vector. The vector length must equal the number of stages. Hd must be generated using fdesign and design. constraincoeffwl uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator rand

Hq = constraincoeffwl(Hd,wordlength,'Ntrials',N) specifies the number of Monte Carlo trials to use. Hq is first filter among the trials to meet the specifications in Hd with a wordlength of at most wordlength.

Hq = constraincoeffwl(Hd,wordlength,...,'NoiseShaping',NSFlag) enables or disables the stochastic noise-shaping procedure in the constraint of the wordlength. By default NSFlag is true. Setting NSFlag to false constrains the wordlength without using noise-shaping.

Hq = constraincoeffwl(Hd,wordlength,...,'Apasstol',Apasstol) specifies the passband ripple tolerance in dB. 'Apasstol' defaults to 1e-4.

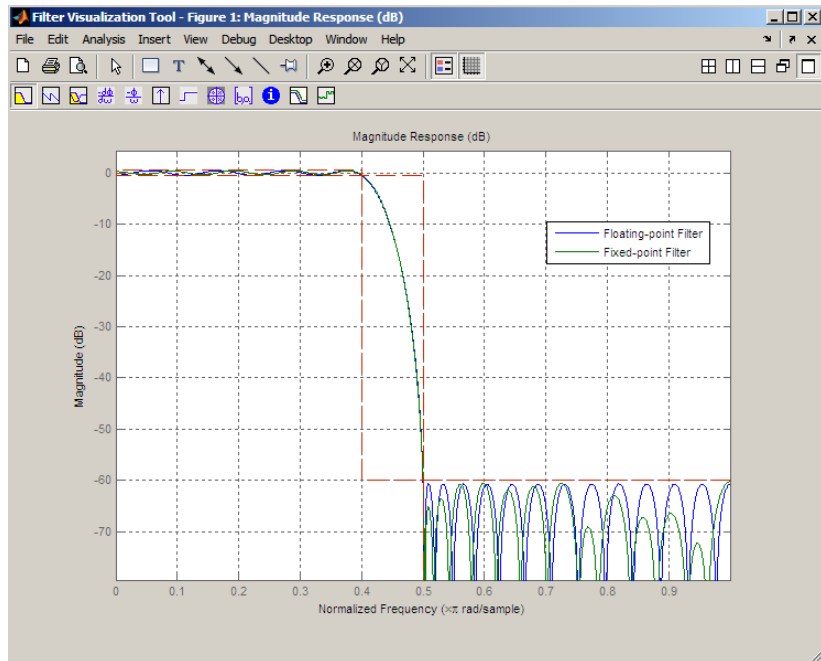
Hq = constraincoeffwl(Hd,wordlength,...,'Astoptol',Astoptol) specifies the stopband tolerance in dB. 'Astoptol' defaults to 1e-2

You must have the Fixed-Point Toolbox™ software installed to use this function.

## Examples

Design fixed-point filter with a wordlength of at most 11 bits:

```
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',.4,.5,1,60);
Hd = design(Hf,'equiripple'); % 43 coefficients
Hq = constraincoeffwl(Hd,11); % 45 11-bit coefficients
hfv = fvtool(Hd,Hq,'showreference','off');
legend(hfv,'Floating-point Filter','Fixed-point Filter');
```



## See Also

[design](#) | [fdesign](#) | [maximizestopband](#) | [minimizecoeffwl](#) | [measure](#) | [rand](#)

## Tutorials

- “Fixed-Point Concepts”

# convert

---

**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
- 'df1sos': Direct-Form I, Second-Order Sections
- 'df1tsos': Direct-Form I Transposed, Second-Order Sections
- 'df2': Direct form II
- 'df2t': Direct form II transposed. Default filter structure
- 'df2sos': Direct-Form II, Second-Order Sections
- 'df2tsos': Direct-Form II Transposed, Second-Order Sections
- 'dffir': FIR
- 'dffirt': Direct form FIR transposed
- 'latcallpass': Lattice allpass
- 'latticeca': Lattice coupled-allpass
- 'latticecapc': Lattice coupled-allpass power-complementary
- 'latticear': Lattice autoregressive (AR)
- 'laticemamax': Lattice moving average (MA) maximum phase
- 'laticemamin': Lattice moving average (MA) minimum 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': Direct form I
- 'df1t': Direct form I transposed
- 'df1sos': Direct-Form I, Second-Order Sections
- 'df1tsos': Direct-Form I Transposed, Second-Order Sections
- 'df2': Direct form II
- 'df2t': Direct form II transposed. Default filter structure
- 'df2sos': Direct-Form II, Second-Order Sections
- 'df2tsos': Direct-Form II Transposed, Second-Order Sections
- 'statespace': Single-input/single-output state-space
- 'symmetricfir': Symmetric FIR. Even and odd forms

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

- Minimum phase FIR filters can be converted to `latticeamin`
- Maximum phase FIR filters can be converted to `latticeamax`
- 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: [1x6 double]  
Denominator: [1x6 double]  
PersistentMemory: false
```

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);
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
```

---

**Note** The above example will generate a warning:

```
Warning: Using reference filter for structure conversion.
Fixed-point attributes will not be converted.
```

Since CIC interpolators only use fixed-point arithmetic, the user may disregard this warning. The fixed-point structure will not be lost on conversion.

---

## See Also

`mfilt`

`dfilt` in Signal Processing Toolbox documentation

# cost

---

**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	<code>nmult</code>	Number of multiplications during the filter run. <code>nmult</code> ignores multiplications by -1, 0, and 1 in the total multiple.
Number of Additions	<code>nadd</code>	Number of additions during the filter run.
Number of States	<code>nstates</code>	Number of states the filter uses.
<code>MultPerInputSample</code>	<code>multperinputsample</code>	Number of multiplication operations performed for each input sample
<code>AddPerInputSample</code>	<code>addperinputsample</code>	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'
```

**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.

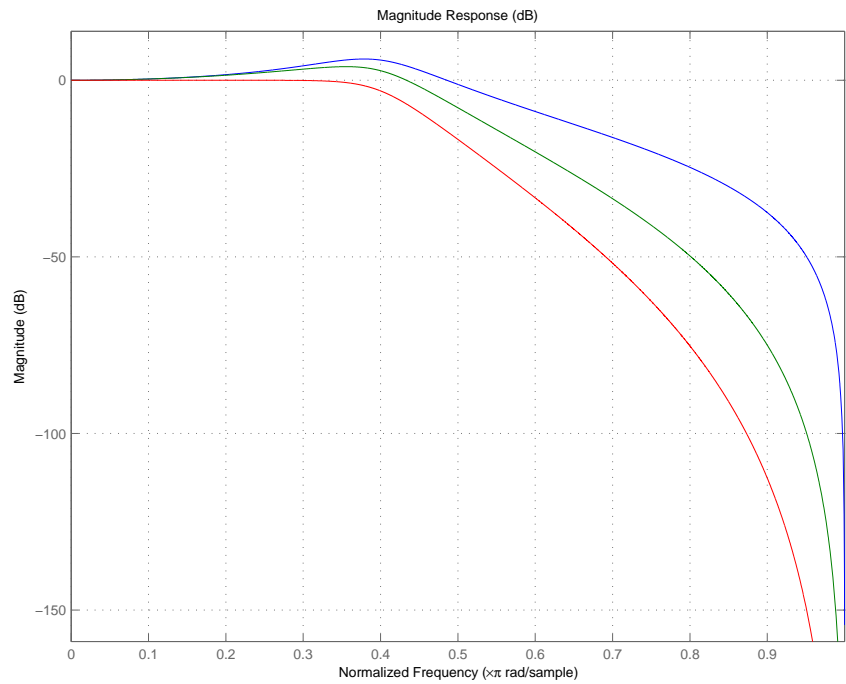
`h = cumsec(hd,indices,secondary)` when `secondary` is true, `cumsec` uses the secondary scaling points in the sections to determine where the sections should be split. This option applies only when `hd` is a `df2sos` and `df1tsos` filter. For these second-order section structures, the secondary scaling 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(hfvf, 'First Section', 'First Two Sections', 'Overall Filter');
```



**See Also** [scale](#), [scalecheck](#)

# denormalize

---

**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,F3dB',14,0.45)
hd =design(d,'butter');
hd.arithmetic='fixed'
g=normalize(hd)';
hd.sosMatrix
denormalize(hd)
hd.sosMatrix
```

**See Also** `normalize`



<b>Purpose</b>	Apply design method to specification object
<b>Syntax</b>	<pre>h = design(d) h = design(d,designmethod) h = design(d,designmethod,specname,specvalue,...)</pre>
<b>Description</b>	<p><code>h = design(d)</code> uses specifications object <code>d</code> to generate a filter <code>h</code>. When you do not provide a design method as an input argument, <code>design</code> chooses the design method to use by following these rules in the order listed.</p> <ol style="list-style-type: none"><li>1 Use <code>equiripple</code> if it applies to the object <code>d</code>.</li><li>2 When <code>equiripple</code> does not apply to <code>d</code>, use another FIR design method, such as <code>firls</code>.</li><li>3 If FIR design methods do not apply to <code>d</code>, use <code>ellip</code>.</li><li>4 When <code>ellip</code> does not apply to <code>d</code>, use another IIR design method, such as <code>butter</code> or <code>cheby2</code>, that applies to the object <code>d</code>.</li></ol> <p>More rules apply.</p> <ul style="list-style-type: none"><li>• <code>design</code> uses an FIR filter design method before using an IIR design method.</li><li>• <code>fdesign.nyquist</code> specifications objects use the <code>kaiserwin</code> design method as the first design choice, rather than <code>equiripple</code>, because <code>kaiserwin</code> produces better filters than <code>equiripple</code>.</li><li>• For decimators, interpolators, and rational sample rate changers that use <code>fdesign.nyquist</code> objects, the default design method is <code>kaiserwin</code>. Otherwise, those objects use the <code>equiripple</code> design method by default.</li></ul> <p>For more guidance about using <code>design</code> to design filters, refer to <i>Filter Design Toolbox™ Getting Started Guide</i>. There you find examples that use <code>design</code> to design filters and use methods in the toolbox to</p>

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.

- `ansis142` — Design method following ANSI standard S1.42–2001. Valid only for `fdesign.audioweighting` objects with A and C weighting types.
- `bell141009`— Design method following the Bell System Technical Reference PUB 41009. Valid only for `fdesign.audioweighting` objects with a C-message weighting type.
- `butter`— Butterworth design
- `cheby1`— Chebyshev type I design. All-pole design that is equiripple in the passband and monotonic in the stopband.
- `cheby2`— Chebyshev type II design. Pole-zero design that is monotonic in the passband and equiripple in the stopband.
- `ciccomp`—Cascaded integrator-comb (CIC) compensator design
- `ellip`— Elliptic (Cauer) design. Elliptic filters are equiripple in both the pass and stopbands.

- `equiripple`—FIR equiripple design
- `firls`—FIR linear phase filter with least-square error minimization
- `freqsamp`—FIR filter design by frequency sampling
- `ifir`—Interpolated FIR filter design
- `iirhilbert`—IIR Hilbert transformer design
- `iirlinphase`—IIR quasi linear phase design
- `iirlpnorm`—Least P-norm IIR design
- `isinclp`—Inverse sinc filter design
- `kaiserwin`—FIR Kaiser window design
- `lagrange`—FIR Lagrange interpolation filter design
- `multistage`—Multistage design
- `window`—FIR window design

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.

<b>Designmethod Keyword</b>	<b>Description of the design Response</b>
<b><code>fir</code></b>	Forces <code>design</code> to produce an FIR filter. When no FIR design method exists for object <code>d</code> , <code>design</code> returns an error.
<b><code>iir</code></b>	Forces <code>design</code> to produce an IIR filter. When no IIR design method exists for object <code>d</code> , <code>design</code> returns an error.

<b>Designmethod Keyword</b>	<b>Description of the design Response</b>
<b>allfir</b>	Produces filters from every applicable FIR design method for the specifications in <code>d</code> , one filter for each design method. As a result, <code>design</code> returns multiple filters in the output object.
<b>alliir</b>	Produces filters from every applicable IIR design method for the specifications in <code>d</code> , one filter for each design method. As a result, <code>design</code> returns multiple filters in the output object.
<b>all</b>	Designs filters using all applicable design methods for the specifications object <code>d</code> . As a result, <code>design</code> returns multiple filters, one for each design method. <code>design</code> uses the design methods in the order that <code>designmethods(d)</code> 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 `equiripple`, `ifir`, `kaiserwin`, and `multistage` design methods, in the order shown by `designmethods(d)`. The first filter in `hallfir` comes from the `equiripple` design method; the second from the `ifir` method; the third from using `kaiserwin` 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`

# designmethods

---

**Purpose** Methods available for designing filter from specification object

**Syntax**

```
m = designmethods(d)
m = designmethods(d, 'default')
m = designmethods(d, type)
m = designmethods(d, 'full')
```

**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
ansis142	IIR. Design method following ANSI standard S1.42–2001. Valid only for <code>fdesign.audioweighting</code> objects with A and C weighting types.
bell41009	IIR. Design method following the Bell System Technical Reference PUB 41009. Valid only for <code>fdesign.audioweighting</code> objects with C-message weighting.
butter	IIR
cheby1	IIR
cheby2	IIR
ellip	IIR
equiripple	FIR
firls	FIR
freqsamp	Frequency-sampled FIR
ifir	Interpolated FIR
irlinphase	IIR filter with linear phase



Design Method	Filter Result
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
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)
designmethods(d)
% Design direct form FIR filter via the window method
hd=window(d)
```

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');
```

## designmethods

---

```
m2 = designmethods(d)
m3 = designmethods(d, 'fir')
% Return full names for IIR methods
m4 = designmethods(d, 'iir', 'full')
```

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')
```

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')
```

**See Also**      design, designmethods, fdesign

# dfilt

**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.

<b>dfilt.structure</b>	<b>Description</b>	<b>Coefficient Mapping Support in <code>realizemdl</code></b>
<code>dfilt.allpass</code>	Allpass filter	Supported
<code>dfilt.cascadeallpass</code>	Cascade of allpass filter sections	Supported
<code>dfilt.cascadewdfallpass</code>	Cascade of allpass wave digital filters	Supported
<code>dfilt.delay</code>	Delay	Not supported
<code>dfilt.df1</code>	Direct-form I	Supported
<code>dfilt.df1sos</code>	Direct-form I, second-order sections	Supported.

<b>dfilt.structure</b>	<b>Description</b>	<b>Coefficient Mapping Support in realizemdl</b>
dfilt.df1t	Direct-form I transposed	Supported
dfilt.df1tsos	Direct-form I transposed, second-order sections	Supported.
dfilt.df2	Direct-form II	Supported
dfilt.df2sos	Direct-form II, second-order sections	Supported.
dfilt.df2t	Direct-form II transposed	Supported
dfilt.df2tsos	Direct-form II transposed, second-order sections	Supported.
dfilt.dffir	Direct-form FIR	Supported
dfilt.dffirt	Direct-form FIR transposed	Supported
dfilt.dfsymfir	Direct-form symmetric FIR	Supported
dfilt.dfasymfir	Direct-form antisymmetric FIR	Supported
dfilt.farrowfd	Generic fractional delay Farrow filter	Supported.
dfilt.farrowlinearfd	Linear fractional delay Farrow filter	Not supported
dfilt.fftfir	Overlap-add FIR	Not supported
dfilt.latticeallpass	Lattice allpass	Supported
dfilt.latticear	Lattice autoregressive (AR)	Supported
dfilt.latticearma	Lattice autoregressive moving-average (ARMA)	Supported
dfilt.latticemamax	Lattice moving-average (MA) for maximum phase	Supported
dfilt.latticemamin	Lattice moving-average (MA) for minimum phase	Supported

<b>dfilt.structure</b>	<b>Description</b>	<b>Coefficient Mapping Support in realizemdl</b>
<code>dfilt.calattice</code>	Coupled, allpass lattice	Supported
<code>dfilt.calatticepc</code>	Coupled, allpass lattice with power complementary output	Supported
<code>dfilt.statespace</code>	State-space	Supported.
<code>dfilt.scalar</code>	Scalar gain object	Supported
<code>dfilt.wdfallpass</code>	Allpass wave digital filter object	Supported
<code>dfilt.cascade</code>	Filters arranged in series	Supported
<code>dfilt.parallel</code>	Filters arranged in parallel	Supported

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.

<b>Design Method String</b>	<b>Filter Design Result</b>
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
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 the specifications object `d` changes, the available methods for designing filters from `d` also change. For instance, if `d` is a lowpass filter with the default specification '`Fp,Fst,Ap,Ast`', the applicable methods are:

```
% Create an object to design a lowpass filter.
d=fdesign.lowpass;
designmethods(d) % What design methods apply to object d?
```

If you change the specification string to 'N,F3dB', the available design methods change:

```
d=fdesign.lowpass('N,F3dB');  
designmethods(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 <code>cascade</code> or <code>parallel</code> object, where a stage is a separate, modular filter. Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .
block	(Available only with Signal Processing Blockset)  <code>block(hd)</code> creates a Signal Processing Blockset block of the <code>dfilt</code> object. The <code>block</code> 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 <code>block</code> method ('on') and defined by <code>Blockname</code> . 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 <code>dfilt</code> objects. Refer to <code>dfilt.cascade</code> .
coeffs	Returns the filter coefficients in a structure containing fields that use the same property names as those in the original <code>dfilt</code> .
convert	Converts a <code>dfilt</code> object from one filter structure, to another filter structure

Method	Description
<code>fcfwrite</code>	<p>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 (<code>mfilt</code>) or adaptive filters (<code>adaptfilt</code>). Default filename is <code>untitled.fcf</code>.</p> <p><code>fcfwrite(hd,filename)</code> writes to a disk file named <code>filename</code> in the current working folder. The <code>.fcf</code> extension is added automatically.</p> <p><code>fcfwrite(...,fmt)</code> writes the coefficients in the format <code>fmt</code>, where valid <code>fmt</code> strings are:</p> <ul style="list-style-type: none"> <li>'hex' for hexadecimal</li> <li>'dec' for decimal</li> <li>'bin' for binary representation.</li> </ul>
<code>fftcoeffs</code>	Returns the frequency-domain coefficients used when filtering with a <code>dfilt.fftfir</code>
<code>filter</code>	Performs filtering using the <code>dfilt</code> object
<code>firtype</code>	Returns the type (1-4) of a linear phase FIR filter
<code>freqz</code>	Plots the frequency response in <code>fvtool</code> . Note that unlike the <code>freqz</code> function, this <code>dfilt</code> <code>freqz</code> method has a default length of 8192.
<code>grpdelay</code>	Plots the group delay in <code>fvtool</code>
<code>impz</code>	Plots the impulse response in <code>fvtool</code>
<code>impzlength</code>	Returns the length of the impulse response
<code>info</code>	Displays <code>dfilt</code> information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length.

<b>Method</b>	<b>Description</b>
isallpass	Returns a logical 1 (i.e., true) if the <code>dfilt</code> object is in an allpass filter or a logical 0 (i.e., false) if it is not
iscascade	Returns a logical 1 if the <code>dfilt</code> object is cascaded or a logical 0 if it is not
isfir	Returns a logical 1 if the <code>dfilt</code> object has finite impulse response (FIR) or a logical 0 if it does not
islinphase	Returns a logical 1 if the <code>dfilt</code> object is linear phase or a logical 0 if it is not
ismaxphase	Returns a logical 1 if the <code>dfilt</code> object is maximum-phase or a logical 0 if it is not
isminphase	Returns a logical 1 if the <code>dfilt</code> object is minimum-phase or a logical 0 if it is not
isparallel	Returns a logical 1 if the <code>dfilt</code> object has parallel stages or a logical 0 if it does not
isreal	Returns a logical 1 if the <code>dfilt</code> object has real-valued coefficients or a logical 0 if it does not
isscalar	Returns a logical 1 if the <code>dfilt</code> object is a scalar or a logical 0 if it is not scalar
issos	Returns a logical 1 if the <code>dfilt</code> object has second-order sections or a logical 0 if it does not
isstable	Returns a logical 1 if the <code>dfilt</code> 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
realizemdl	<p>(Available only with Simulink )</p> <p>realizemdl(hd) creates a Simulink model containing a subsystem block realization of your dfilt.</p> <p>realizemdl(hd,p1,v1,p2,v2,...) creates the block using the properties p1, p2,... and values v1, v2,... specified.</p> <p>The following properties are available:</p> <p>'Blockname' specifies the name of the block. The default value is 'Filter'.</p> <p>'Destination' specifies whether to add the block to a current Simulink model or create a</p>

Method	Description
	<p>new model. Valid values are 'Current' and 'New'.</p> <p>'OverwriteBlock' specifies whether to overwrite an existing block that was created by <code>realizemdl</code> or create a new block. Valid values are 'on' and 'off'. Note that only blocks created by <code>realizemdl</code> are overwritten.</p> <p>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'.</p> <p>'OptimizeZeros' removes zero-gain blocks.</p> <p>'OptimizeOnes' replaces unity-gain blocks with a direct connection.</p> <p>'OptimizeNegOnes' replaces negative unity-gain blocks with a sign change at the nearest summation block.</p> <p>'OptimizeDelayChains' replaces cascaded chains of delay block with a single integer delay block set to the appropriate delay.</p>
removestage	Removes a stage from a cascade or parallel <code>dfilt</code> . Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .
setstage	Overwrites a stage of a cascade or parallel <code>dfilt</code> . Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .

Method	Description
sos	<p>Converts the <code>dfilt</code> to a second-order sections <code>dfilt</code>. If <code>hd</code> has a single section, the returned filter has the same class.</p> <p><code>sos(hd, flag)</code> specifies the ordering of the second-order sections. If <code>flag='UP'</code>, the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If <code>flag='down'</code>, the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them.</p> <p><code>sos(hd, flag, scale)</code> specifies the scaling of the gain and the numerator coefficients of all second-order sections. <code>scale</code> can be <code>'none'</code>, <code>'inf'</code> (infinity-norm) or <code>'two'</code> (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.</p>
ss	<p>Converts the <code>dfilt</code> to state-space. To see the separate <code>A, B, C, D</code> matrices for the state-space model, use <code>[A, B, C, D]=ss(hd)</code>.</p>
stepz	<p>Plots the step response in <code>fvtool</code></p> <p><code>stepz(hd, n)</code> computes the first <code>n</code> samples of the step response.</p> <p><code>stepz(hd, n, Fs)</code> separates the time samples by <math>T = 1/Fs</math>, where <code>Fs</code> is assumed to be in Hz.</p>
tf	<p>Converts the <code>dfilt</code> to a transfer function</p>
zerophase	<p>Plots the zero-phase response in <code>fvtool</code></p>



---

Method	Description
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 Fixed-Point Toolbox software, 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);
isstable(hd)
```

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

```
B = get(hd,'numerator');
% or
B1 = hd.numerator;
```

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 filter object
[b,a] = butter(5,.5);        % Create IIR filter
hd(2) = dfilt.df2t(b,a);     % Create DF2T object and place
                             % in the second column of hd.

[h,w] = freqz(hd);
test_fir = isfir(hd)
% 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

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \dots + z^{-n}}{1 + c(1)z^{-1} + \dots + c(n)z^{-n}}$$

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. <b>False</b> 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 <code>filtstates</code> 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)
```

## See Also

`dfilt`, `dfilt.cascadeallpass`, `dfilt.cascadewdfallpass`, `dfilt.latticeallpass`, `mfilt.iirdecim`, `mfilt.iirinterp`

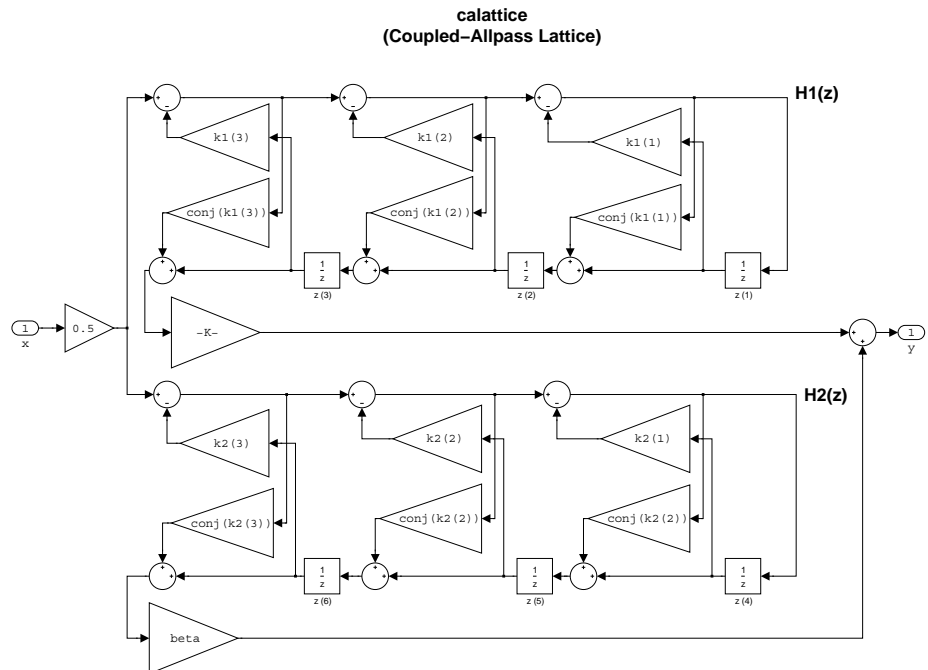
# dfilt.calattice

**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)
```

The `Allpass1` and `Allpass2` properties store vectors of coefficients.

```
hd.Allpass1
hd.Allpass2
```

**See Also**

`dfilt.calatticepc`  
`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`,  
`dfilt.latticearma`, `dfilt.latticemamax`, `dfilt.latticemamin` in  
Signal Processing Toolbox documentation

# dfilt.calatticepc

---

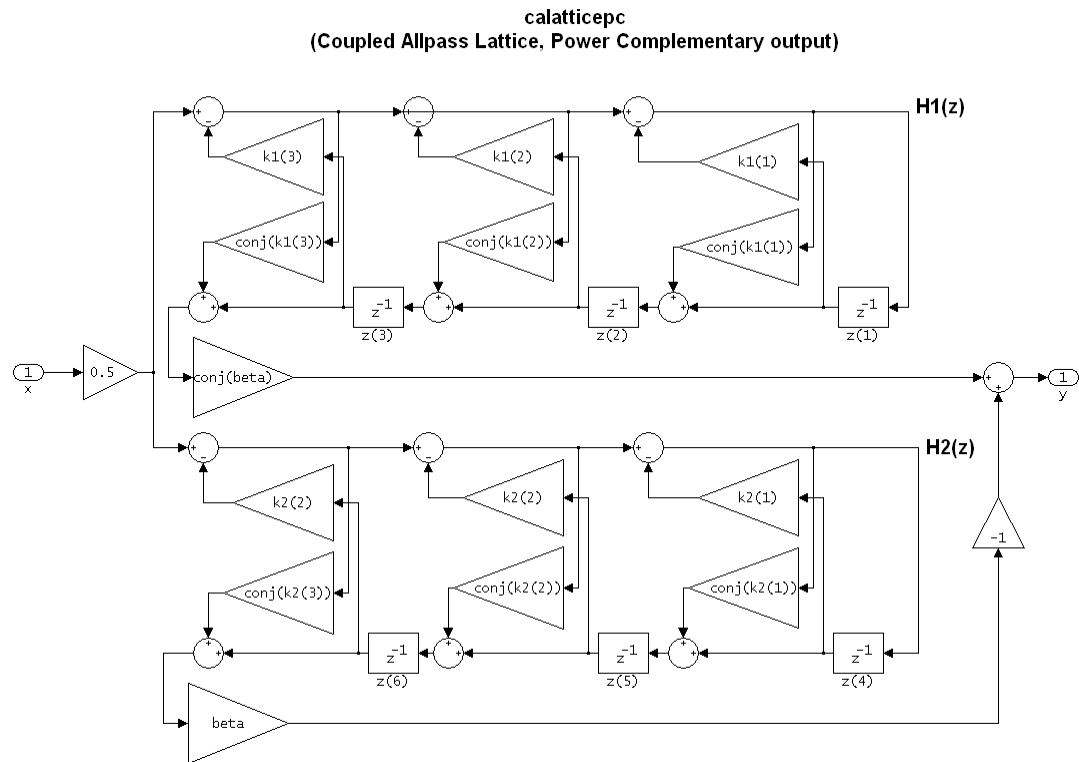
**Purpose** Coupled-allpass, power-complementary lattice filter

**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  $h_d$  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);
```

To see the coefficients for Allpass1, check the property values.

## dfilt.calatticepc

---

```
get(hd, 'Allpass1')
```

### 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.

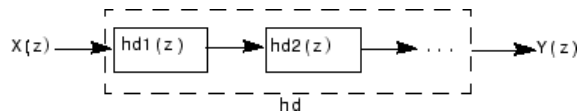
**Description**

`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
```

To view the details of one filter stage, use

# dfilt.cascade

---

`hcas.Stage(1)`

## **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
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. <code>False</code> 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 <code>filtstates</code> 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);
```

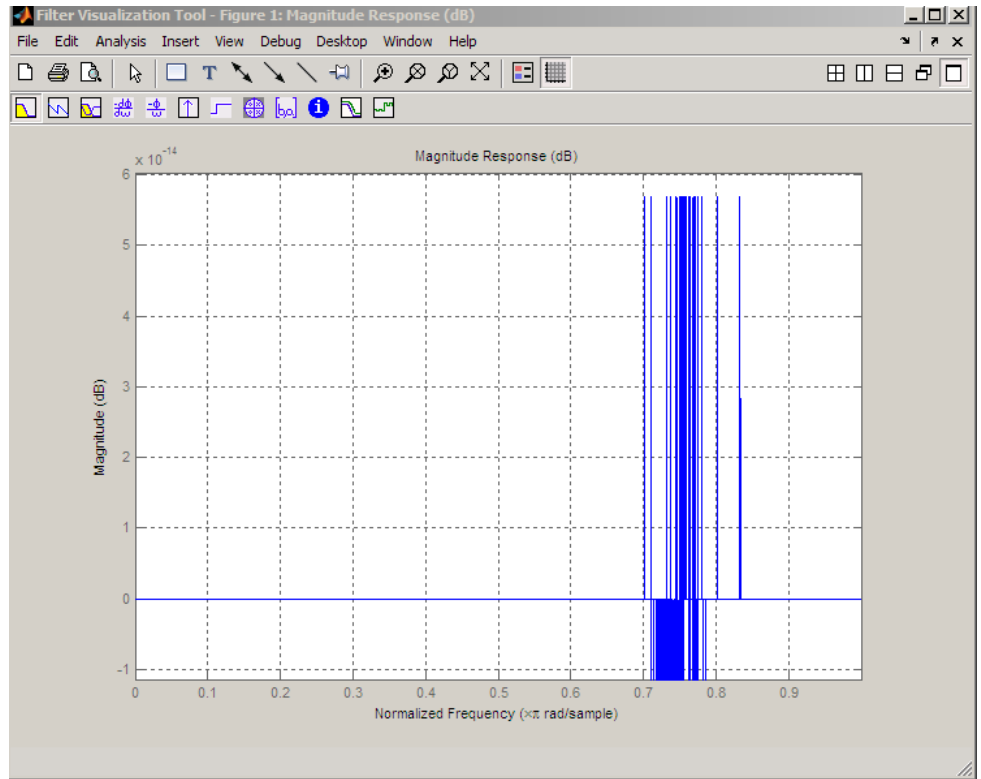
Now perform the actual filter design. `hd` contains two `dfilt.cascadeallpass` objects.

```
hd = design(d,'ellip','filterstructure','cascadeallpass');  
% Get summary information about one dfilt.cascadeallpass stage.  
hd.Stage(1).Stage(1)
```

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.
```

# dfilt.cascadeallpass



## 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.

# dfilt.cascadewdfallpass

---

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. <code>False</code> 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 <code>filtstates</code> 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.

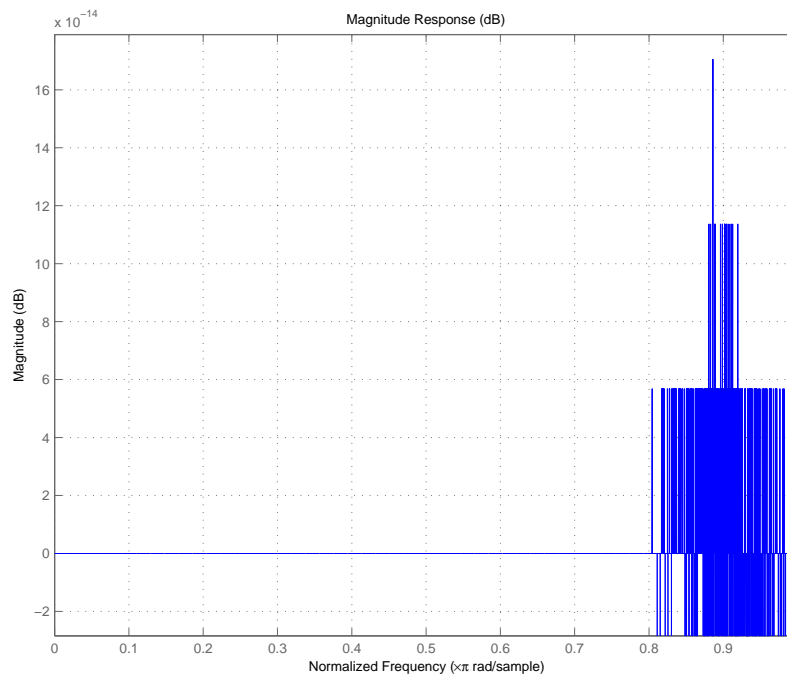
```
hd = design(d,'ellip','filterstructure','cascadewdfallpass');
hd.stage(1).stage(2) % Summary info on dfilt.cascadewdfallpass.
% Requires Simulink to realize model.
realizemdl(hd.stage(1).stage(2))
```

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);
fvtool(hd) % Visualize the filter.
info(hd)
```

# dfilt.cascadewdfallpass

---



**See Also** [dfilt](#), [dfilt.wdfallpass](#)

**Purpose** Delay filter

**Syntax**  
Hd = dfilt.delay  
Hd = dfilt.delay(latency)

**Description** Hd = dfilt.delay returns a discrete-time filter, Hd, of type delay, which adds a single delay to any signal filtered with Hd. The filtered signal has its values shifted by one sample.

Hd = dfilt.delay(latency) returns a discrete-time filter, Hd, of type delay, which adds the number of delay units specified in latency to any signal filtered with Hd. The filtered signal has its values shifted by the latency number of samples. The values that appear before the shifted signal are the filter states.

**Examples** Create a delay filter with a latency of 4 and filter a simple signal to view the impact of applying a delay.

```
h = dfilt.delay(4)
h =
    FilterStructure: 'Delay'
        Latency: 4
    PersistentMemory: false

sig = 1:7      % Create some simple signal data
sig =
     1     2     3     4     5     6     7

states = h.states % Filter states before filtering
states =
     0
     0
     0
     0

filter(h,sig) % Filter using the delay filter
ans =
```

# dfilt.delay

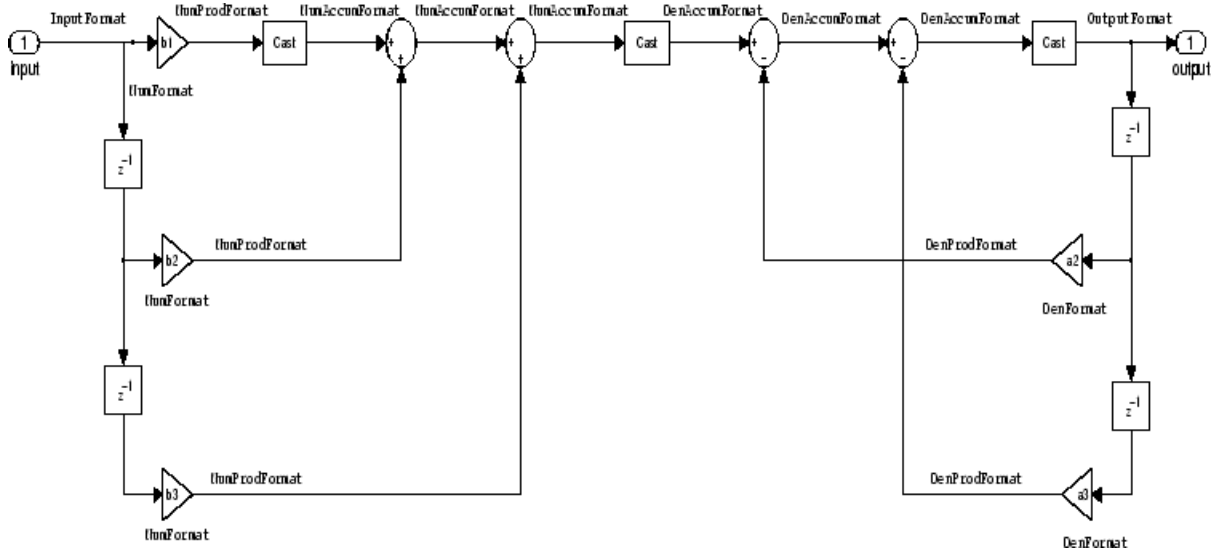
---

```
          0    0    0    0    1    2    3
states=h.states    % Filter states after filtering
states =
    4
    5
    6
    7
```

**See Also**      [dfilt](#)

---

<b>Purpose</b>	Discrete-time, direct-form I filter
<b>Syntax</b>	Refer to <code>dfilt.df1</code> in Signal Processing Toolbox documentation.
<b>Description</b>	<p><code>hd = dfilt.df1</code> returns a default discrete-time, direct-form I filter object that uses double-precision arithmetic. By default, the numerator and denominator coefficients <code>b</code> and <code>a</code> are set to 1. With these coefficients the filter passes the input to the output without changes.</p> <p>Make this filter a fixed-point or single-precision filter by changing the value of the <code>Arithmetic</code> property for the filter <code>hd</code> as follows:</p> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <pre>set(hd,'arithmetic','single');</pre></li><li>• To change to fixed-point filtering, enter <pre>set(hd,'arithmetic','fixed');</pre></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “Arithmetic” on page 3-20.</p> <hr/> <p><b>Note</b> <code>a(1)</code>, the leading denominator coefficient, cannot be 0. To allow you to change the arithmetic setting to <code>fixed</code> or <code>single</code>, <code>a(1)</code> must be equal to 1.</p> <hr/>
<b>Fixed-Point Filter Structure</b>	<p>The following figure shows the signal flow for the direct-form I filter implemented by <code>dfilt.df1</code>. 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.</p>



## 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.



<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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” on page 3-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .
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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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.

<b>Property Name</b>	<b>Brief Description</b>
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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .
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 <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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 ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) 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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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. <code>False</code> is the default setting.

Property Name	Brief Description
RoundMode	<p>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).</p> <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>

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 <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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)
```

Now convert `hd` to a fixed-point filter:

```
set(hd,'arithmetic','fixed')
```

## 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`” on page 3-20.

`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.

---

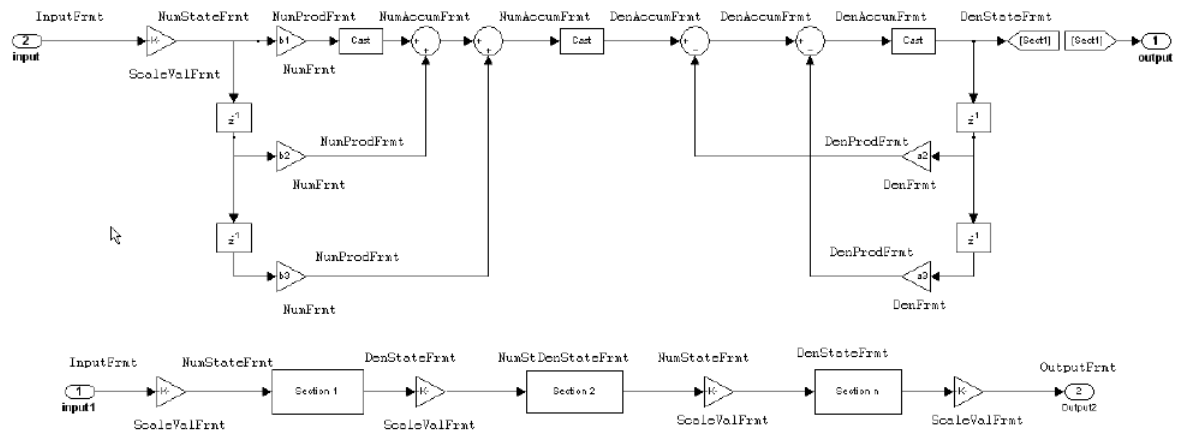
**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---



## 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.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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” on page 3-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .

<b>Property Name</b>	<b>Brief Description</b>
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .

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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
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.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
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: <ul style="list-style-type: none"><li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>
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 <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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 ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) 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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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. <code>False</code> is the default setting.

Property Name	Brief Description
RoundMode	<p>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).</p> <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>
ScaleValueFracLength	<p>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 <code>AutoScaleMode</code> by setting it to <code>false</code>.</p>



Property Name	Brief Description
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 <code>double</code> 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 <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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.arithmetic='fixed'
```

## 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`” on page 3-20.

`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.

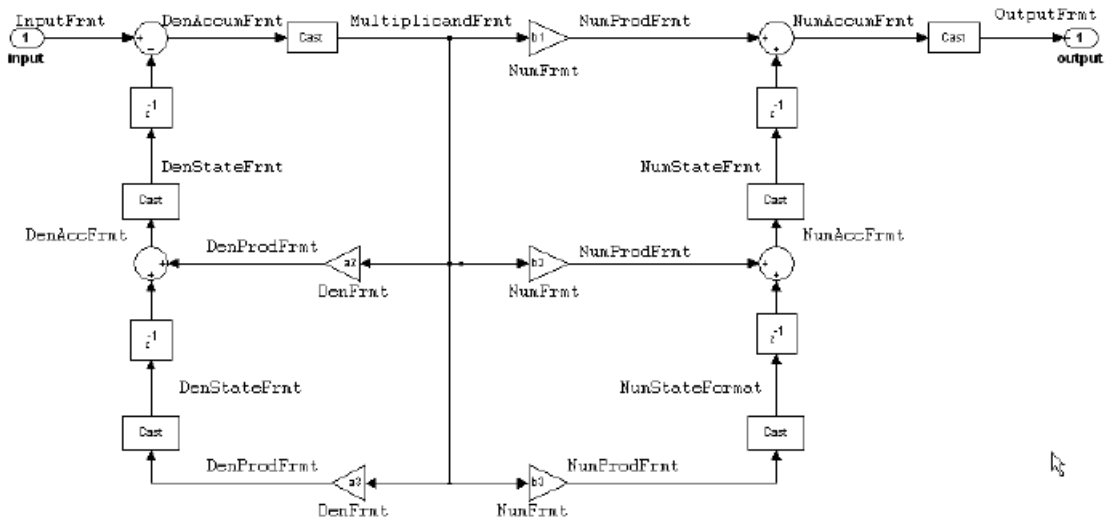
---

**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 `InputFrm` 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 `NumFrm`, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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 dflt 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” on page 3-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
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 <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .
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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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 <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"><li>• <code>AvoidOverflow</code> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <code>BestPrecision</code> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li></ul>



Property Name	Brief Description
	<ul style="list-style-type: none"> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul>
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 <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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 ( <b>FullPrecision</b> ), or whether to keep the most significant bit ( <b>KeepMSB</b> ) or least significant bit ( <b>KeepLSB</b> ) 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 <b>ProductMode</b> to <b>SpecifyPrecision</b> .

<b>Property Name</b>	<b>Brief Description</b>
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.
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). <ul style="list-style-type: none"><li>• <b>ceil</b> - Round toward positive infinity.</li><li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <b>fix</b> - Round toward zero.</li><li>• <b>floor</b> - Round toward negative infinity.</li><li>• <b>nearest</b> - Round toward nearest. Ties round toward positive infinity.</li><li>• <b>round</b> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul>

Property Name	Brief Description
	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.
StateAutoScale	Setting autoscaling for filter states to <code>true</code> reduces the possibility of overflows occurring during fixed-point operations. Set to <code>false</code> , <code>StateAutoScale</code> 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)
```

Now convert the filter to single-precision filtering arithmetic.

```
set(hd,'arithmetic','single')
```

## dfilt.df1t

---

### **See Also**

dfilt, dfilt.df1, dfilt.df2, dfilt.df2t

<b>Purpose</b>	Discrete-time, SOS direct-form I transposed filter
<b>Syntax</b>	Refer to <code>dfilt.df1tsos</code> in Signal Processing Toolbox documentation.
<b>Description</b>	<p><code>hd = dfilt.df1tsos(s)</code> returns a discrete-time, second-order section, direct-form I, transposed filter object <code>hd</code>, with coefficients given in the <code>s</code> matrix.</p> <p>Make this filter a fixed-point or single-precision filter by changing the value of the <code>Arithmetic</code> property for the filter <code>hd</code> as follows:</p> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <pre>set(hd,'arithmetic','single');</pre></li><li>• To change to fixed-point filtering, enter <pre>set(hd,'arithmetic','fixed');</pre></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to .</p> <p><code>hd = dfilt.df1tsos(b1,a1,b2,a2,...)</code> returns a discrete-time, second-order section, direct-form I, transposed filter object <code>hd</code>, with coefficients for the first section given in the <code>b1</code> and <code>a1</code> vectors, for the second section given in the <code>b2</code> and <code>a2</code> vectors, etc.</p> <p><code>hd = dfilt.df1tsos(...,g)</code> includes a gain vector <code>g</code>. The elements of <code>g</code> are the gains for each section. The maximum length of <code>g</code> is the number of sections plus one. If <code>g</code> is not specified, all gains default to one.</p> <p><code>hd = dfilt.df1tsos</code> returns a default, discrete-time, second-order section, direct-form I, transposed filter object, <code>hd</code>. This filter passes the input through to the output unchanged.</p>

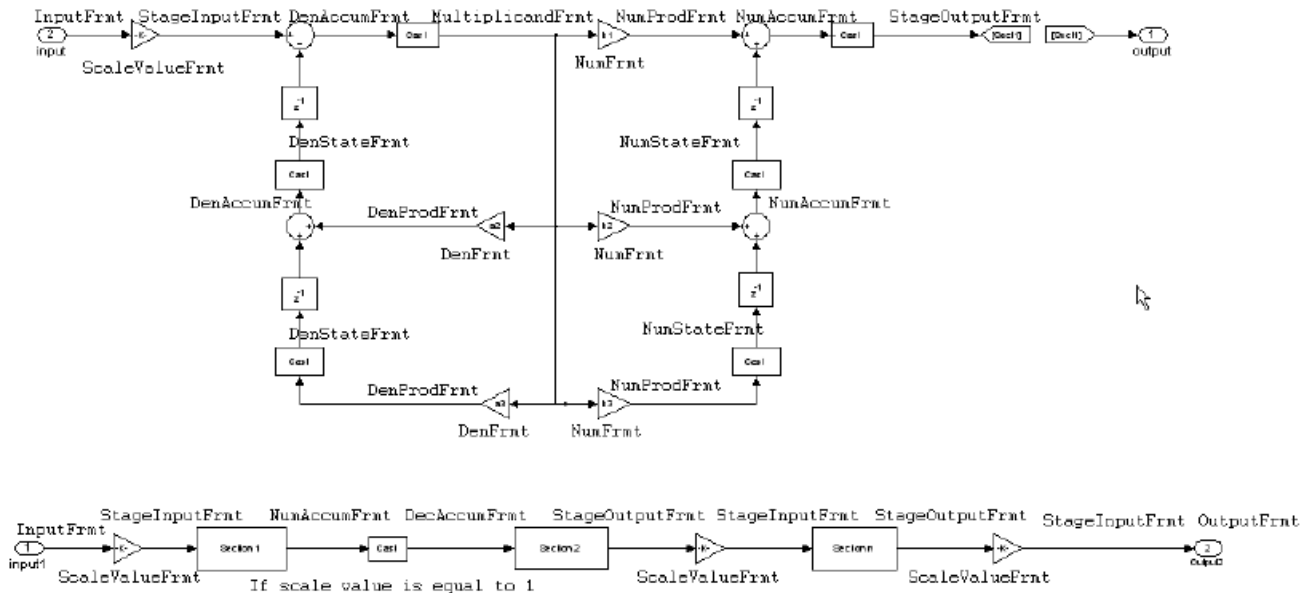
---

**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.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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	NumStateWordLength	NumStateFracLength	States
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFrmt	CoeffWordLength	ScaleValue-FracLength	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 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” on page 3-2.



Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used.

<b>Property Name</b>	<b>Brief Description</b>
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.
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.

<b>Property Name</b>	<b>Brief Description</b>
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.
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.

Property Name	Brief Description
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.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
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: <ul style="list-style-type: none"><li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>

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 <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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 ( <b>FullPrecision</b> ), or whether to keep the most significant bit ( <b>KeepMSB</b> ) or least significant bit ( <b>KeepLSB</b> ) 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 <b>ProductMode</b> to <b>SpecifyPrecision</b> .

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.
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). <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li></ul>

Property Name	Brief Description
	<ul style="list-style-type: none"><li>• round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>
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.

Property Name	Brief Description
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.
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)  
set(hd, 'arithmetic', 'fixed')
```



**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`” on page 3-20.

`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.

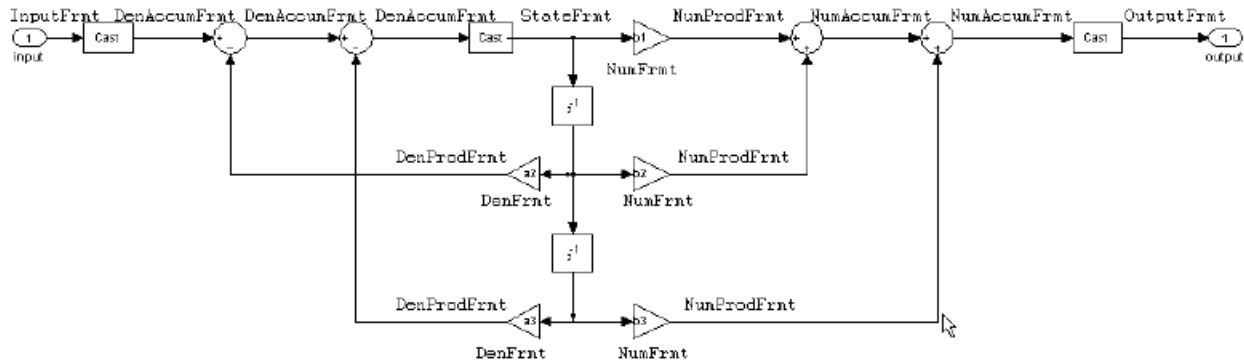
---

**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.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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” on page 3-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.

<b>Property Name</b>	<b>Brief Description</b>
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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .
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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .

<b>Property Name</b>	<b>Brief Description</b>
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	<p>Sets the mode the filter uses to scale the filtered data for output. You have the following choices:</p> <ul style="list-style-type: none"><li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>
OutputWordLength	<p>Determines the word length used for the output data.</p>
OverflowMode	<p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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.</p>



Property Name	Brief Description
ProductMode	<p>Determines how the filter handles the output of product operations. Choose from full precision (<code>FullPrecision</code>), or whether to keep the most significant bit (<code>KeepMSB</code>) or least significant bit (<code>KeepLSB</code>) 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 <code>ProductMode</code> to <code>SpecifyPrecision</code>.</p>
PersistentMemory	<p>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. <code>False</code> is the default setting.</p>
RoundMode	<p>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).</p> <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative</li></ul>

Property Name	Brief Description
	numbers, and toward positive infinity for positive numbers.  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.
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)
```

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`

- 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`” on page 3-20.

`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.

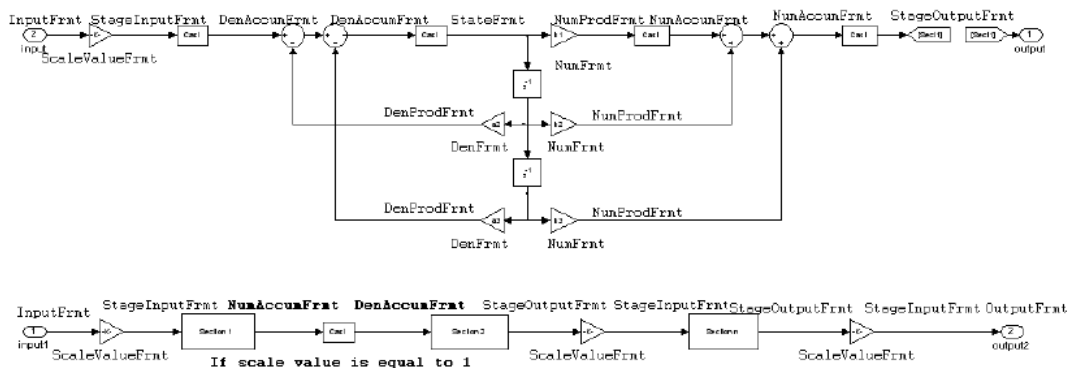
---

**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.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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
ScaleValueFrmnt	CoeffWordLength	ScaleValue-FracLength	CoeffAutoScale, ScaleValues
SectionInputFormt	SectionInput-WordLength	SectionInput-FracLength	SectionInput-AutoScale
SectionOutputFrmt	SectionOutput-WordLength	SectionOutput-FracLength	SectionOutput-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” on page 3-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.



<b>Property Name</b>	<b>Brief Description</b>
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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
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.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
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: <ul style="list-style-type: none"><li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>
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 <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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 ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) 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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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. <code>False</code> is the default setting.

Property Name	Brief Description
RoundMode	<p>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).</p> <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>
ScaleValueFracLength	<p>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 <code>AutoScaleMode</code> by setting it to <code>false</code>.</p>

Property Name	Brief Description
ScaleValues	Scaling for the filter objects in SOS filters.
SectionInputAutoScale	Tells the filter whether to set the stage input data format to minimize the occurrence of overflow conditions.
SectionInputFracLength	Lets you set the fraction length for section inputs in SOS filters, if you set SectionInputAutoScale to false.
SectionInputWordLength	Lets you set the word length for section inputs in SOS filters, if you set SectionInputAutoScale to false.
SectionOutputAutoScale	Tells the filter whether to set the section output data format to minimize the occurrence of overflow conditions.
SectionOutputFracLength	Lets you set the fraction length for section outputs in SOS filters, if you set SectionOutputAutoScale to off.
SectionOutputWordLength	Lets you set the word length for section outputs in SOS filters, if you set SectionOutputAutoScale to false.
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.

# dfilt.df2sos

---

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 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)
```

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.arithmetic='fixed';
```

## See Also

`dfilt`, `dfilt.df1sos`, `dfilt.df1tsos`, `dfilt.df2tsos`

---

<b>Purpose</b>	Discrete-time, direct-form II transposed filter
<b>Syntax</b>	Refer to <code>dfilt.df2t</code> in Signal Processing Toolbox documentation.
<b>Description</b>	<p><code>hd = dfilt.df2t(b,a)</code> returns a discrete-time, direct-form II transposed filter object <code>hd</code>, with numerator coefficients <code>b</code> and denominator coefficients <code>a</code>.</p> <p>Make this filter a fixed-point or single-precision filter by changing the value of the <code>Arithmetic</code> property for the filter <code>hd</code> as follows:</p>

- 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” on page 3-20.

`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.

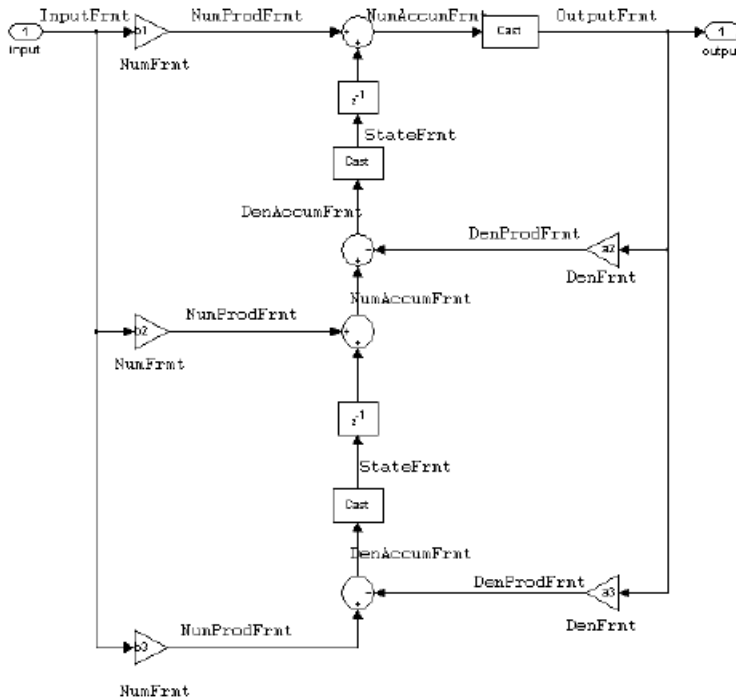
---

**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.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
<code>DenAccumFrmt</code>	<code>AccumWordLength</code>	<code>DenAccumFracLength</code>	<code>AccumMode</code> , <code>CastBeforeSum</code>
<code>DenFrmt</code>	<code>CoeffWordLength</code>	<code>DenFracLength</code>	<code>CoeffAutoScale</code> , <code>Signed</code> , <code>Denominator</code>
<code>DenProdFrmt</code>	<code>CoeffWordLength</code>	<code>DenProdFracLength</code>	<code>ProductMode</code> , <code>ProductWordLength</code>
<code>InputFrmt</code>	<code>InputWordLength</code>	<code>InputFracLength</code>	None
<code>NumAccumFrmt</code>	<code>AccumWordLength</code>	<code>NumAccumFracLength</code>	<code>AccumMode</code> , <code>CastBeforeSum</code>
<code>NumFrmt</code>	<code>CoeffWordLength</code>	<code>NumFracLength</code>	<code>CoeffAutoScale</code> , <code>Signed</code> , <code>Numerator</code>
<code>NumProdFrmt</code>	<code>CoeffWordLength</code>	<code>NumProdFracLength</code>	<code>ProductWordLength</code> , <code>ProductMode</code>
<code>OutputFrmt</code>	<code>OutputWordLength</code>	<code>OutputFracLength</code>	<code>OutputMode</code>
<code>StateFrmt</code>	<code>StateWordLength</code>	<code>StateFracLength</code>	<code>States</code>

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” on page 3-2.

Property Name	Brief Description
<code>AccumMode</code>	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
<code>AccumWordLength</code>	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 <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .
Denominator	Holds the denominator coefficients for IIR filters.

<b>Property Name</b>	<b>Brief Description</b>
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .

Property Name	Brief Description
OutputMode	<p>Sets the mode the filter uses to scale the filtered data for output. You have the following choices:</p> <ul style="list-style-type: none"><li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>
OutputWordLength	<p>Determines the word length used for the output data.</p>
OverflowMode	<p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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.</p>

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) 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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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. <code>False</code> 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). <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li></ul>

Property Name	Brief Description
	<ul style="list-style-type: none"> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>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.</p>
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 <code>true</code> reduces the possibility of overflows occurring during fixed-point operations. Set to <code>false</code> , <code>StateAutoScale</code> lets the filter select the fraction length to limit the overflow potential.
StateFracLength	When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> 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`

<b>Purpose</b>	Discrete-time, SOS direct-form II transposed filter
<b>Syntax</b>	Refer to <code>dfilt.df2tsos</code> in Signal Processing Toolbox documentation.
<b>Description</b>	<p><code>hd = dfilt.df2tsos(s)</code> returns a discrete-time, second-order section, direct-form II, transposed filter object <code>hd</code>, with coefficients given in the matrix <code>s</code>.</p> <p>Make this filter a fixed-point or single-precision filter by changing the value of the <code>Arithmetic</code> property for the filter <code>hd</code> as follows:</p> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <pre>set(hd,'arithmetic','single');</pre></li><li>• To change to fixed-point filtering, enter <pre>set(hd,'arithmetic','fixed');</pre></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “<code>Arithmetic</code>” on page 3-20.</p> <p><code>hd = dfilt.df2tsos(b1,a1,b2,a2,...)</code> returns a discrete-time, second-order section, direct-form II, transposed filter object <code>hd</code>, with coefficients for the first section given in the <code>b1</code> and <code>a1</code> vectors, for the second section given in the <code>b2</code> and <code>a2</code> vectors, etc.</p> <p><code>hd = dfilt.df2tsos(...,g)</code> includes a gain vector <code>g</code>. The elements of <code>g</code> are the gains for each section. The maximum length of <code>g</code> is the number of sections plus one. If <code>g</code> is not specified, all gains default to one.</p> <p><code>hd = dfilt.df2tsos</code> returns a default, discrete-time, second-order section, direct-form II, transposed filter object, <code>hd</code>. This filter passes the input through to the output unchanged.</p>

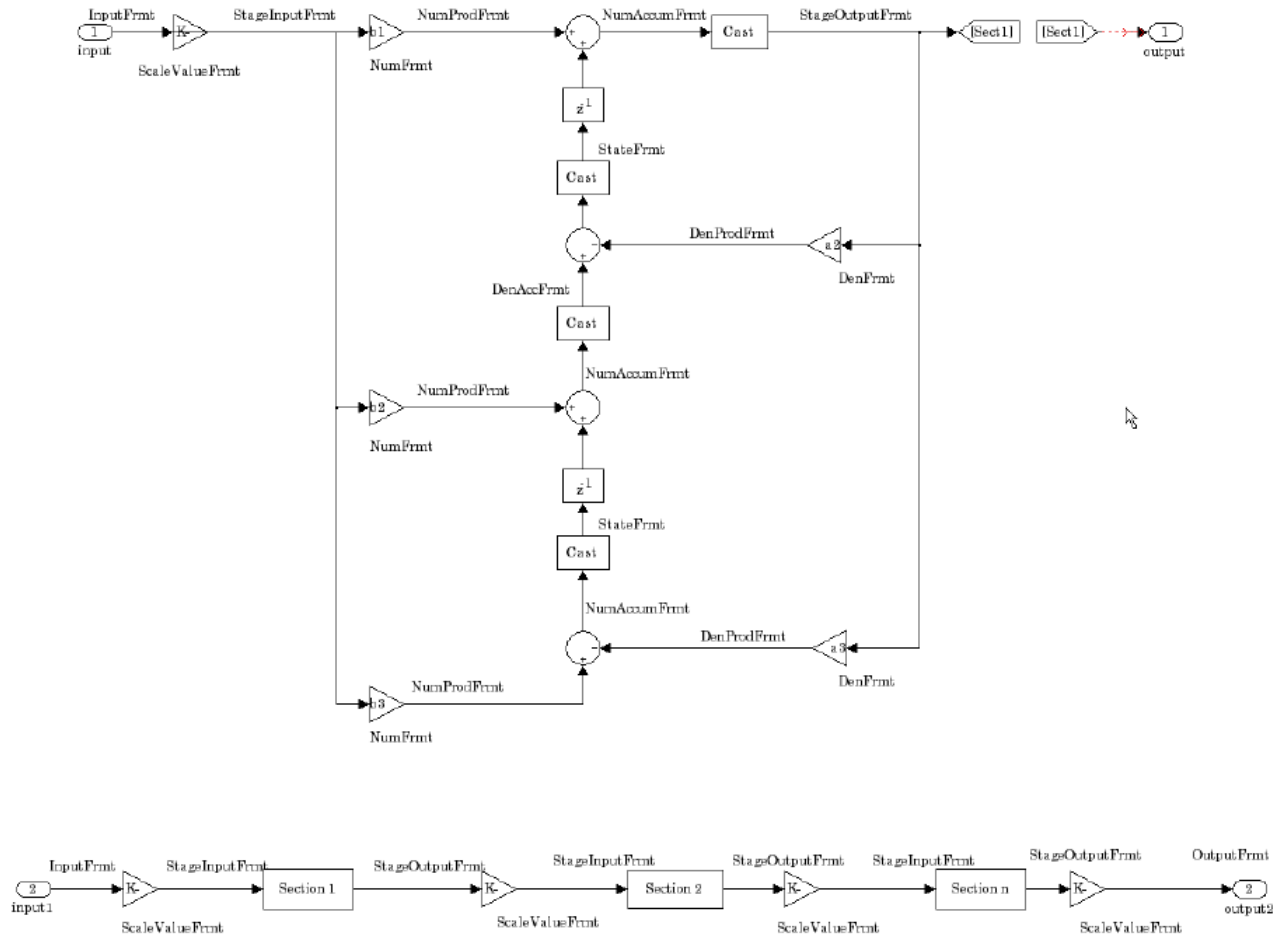
---

**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.df2tsos`. 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.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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
SectionInputFormt	SectionInput-WordLength	SectionInput-FracLength	
SectionOutputFormt	SectionOutput-WordLength	SectionOutput-FracLength	
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” on page 3-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .



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.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	<p>Sets the mode the filter uses to scale the filtered data for output. You have the following choices:</p> <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul>
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 <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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 ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) 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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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. <code>False</code> is the default setting.

Property Name	Brief Description
RoundMode	<p>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).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>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.</p>
ScaleValueFracLength	<p>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 <code>AutoScaleMode</code> by setting it to <code>false</code>.</p>
ScaleValues	<p>Scaling for the filter objects in SOS filters.</p>
Signed	<p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>

Property Name	Brief Description
SosMatrix	Holds the filter coefficients as property values — you use <code>set</code> and <code>get</code> to modify them. Displays the matrix in the format [sections x coefficients/section data type].
SectionInputFracLength	Lets you set the fraction length for section inputs in SOS filters, if you set <code>SectionInputAutoScale</code> to <code>false</code> .
SectionInputWordLength	Lets you set the word length for section inputs in SOS filters, if you set <code>SectionInputAutoScale</code> to <code>false</code> .
SectionOutputFracLength	Lets you set the fraction length for section outputs in SOS filters, if you set <code>SectionOutputAutoScale</code> to <code>off</code> .
SectionOutputWordLength	Lets you set the word length for section outputs in SOS filters, if you set <code>SectionOutputAutoScale</code> to <code>false</code> .
StateAutoScale	Setting autoscaling for filter states to <code>true</code> reduces the possibility of overflows occurring during fixed-point operations. Set to <code>false</code> , <code>StateAutoScale</code> lets the filter select the fraction length to limit the overflow potential.
StateFracLength	When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> 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

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: 'Direct-Form II Transposed, Second-Order Sections'
      Arithmetic: 'double'
      sosMatrix: [15x6 double]
      ScaleValues: [2.51598549209694e-008;1;1;1;1;1;1;1;1;1;1;1;1;1;1]
  OptimizeScaleValues: true
    PersistentMemory: false
```

Now change the setting of the property `Arithmetic` to convert the filter to fixed-point operation.

```
>> hd.arithmetic='fixed'
```

```
hd =
```

```
    FilterStructure: 'Direct-Form II Transposed, Second-Order Sections'
      Arithmetic: 'fixed'
      sosMatrix: [15x6 double]
      ScaleValues: [2.51602614298463e-008;1;1;1;1;1;1;1;1;1;1;1;1;1;1]
  OptimizeScaleValues: true
    PersistentMemory: false
```

```
    CoeffWordLength: 16
      CoeffAutoScale: true
          Signed: true
```

```
    InputWordLength: 16
      InputFracLength: 15
```

```
    SectionInputWordLength: 16
      SectionInputFracLength: 15
```

```
    SectionOutputWordLength: 16
```

```
SectionOutputFracLength: 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`” on page 3-20.

`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.

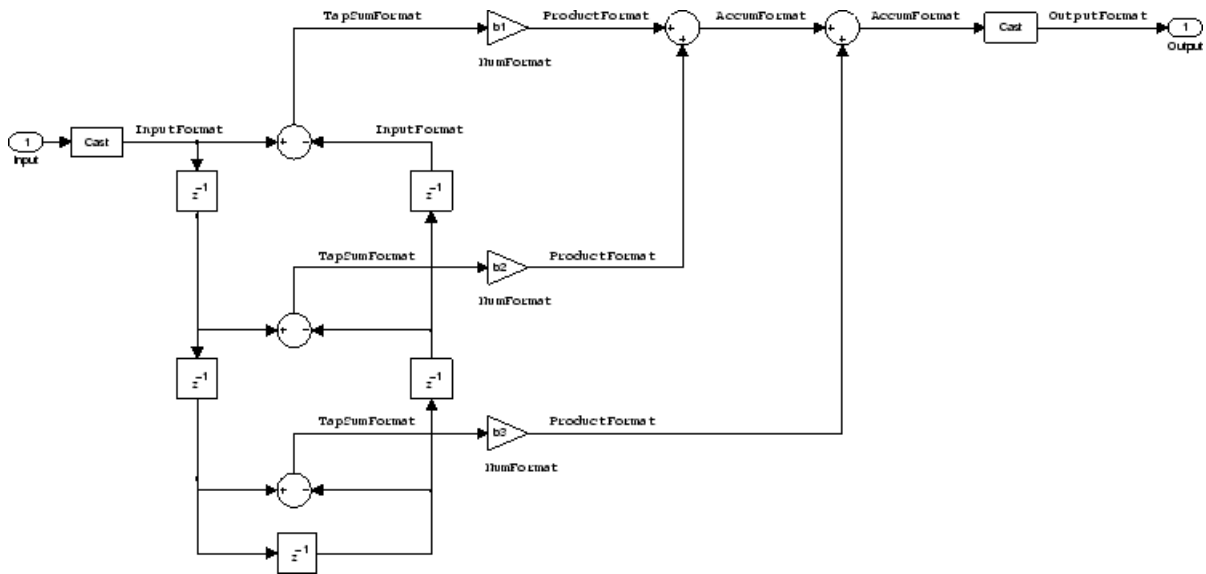
---

**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.



<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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” on page 3-2.

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 <code>fixed</code> 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 <code>false</code> enables you to change the <code>NumFracLength</code> 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 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.

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
RoundMode	[convergent], ceil, fix, floor, nearest, 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). <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li></ul>

Name	Values	Description
		<ul style="list-style-type: none"> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>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.</p>
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 Fixed-Point Toolbox documentation.

## 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);
set(hd,'arithmetic','fixed');
```

## dfilt.dfasymfir

---

Now look at the coefficients after converting `hd` to fixed-point format.

```
get(hd, 'numerator')
```

You can also obtain the filter coefficients with:

```
hd.numerator
```

### Even Order

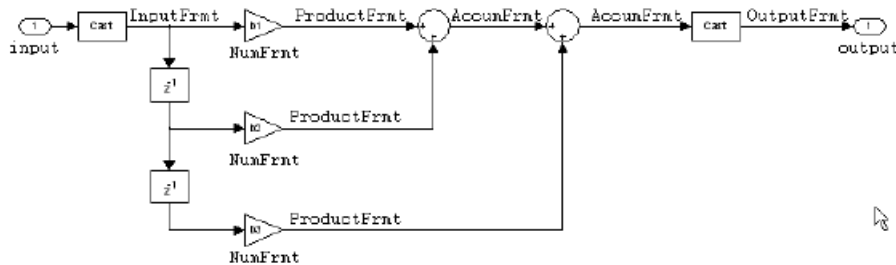
Specify a fourth-order direct-form antisymmetric FIR filter structure for `dfilt` object `hd`, with the following code:

```
b = [-0.01 0.1 0.0 -0.1 0.01];  
hd = dfilt.dfasymfir(b);  
hd.arithmetic='fixed';  
Filter_coefs = get(hd, 'numerator');  
% or equivalently  
Filter_coefs = hd.numerator;
```

### See Also

`dfilt`, `dfilt.dffir`, `dfilt.dffirt`, `dfilt.dfsymfir`

<b>Purpose</b>	Discrete-time, direct-form FIR filter
<b>Syntax</b>	Refer to <code>dfilt.dffir</code> in Signal Processing Toolbox documentation.
<b>Description</b>	<p><code>hd = dfilt.dffir(b)</code> returns a discrete-time, direct-form finite impulse response (FIR) filter object <code>hd</code>, with numerator coefficients <code>b</code>.</p> <p>Make this filter a fixed-point or single-precision filter by changing the value of the <code>Arithmetic</code> property for the filter <code>hd</code> as follows:</p> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <pre>set(hd,'arithmetic','single');</pre></li><li>• To change to fixed-point filtering, enter <pre>set(hd,'arithmetic','fixed');</pre></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “Arithmetic” on page 3-20.</p> <p><code>hd = dfilt.dffir</code> returns a default, discrete-time, direct-form FIR filter object <code>hd</code>, with <code>b=1</code>. This filter passes the input through to the output unchanged.</p>
<b>Fixed-Point Filter Structure</b>	<p>The following figure shows the signal flow for the direct-form FIR filter implemented by <code>dfilt.dffir</code>. 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.</p>



## 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

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” on page 3-2.

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 <code>fixed</code> 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 <code>false</code> enables you to change the <code>NumFracLength</code> 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, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> 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 <code>OutputFracLength</code> when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .
OutputWordLength	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting <code>FilterInternals</code> to <code>SpecifyPrecision</code> .
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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.
RoundMode	[convergent], ceil, fix, floor, nearest, 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). <ul style="list-style-type: none"><li>• <b>ceil</b> - Round toward positive infinity.</li><li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <b>fix</b> - Round toward zero.</li><li>• <b>floor</b> - Round toward negative infinity.</li><li>• <b>nearest</b> - Round toward nearest. Ties round toward positive infinity.</li><li>• <b>round</b> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> The choice you make affects only the accumulator and output arithmetic. Coefficient

Name	Values	Description
		and input arithmetic always round. Finally, products never overflow — they maintain full precision.
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 Fixed-Point Toolbox documentation.

## 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.arithmetic='fixed';
hd.filterInternals='specifyPrecision';
```

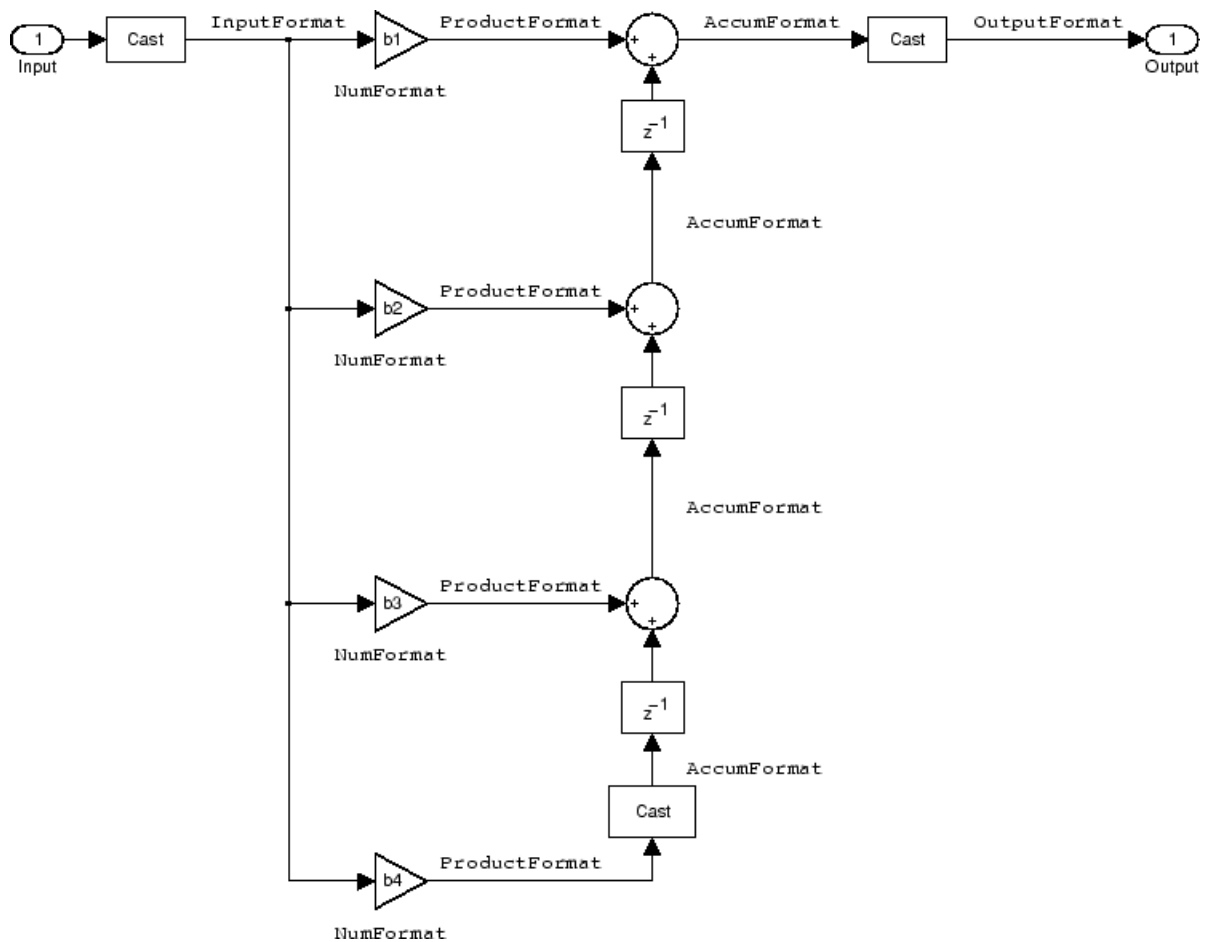
## See Also

`dfilt`, `dfilt.dfasymfir`, `dfilt.dffirt`, `dfilt.dfsymfir`

# dfilt.dffirt

---

<b>Purpose</b>	Discrete-time, direct-form FIR transposed filter
<b>Syntax</b>	Refer to <code>dfilt.dffirt</code> in Signal Processing Toolbox documentation.
<b>Description</b>	<p><code>hd = dfilt.dffirt(b)</code> returns a discrete-time, direct-form FIR transposed filter object <code>hd</code>, with numerator coefficients <code>b</code>.</p> <p>Make this filter a fixed-point or single-precision filter by changing the value of the <code>Arithmetic</code> property for the filter <code>hd</code> as follows:</p> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <pre>set(hd, 'arithmetic', 'single');</pre></li><li>• To change to fixed-point filtering, enter <pre>set(hd, 'arithmetic', 'fixed');</pre></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “<code>Arithmetic</code>” on page 3-20.</p> <p><code>hd = dfilt.dffirt</code> returns a default, discrete-time, direct-form FIR transposed filter object <code>hd</code>, with <code>b = 1</code>. This filter passes the input through to the output unchanged.</p>
<b>Fixed-Point Filter Structure</b>	<p>The following figure shows the signal flow for the transposed direct-form FIR filter implemented by <code>dfilt.dffirt</code>. 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.</p>



### 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.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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” on page 3-2.

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 <code>fixed</code> 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 <code>false</code> enables you to change the <code>NumFracLength</code> 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 FilterInternals 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]	<p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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.</p>
RoundMode	[convergent], ceil, fix, floor, nearest, round	<p>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).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul>

# dfilt.dffirt

Name	Values	Description
		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	[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 Fixed-Point Toolbox documentation.

## 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);
```

Now use the filter property `Arithmetic` to change the filter to fixed-point format.

```
set(hd,'arithmetic','fixed')
```

## 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” on page 3-20.

`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.

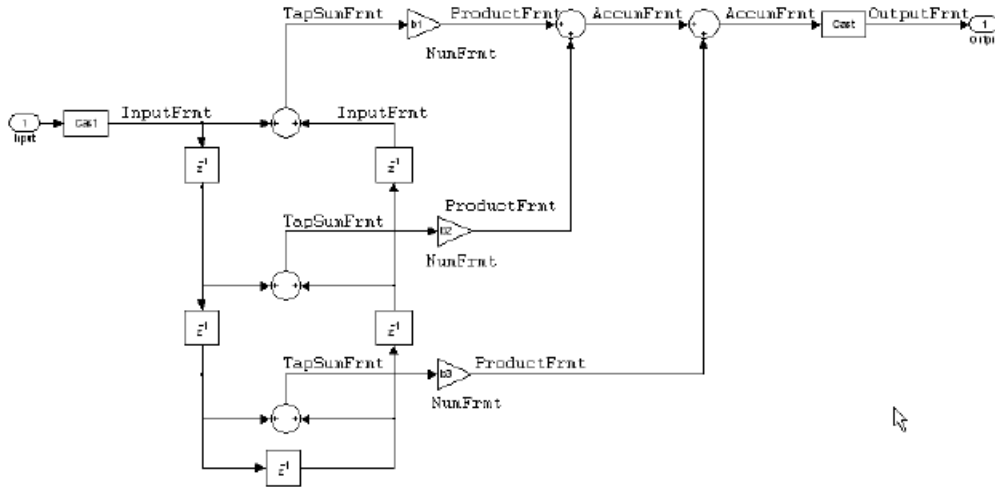
---

**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.

<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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” on page 3-2.

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 <code>fixed</code> 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 <code>false</code> enables you to change the <code>NumFracLength</code> 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 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.

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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 <b>ProductMode</b> to <b>SpecifyPrecision</b> .
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 <b>ProductMode</b> to <b>SpecifyPrecision</b> .

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	<p>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).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>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.</p>

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 Fixed-Point Toolbox documentation.

## 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
```

# dfilt.dfsymfir

---

```
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);  
set(hd, 'arithmetic', 'fixed');
```

## See Also

`dfilt`, `dfilt.dfasymfir`, `dfilt.dffir`, `dfilt.dffirt`

**Purpose** Fractional Delay Farrow filter

**Syntax** Hd = dfilt.farrowd(D, COEFFS)

**Description** Hd = dfilt.farrowd(D, COEFFS)  
Constructs a discrete-time fractional delay Farrow filter with COEFFS coefficients and D delay.

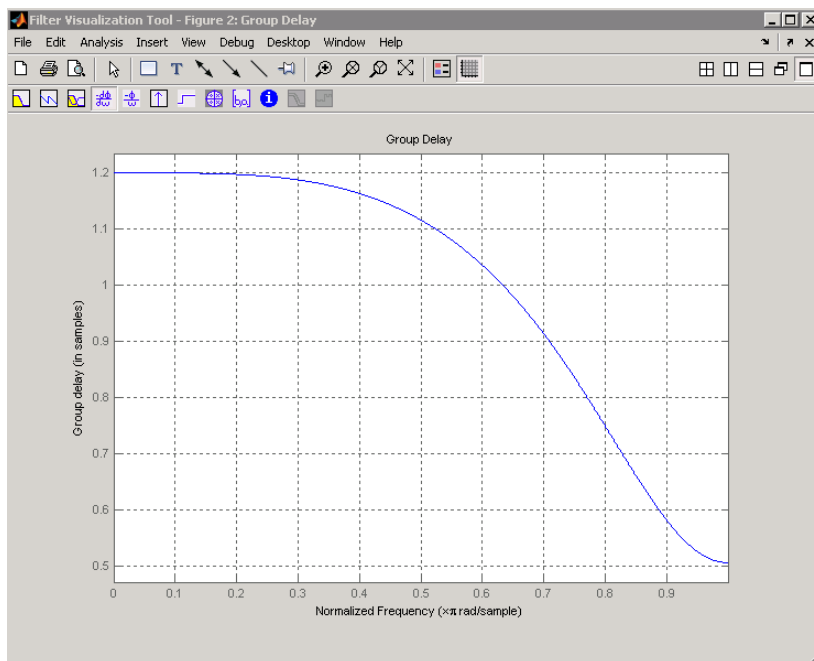
**Examples** Farrow filters can be designed with the dfilt.farrowfd filter designer.

```
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(0.5, coeffs);
```

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', 'farrowfd');  
fvtool(Hd, 'Analysis', 'grpdelay');
```

# dfilt.farrowfd



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, `farrowdemo`.

**See Also** `dfilt`

**Purpose** Discrete-time, overlap-add, FIR filter

**Syntax**  
`Hd = dfilt.fftfir(b,len)`  
`Hd = dfilt.fftfir(b)`  
`Hd = dfilt.fftfir`

**Description** This object uses the overlap-add method of block FIR filtering, which is very efficient for streaming data.

`Hd = dfilt.fftfir(b,len)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`. The block length is the number of input points to use for each overlap-add computation.

`Hd = dfilt.fftfir(b)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len=100`.

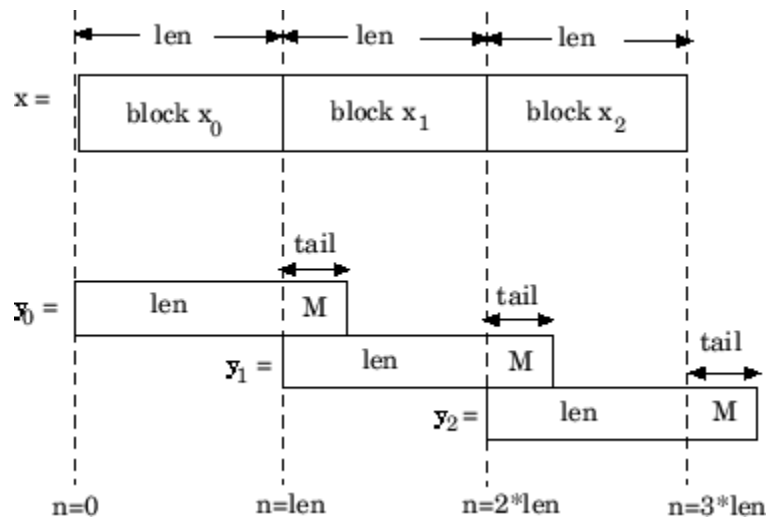
`Hd = dfilt.fftfir` returns a default, discrete-time, FFT, FIR filter, `Hd`, with the numerator `b=1` and block length, `len=100`. This filter passes the input through to the output unchanged.

---

**Note** When you use a `dfilt.fftfir` object to filter, the input signal length must be an integer multiple of the object's block length, `len`. The resulting number of FFT points = (filter length + the block length - 1). The filter is most efficient if the number of FFT points is a power of 2.

---

The `fftfir` uses an overlap-add block processing algorithm, which is represented as follows,



where  $len$  is the block length and  $M$  is the length of the numerator-1,  $(length(b) - 1)$ , which is also the number of states. The output of each convolution is a block that is longer than the input block by a tail of  $(length(b) - 1)$  samples. These tails overlap the next block and are added to it. The states reported by `dfilt.fftfir` are the tails of the final convolution.

## Examples

Create an FFT FIR discrete-time filter with coefficients from a 30<sup>th</sup> order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.fftfir(b)
Hd =
    FilterStructure: 'Overlap-Add FIR'
        Numerator: [1x31 double]
        BlockLength: 100
    NonProcessedSamples: []
        PersistentMemory: false
```

To view the frequency domain coefficients used in the filtering, use the following command.

```
fftcoeffs(Hd)
```

### **See Also**

dfilt, dfilt.dffir, dfilt.dfasymfir, dfilt.dffirt,  
dfilt.dfsymfir

## 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');
```

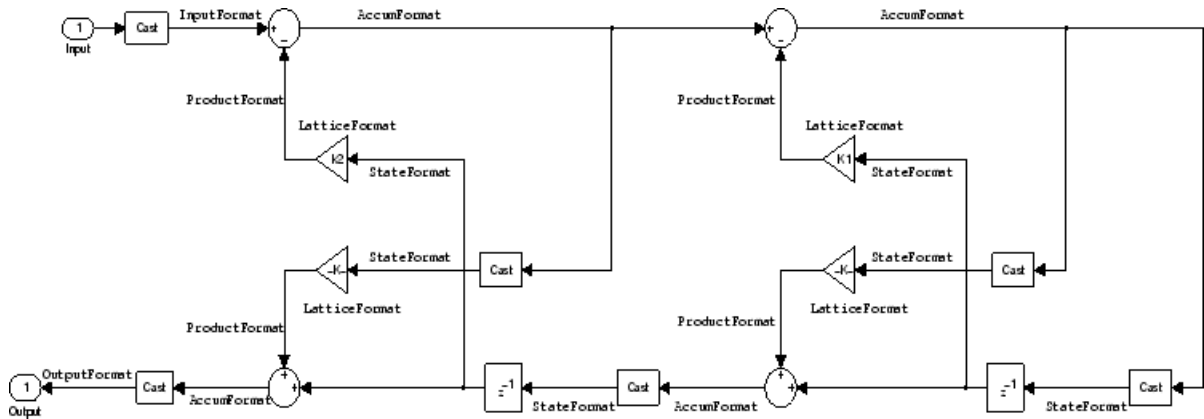
For more information about the property `Arithmetic`, refer to “Arithmetic” on page 3-20.

`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.

# dfilt.latticeallpass



## 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 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.

---

## dfilt.latticeallpass

---

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties” on page 3-2.

Property Name	Brief Description
<code>AccumFracLength</code>	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 — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.
<code>AccumMode</code>	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
<code>AccumWordLength</code>	Sets the word length used to store data in the accumulator/buffer.
<code>Arithmetic</code>	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.
<code>CastBeforeSum</code>	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 <code>false</code> enables you to change the <code>LatticeFracLength</code> 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 <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .

# dfilt.latticeallpass

Property Name	Brief Description
OutputMode	<p>Sets the mode the filter uses to scale the filtered data for output. You have the following choices:</p> <ul style="list-style-type: none"><li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>
OutputWordLength	<p>Determines the word length used for the output data.</p>
OverflowMode	<p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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.</p>
ProductFracLength	<p>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 <b>ProductMode</b> to <b>SpecifyPrecision</b>.</p>

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.
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). <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> </ul>

Property Name	Brief Description
	<ul style="list-style-type: none"><li>• nearest - Round toward nearest. Ties round toward positive infinity.</li><li>• round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> 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 <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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';
```

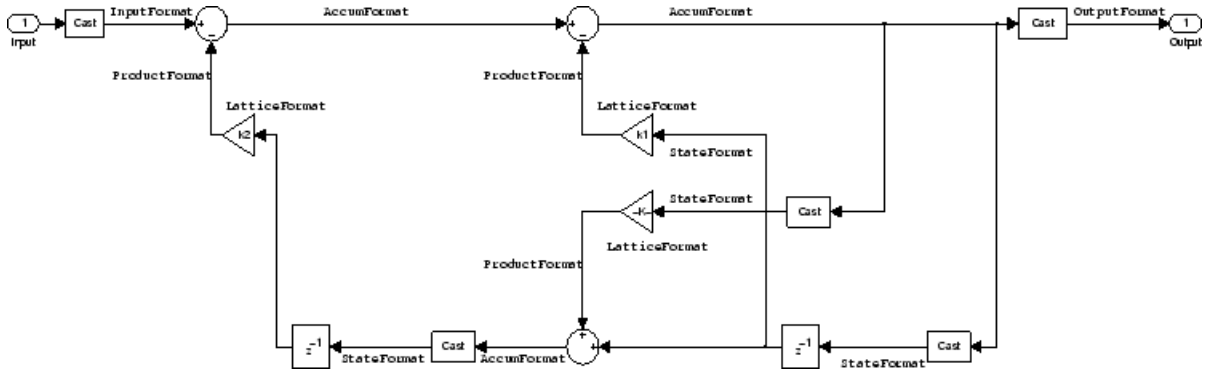
### See Also

dfilt, dfilt.latticear, dfilt.latticearma, dfilt.latticemamax,  
dfilt.latticemamin

# dfilt.latticear

---

<b>Purpose</b>	Discrete-time, lattice, autoregressive filter
<b>Syntax</b>	Refer to <code>dfilt.latticear</code> in Signal Processing Toolbox documentation.
<b>Description</b>	<p><code>hd = dfilt.latticear(k)</code> returns a discrete-time, lattice autoregressive filter object <code>hd</code>, with lattice coefficients, <code>k</code>.</p> <p>Make this filter a fixed-point or single-precision filter by changing the value of the <code>Arithmetic</code> property for the filter <code>hd</code> as follows:</p> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <pre>set(hd, 'arithmetic', 'single');</pre></li><li>• To change to fixed-point filtering, enter <pre>set(hd, 'arithmetic', 'fixed');</pre></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “<code>Arithmetic</code>” on page 3-20.</p> <p><code>hd = dfilt.latticear</code> returns a default, discrete-time, lattice autoregressive filter object <code>hd</code>, with <code>k=[]</code>. This filter passes the input through to the output unchanged.</p>
<b>Fixed-Point Filter Structure</b>	<p>The following figure shows the signal flow for the autoregressive lattice filter implemented by <code>dfilt.latticear</code>. 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.</p>



**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” on page 3-2.



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.

<b>Property Name</b>	<b>Brief Description</b>
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 <code>false</code> enables you to change the <code>LatticeFracLength</code> 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 <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .

Property Name	Brief Description
OutputMode	<p>Sets the mode the filter uses to scale the filtered data for output. You have the following choices:</p> <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul>
OutputWordLength	<p>Determines the word length used for the output data.</p>
OverflowMode	<p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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.</p>
ProductFracLength	<p>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 <b>ProductMode</b> to <b>SpecifyPrecision</b>.</p>

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision ( <b>FullPrecision</b> ), or whether to keep the most significant bit ( <b>KeepMSB</b> ) or least significant bit ( <b>KeepLSB</b> ) 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 <b>ProductMode</b> to <b>SpecifyPrecision</b> .
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <b>ProductMode</b> to <b>SpecifyPrecision</b> .
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. <b>False</b> 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). <ul style="list-style-type: none"><li>• <b>ceil</b> - Round toward positive infinity.</li><li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <b>fix</b> - Round toward zero.</li><li>• <b>floor</b> - Round toward negative infinity.</li></ul>

Property Name	Brief Description
	<ul style="list-style-type: none"> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>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.</p>
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> 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 <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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.

## dfilt.latticear

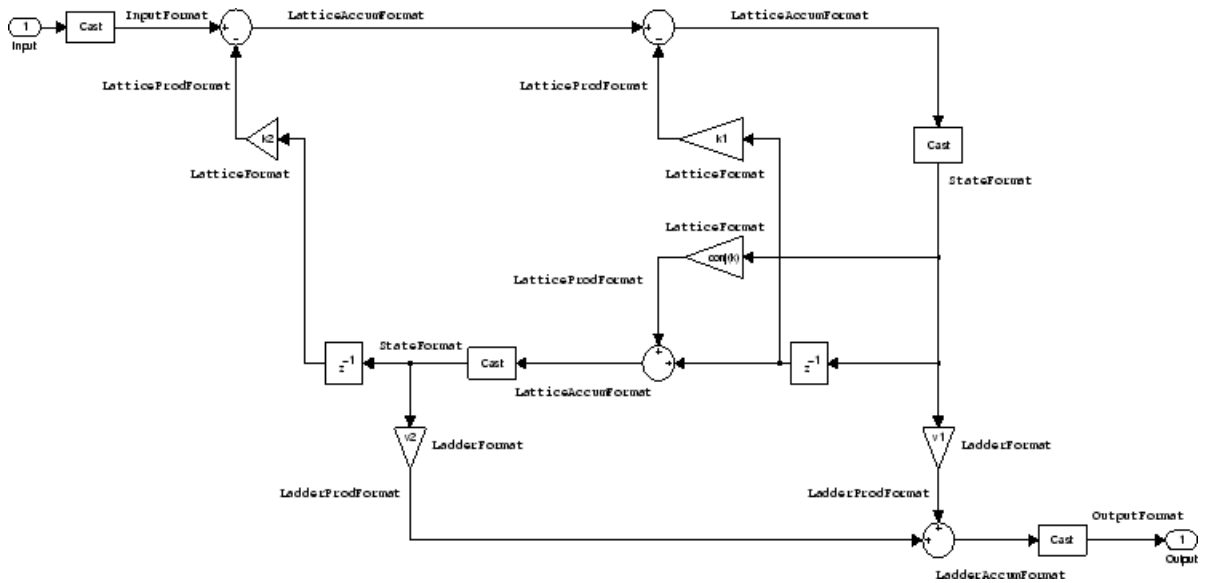
---

```
k = [.66 .7 .44];  
hd1=dfilt.latticear(k)  
hd1.arithmetic='fixed';  
specifyall(hd1)
```

### See Also

dfilt, dfilt.latticeallpass, dfilt.latticearma,  
dfilt.latticemamax, dfilt.latticemamin

<b>Purpose</b>	Discrete-time, lattice, autoregressive, moving-average filter
<b>Syntax</b>	Refer to <code>dfilt.latticearma</code> in Signal Processing Toolbox documentation.
<b>Description</b>	<p><code>hd = dfilt.latticearma(k)</code> returns a discrete-time, lattice moving-average autoregressive filter object <code>hd</code>, with lattice coefficients, <code>k</code>.</p> <p>Make this filter a fixed-point or single-precision filter by changing the value of the <code>Arithmetic</code> property for the filter <code>hd</code> as follows:</p> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <pre>set(hd,'arithmetic','single');</pre></li><li>• To change to fixed-point filtering, enter <pre>set(hd,'arithmetic','fixed');</pre></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “Arithmetic” on page 3-20.</p> <p><code>hd</code> <code>dfilt.latticearma</code> returns a default, discrete-time, lattice moving-average, autoregressive filter object <code>hd</code>, with <code>k = []</code>. This filter passes the input through to the output unchanged.</p>
<b>Fixed-Point Filter Structure</b>	<p>The following figure shows the signal flow for the autoregressive lattice filter implemented by <code>dfilt.latticearma</code>. 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.</p>



## 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.



<b>Signal Flow Label</b>	<b>Corresponding Word Length Property</b>	<b>Corresponding Fraction Length Property</b>	<b>Related Properties</b>
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” on page 3-2.

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 — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .

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 <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>LatticeFracLength</code> 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 ( <code>dfilt.latticearma</code> ) 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: <ul style="list-style-type: none"><li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>

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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) 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 <code>ProductMode</code> to <code>SpecifyPrecision</code> .

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.
RoundMode	<p>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).</p> <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>The choice you make affects only the accumulator and output arithmetic.</p>

Property Name	Brief Description
	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.
StateFracLength	When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> 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 <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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`

# dfilt.latticemamax

---

**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” on page 3-20.

---

**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.

---

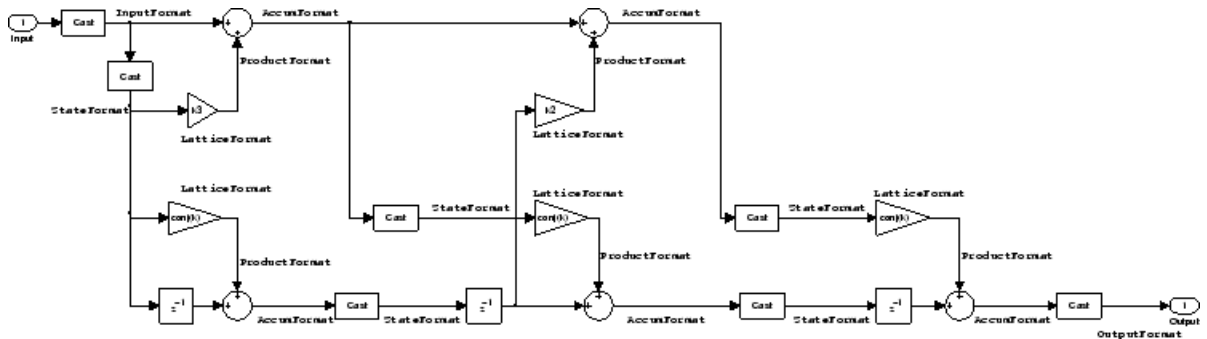
`hd = dfilt.latticemamax` returns a default discrete-time, lattice, moving-average filter object `hd`, with `k = []`. This filter passes the input through to the output unchanged.

## 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
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” on page 3-2.

Property Name	Brief Description
<code>AccumFracLength</code>	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 — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.
<code>AccumMode</code>	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
<code>AccumWordLength</code>	Sets the word length used to store data in the accumulator/buffer.
<code>Arithmetic</code>	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.
<code>CastBeforeSum</code>	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

<b>Property Name</b>	<b>Brief Description</b>
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 <code>false</code> enables you to change the <code>LatticeFracLength</code> 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 <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .

Property Name	Brief Description
OutputMode	<p>Sets the mode the filter uses to scale the filtered data for output. You have the following choices:</p> <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul>
OutputWordLength	<p>Determines the word length used for the output data.</p>
OverflowMode	<p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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.</p>
ProductFracLength	<p>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 <b>ProductMode</b> to <b>SpecifyPrecision</b>.</p>

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision ( <b>FullPrecision</b> ), or whether to keep the most significant bit ( <b>KeepMSB</b> ) or least significant bit ( <b>KeepLSB</b> ) 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 <b>ProductMode</b> to <b>SpecifyPrecision</b> .
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <b>ProductMode</b> to <b>SpecifyPrecision</b> .
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. <b>False</b> 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). <ul style="list-style-type: none"><li>• <b>ceil</b> - Round toward positive infinity.</li><li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <b>fix</b> - Round toward zero.</li><li>• <b>floor</b> - Round toward negative infinity.</li></ul>

Property Name	Brief Description
	<ul style="list-style-type: none"> <li>• nearest - Round toward nearest. Ties round toward positive infinity.</li> <li>• round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>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.</p>
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> 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 <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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:

## dfilt.latticemamax

---

```
k = [.66 .7 .44 .33];  
hd = dfilt.latticemamax(k)
```

### See Also

dfilt, dfilt.latticeallpass, dfilt.latticear,  
dfilt.latticearma, dfilt.latticemamin



**Purpose** Discrete-time, lattice, moving-average filter 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” on page 3-20.

---

**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.

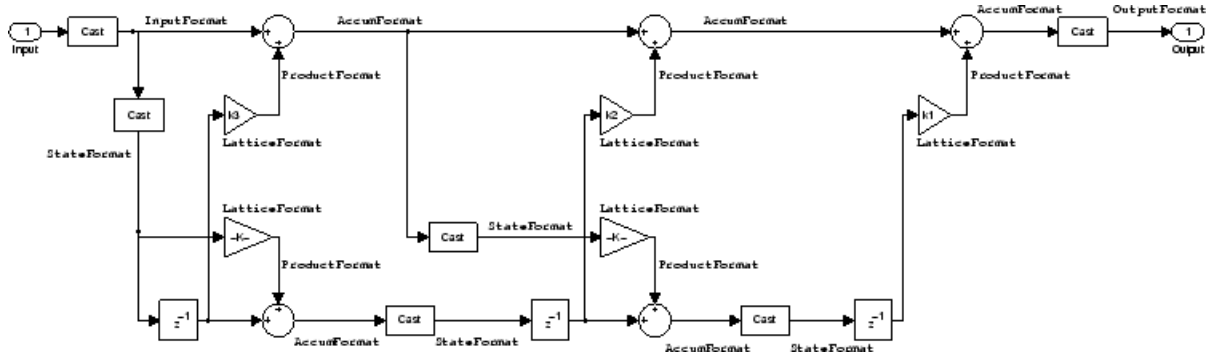
---

`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.

---

## dfilt.latticemamin

---

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties” on page 3-2.

Property Name	Brief Description
<code>AccumFracLength</code>	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 — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.
<code>AccumMode</code>	Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) 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 <code>AccumMode</code> to <code>SpecifyPrecision</code> .
<code>AccumWordLength</code>	Sets the word length used to store data in the accumulator/buffer.
<code>Arithmetic</code>	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.
<code>CastBeforeSum</code>	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 <code>false</code> enables you to change the <code>LatticeFracLength</code> 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 <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .

Property Name	Brief Description
OutputMode	<p>Sets the mode the filter uses to scale the filtered data for output. You have the following choices:</p> <ul style="list-style-type: none"><li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>
OutputWordLength	<p>Determines the word length used for the output data.</p>
OverflowMode	<p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (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.</p>
ProductFracLength	<p>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 <b>ProductMode</b> to <b>SpecifyPrecision</b>.</p>

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.
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). <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> </ul>

Property Name	Brief Description
	<ul style="list-style-type: none"><li>• nearest - Round toward nearest. Ties round toward positive infinity.</li><li>• round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> 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 <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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
```

# dfilt.latticemamin

---

```
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.

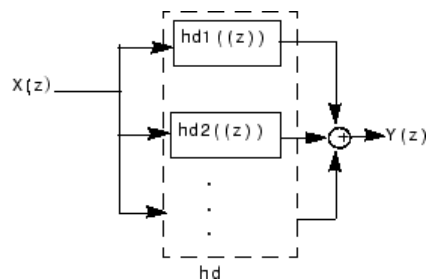
**Description**

`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`” on page 3-20.

`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” on page 3-2.

Property Name	Brief Description
Arithmetic	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>CoeffFracLength</code> property to specify the precision used.
CoeffFracLength	Set the fraction length the filter uses to interpret coefficients. <code>CoeffFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .
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 <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"><li>• <code>AvoidOverflow</code> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>• <code>BestPrecision</code> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>• <code>SpecifyPrecision</code> — lets you set the word and fraction lengths used by the output data from filtering.</li></ul>
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 <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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	<p>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. <code>False</code> is the default setting.</p>
RoundMode	<p>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).</p> <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>

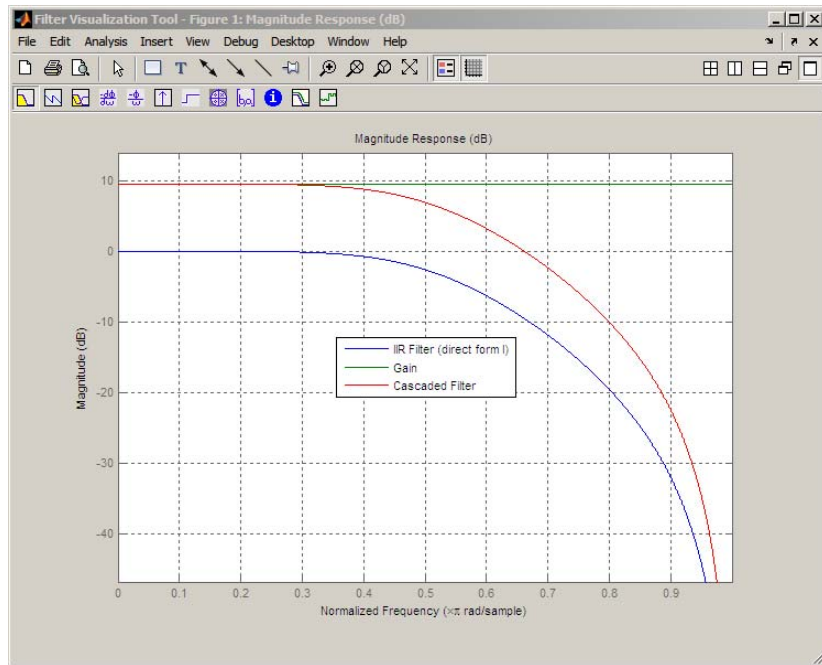
Property Name	Brief Description
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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=cascade(hd_gain,hd_filt);
fvtool_handle = fvtool(hd_filt,hd_gain,hd_cascade);
legend(fvtool_handle,'IIR Filter (direct form I)',...
'Gain','Cascaded Filter');
```





To view the stages of the cascaded filter, use

```
hd.Stage(1)
```

and

```
hd.Stage(2)
```

### See Also

`dfilt`, `dfilt.cascade`

# dfilt.wdfallpass

---

**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

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \dots + z^{-n}}{1 + c(1)z^{-1} + \dots + c(n)z^{-n}}$$

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. <code>False</code> 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 <code>filtstates</code> 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, `realizemd1` 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);
% If you have Simulink
realizemd1(hd1)

section21=[0.8,0.4];

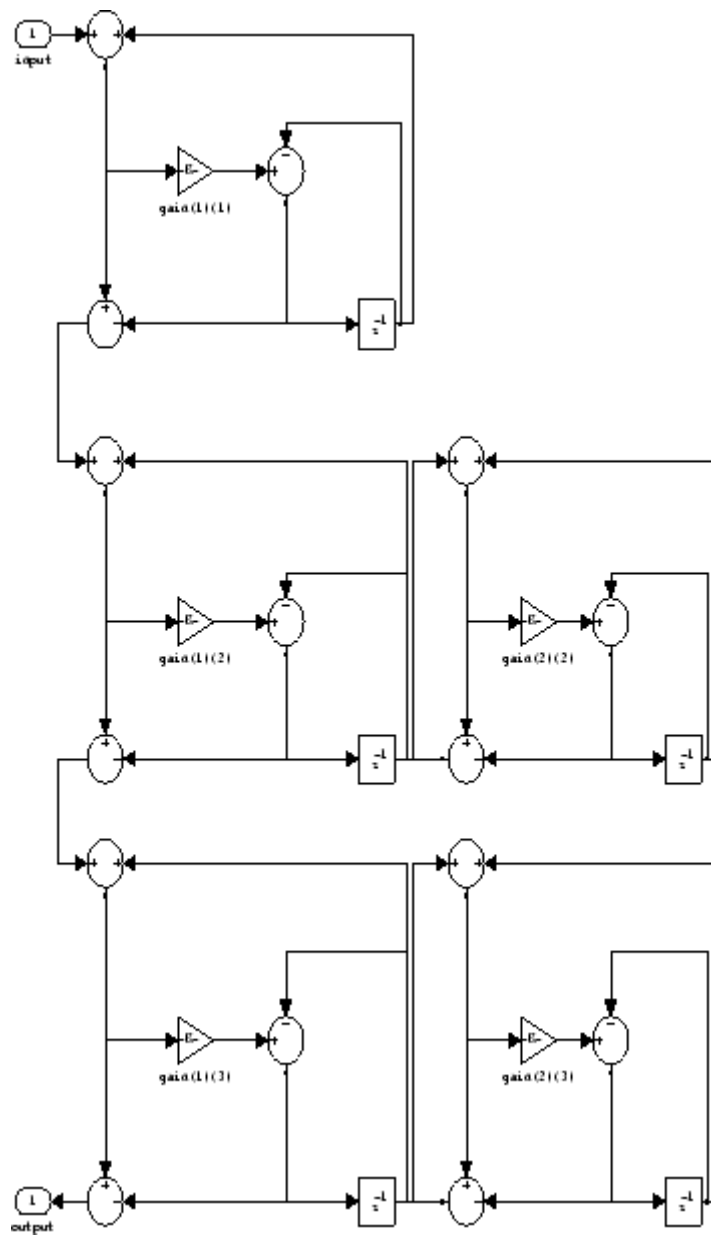
```

## dfilt.wdfallpass

---

```
section22=[0,1.5,0,0.7];  
section23=[0,1.8,0,0.9];  
hd2=dfilt.cascadewdfallpass(section21,section22,section23);  
% If you have Simulink  
realizemd1(hd2)
```

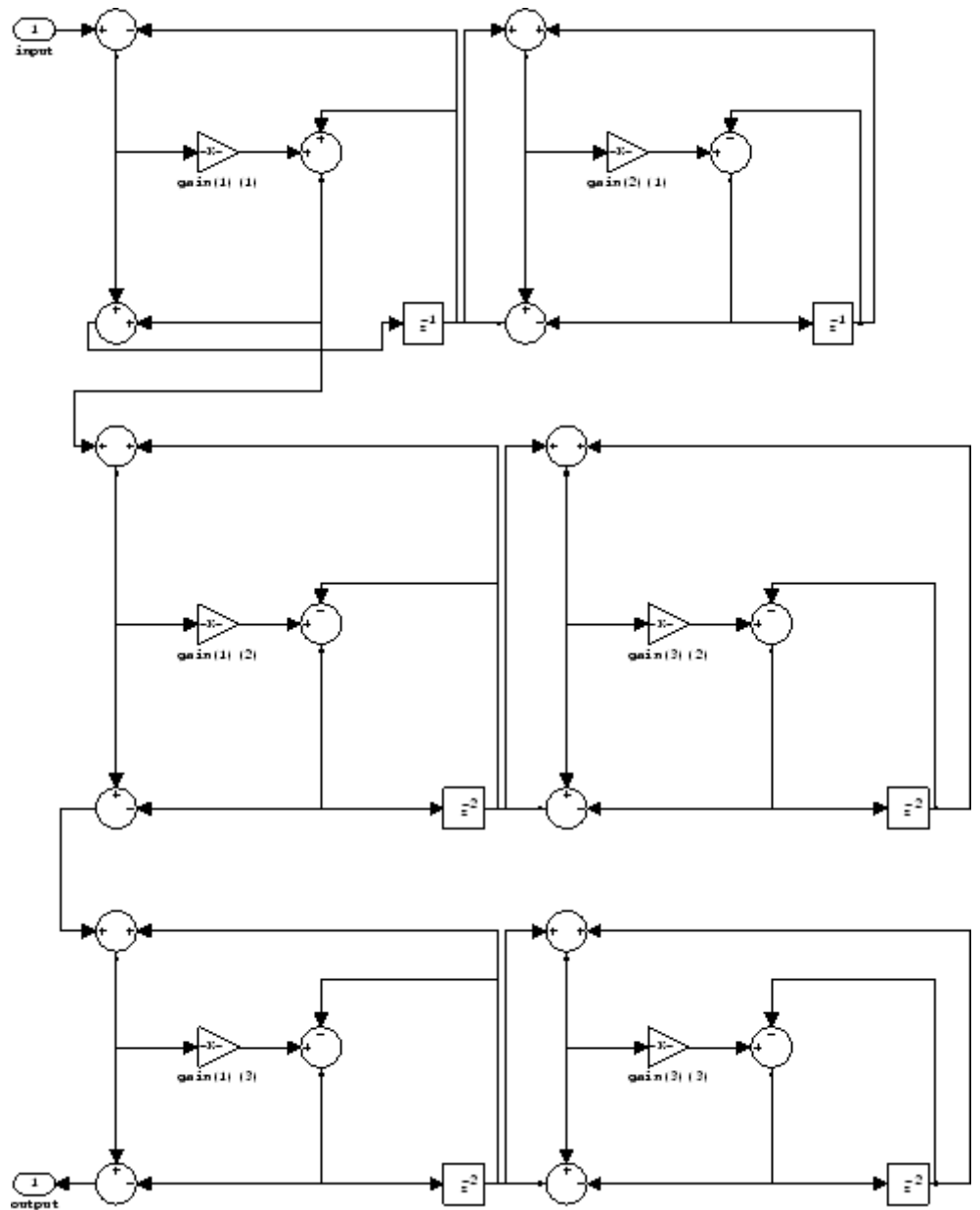
hd1 has this filter structure with three sections.



## **dfilt.wdfallpass**

---

The filter structure for `hd2` is somewhat different, with the different orders and interconnections between the three sections.



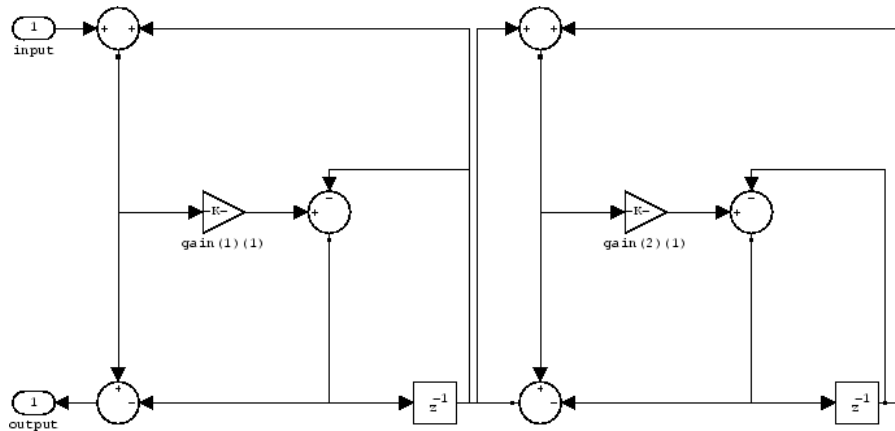
# dfilt.wdfallpass

## 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)  
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

# double

---

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 Causer 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**

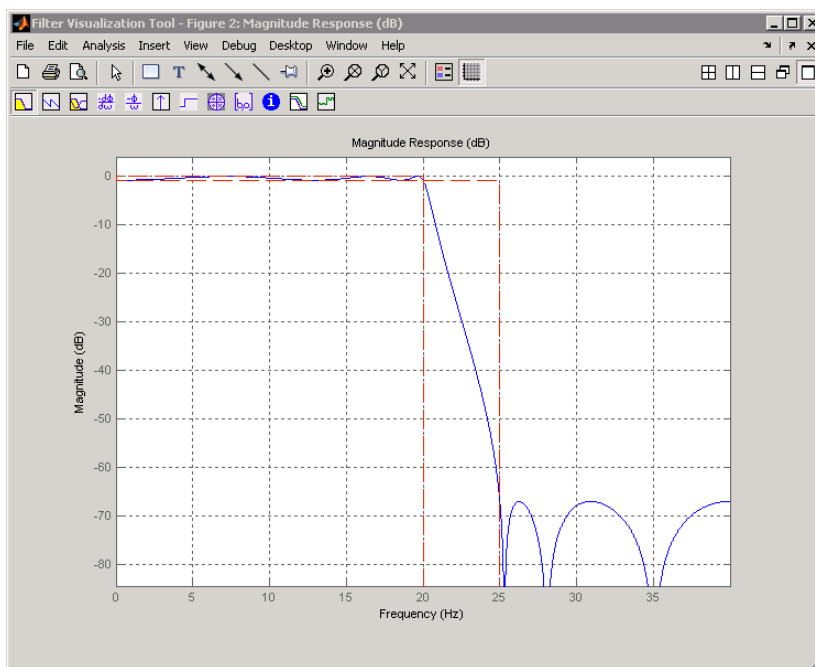
Construct the default bandpass filter specification object and design an elliptic filter.

```
d = fdesign.bandpass;
designopts(d,'ellip')
hd = design(d,'ellip','matchexactly','both');
```

## Example 2

Construct a lowpass object with order, passband-edge frequency, stopband-edge frequency, and passband ripple specifications, and then design an elliptic filter.

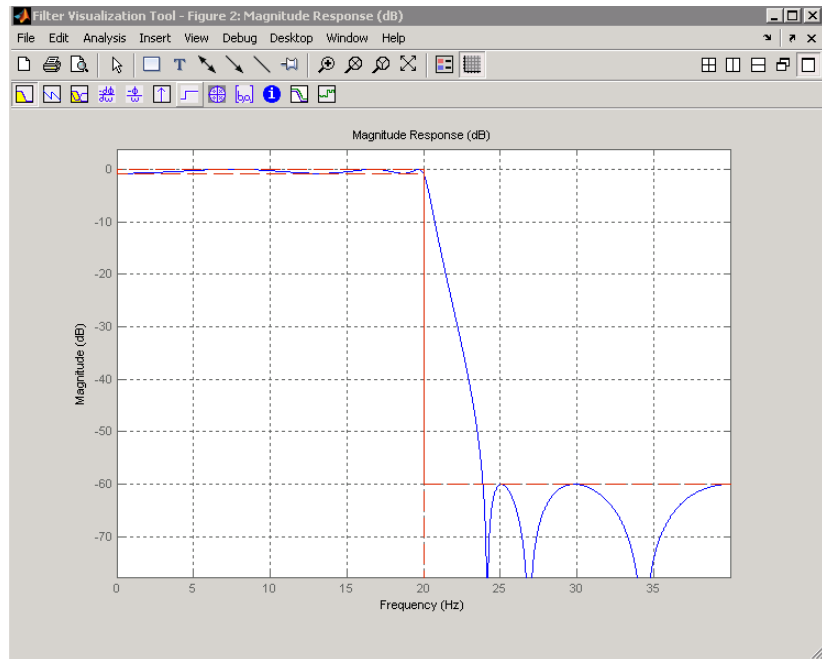
```
d = fdesign.lowpass('n,fp,fst,ap',6,20,25,.8,80);
design(d,'ellip'); % Starts FVtool to display the filter.
```



## 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

# euclidfactors

---

**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

# equiripple

---

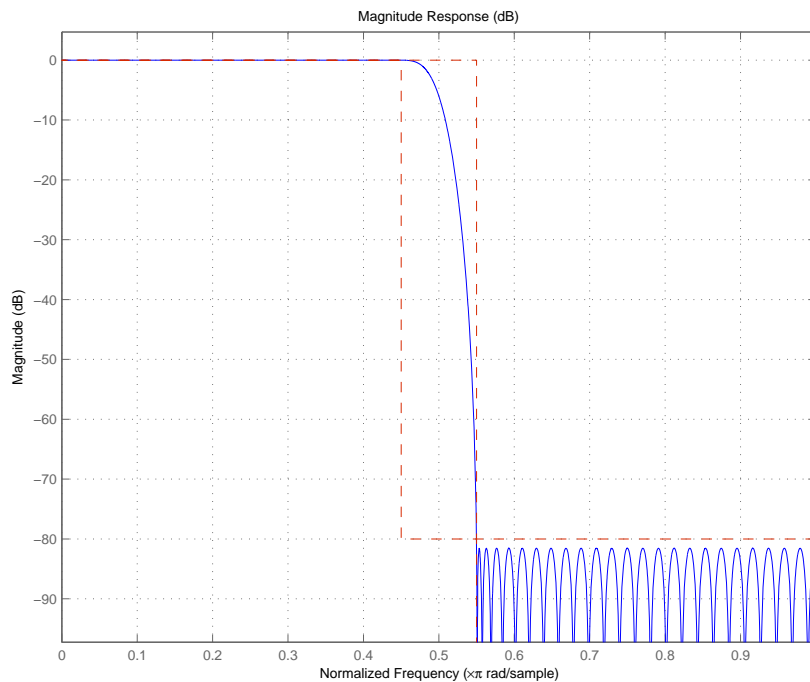
<b>Purpose</b>	Equiripple single-rate or multirate FIR filter from specification object
<b>Syntax</b>	<pre>hd = design(d,'equiripple') hd = design(d,'equiripple',designoption,value,designoption, ...value,...)</pre>
<b>Description</b>	<p><code>hd = design(d,'equiripple')</code> designs an equiripple FIR digital filter or multirate filter using the specifications supplied in the object <code>d</code>. Equiripple filter designs minimize the maximum ripple in the passbands and stopbands.</p> <p><code>hd</code> is either a <code>dfilt</code> object (a single-rate digital filter) or an <code>mfilt</code> object (a multirate digital filter) depending on the <code>Specification</code> property of the filter specification object <code>d</code> and the specifications object type — halfband or interpolator.</p> <p>When you use <code>equiripple</code> 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 <code>fdesign.nyquist</code> in the online Help system.</p> <p><code>hd = design(d,'equiripple',designoption,value,designoption,...value,...)</code> returns an equiripple FIR filter where you specify design options as input arguments.</p> <p>To determine the available design options, use <code>designopts</code> with the specification object and the design method as input arguments as shown.</p> <pre>designopts(d,'method')</pre> <p>For complete help about using <code>equiripple</code>, refer to the command line help system. For example, to get specific information about using <code>equiripple</code> with <code>d</code>, the specification object, enter the following at the MATLAB prompt.</p> <pre>help(d,'equiripple')</pre>

## 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)
help(d,'equiripple')
designopts(d,'equiripple')
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.

# equiripple

---

```
d = fdesign.interpolator(2); % Interpolation factor = 2.  
hd = design(d,'equiripple');
```

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');  
opts=designopts(d,'equiripple')  
opts.FilterStructure='dffirt'  
opts.MinPhase=1;  
opts.DensityFactor=20;  
opts  
hd=design(d,'equiripple',opts)
```

---

**Note** The MaxPhase design option for equiripple FIR filters is currently only available for lowpass, highpass, bandpass, and bandstop filters.

---

## See Also

`fdesign.nyquist`, `firls`, `kaiserwin`

**Purpose** Farrow filter

**Syntax** `hd = farrow.structure(delay,...)`

---

**Note** The `farrow` function has been deprecated, use `dfilt.farrowd` and `dfilt.farrowlinearfd` 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
<code>fd</code>	Generic fractional delay Farrow filter
<code>linearfd</code>	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
<code>cost</code>	Estimate the hardware implementation cost in terms of mathematical operations like add and multiply
<code>filter</code>	Execute the filter by using it to filter data
<code>fvtool</code>	Display and analyze the filter
<code>freqrespest</code>	Use filtering to estimate filter frequency response
<code>freqz</code>	Compute the instantaneous frequency response of the filter
<code>realizemdl</code>	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');
```

---

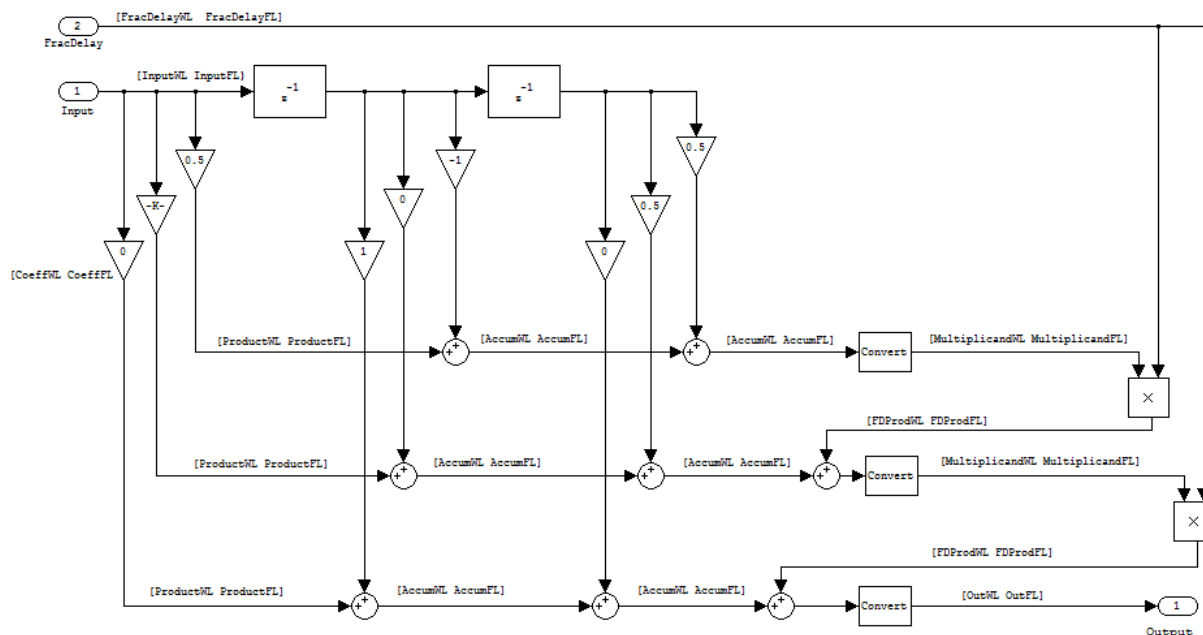
**Note** `a(1)`, the leading denominator coefficient, cannot be 0. To be able to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

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.

<b>Signal Flow Label</b>	<b>Corresponding Filter Property</b>
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.

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 <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>false</code> enables you to change the <code>CoeffWordLength</code> and <code>CoeffFracLength</code> properties to specify the data format used.
CoeffFracLength	Specifies the fraction length to apply to filter coefficients.
Coefficients	Contains the coefficients for the filter.
CoeffWordLength	Specifies the word length to apply to filter coefficients.

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.
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 <code>false</code> enables you to change the <code>FDWordLength</code> and <code>FDfracLength</code> 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.

<b>Property Name</b>	<b>Brief Description</b>
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, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> 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 <code>FilterInternals</code> to <code>SpecifyPrecision</code> .

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 <code>FilterInternals</code> to <code>SpecifyPrecision</code> .
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .
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 <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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. <code>False</code> is the default setting.

Property Name	Brief Description
ProductFracLength	<p>Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code>.</p>
ProductWordLength	<p>Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code>.</p>
RoundMode	<p>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).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic.</p>

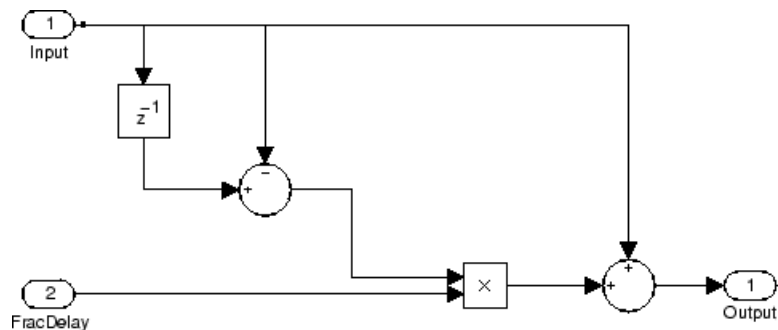
Property Name	Brief Description
	Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> 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 in a folder you choose, or your current MATLAB working folder. `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 folder 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 folder 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

```

# fcfwrite

---

```
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

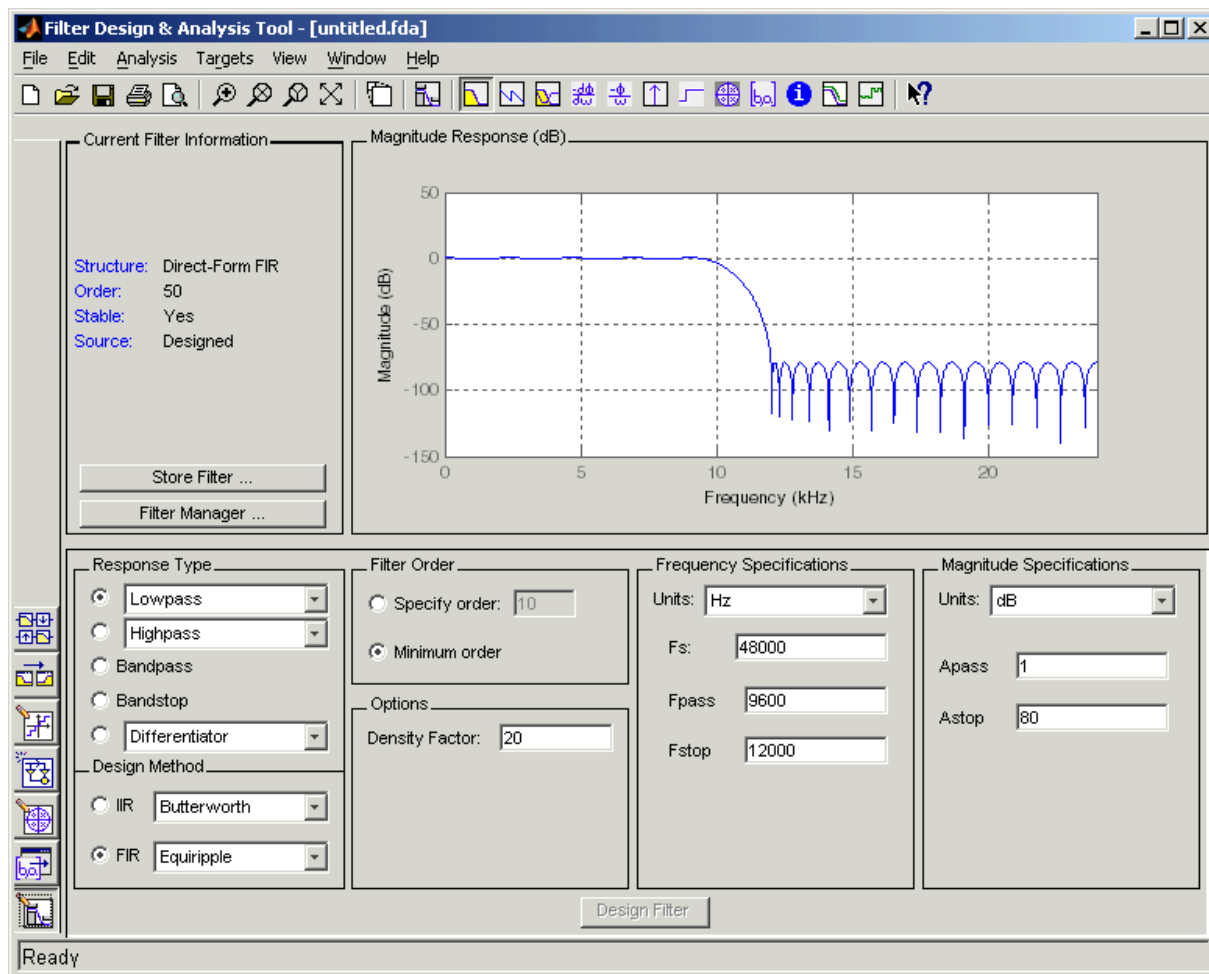
**Syntax** `fdatool`

**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)
- 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.

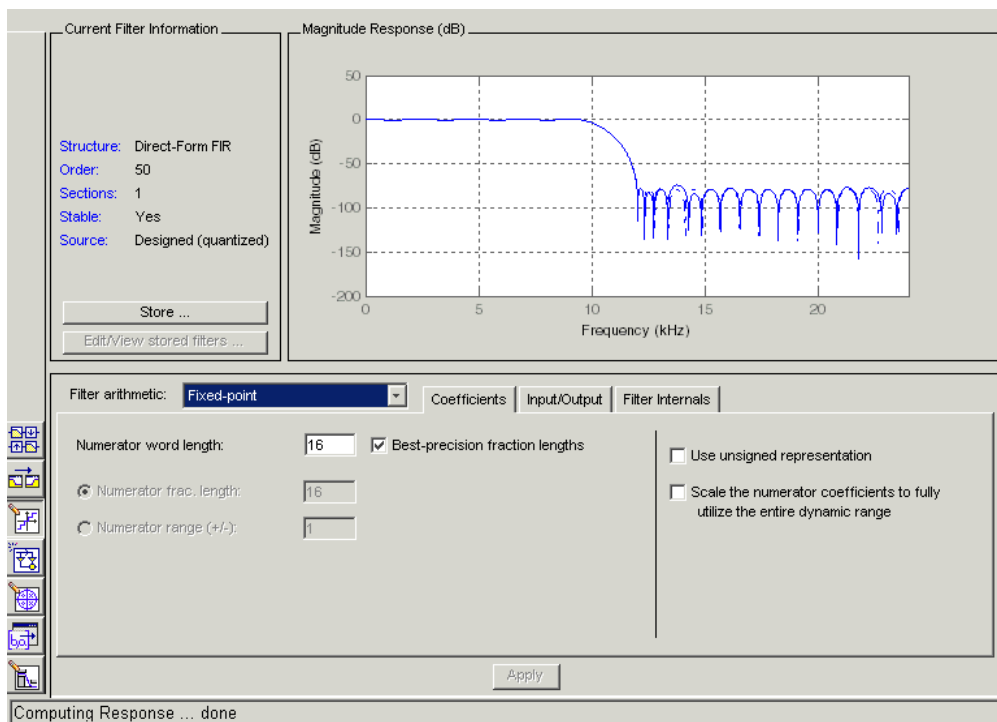
**Set Quantization Parameters** — provides access to the properties of the quantizers that compose a quantized filter. When you click **Set Quantization Parameters**, you see FDATool displaying the

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.



# fdatool

---

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 2-510

---

**Note** Several of the filter response types described below are only available if your installation includes the Filter Design Toolbox. The Filter Design Toolbox significantly expands the functionality available for the specification, design, and analysis of filters.

---

Here is how you design filters using `fdesign`.

- 1** Use `fdesign.response` to construct a filter specification object.
- 2** Use `designmethods` to determine which filter design methods work for your new filter specification object.
- 3** Use `design` to apply your filter design method from step 2 to your filter specification object to construct a filter object.
- 4** Use `FVTool` to inspect and analyze your filter object.

---

**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.

<b>fdesign Response String</b>	<b>Description</b>
<code>arbmag</code>	<code>fdesign.arbmag</code> creates an object to specify IIR filters that have arbitrary magnitude responses defined by the input arguments.
<code>arbmagnphase</code>	<code>fdesign.arbmagnphase</code> creates an object to specify IIR filters that have arbitrary magnitude and phase responses defined by the input arguments. Requires the Filter Design Toolbox.
<code>audioweighting</code>	<code>fdesign.audioweighting</code> creates a filter specification object for audio weighting filters. The supported audio weighting types are: A, C, C-message, ITU-T 0.41, and ITU-R 468-4 weighting. Requires the Filter Design Toolbox.
<code>bandpass</code>	<code>fdesign.bandpass</code> creates an object to specify bandpass filters.

<b>fdesign Response String</b>	<b>Description</b>
bandstop	<code>fdesign.bandstop</code> creates an object to specify bandstop filters.
ciccomp	<code>fdesign.ciccomp</code> creates an object to specify filters that compensate for the CIC decimator or interpolator response curves. Requires the Filter Design Toolbox.
comb	<code>fdesign.comb</code> creates an object to specify a notching or peaking comb filter. Requires the Filter Design Toolbox.
decimator	<code>fdesign.decimator</code> creates an object to specify decimators. Requires the Filter Design Toolbox
differentiator	<code>fdesign.differentiator</code> creates an object to specify differentiators.
fracdelay	<code>fdesign.fracdelay</code> creates an object to specify fractional delay filters. Requires the Filter Design Toolbox.
halfband	<code>fdesign.halfband</code> creates an object to specify halfband filters. Requires the Filter Design Toolbox.
highpass	<code>fdesign.highpass</code> creates an object to specify highpass filters.
hilbert	<code>fdesign.hilbert</code> creates an object to specify Hilbert filters.
interpolator	<code>fdesign.interpolator</code> creates an object to specify interpolators. Requires the Filter Design Toolbox.

<b>fdesign Response String</b>	<b>Description</b>
isinclp	<code>fdesign.isinclp</code> creates an object to specify lowpass filters that use inverse-sinc form. Requires the Filter Design Toolbox.
lowpass	<code>fdesign.lowpass</code> creates an object to specify lowpass filters.
notch	<code>fdesign.notch</code> creates an object to specify notch filters. Requires the Filter Design Toolbox.
nyquist	<code>fdesign.nyquist</code> creates an object to specify nyquist filters. Requires the Filter Design Toolbox.
octave	<code>fdesign.octave</code> creates an object to specify octave and fractional octave filters. Requires the Filter Design Toolbox.
parameq	<code>fdesign.parameq</code> creates an object to specify parametric equalizer filters. Requires the Filter Design Toolbox.
peak	<code>fdesign.peak</code> creates an object to specify peak filters. Requires the Filter Design Toolbox.
polysrc	<code>fdesign.polysrc</code> creates an object to specify polynomial sample-rate converter filters. Requires the Filter Design Toolbox.
pulseshaping	<code>fdesign.pulseshaping</code> creates an object to specify pulse-shaping filters.
rsrc	<code>fdesign.rsrc</code> creates an object to specify rational-factor sample-rate converters. Requires the Filter Design Toolbox.

Use the doc `fdesign.response` syntax at the MATLAB prompt to get help on a specific structure. Using `doc` in a syntax like

```
doc fdesign.lowpass
doc fdesign.bandstop
```

gets more information about the lowpass or bandstop structure objects.

Each response has a property `Specification` that defines the specifications to use to design your filter. You can use defaults or specify the `Specification` property when you construct the specifications object.

With the strings for the `Specification` property, you provide filter constraints such as the filter order or the passband attenuation to use when you construct your filter from the specification object.

## Properties

`fdesign` returns a filter specification object. Every filter specification object has the following properties.

Property Name	Default Value	Description
Response	Depends on the chosen type	Defines the type of filter to design, such as an interpolator or bandpass filter. This is a read-only value.
Specification	Depends on the chosen type	Defines the filter characteristics used to define the desired filter performance, such as the cutoff frequency $F_c$ 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 $F_s/2$ , the sampling frequency. Accepts either true or false without single quotation marks. Audio weighting filters do not support normalized frequency.

In addition to these properties, filter specification objects may have other properties as well, depending on whether they design `dfilt` objects or `mfilt` objects.

Added Properties for <code>mfilt</code> Objects	Description
DecimationFactor	Specifies the amount to decrease the sampling rate. Always a positive integer.
InterpolationFactor	Specifies the amount to increase the sampling rate. Always a positive integer.
PolyphaseLength	Polyphase length is the length of each polyphase subfilter that composes the decimator or interpolator or rate-change

Added Properties for mfilt Objects	Description
	factor filters. Total filter length is the product of p1 and the rate change factors. p1 must be an even integer.

`d = fdesign.response(spec)`. In `spec`, you specify the variables to use that define your filter design, such as the passband frequency or the stopband attenuation. The specifications are applied to the filter design method you choose to design your filter.

For example, when you create a default lowpass filter specification object, `fdesign.lowpass` sets the passband frequency `Fp`, the stopband frequency `Fst`, the stopband attenuation `Ast`, and the passband ripple `Ap` :

```
H = fdesign.lowpass
% Without a terminating semicolon
% the filter specifications are displayed
```

The default specification '`Fp,Fst,Ap,Ast`' is only one of the possible specifications for `fdesign.lowpass`. To see all available specifications:

```
H = fdesign.lowpass;
set(H, 'specification')
```

The Filter Design Toolbox software supports all available specification strings. The Signal Processing Toolbox supports a subset of the specification strings. See the reference pages for the filter specification object to determine which specification strings your installation supports.

One important note is that the specification string you choose determines which design method apply to the filter specifications object.

For the default lowpass filter specification object `H`, you can use `butter`, `cheby1`, `cheby2`, `ellip`, `equiripple`, and `kaiserwin`. However, if the specification string is '`N,Fp,Fst`', the valid design methods are `equiripple`, `firls`, and `iirlpnorm`.

Specifications that do not contain the filter order result in minimum order designs when you invoke the `design` method:

```
d = fdesign.lowpass;  
% Specification is Fp,Fst,Ap,Ast  
Hd = design(d,'equiripple');  
length(Hd.Numerator) % returns 43  
% Filter order is 42  
fvtool(Hd) %view magnitude
```

`d = fdesign.response(...,Fs)` specifies the sampling frequency in Hz to use in the filter specifications. The sampling frequency is a scalar trailing all other input arguments. If you specify a sampling frequency, all frequency specifications are in Hz.

`d = fdesign.response(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of the following strings:

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in decibels
- 'squared' — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Using Filter Design Methods with Specification Objects

After you create a filter specification object, you use a filter design method to implement your filter with a selected algorithm. 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.



<b>Design Function</b>	<b>Description</b>
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
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.

<b>Supporting Function</b>	<b>Description</b>
setspecs	Set all of the specifications simultaneously.
designmethods	Return the design methods.
designopts	Return the input arguments and default values that apply to a specifications object and method

You can set filter specification values by passing them after the `Specification` argument, or by passing the values without the `Specification` string.

Filter object constructors take the input arguments in the same order as `setspecs` and the order in the strings for `Specification`. Enter `doc setspecs` at the prompt for more information about using `setspecs`.

When the first input to `fdesign` is not a valid `Specification` string like `'n,fc'`, `fdesign` assumes that the input argument is a filter specification and applies it using the default `Specification` string —`fp,fst,ap,ast` for a lowpass object, for example.

## Examples

The following examples require only the Signal Processing Toolbox.

### Example 1—Bandstop Filter

A bandstop filter specification object for data sampled at 8 kHz. The stopband between 2 and 2.4 kHz is attenuated at least 80 dB:

```
H = fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',...  
                  1600,2000,2400,2800,1,80,1,8000);
```

### Example 2—Lowpass Filter

A lowpass filter specification object for data sampled at 10 kHz. The passband frequency is 500 Hz and the stopband frequency is 750 Hz. The passband ripple is set to 1 dB and the required attenuation in the stopband is 80 dB.

```
H = fdesign.lowpass('Fp,Fst,Ap,Ast',500,750,1,80,10000);
```

### Example 3—Highpass Filter

A default highpass filter specification object.

```
H =fdesign.highpass % Creates specifications object.
```

```
H =
```

```
           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
```

H.Description

```
ans =  
  
    'Stopband Frequency'  
    'Passband Frequency'  
    'Stopband Attenuation (dB)'  
    'Passband Ripple (dB)'
```

Notice the correspondence between the property values in Specification and Description — in Description you see in words the definitions of the variables shown in Specification.

### Example 4—Filter Specification and Design

Lowpass Butterworth filter specification object

Use a filter specification object to construct a lowpass Butterworth filter with default Specification `fp`, `fst`, `ap`, `ast` — 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);
```

Determine which design methods apply to `d`. With only the Signal Processing Toolbox installed, you can choose among the following algorithms:

```
>>designmethods(d)
```

Design Methods for class `fdesign.lowpass`:

```
butter  
cheby1  
cheby2  
ellip
```

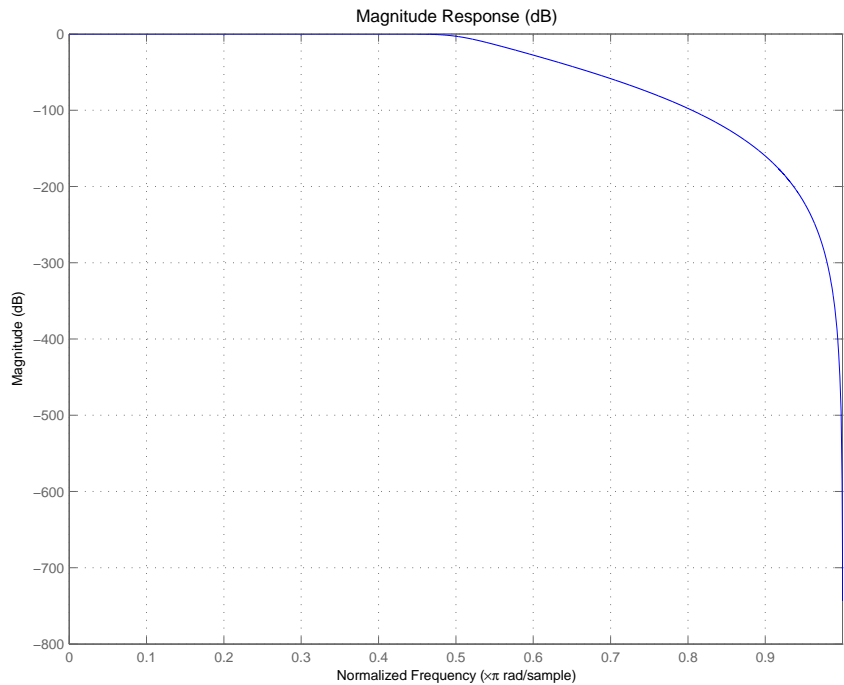
With the Filter Design Toolbox installed, you have additional algorithms available.

```
>>designmethods(d)  
  
Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):  
  
butter  
cheby1  
cheby2  
ellip  
equiripple  
ifir  
kaiserwin  
multistage
```

You can use `d` and the `butter` design method to design a Butterworth filter.

```
hd = design(d, 'butter', 'matchexactly', 'passband');  
fvtool(hd);
```

The resulting filter magnitude response shown by FVTool appears in the following figure.



If you 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, fir1s, fvtool, kaiserwin, lagrange, multistage, setspecs, validstructures

## Purpose

Audio weighting filter specification object

## Syntax

```
HAWf = fdesign.audioweighting  
HAWf = fdesign.audioweighting(spec)  
HAWf = fdesign.audioweighting(spec,specvalue1,specvalue2)  
HAWf = fdesign.audioweighting(specvalue1,specvalue2)  
HAWf = fdesign.audioweighting(...,Fs)
```

## Description

Supported audio weighting filter types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468–4 weighting.

*HAWf* = fdesign.audioweighting constructs an audio weighting filter specification object *HAWf* with a weighting type of A and a filter class of 1. Use the `design` method to instantiate a `dfilt` object based on the specifications in *HAWf*. Use `designmethods` to find valid filter design methods. Because the standards for audio weighting filters are in Hz, normalized frequency specifications are not supported for `fdesign.audioweighting` objects. The default sampling frequency for A weighting, C weighting, C-message, and ITU-T 0.41 filters is 48 kHz. The default sampling frequency for the ITU-R 468–4 filter is 80 kHz. If you invoke the `normalizefreq` method, a warning is issued when you instantiate the `dfilt` object and the default sampling frequencies in Hz are used.

*HAWf* = fdesign.audioweighting(*spec*) returns an audio weighting filter specification object using default values for the specification string in *spec*. The following are valid entries for *spec*. The entries are not case sensitive.

- 'WT,Class' (default *spec*)

The 'WT,Class' specification is valid for A weighting and C weighting filters of class 1 or 2.

The weighting type is specified by the string: 'A' or 'C'. The class is the scalar 1 or 2.

The default values for 'WT,Class' are 'A',1.

- 'WT'

The 'WT' specification is valid for C-message (default), ITU-T 0.41, and ITU-R 468–4 weighting filters.

The weighting type is specified by the string: 'Cmessage', 'ITUT041', or 'ITUR4684'.

*HAWf* = fdesign.audioweighting(*spec*,*specvalue1*,*specvalue2*) constructs an audio weighting filter specification object *HAWf* and sets its specifications at construction time.

*HAWf* = fdesign.audioweighting(*specvalue1*,*specvalue2*) constructs an audio weighting filter specification object *HAWf* with the specification 'WT,Class' using the values you provide. The valid weighting types are 'A' or 'C'.

*HAWf* = fdesign.audioweighting(...,*Fs*) specifies the sampling frequency in Hz. The sampling frequency is a scalar trailing all other input arguments.

## Input Arguments

### Parameter Name/Value Pairs

WT

Weighting type

The weighting type defines the frequency response of the filter. The valid weighting types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468–4 weighting. The weighting types are described in “Definitions” on page 2-519.

Class

Filter Class

Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards [1], [2]. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in *fvtool* for the analysis of the filter design.



Default: 1

## Definitions

### A weighting

The specifications for the A weighting filter are found in ANSI standard S1.42-2001. The A weighting filter is based on the 40-phon Fletcher-Munson equal loudness contour [3]. One phon is equal to one dB sound pressure level (SPL) at one kHz. The Fletcher-Munson equal loudness contours are designed to account for frequency and level dependent differences in the perceived loudness of tonal stimuli. The 40-phon contour reflects the fact that the human auditory system is more sensitive to frequencies around 1–2 kHz than lower and higher frequencies. The filter roll off is more pronounced at lower frequencies and more modest at higher frequencies. While A weighting is based on the equal loudness contour for low-level (40-phon) tonal stimuli, it is commonly used in the United States for assessing potential health risks associated with noise exposure to narrowband and broadband stimuli at high levels.

### C weighting

The specifications for the C weighting filter are found in ANSI standard S1.42-2001. The C weighting filter approximates the 100-phon Fletcher-Munson equal loudness contour for tonal stimuli. The C weighting magnitude response is essentially flat with 3-dB frequencies at 31.5 Hz and 8000 Hz. While C weighting is not as common as A weighting, sound level meters frequently have a C weighting filter option.

### C-message

The specifications for the C-message weighting filter are found in Bell System Technical Reference, PUB 41009. C-message weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [6]. C-message weighting filters are commonly used in North America, while the ITU-T 0.41 filter is more commonly used outside of North America.

## ITU-R 468-4

The specifications for the ITU-R 468-4 weighting filter are found in the International Telecommunication Union Recommendation ITU-R BS.468-4. ITU-R 468-4 is an equal loudness contour weighting filter. Unlike the A weighting filter, the ITU-R 468-4 filter describes subjective loudness judgements for broadband stimuli [4]. A common criticism of the A weighting filter is that it underestimates the loudness judgement of real-world stimuli particularly in the frequency band from about 1–9 kHz. A comparison of A weighting and ITU-R 468-4 weighting curves shows that the ITU-R 468-4 curve applies more gain between 1 and 10 kHz with a peak difference of approximately 12 dB around 6–7 kHz.

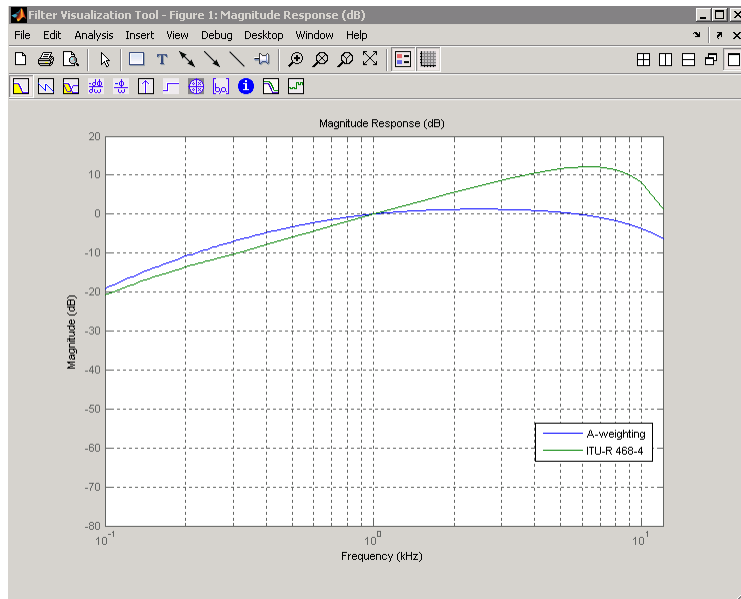
## ITU-T 0.41

The specifications for the ITU-T 0.41 filter are found in the ITU-T Recommendation 0.41. ITU-T 0.41 weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [5]. ITU-T 0.41 weighting filters are commonly used outside of North America, while the C-message weighting filter is more common in North America.

## Examples

Compare class 1 A weighting and ITU-R 468-4 filters between 0.1 and 12 kHz:

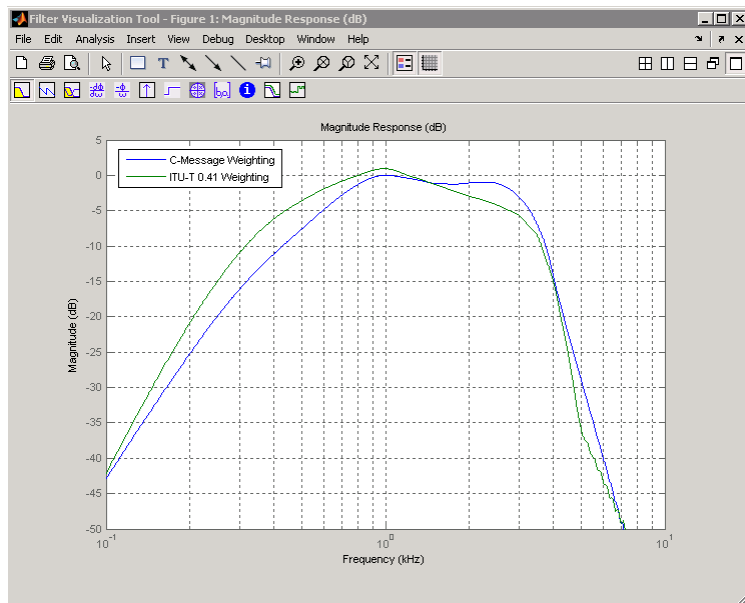
```
HawfA = fdesign.audioweighting('WT,Class','A',1,44.1e3);  
% Sampling frequency is 44.1 kHz  
HawfITUR = fdesign.audioweighting('WT','ITUR4684',44.1e3);  
Afilter = design(HawfA);  
ITURfilter = design(HawfITUR);  
hfvt = fvtool([Afilter ITURfilter]);  
axis([0.1 12 -80 20]);  
legend(hfvt,'A-weighting','ITU-R 468-4');
```



Compare C-message and ITU-T 0.41 filters:

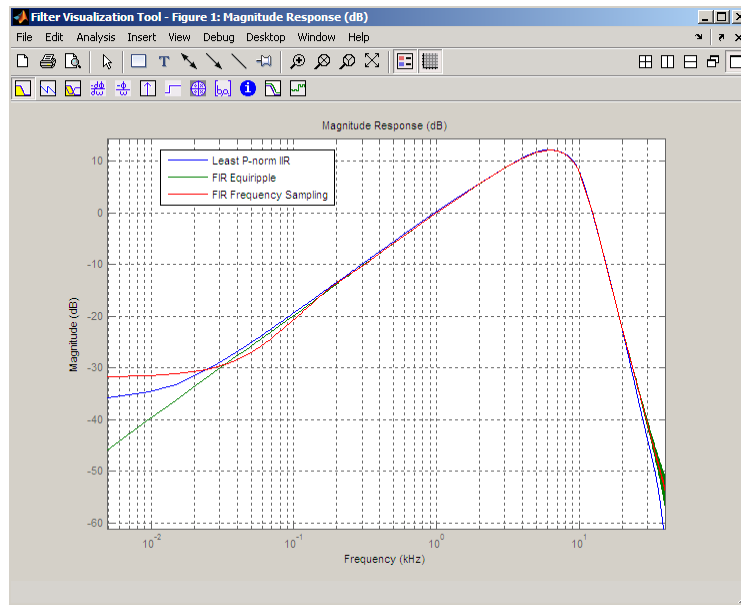
```
hCmessage = fdesign.audioweighting('WT','Cmessage',24e3);
hITUT = fdesign.audioweighting('WT','ITUT041',24e3);
dCmessage = design(hCmessage);
dITUT = design(hITUT);
hfvtool = fvtool([dCmessage dITUT]);
legend(hfvtool,'C-Message Weighting','ITU-T 0.41 Weighting');
axis([0.1 10 -50 5]);
```

# fdesign.audioweighting



Construct an ITU-R 468–4 filter using all available design methods:

```
HAwf = fdesign.audioweighting('WT','ITUR4684');  
designmethods(HAwf)  
% returns iirlpnorm,equiripple,freqsamp  
D = design(HAwf,'all'); % returns all designs  
hfvtool(D);  
legend(hfvtool,'Least P-norm IIR','FIR Equiripple',...  
'FIR Frequency Sampling')
```



## References

- [1] *American National Standard Design Response of Weighting Networks for Acoustical Measurements*, ANSI S1.42-2001, Acoustical Society of America, New York, NY, 2001.
- [2] *Electroacoustics Sound Level Meters Part 1: Specifications*, IEC 61672-1, First Edition 2002-05.
- [3] Fletcher, H. and W.A. Munson. "Loudness, its definition, measurement and calculation." *Journal of the Acoustical Society of America*, Vol. 5, 1933, pp. 82–108.
- [4] *Measurement of Audio-Frequency Noise Voltage Level in Sound Broadcasting*, International Telecommunication Union Recommendation ITU-R BS.468-4, 1986.
- [5] *Psophometer for Use on Telephone-Type Circuits*, ITU-T Recommendation 0.41.

# fdesign.audioweighting

---

[6] *Transmission Parameters Affecting Voiceband Data*  
*Transmission-Measuring Techniques*, Bell System Technical Reference,  
PUB 41009, 1972.

## See Also

`design` | `designmethods` | `fdesign` | `fvtool`

## How To

- Audio Weighting Filters Demo
- “Designing a Filter in Fdesign — Process Overview”

## 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
<code>n, f, a</code>	Single band design (default). FIR and IIR ( <code>n</code> is the order for both numerator and denominator)
<code>n, b, f, a</code>	Multiband design where <code>b</code> defines the number of bands
<code>nb, na, f, a</code>	IIR single band design
<code>nb, na, b, f, a</code>	IIR multiband design where <code>b</code> 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]

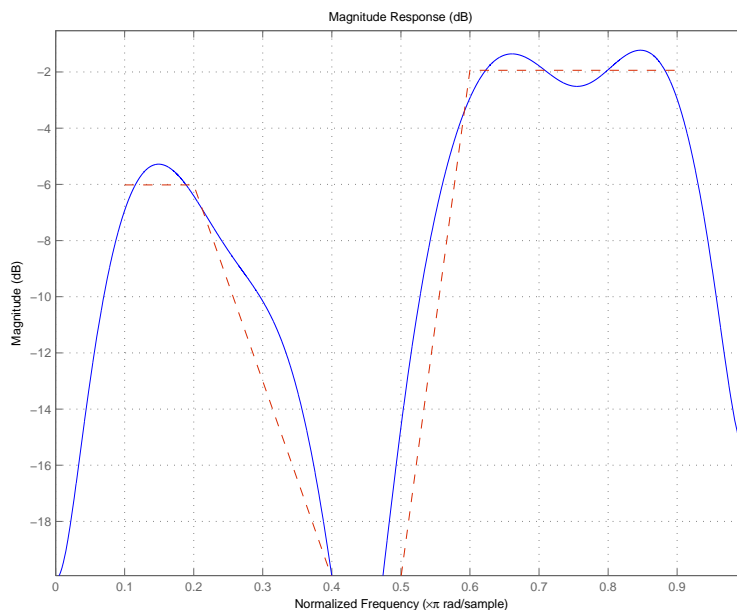
Define the response vector a as [0 0.5 0.5 0.1 0.1 0.8 0.8 0]

These specifications connect f and a as shown in the following table.



<b>f (Normalized Frequency)</b>	<b>a (Response Desired at f)</b>
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 .



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`, ... `specn`.

`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]
```

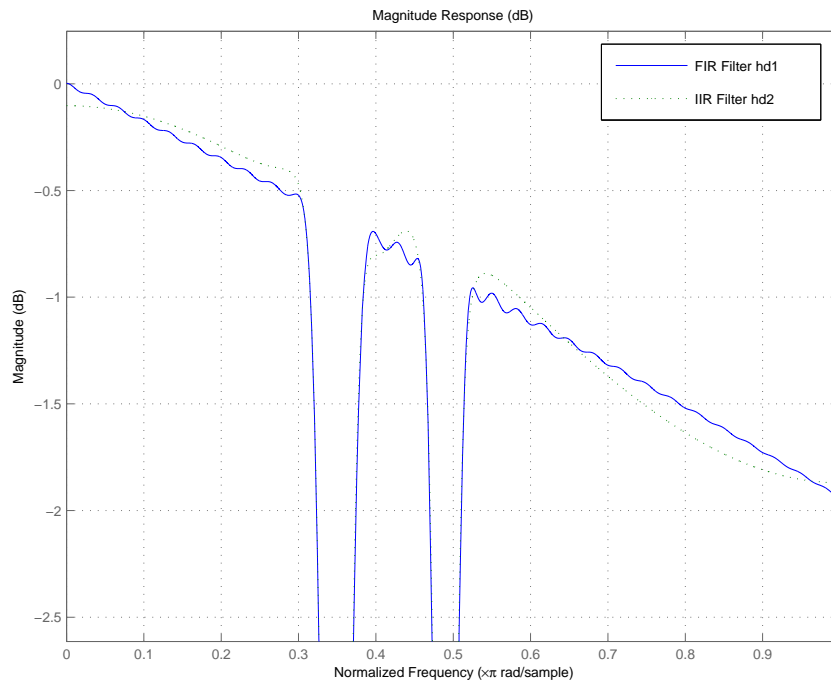
```
hd2=design(d,'iirlpnorm') % Design an IIR filter from the same object.
```

```
hd2 =
```

```
FilterStructure: 'Direct-Form II, Second-Order Sections'  
Arithmetic: 'double'  
sosMatrix: [5x6 double]  
ScaleValues: [0.85714867585342;1;1;1;1;1]  
PersistentMemory: false
```

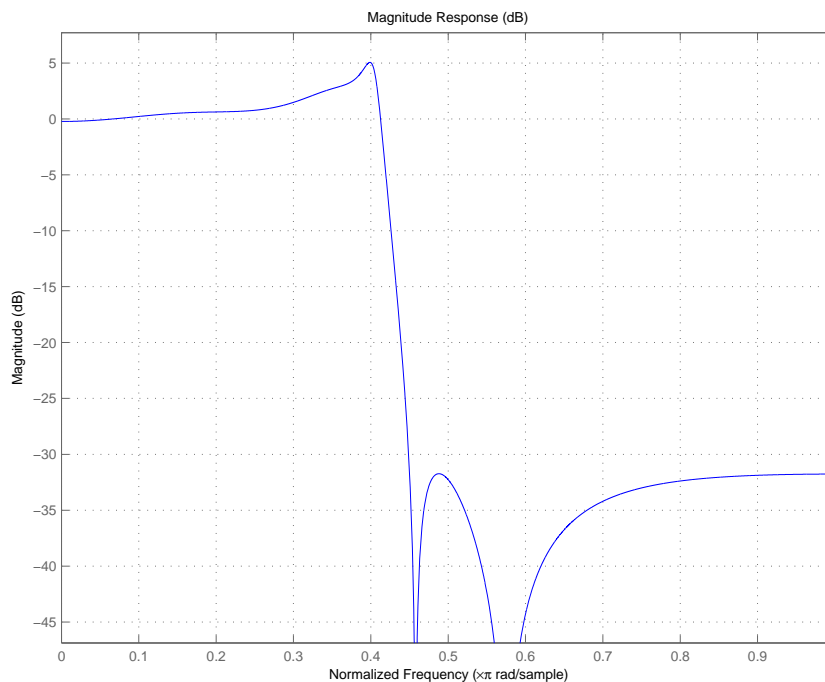
```
fvtool(hd1,hd2)
```

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, designmethods, fdesign

**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
<code>n, f, h</code>	Single band design (default). FIR and IIR ( <code>n</code> is the order for both numerator and denominator).
<code>n, b, f, h</code>	FIR multiband design where <code>b</code> defines the number of bands.
<code>nb, na, f, h</code>	IIR single band design.

The following table describes the arguments in the strings.

Argument	Description
<code>b</code>	Number of bands in the multiband filter.
<code>f</code>	Frequency vector. Frequency values specified in <code>f</code> indicate locations where you provide specific filter response amplitudes. When you provide <code>f</code> you must also provide <code>h</code> 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]

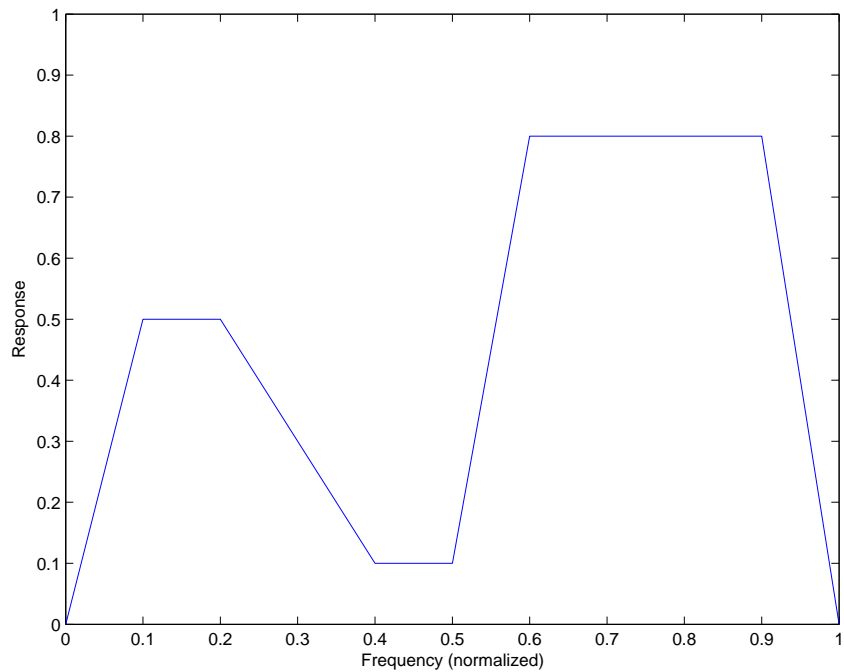
These specifications connect f and h as shown in the following table.

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



<b>f (Normalized Frequency)</b>	<b>h (Response Desired at f)</b>
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.

# fdesign.arbmagnphase

---

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, ...specn.
```

```
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

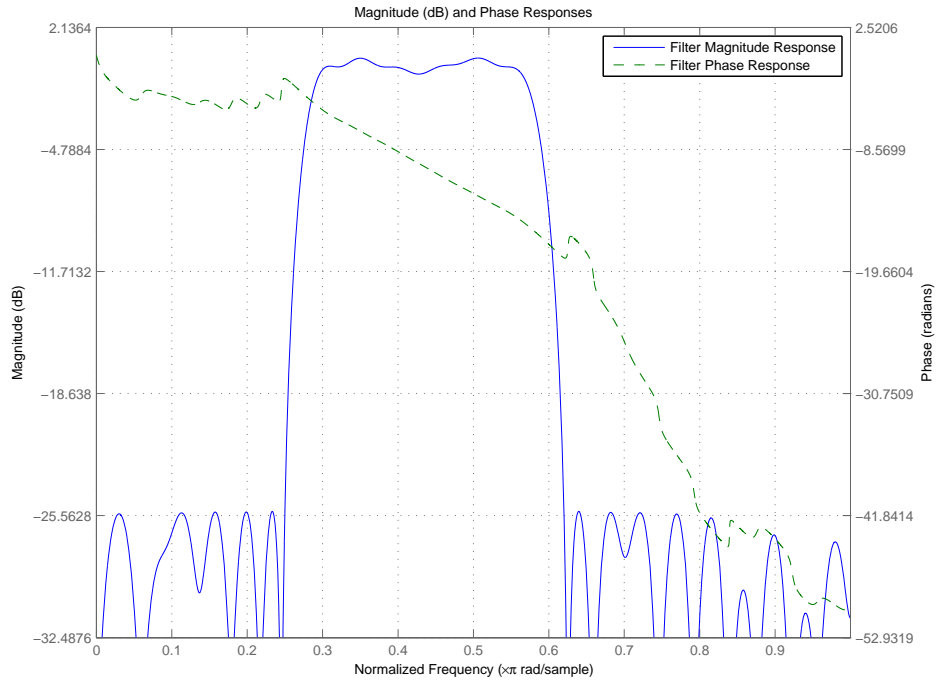
Use `fdesign.arbmagnphase` to model a complex analog filter:

```
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`

# fdesign.bandpass

---

**Purpose** Bandpass filter specification object

**Syntax**

```
d = fdesign.bandpass
d = fdesign.bandpass(spec)
d = fdesign.bandpass(spec,specvalue1,specvalue2,specvalue3,
    ...)
d = fdesign.bandpass(specvalue1,specvalue2,specvalue3,
    specvalue4,specvalue4,specvalue5,specvalue6,specvalue7)
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 valid 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,Fc1,Fc2,Ast1,Ap,Ast2'

- 'N,Fp1,Fp2,Ap'
- 'N,Fp1,Fp2,Ast1,Ap,Ast2'
- 'N,Fst1,Fp1,Fp2,Fst2'
- 'N,Fst1,Fp1,Fp2,Fst2,Ap'
- 'N,Fst1,Fst2,Ast'
- 'Nb,Na,Fst1,Fp1,Fp2,Fst2'

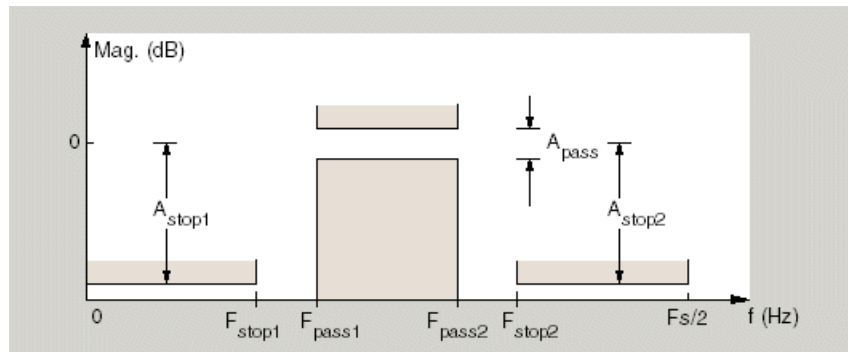
The string entries are defined as follows:

- Ap — passband ripple in dB (the default units).
- Ast1 — attenuation in the first stopband in dB (the default units).
- Ast2 — attenuation in the second stopband in dB (the default units).
- BWp — bandwidth of the filter passband. Specified in normalized frequency units by default.
- BWst — bandwidth of the filter stopband. Specified in normalized frequency units by default.
- F3dB1 — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units by default (IIR filters).
- F3dB2 — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units by default (IIR filters).
- Fc1 — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. Specified in normalized frequency units by default (FIR filters).
- Fc2 — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. Specified in normalized frequency units by default (FIR filters).
- Fp1 — frequency at the edge of the start of the pass band. Specified in normalized frequency units by default.

# fdesign.bandpass

- $F_{p2}$  — frequency at the edge of the end of the pass band. Specified in normalized frequency units by default.
- $F_{st1}$  — frequency at the edge of the end of the first stop band. Specified in normalized frequency units by default.
- $F_{st2}$  — frequency at the edge of the start of the second stop band. Specified in normalized frequency units by default.
- $N$  — filter order. filter order. Results in  $N+1$  filter coefficients, or taps for an FIR filter design and  $N+1$  numerator and denominator coefficients, or taps for an IIR design.
- $N_a$  — denominator order for IIR filters
- $N_b$  — numerator order for IIR filters

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like  $f_{st1}$  and  $f_{p1}$  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, specvalue3, ...)  
constructs an object d and sets its specifications at construction time.
```

```
d =  
fdesign.bandpass(specvalue1, specvalue2, specvalue3, specvalue4, specvalue5, ...)  
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, ...  
.
```

```
d = fdesign.bandpass(..., Fs) specifies the sampling frequency Fs in  
Hz as a scalar trailing all other numeric input arguments. If you provide  
a sampling frequency, all frequencies in the specification are in Hz.
```

```
d = fdesign.bandpass(..., MAGUNITS) specifies the units for any  
magnitude specification you provide in the input arguments. MAGUNITS  
can be one of the following strings:
```

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in 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'
```

## fdesign.bandpass

---

```
        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(10,12,14,16,80,0.5,60,48)
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
```

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 =
```



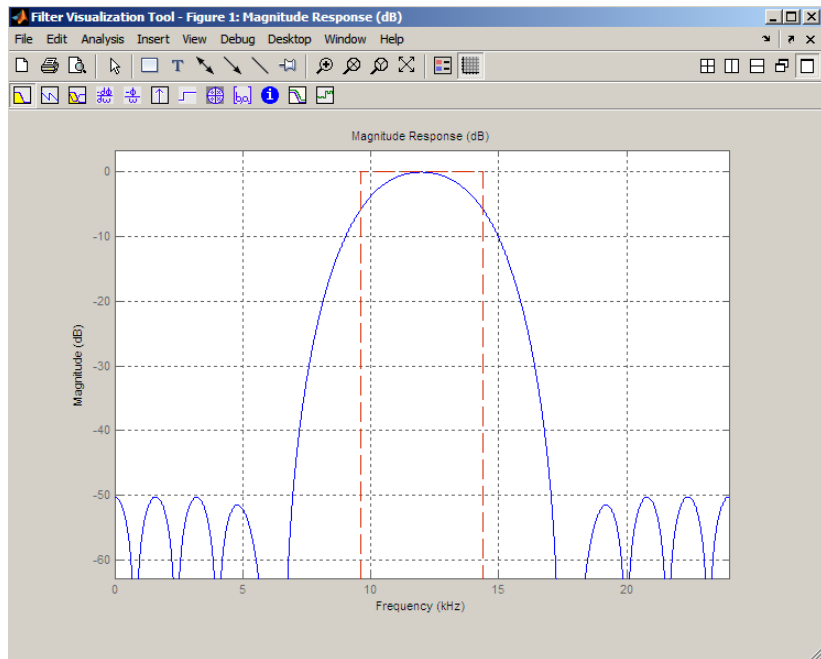
```

Response: 'Bandpass with cutoff'
Specification: 'N,Fc1,Fc2'
Description: {3x1 cell}
NormalizedFrequency: true
FilterOrder: 10
Fcutoff1: 0.4000
Fcutoff2: 0.6000
    
```

Pass the specification values to the object rather than accepting the default values for  $N$ ,  $Fc1$ , and  $Fc2$ . Include the sampling frequency  $F_s$  as the final input argument. Design the filter and plot the results.

```

d = fdesign.bandpass('n,fc1,fc2',30,9600,14400,48000);
% Fc1 and Fc2 are 6-dB down points (FIR filter)
Hd = design(d);
fvtool(Hd);
    
```



# **fdesign.bandpass**

---

The following topics include examples of `fdesign.bandpass`:

“Basic Filter Design Process”

“Floating-Point to Fixed-Point Conversion”

“Process Flow Diagram and Filter Design Methodology”

## **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,value1,value2,...)
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,Fc1,Fc2,Ap1,Ast,Ap2'
- 'N,Fp1,Fp2,Ap'

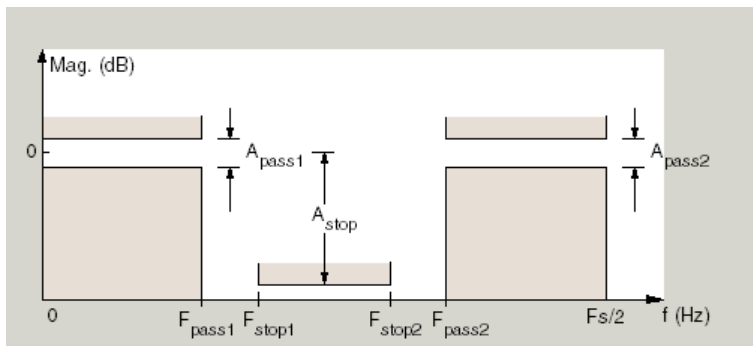
- 'N,Fp1,Fp2,Ap,Ast'
- 'N,Fp1,Fst1,Fst2,Fp2'
- 'N,Fp1,Fst1,Fst2,Fp2,Ap'
- 'N,Fst1,Fst2,Ast'
- 'Nb,Na,Fp1,Fst1,Fst2,Fp2'

The string entries are defined as follows:

- Ap — passband ripple in dB (the default units).
- Ap1 — amount of ripple in the 1st passband in dB (the default units).
- Ap2 — amount of ripple in the 2nd passband in dB (the default units).
- Ast — stopband attenuation in dB (the default units).
- BWp — bandwidth of the filter passband. Specified in normalized frequency units by default.
- BWst — bandwidth of the filter stopband. Specified in normalized frequency units by default.
- F3dB1 — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units by default.
- F3dB2 — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units by default.
- Fc1 — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. Specified in normalized frequency units by default.
- Fc2 — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. Specified in normalized frequency units by default.
- Fp1 — frequency at the start of the pass band. Specified in normalized frequency units by default.

- $F_{p2}$  — frequency at the end of the pass band. Specified in normalized frequency units by default.
- $F_{st1}$  — frequency at the end of the first stop band. Specified in normalized frequency units by default.
- $F_{st2}$  — frequency at the start of the second stop band. Specified in normalized frequency units by default.
- $N$  — filter order. FIR designs result in  $N+1$  filter coefficients, or taps. IIR designs results in  $N+1$  numerator and denominator coefficients, or taps.
- $N_a$  — denominator order for IIR filters.
- $N_b$  — numerator order for IIR filters.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like  $F_{p1}$  and  $F_{st1}$  are transition, or “don’t care”, 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.

# fdesign.bandstop

---

`d = fdesign.bandstop(spec,specvalue1,specvalue2,...)` constructs a specification 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)` specifies the sampling frequency `Fs` in Hz as a scalar trailing all other numeric input arguments. If you provide a sampling frequency, all frequencies in the specification are in Hz.

`d = fdesign.bandstop(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of the following strings:

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in 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
```

Create an object by passing the specification type string 'N,F3dB1,F3dB2' — the resulting object uses default values for N, F3dB1, and F3dB2.

```
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)
```

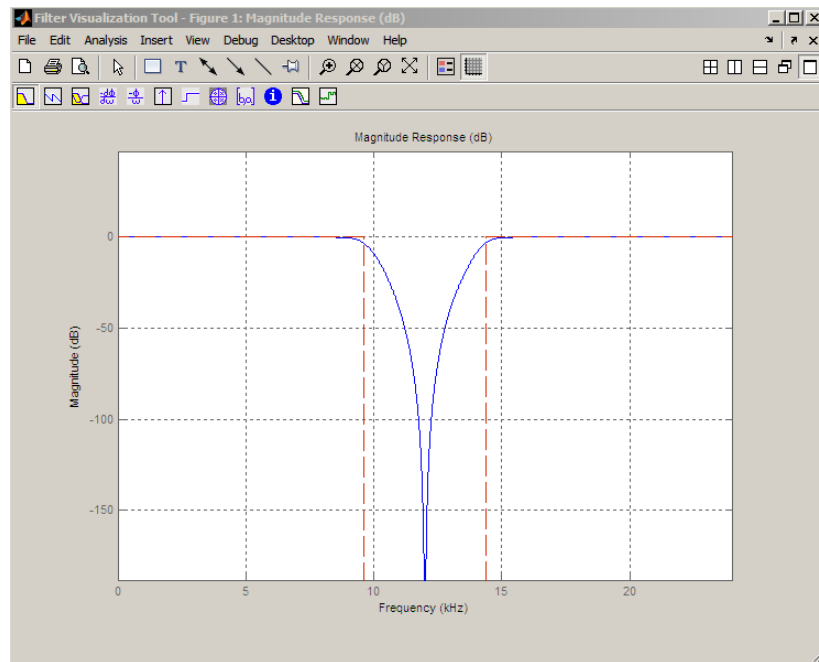
```
Design Methods for class fdesign.bandstop:
```

```
butter  
cheby1  
cheby2  
ellip
```

# fdesign.bandstop

Create another bandstop filter, passing the specification values to the object rather than accepting the default values for  $N$ ,  $F_{3dB1}$ , and  $F_{3dB2}$ . You can add  $F_s$  as the final input argument to specify the sampling frequency of 48 kHz. Adding the sampling frequency automatically converts all frequency parameters to Hz. Design the filter and plot the results.

```
d = fdesign.bandstop('n,f3db1,f3db2',10,9600,14400,48000);  
Hd = design(d);  
fvtool(Hd)
```



Pass the filter specifications that correspond to the default Specification —  $F_{p1}, F_{st1}, F_{st2}, F_{p2}, A_{p1}, A_{st}, A_{p2}$ .

```
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
```

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**

```
d= fdesign.ciccomp
d= fdesign.ciccomp(d,nsections)
d= fdesign.ciccomp(...,spec)
h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)
```

**Description** `d= 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.

`d= 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.

`d= 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.

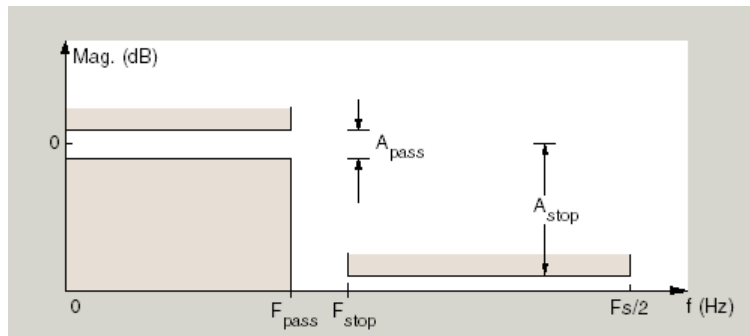
- 'fp,fst,ap,ast' (default spec)
- 'n,fc,ap,ast'
- 'n,fp,ap,ast'
- 'n,fp,fst'
- 'n,fst,ap,ast'

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 6 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.

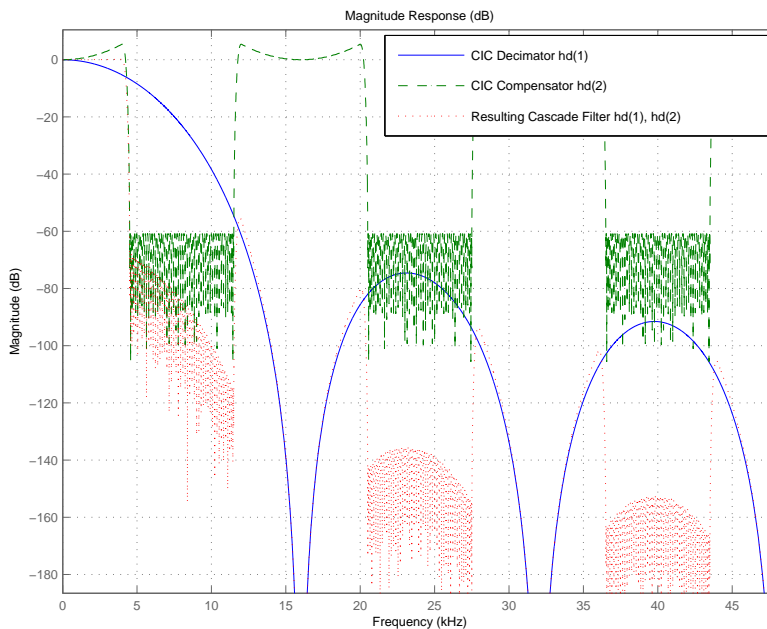
`h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)` constructs an object and sets the specifications in the order they are specified in the `spec` input when you construct the object.

## 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.

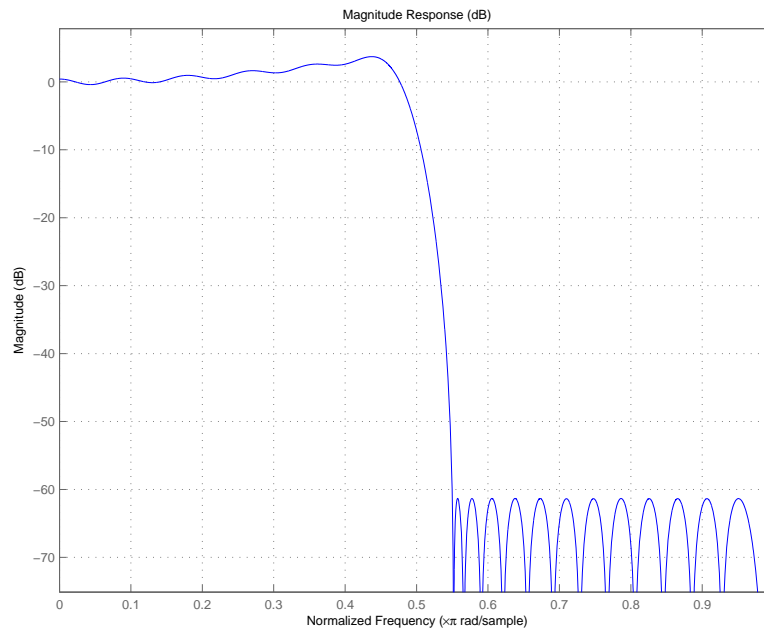
```
fs = 96e3; % Input sampling frequency.
fpass = 4e3; % Frequency band of interest.
m = 6; % Decimation factor.
hcic = design(fdesign.decimator(m,'cic',1,fpass,60,fs));
hd = cascade(dfilt.scalar(1/gain(hcic)),hcic);
hd(2) = design(fdesign.ciccomp(hcic.differentialdelay, ...
    hcic.numberofsections,fpass,4.5e3,.1,60,fs/m));
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 rolloff 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** IIR comb filter specification object

**Syntax**

```
d=fdesign.comb
d=fdesign.comb(combtype)
d=fdesign(combtype,specstring)
d=fdesign(combtype,specstring,specvalue1,specvalue2,...)
d=fdesign.comb(...,Fs)
```

**Description** `fdesign.comb` specifies a peaking or notching comb filter. Comb filters amplify or attenuate a set of harmonically related frequencies.

`d=fdesign.comb` creates a notching comb filter specification object and applies default values for the filter order (N=10) and quality factor (Q=16).

`d=fdesign.comb(combtype)` creates a comb filter specification object of the specified type and applies default values for the filter order and quality factor. The valid entries for `combtype` are shown in the following table. The entries are not case-sensitive.

Argument	Description
notch	creates a comb filter that attenuates a set of harmonically related frequencies.
peak	creates a comb filter that amplifies a set of harmonically related frequencies.

`d=fdesign(combtype,specstring)` creates a comb filter specification object of type `combtype` and sets its `Specification` property to `specstring` with default values. The entries in `specstring` determine the number of peaks or notches in the comb filter as well as their bandwidth and slope. Valid entries for `specstring` are shown below. The entries are not case-sensitive.

- 'N,Q' (default)

- 'N, BW'
- 'L, BW, GWB, Nsh'

The following table describes the arguments in *specstring*.

Argument	Description
BW	Bandwidth of the notch or peak. By default the bandwidth is calculated at the point $-3$ dB down from the center frequency of the peak or notch. For example, setting $BW=0.01$ specifies that the $-3$ dB point will be $\pm 0.005$ (in normalized frequency) from the center of the notch or peak.
GWB	Gain at which the bandwidth is measured. This allows the user to specify the bandwidth of the notch or peak at a gain different from the $-3$ dB default.
L	Upsampling factor for a shelving filter of order $Nsh$ . $L$ determines the number of peaks or notches, which are equally spaced over the normalized frequency interval $[-1,1]$ .
N	Filter order. Specifies a filter with $N+1$ numerator and denominator coefficients. The filter will have $N$ peaks or notches equally spaced over the interval $[-1,1]$ .



Argument	Description
Nsh	Shelving filter order. Nsh represents a positive integer that determines the sharpness of the peaks or notches. The greater the value of the shelving filter order, the steeper the slope of the peak or notch. This results in a filter of order $L \cdot Nsh$ .
Q	Peak or notch quality factor. Q represents the ratio of the lowest center frequency peak or notch (not including DC) to the bandwidth calculated at the $-3$ dB point.

`d=fdesign(combtype,specstring,specvalue1,specvalue2,...)` creates an IIR comb filter specification object of type `combtype` and sets its `Specification` property to the values in `specvalue1,specvalue2,...`

`d=fdesign.comb(...,Fs)` creates an IIR comb filter specification object using the sampling frequency, `Fs`, of the signal to be filtered. The function assumes that `Fs` is in Hertz and must be specified as a scalar trailing all other provided values.

## Examples

These examples demonstrate how to create IIR comb filter specification objects. First, create a default specification object.

```
d=fdesign.comb
```

```
d =
```

```

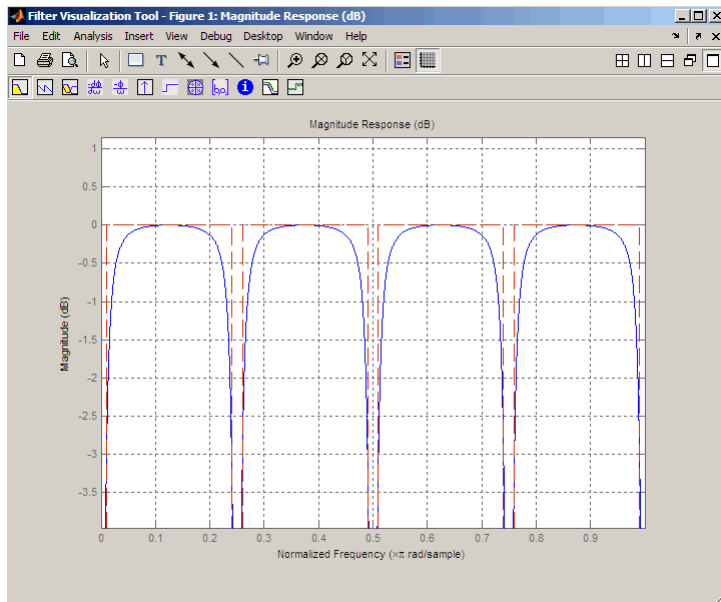
    Response: 'Comb Filter'
    CombType: 'Notch'
    Specification: 'N,Q'
    Description: {'Filter Order';'Quality Factor'}

```

```
NormalizedFrequency: true
NotchFrequencies: [0 0.2 0.4 0.6 0.8 1]
FilterOrder: 10
Q: 16
```

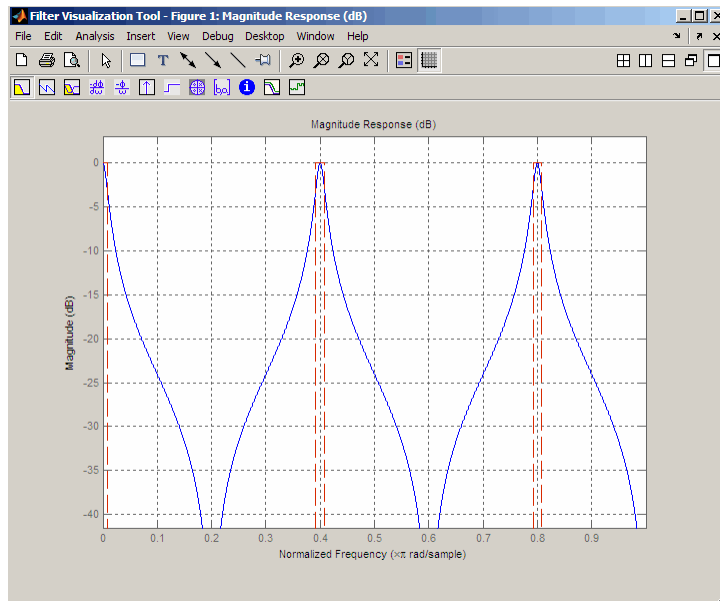
In the next example, create a notching filter of order 8 with a bandwidth of 0.02 (normalized frequency) referenced to the  $-3$  dB point.

```
d = fdesign.comb('notch','N,BW',8,0.02);
Hd = design(d);
fvtool(Hd)
```



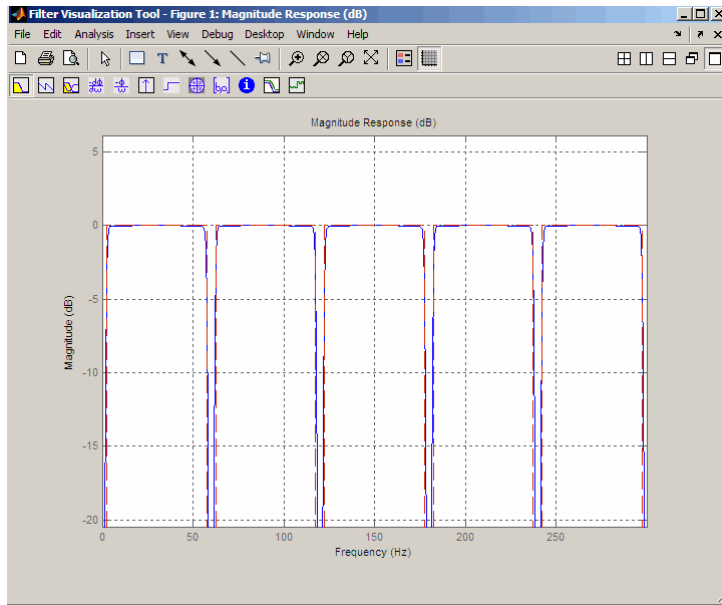
Next, create a peaking comb filter with 5 peaks and a quality factor of 25.

```
d = fdesign.comb('peak','N,Q',5,25);
Hd = design(d);
fvtool(Hd)
```



In the next example, create a notching filter to remove interference at 60 Hz and its harmonics. The following code creates a filter with 10 notches and a notch bandwidth of 5 Hz referenced to the -4dB level. The filter has a shelving filter order of 4 and a sampling frequency of 600 Hz. Because the notches are equidistantly spaced in the interval [-300, 300] Hz, they occur at multiples of 60 Hz.

```
d = fdesign.comb('notch', 'L,BW,GBW,Nsh',10,5,-4,4,600);
Hd=design(d);
fvtool(Hd)
```



**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.

## fdesign.decimator

---

Design String	Description
ciccomp	Sets the design for the decimator specifications object to CIC compensator.
Gaussian	Sets the design for the decimator specifications object to Gaussian (pulseshaping filter)
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.
Raised cosine	Sets the design for the decimator specifications object to raised cosine (pulseshaping filter).
Square root raised cosine	Sets the design for the decimator specifications object to square root raised cosine (pulseshaping filter).

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	<ul style="list-style-type: none"> <li>• <i>n, f, a</i> (default string)</li> <li>• <i>n, b, f, a</i></li> </ul>
Arbitrary Magnitude and Phase	<ul style="list-style-type: none"> <li>• <i>n, f, h</i> (default string)</li> <li>• <i>n, b, f, h</i></li> </ul>
Bandpass	<ul style="list-style-type: none"> <li>• <i>fst1, fp1, fp2, fst2, ast1, ap, ast2</i> (default string)</li> <li>• <i>n, fc1, fc2</i></li> <li>• <i>n, fst1, fp1, fp2, fst2</i></li> </ul>
Bandstop	<ul style="list-style-type: none"> <li>• <i>n, fc1, fc2</i></li> <li>• <i>n, fp1, fst1, fst2, fp2</i></li> <li>• <i>fp1, fst1, fst2, fp2, ap1, ast, ap2</i> (default string)</li> </ul>
CIC	<ul style="list-style-type: none"> <li>• <i>fp, ast</i> (default and only string)</li> </ul>

# fdesign.decimator

---

<b>Design Type</b>	<b>Valid Specification Strings</b>
CIC Compensator	<ul style="list-style-type: none"><li>• fp, fst, ap, ast (default string)</li><li>• n, fc, ap, ast</li><li>• n, fp, ap, ast</li><li>• n, fp, fst</li><li>• n, fst, ap, ast</li></ul>
Gaussian	<p>All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code>. See <code>fdesign.pulseshaping</code></p> <ul style="list-style-type: none"><li>• nsym, bt</li></ul>
Halfband	<ul style="list-style-type: none"><li>• tw, ast (default string)</li><li>• n, tw</li><li>• n</li><li>• n, ast</li></ul>
Highpass	<ul style="list-style-type: none"><li>• fst, fp, ast, ap (default string)</li><li>• n, fc</li><li>• n, fc, ast, ap</li><li>• n, fp, ast, ap</li><li>• n, fst, fp, ap</li><li>• n, fst, fp, ast</li><li>• n, fst, ast, ap</li><li>• n, fst, fp</li></ul>



Design Type	Valid Specification Strings
Inverse-Sinc Lowpass	<ul style="list-style-type: none"> <li>• fp, fst, ap, ast (default string)</li> <li>• n, fc, ap, ast</li> <li>• n, fst, ap, ast</li> <li>• n, fp, ap, ast</li> <li>• n, fp, fst</li> </ul>
Lowpass	<ul style="list-style-type: none"> <li>• fp, fst, ap, ast (default string)</li> <li>• n, fc</li> <li>• n, fc, ap, ast</li> <li>• n, fp, ap, ast</li> <li>• n, fp, fst</li> <li>• n, fp, fst, ap</li> <li>• n, fp, fst, ast</li> <li>• n, fst, ap, ast</li> </ul>
Nyquist	<ul style="list-style-type: none"> <li>• tw, ast (default string)</li> <li>• n, tw</li> <li>• n</li> <li>• n, ast</li> </ul>

Design Type	Valid Specification Strings
Raised cosine	All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code> . See <code>fdesign.pulseshaping</code> <ul style="list-style-type: none"><li>• <code>ast,beta</code> (default string)</li><li>• <code>nsym,beta</code></li><li>• <code>n,beta</code></li></ul>
Square root raised cosine	All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code> . See <code>fdesign.pulseshaping</code> <ul style="list-style-type: none"><li>• <code>ast,beta</code> (default string)</li><li>• <code>nsym,beta</code></li><li>• <code>n,beta</code></li></ul>

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
- **beta** — Rolloff factor for pulse shaping filters (raised cosine and square root raised cosine) expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.
- **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.

# fdesign.decimator

---

- `h` — complex frequency response values
- `n` — filter order.
- `nsym`— Pulse-shaping filter order in symbols. The length of the impulse response is `nsym*SamplesPerSymbol+1`. The product `nsym*SamplesPerSymbol` must be even.
- `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.
```

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');
```

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);
```

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);
```

Now design a decimator using the `equiripple` design method.

```
hm = equiripple(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);
```

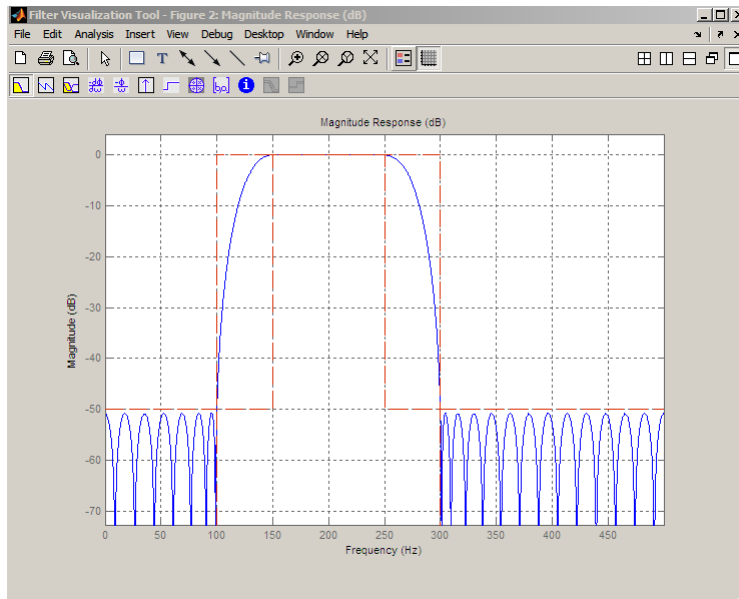
In this example, you specify a sampling frequency as the last input argument. Here is it 1000 Hz.

```
d=fdesign.decimator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,ap,ast2',...  
100,150,250,300,50,.05,50,1000);
```

Design the filter and display the magnitude response in `FVTool`.

# fdesign.decimator

```
fvtool(design(d,'equiripple'))
```



## 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.

# fdesign.differentiator

---

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

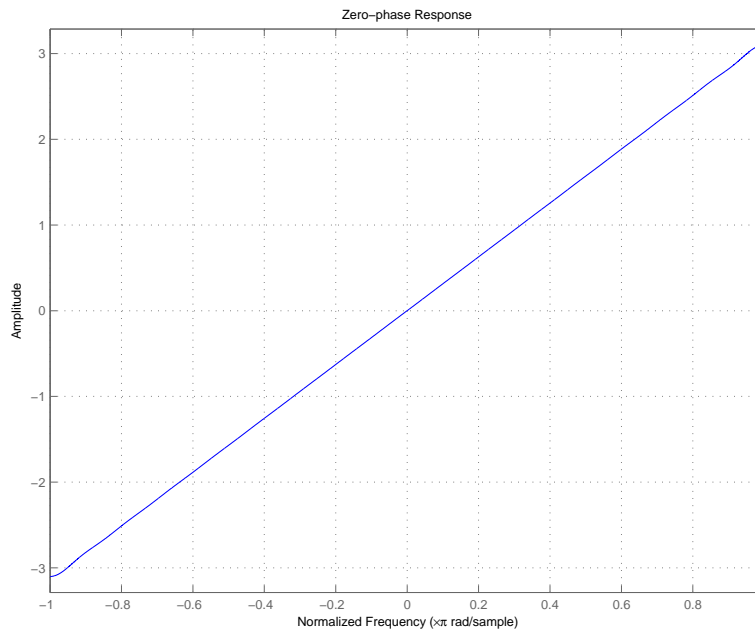


full band differentiator. The FVTool plot following the code shows the resulting 33rd-order filter.

```
d = fdesign.differentiator(33); % Filter order is 33.  
hd = design(d, 'firls');  
fvtool(hd, 'magnitudedisplay', 'zero-phase', ...  
    'frequencyrange', '[-pi, pi]')
```

Design Methods for class fdesign.differentiator (N):

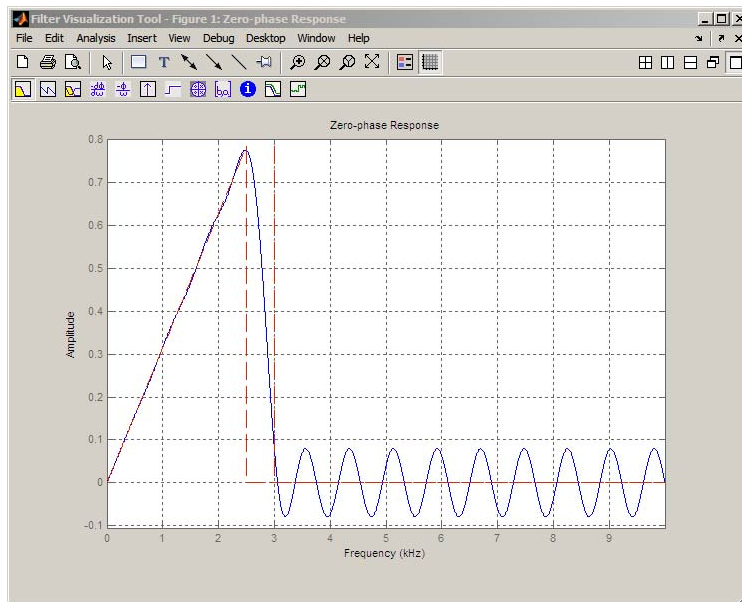
equiripple  
firls



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.

# fdesign.differentiator

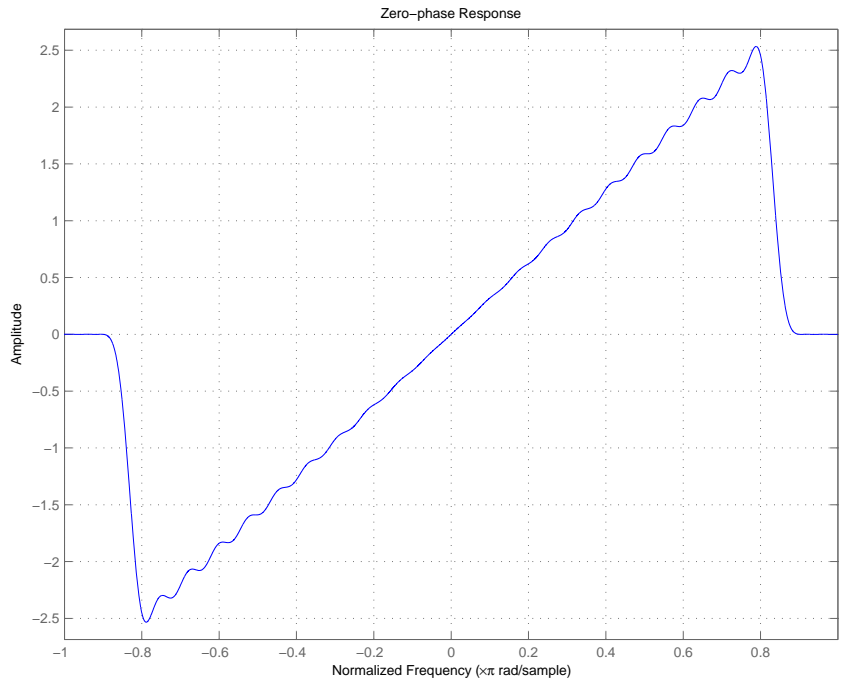
```
Fs=20000; %sampling frequency
d = fdesign.differentiator('n,fp,fst',54,2500,3000,Fs);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange',[0, Fs/2])
```



Finally, design a minimum-order, wide-band differentiator.

```
d = fdesign.differentiator('fp,fst,ap,ast',0.8,0.9,1,80);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange',[-pi, pi])
```

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. Different filter design methods also have options that you can specify. Use `designopts` with the design method string to see the available options. For example:

```
>>f=fdesign.halfband('N,TW');  
>>designmethods(f)
```

```
Design Methods for class fdesign.halfband (N,TW):
```

```
ellip  
iirlinphase  
equiripple  
firls  
kaiserwin  
>>designopts(f,'equiripple')  
ans =
```

```
FilterStructure: 'dffir'  
MinPhase: 0  
ZeroPhase: 0  
StopbandShape: 'flat'  
StopbandDecay: 0
```

```
d = fdesign.halfband(spec,specvalue1,specvalue2,...)  
constructs an object d and sets its specifications at construction time.
```

# fdesign.halfband

---

`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

Create a default halfband filter specifications object:

```
>> d=fdesign.halfband

d =

    Response: 'Halfband'
  Specification: 'TW,Ast'
  Description: {'Transition Width';'Stopband Attenuation (dB)'}
        Type: 'Lowpass'
NormalizedFrequency: true
  TransitionWidth: 0.1
          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 (dB)'}  
    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'
```

# fdesign.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 =

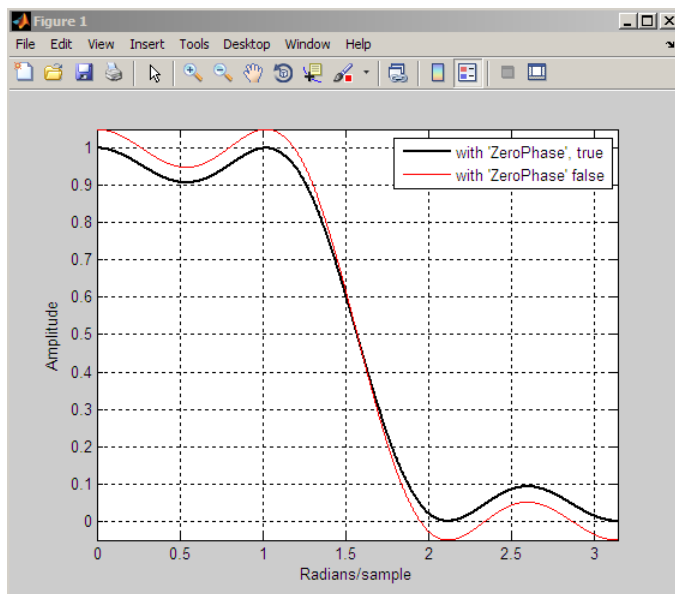
    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'double'
      Numerator: [1x43 double]
 PersistentMemory: false
```

Create two equiripple halfband filters with and without a nonnegative zero phase response:

```
f=fdesign.halfband('N,TW',12,0.2);
% Equiripple halfband filter with nonnegative zero phase response
Hd1=design(f,'equiripple','ZeroPhase',true);
% Equiripple halfband filter with zero phase false
% 'zerophase',false is the default
Hd2=design(f,'equiripple','ZeroPhase',false);
%Obtain real-valued amplitudes (not magnitudes)
[Hr_zerophase,W]=zerophase(Hd1);
[Hr,W]=zerophase(Hd2);
% Plot and compare response
plot(W,Hr_zerophase,'k','linewidth',2);
xlabel('Radians/sample'); ylabel('Amplitude');
hold on;
plot(W,Hr,'r');
axis tight; grid on;
legend('with ''ZeroPhase'', true','with ''ZeroPhase'' false');
```

Note that the amplitude of the zero phase response (black line) is nonnegative for all frequencies.

# fdesign.halfband



The 'ZeroPhase' option is valid only for equiripple halfband designs with the 'N,TW' specification. You cannot specify 'MinPhase' and 'ZeroPhase' to be simultaneously 'true'.

## See Also

fdesign, fdesign.decimator, design, fdesign.interpolator, fdesign.nyquist, setspecs, zerophase

## 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,Ap'`.

Using `fdesign.highpass` with a valid 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.

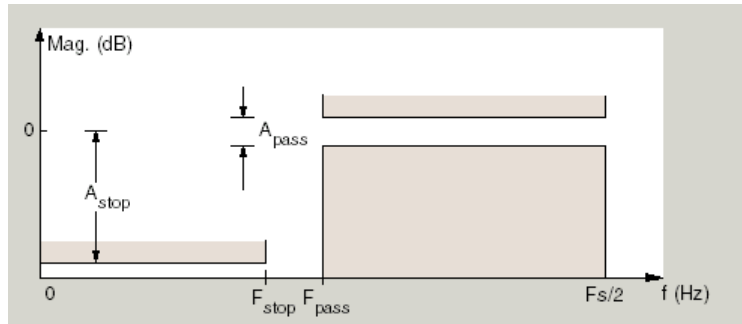
- `'Fst,Fp,Ast,Ap'` (default `spec`)
- `'N,F3db'`
- `'N,F3db,Ap'`
- `'N,F3db,Ast'`
- `'N,F3db,Ast,Ap'`
- `'N,F3db,Fp'`
- `'N,Fc'`
- `'N,Fc,Ast,Ap'`
- `'N,Fp,Ap'`
- `'N,Fp,Ast,Ap'`
- `'N,Fst,Ast'`

- '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 — passband ripple in dB (the default units).
- Ast — stopband attenuation in dB (the default units)
- F3db — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units by default.
- Fc — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units by default.
- Fp — frequency at the start of the passband. Specified in normalized frequency units by default.
- Fst — frequency at the end of the stopband. Specified in normalized frequency units by default.
- N — filter order. FIR designs result in  $N+1$  filter coefficients, or taps. IIR designs results in  $N+1$  numerator and denominator coefficients, or taps.
- Na and Nb denominator and numerator filter orders. Only IIR designs are possible.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like  $F_{st}$  and  $F_p$  are transition, or “don’t care”, 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 you provide for the default `Specification` properties `specvalue1,specvalue2,specvalue3,specvalue4`.

`d = fdesign.highpass(...,Fs)` specifies the sampling frequency  $F_s$  in Hz as a scalar trailing all other numeric input arguments. If you provide a sampling frequency, all frequencies in the specification are in Hz.

`d = fdesign.highpass(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of the following strings:

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)

# fdesign.highpass

---

- '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 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
```

Pass numeric specifications that correspond to the default Specification string.

```
d = fdesign.highpass(0.4,0.5,80,1);

d =

    Response: 'Minimum-order highpass'
  Specification: 'Fst,Fp,Ast,Ap'
    Description: {4x1 cell}
  NormalizedFrequency: true
                Fstop: 0.4000
                Fpass: 0.5000
```



```
Astop: 80  
Apass: 1
```

Create an object with the specification type string 'N,Fc' — the resulting object uses default values for N and Fc.

```
d = fdesign.highpass('N,Fc')
```

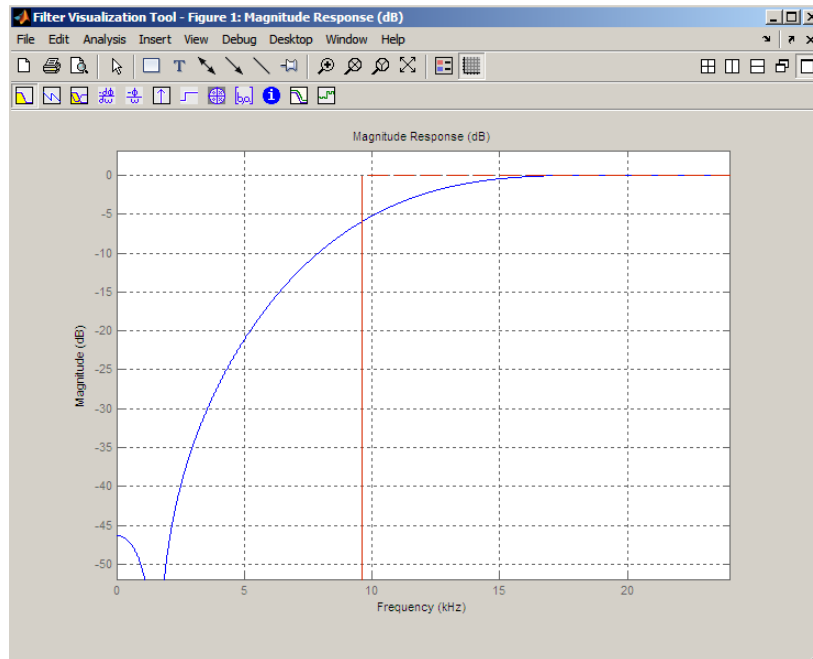
```
d =
```

```
Response: 'Highpass with cutoff'  
Specification: 'N,Fc'  
Description: {2x1 cell}  
NormalizedFrequency: true  
FilterOrder: 10  
Fcutoff: 0.5000
```

Create another filter object using the specification string 'N,Fc'. Include the sampling frequency Fs as the final input argument. Design the filter and plot the result.

```
d =fdesign.highpass('n,fc',10,9600,48000);  
%FIR window design. Fc is the 6-dB down point.  
Hd = design(d);  
fvtool(Hd);
```

# fdesign.highpass



The next example adds the `MAGUNITS` option.

```
d = fdesign.highpass('N,Fc',10,9600,48000,'squared');
```

## 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

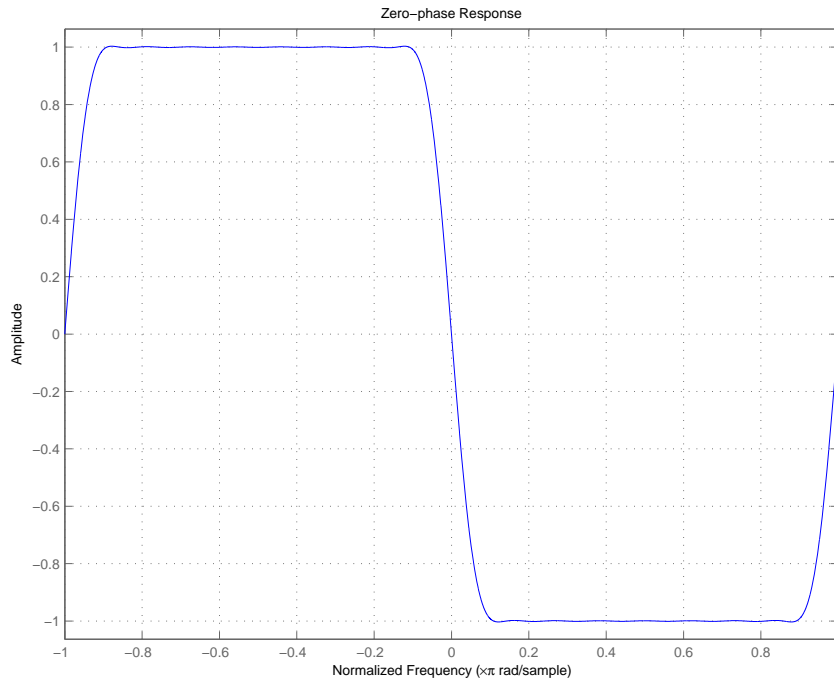
The toolbox lets you design a range of Hilbert filters. These examples present a few possible designs. The first example designs a 30th-order type III Hilbert transformer filter. The FVTool plot following the code shows the resulting filter.

```
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
```

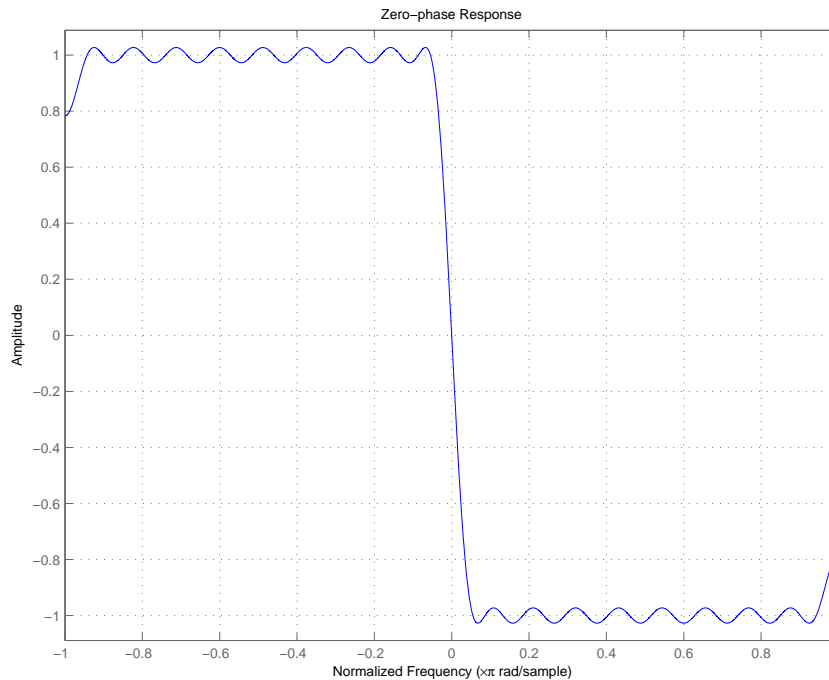


For the second example, design a 35th-order type IV Hilbert transformer.

# fdesign.hilbert

```
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.

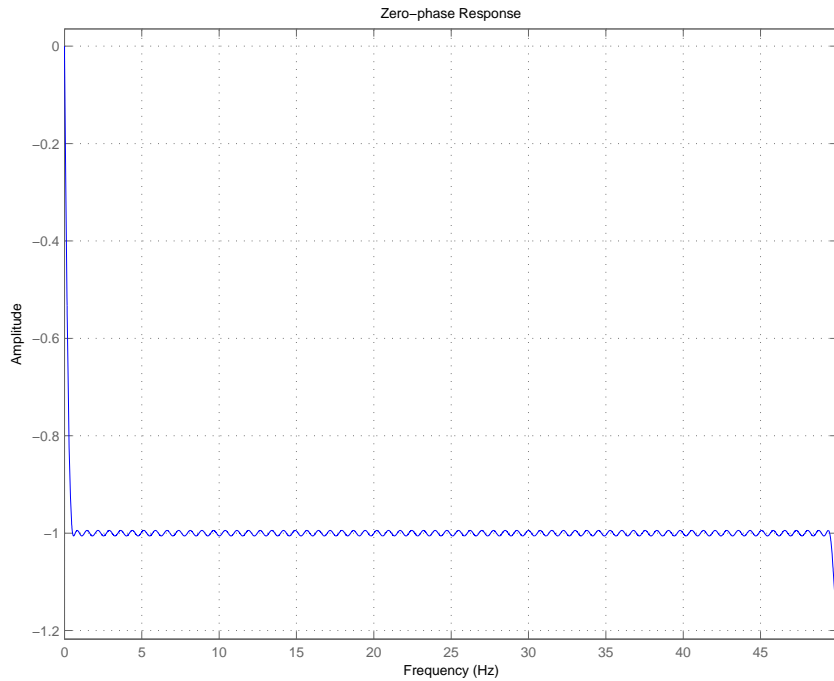


Finally, design a minimum-order transformer that has a sampling frequency of 100 Hz — add  $F_s$  as an input argument in Hz.

```
d = fdesign.hilbert('tw,ap',1,0.1,100); %  $F_s = 100$  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.
arbmagphase	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.
Gaussian	Sets the response for the interpolator specifications object to Gaussian pulse-shaping
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.
raised cosine	Sets the response for the interpolator specifications object to raised cosine pulse-shaping
square root raised cosine	Sets the response for the interpolator specifications object to square root raised cosine pulse-shaping

`d = fdesign.interpolator(1,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.

# fdesign.interpolator

---

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	<ul style="list-style-type: none"><li>• <code>n, b, f, a</code></li><li>• <code>n, f, a</code> (default string)</li></ul>
Arbitrary Magnitude and Phase	<ul style="list-style-type: none"><li>• <code>n, b, f, h</code></li><li>• <code>n, f, h</code> (default string)</li></ul>
Bandpass	<ul style="list-style-type: none"><li>• <code>fst1, fp1, fp2, fst2, ast1, ap, ast2</code> (default string)</li><li>• <code>n, fc1, fc2</code></li><li>• <code>n, fst1, fp1, fp2, fst2</code></li></ul>
Bandstop	<ul style="list-style-type: none"><li>• <code>n, fc1, fc2</code></li><li>• <code>n, fp1, fst1, fst2, fp2</code></li><li>• <code>fp1, fst1, fst2, fp2, ap1, ast, ap2</code> (default string)</li></ul>
CIC	<ul style="list-style-type: none"><li>• <code>fp, ast</code> (default and only string)</li></ul>
CIC Compensator	<ul style="list-style-type: none"><li>• <code>fp, fst, ap, ast</code> (default string)</li><li>• <code>n, fc, ap, ast</code></li><li>• <code>n, fp, ap, ast</code></li><li>• <code>n, fp, fst</code></li><li>• <code>n, fst, ap, ast</code></li></ul>

Design Type	Valid Specification Strings
Gaussian	<p>All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code>. See <code>fdesign.pulseshaping</code></p> <ul style="list-style-type: none"> <li>• <code>nsym, bt</code></li> </ul>
Halfband	<ul style="list-style-type: none"> <li>• <code>tw, ast</code> (default string)</li> <li>• <code>n, tw</code></li> <li>• <code>n</code></li> <li>• <code>n, ast</code></li> </ul>
Highpass	<ul style="list-style-type: none"> <li>• <code>fst, fp, ast, ap</code> (default string)</li> <li>• <code>n, fc</code></li> <li>• <code>n, fc, ast, ap</code></li> <li>• <code>n, fp, ast, ap</code></li> <li>• <code>n, fst, fp, ap</code></li> <li>• <code>n, fst, fp, ast</code></li> <li>• <code>n, fst, ast, ap</code></li> <li>• <code>n, fst, fp</code></li> </ul>
Inverse-Sinc Lowpass	<ul style="list-style-type: none"> <li>• <code>fp, fst, ap, ast</code> (default string)</li> <li>• <code>n, fc, ap, ast</code></li> <li>• <code>n, fst, ap, ast</code></li> <li>• <code>n, fp, ap, ast</code></li> <li>• <code>n, fp, fst</code></li> </ul>
Lowpass	<ul style="list-style-type: none"> <li>• <code>fp, fst, ap, ast</code> (default string)</li> </ul>

# fdesign.interpolator

---

Design Type	Valid Specification Strings
Nyquist	<ul style="list-style-type: none"><li>• <code>tw,ast</code> (default string)</li><li>• <code>n,tw</code></li><li>• <code>n</code></li><li>• <code>n,ast</code></li></ul>
Raised cosine	<p>All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code>. See <code>fdesign.pulseshaping</code></p> <ul style="list-style-type: none"><li>• <code>ast,beta</code> (default string)</li><li>• <code>nsym,beta</code></li><li>• <code>n,beta</code></li></ul>
Square root raised cosine	<p>All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code>. See <code>fdesign.pulseshaping</code></p> <ul style="list-style-type: none"><li>• <code>ast,beta</code> (default string)</li><li>• <code>nsym,beta</code></li><li>• <code>n,beta</code></li></ul>

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.
- `beta` — Rolloff factor for pulse shaping filters (raised cosine and square root raised cosine) expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.
- `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 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 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.

## fdesign.interpolator

---

- `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.
- `nsym` — Pulse-shaping filter order in symbols. The length of the impulse response is `nsym*SamplesPerSymbol+1`. The product `nsym*SamplesPerSymbol` must be even.
- `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
```

# fdesign.interpolator

---

```
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 `equiripple` design method.

```
hm = design(d,'equiripple')

hm =

    FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
    Arithmetic: 'double'
    Numerator: [1x21 double]
    InterpolationFactor: 3
    PersistentMemory: false
```

Pass a new specification type for the filter, specifying the filter order.

```
d = fdesign.interpolator(5,'CIC',1,'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
```

With the specifications object in your workspace, design an interpolator using the `multisection` design method.

# fdesign.interpolator

---

```
hm = design(d,'multisection')
```

```
hm =
```

```
    FilterStructure: 'Cascaded Integrator-Comb Interpolator'  
      Arithmetic: 'fixed'  
DifferentialDelay: 1  
  NumberOfSections: 28  
InterpolationFactor: 5  
  PersistentMemory: false  
  
    InputWordLength: 16  
    InputFracLength: 15  
  
    FilterInternals: 'FullPrecision'
```

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  $(1 \times fs)$  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(1,'cic',dd,'fp,ast',fp,ast,l*fs);  
d =
```

# fdesign.interpolator

---

```
        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.

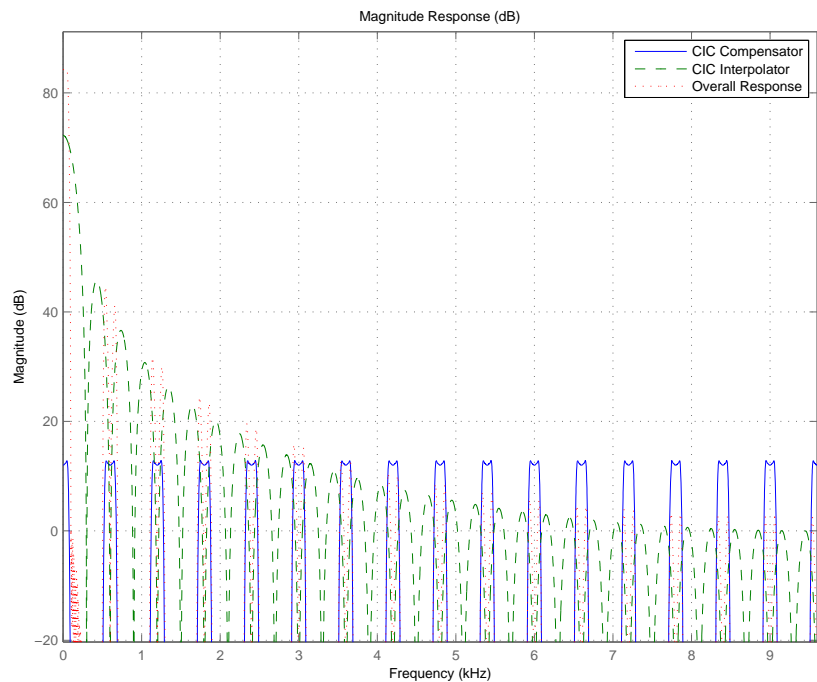
```
nsecs = hm.numberofsections;
d = fdesign.interpolator(4,'ciccomp',dd,nsecs,...
50,100,0.1,80,fs);
```

```
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,1*fs,1*fs],...
'showreference','off');
legend('CIC Compensator','CIC Interpolator',...
'Overall Response');
```

FVTool returns with this plot.



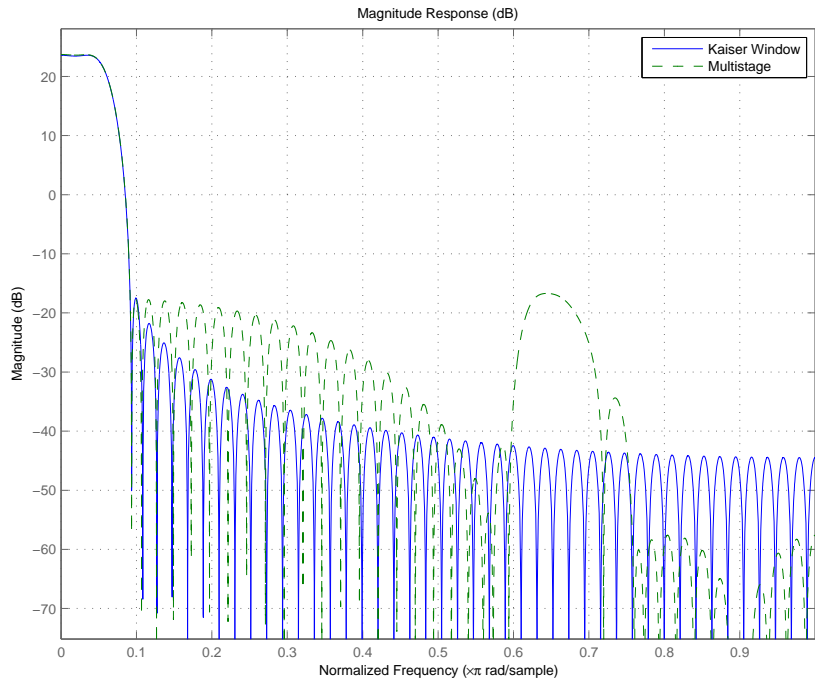
# fdesign.interpolator

---

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(1,'nyquist',l,tw,ast);
hm = design(d,'kaiserwin');
hm2 = design(d,'multistage'); % Design the multistage interpolator.
fvtool(hm,hm2);
legend('Kaiser Window','Multistage')
```

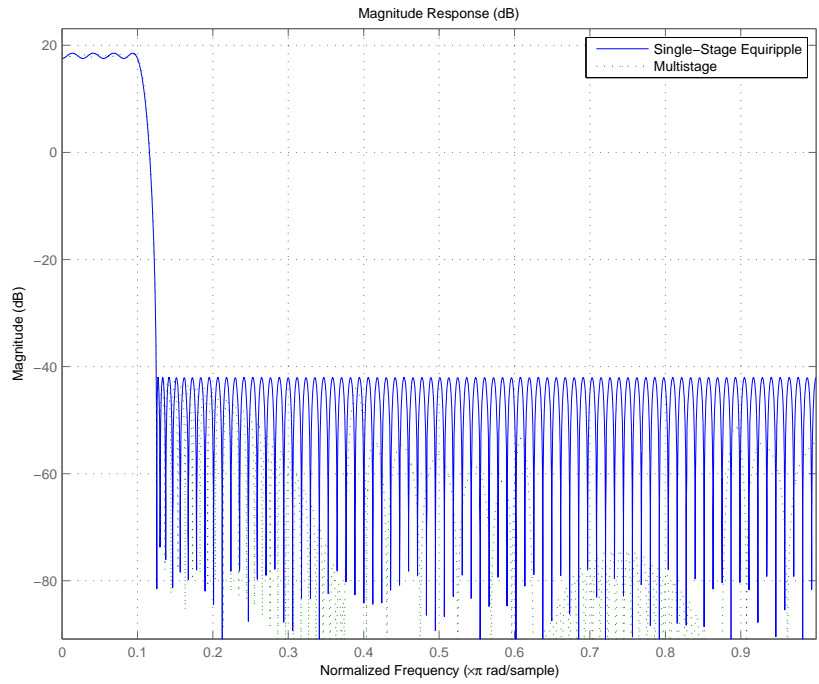
FVTool shows both responses.



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')
```

# fdesign.interpolator



## 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`.

# fdesign.isinclp

---

- `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(...,MAGNUNITS)
```

**Description** `d = fdesign.lowpass` constructs a lowpass filter specification object `d` applying default values for the specifications `Fp`, `Fst`, `Ap`, and `Ast`.

Using the `fdesign.lowpass` specification object with a valid design method generates a `dfilt` object.

`d = fdesign.lowpass(spec)` constructs the specification 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'
- 'Nb,Na,F3dB'
- 'N,F3db,Ap'
- 'N,F3db,Ap,Ast'
- 'N,F3db,Ast'
- 'N,F3db,Fst'
- 'N,Fc'
- 'N,Fc,Ap,Ast'
- 'N,Fp,Ap'
- 'N,Fp,Ap,Ast'

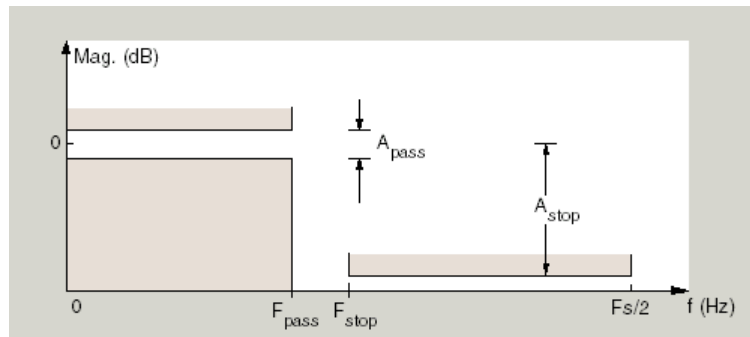
- 'N,Fp,Fst,Ap'
- 'N,Fp,F3db'
- 'N,Fp,Fst'
- 'N,Fp,Fst,Ast'
- 'N,Fst,Ap,Ast'
- 'N,Fst,Ast'
- 'Nb,Na,Fp,Fst'

The string entries are defined as follows:

- Ap — passband ripple in dB (the default units).
- Ast — stopband attenuation in dB (the default units).
- F3db — cutoff frequency for the point 3 dB below the passband value. Specified in normalized frequency units by default.
- Fc — cutoff frequency for the point 6 dB below the passband value. Specified in normalized frequency units by default.
- Fp — frequency at the edge of the passband. Specified in normalized frequency units by default.
- Fst — frequency at the beginning of the stopband. Specified in normalized frequency units by default.
- N — filter order. FIR designs result in N+1 filter coefficients, or taps. IIR designs result in N+1 numerator and denominator filter coefficients, or taps.
- Na and Nb — denominator and numerator filter orders. Only IIR designs are possible.

Graphically, the filter specifications look similar to those shown in the following figure.

# fdesign.lowpass



Regions between specification values like  $F_p$  and  $F_{st}$  are transition, or “don’t care” regions where the filter response is not explicitly defined.

`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 string `Fp,Fst,Ap,Ast` you provide as the input arguments `specvalue1,specvalue2,specvalue3,specvalue4`.

`d = fdesign.lowpass(...,Fs)` specifies the sampling frequency  $F_s$  in Hz as a scalar trailing all other numeric input arguments. If you provide a sampling frequency, all frequencies in the specifications are in Hz.

`d = fdesign.lowpass(...,MAGUNITS)` specifies the units for any magnitude specifications you provide in the input arguments. When a sampling frequency is specified, `MAGUNITS` must follow the sampling frequency. `MAGUNITS` is one of the following strings:

- '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 demonstrate 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
```

You can directly assign values for the specifications in the default string when you create the object:

```
hs = fdesign.lowpass(0.4,0.5,1,80);
hs

hs =

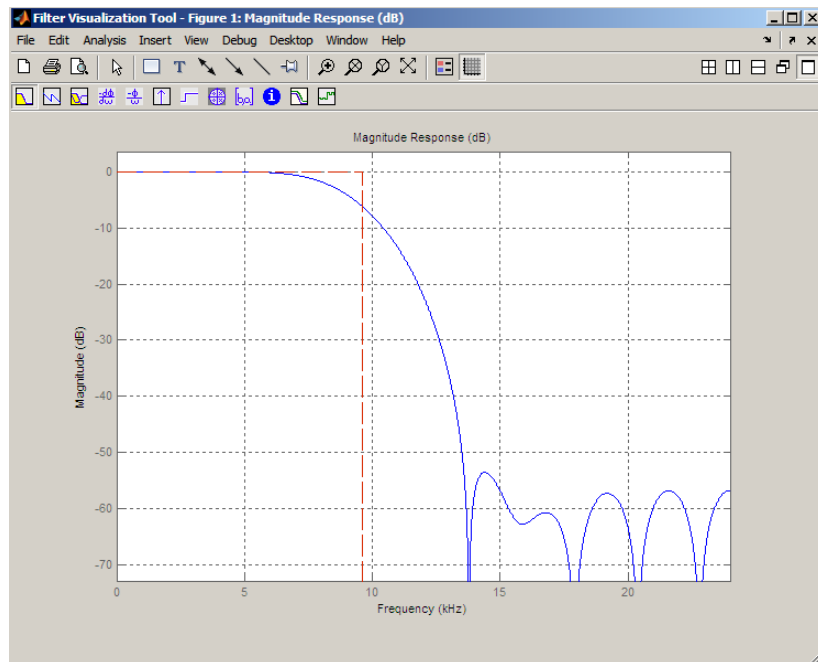
    Response: 'Minimum-order lowpass'
  Specification: 'Fp,Fst,Ap,Ast'
    Description: {4x1 cell}
  NormalizedFrequency: true
           Fpass: 0.4000
           Fstop: 0.5000
           Apass: 1
```

# fdesign.lowpass

Astop: 80

Create another filter object using the specification string 'N,Fc'. Include the sampling frequency Fs as the final input argument. Design the filter and plot the result.

```
d = fdesign.lowpass('N,Fc',20,9600,48000);  
Hd = design(d); % Uses the FIR window method  
fvtool(Hd)
```



The next example adds the MAGUNIT option.

```
d = fdesign.lowpass('N,Fp,Ap',10,9600,0.5,48000,'squared');
```

## 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.

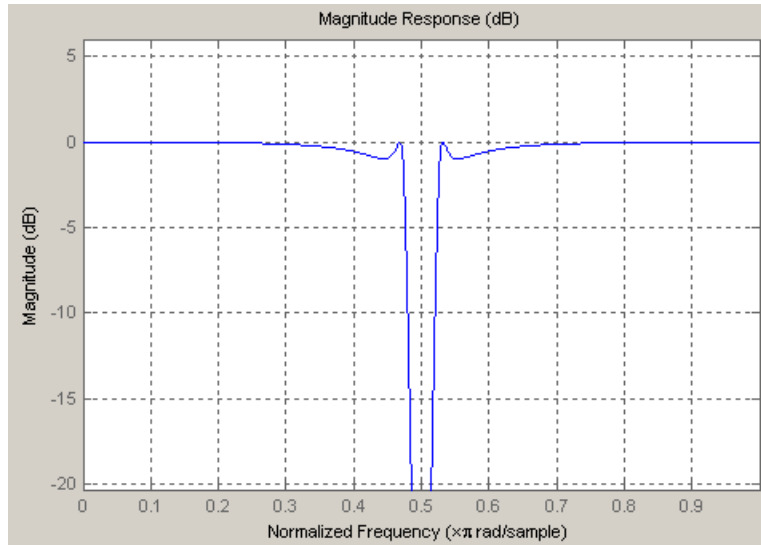
## Examples

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

**Purpose** Nyquist filter specification

**Syntax**

```
d = fdesign.nyquist
d = fdesign.nyquist(1,spec)
d = fdesign.nyquist(1,spec,specvalue1,specvalue2,...)
d = fdesign.nyquist(1,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(1,spec)` constructs object `d` and sets its `Specification` property to `spec`. Use `1` to specify the desired value for `L`.  $L = 2$  designs 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 a Nyquist filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string. Different filter design methods also have options that you can specify. Use `designopts` with the design method string to see the available options. For example:

```
f=fdesign.nyquist(4,'N,TW');  
designmethods(f)
```

`d = fdesign.nyquist(1,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(1,specvalue1,specvalue2)` constructs an object `d` with the values you provide in `1`, `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  $\text{band} = 2$ ) 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
```

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')
```

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)
```

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)
```

Now design a Nyquist filter using the kaiserwin design method.

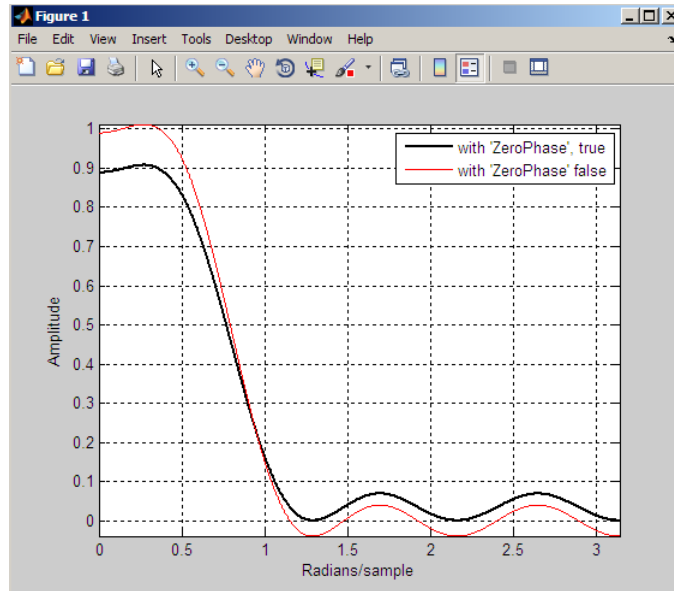
```
hd = design(d,'kaiserwin')
```

Create two equiripple Nyquist 4th-band filters with and without a nonnegative zero phase response:

```
f=fdesign.nyquist(4,'N,TW',12,0.2);
% Equiripple Nyquist 4th-band filter with
% nonnegative zero phase response
Hd1=design(f,'equiripple','zerophase',true);
% Equiripple Nyquist 4th-band filter with 'ZeroPhase' set to false
% 'zerophase',false is the default
Hd2=design(f,'equiripple','zerophase',false);
%Obtain real-valued amplitudes (not magnitudes)
[Hr_zerophase,W]=zerophase(Hd1);
[Hr,W]=zerophase(Hd2);
% Plot and compare response
plot(W,Hr_zerophase,'k','linewidth',2);
xlabel('Radians/sample'); ylabel('Amplitude');
hold on;
plot(W,Hr,'r');
axis tight; grid on;
```

```
legend('with 'ZeroPhase'', true','with 'ZeroPhase'' false');
```

Note that the amplitude of the zero phase response (black line) is nonnegative for all frequencies.



The 'ZeroPhase' option is valid only for equiripple Nyquist designs with the 'N,TW' specification. You cannot specify 'MinPhase' and 'ZeroPhase' to be simultaneously 'true'.

## See Also

fdesign, fdesign.interpolator, fdesign.halfband,  
fdesign.interpolator, fdesign.rsrc, zerophase



**Purpose** Octave filter specification

**Syntax**

```
d = fdesign.octave(1)
d = fdesign.octave(1, MASK)
d = fdesign.octave(1, MASK, spec)
d = fdesign.octave(..., Fs)
```

**Description** `d = fdesign.octave(1)` constructs an octave filter specification object `d`, with 1 bands per octave. The default value for 1 is 1.

`d = fdesign.octave(1, MASK)` constructs an octave filter specification object `d` with 1 bands per octave and `MASK` specification for the FVTool. The available values for `mask` are:

- 'class 0'
- 'class 1'
- 'class 2'

`d = fdesign.octave(1, MASK, spec)` constructs an octave filter specification object `d` with 1 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).

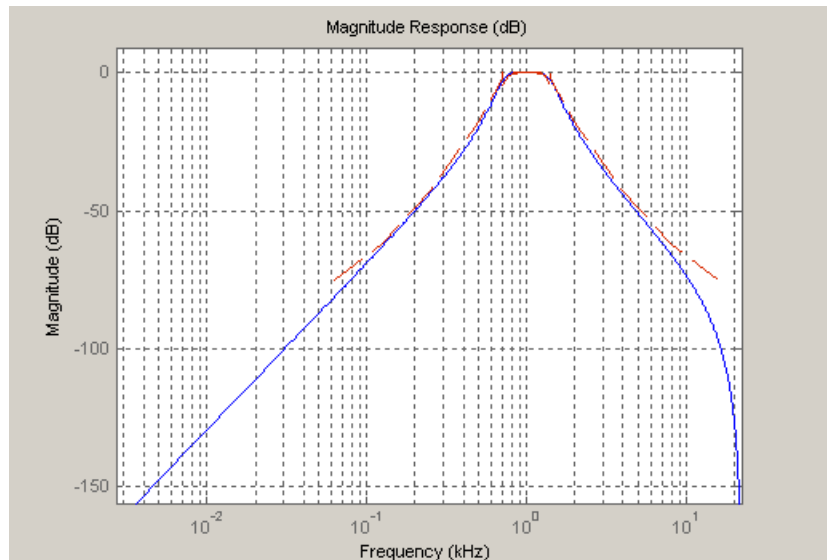
# fdesign.octave

## 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.parmeq(spec, specvalue1, specvalue2, ...)  
d = fdesign.parmeq(... fs)
```

## Description

`d = fdesign.parmeq(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, Fc, Qa, GO'
- 'N, FO, Fc, S, GO'
- '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:

- 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
- F0 — Center Frequency
- Fc — Cutoff frequency
- Fhigh - Higher Frequency at Gain GBW
- Flow - Lower Frequency at Gain GBW
- Qa-Quality Factor
- S-Slope Parameter for Shelving Filters

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.parmeq(... 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

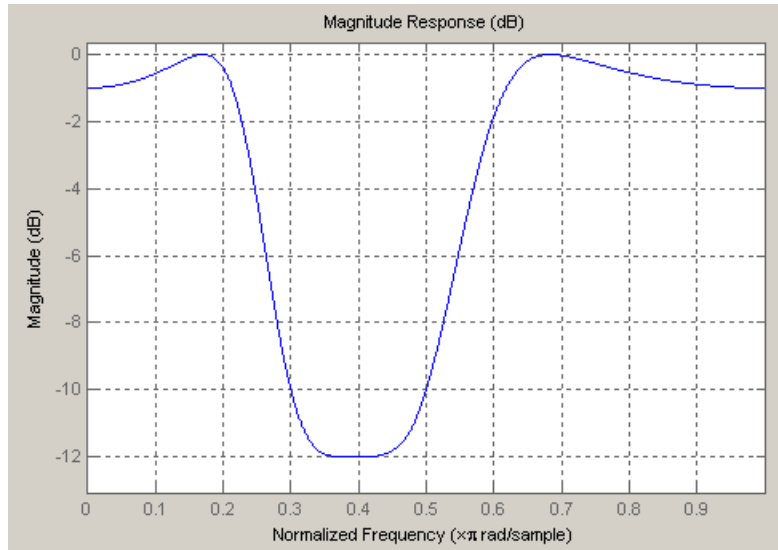
Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB:

```
d = fdesign.parmeq('N,Flow,Fhigh,Gref,G0,GBW,Gst',...
```

```

    4, .3, .5, 0, -12, -10, -1);
Hd = design(d, 'cheby2');
fvtool(Hd)

```

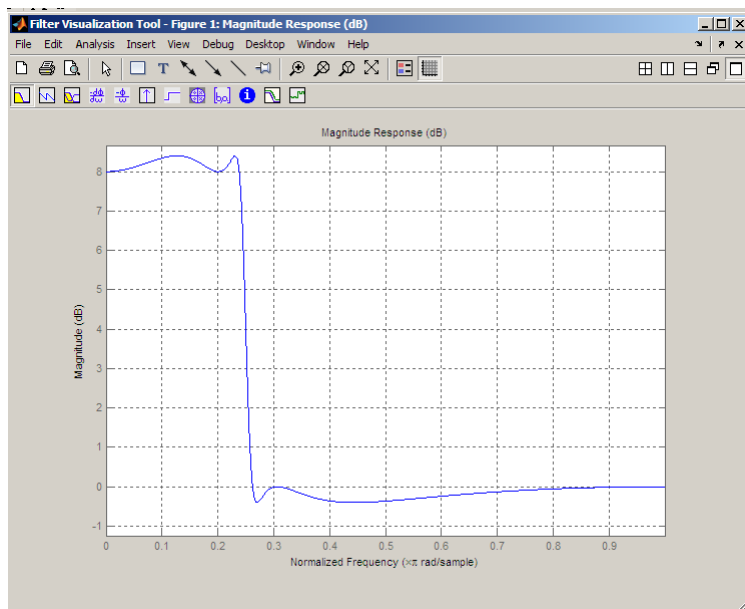


Design a 4th order audio lowpass ( $F_0 = 0$ ) shelving filter with cutoff frequency of  $F_c = 0.25$ , quality factor  $Q_a = 10$ , and boost gain of  $G_0 = 8$  dB:

```

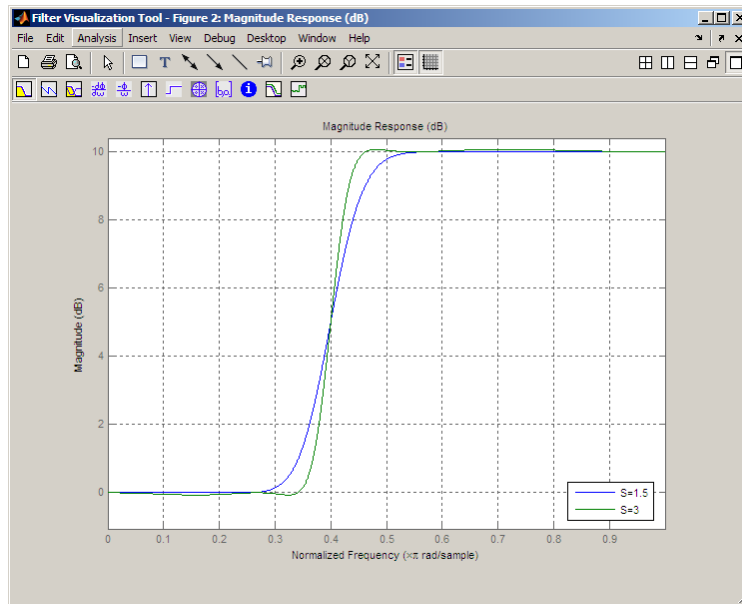
d = fdesign.parmeq('N,F0,Fc,Qa,G0',4,0,0.25,10,8);
Hd = design(d);
fvtool(Hd)

```



Design 4th-order highpass shelving filters with  $S=1.5$  and  $S=3$ :

```
N=4;  
F0 = 1;  
Fc = .4; % Cutoff Frequency  
G0 = 10;  
S = 1.5;  
S2=3;  
f = fdesign.parmeq('N,F0,Fc,S,G0',N,F0,Fc,S,G0);  
h1 = design(f);  
f.S=3;  
h2=design(f);  
hfvf=fvtool([h1 h2]);  
set(hfvf,'Filters',[h1 h2]);  
legend(hfvf,'S=1.5','S=3');
```



**See Also** [fdesign](#), the demo for Parametric Equalizer Design

**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

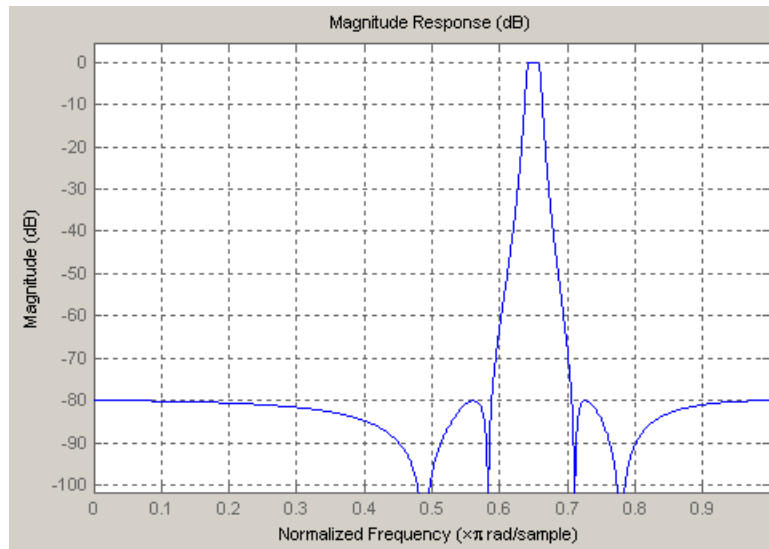
Design a Chebyshev Type II peaking filter with a stopband attenuation of 80 dB:

# fdesign.peak

---

```
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, fdesign.parmeq

**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.

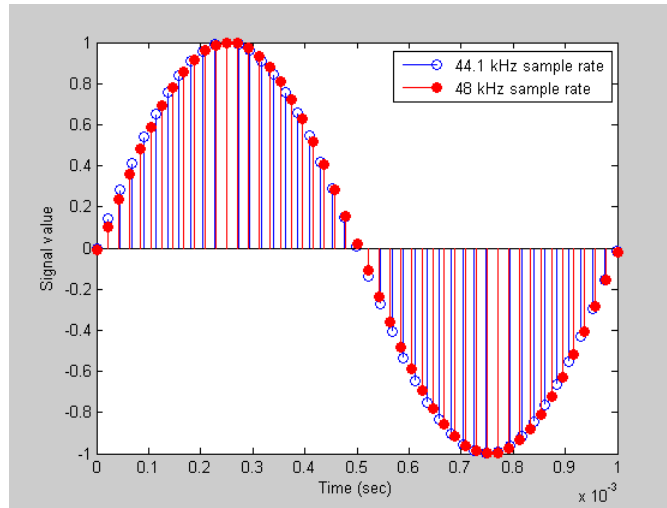
`d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np)` initializes the filter designer specification with `Np` and sets the polynomial order to the value `Np`. If omitted `Np` defaults to 3.

`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, `efficientsredemo`.

## See Also

`fdesign`

**Purpose** Pulse-shaping filter specification object

**Syntax**

```
d = fdesign.pulseshaping
d = fdesign.pulseshaping(sps)
d = fdesign.pulseshaping(sps,shape)
d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)
d = fdesign.pulseshaping(...,fs)
d = fdesign.pulseshaping(...,magunits)
```

**Description** `d = fdesign.pulseshaping` constructs a specification object `d`, which can be used to design a minimum-order raised cosine filter object with a stop band attenuation of 60dB and a rolloff factor of 0.25.

```
d =
```

```
        Response: 'Pulse Shaping'
        PulseShape: 'Raised Cosine'
SamplesPerSymbol: 8
        Specification: 'Ast,Beta'
        Description: {'Stopband Attenuation (dB)';'Rolloff Factor'}
NormalizedFrequency: true
        Astop: 60
        RolloffFactor: 0.25
```

`d = fdesign.pulseshaping(sps)` constructs a minimum-order raised cosine filter specification object `d` with a positive integer-valued oversampling factor, `SamplesPerSymbol`.

`d = fdesign.pulseshaping(sps,shape)` constructs `d` where `shape` specifies the `PulseShape` property. Valid entries for `shape` are:

- 'Raised Cosine'
- 'Square Root Raised Cosine'
- 'Gaussian'

`d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)` constructs `d` where `spec` defines the `Specification` properties. The

string entries for `spec` specify various properties of the filter, including the order and frequency response. Valid entries for `spec` depend upon the `shape` property. For 'Raised Cosine' and 'Square Root Raised Cosine' filters, the valid entries for `spec` are:

- 'Ast,Beta' (minimum order; default)
- 'Nsym,Beta'
- 'N,Beta'

The string entries are defined as follows:

- `Ast` —stopband attenuation (in dB). The default stopband attenuation for a raised cosine filter is 60 dB. The default stopband attenuation for a square root raised cosine filter is 30 dB. If `Ast` is specified, the minimum-order filter is returned.
- `Beta` —rolloff factor expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.
- `Nsym` —filter order in symbols. The length of the impulse response is given by `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.
- `N` —filter order (must be even). The length of the impulse response is `N+1`.

If the `shape` property is specified as 'Gaussian', the valid entries for `spec` are:

- 'Nsym,BT' (default)

The string entries are defined as follows:

- `Nsym`—filter order in symbols. `Nsym` defaults to 6. The length of the filter impulse response is `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.

- **BT** —the 3-dB bandwidth-symbol time product. **BT** is a positive real-valued scalar, which defaults to 0.3. Larger values of **BT** produce a narrower pulse width in time with poorer concentration of energy in the frequency domain.

`d = fdesign.pulseshaping(...,fs)` specifies the sampling frequency of the signal to be filtered. `fs` must be specified as a scalar trailing the other numerical values provided. For this case, `fs` is assumed to be in Hz and is used for analysis and visualization.

`d = fdesign.pulseshaping(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. Valid entries for `magunits` are:

- **linear** — specify the magnitude in linear units
- **dB** — specify the magnitude in dB (decibels)
- **squared** — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

After creating the specification object `d`, you can use the `design` function to create a filter object such as `h` in the following example:

```
d = fdesign.pulseshaping(8,'Raised Cosine','Nsym,Beta',6,0.25);  
h = design(d);
```

Normally, the `Specification` property of the specification object also determines which design methods you can use when you create the filter object. Currently, regardless of the `Specification` property, the `design` function uses the window design method with all `fdesign.pulseshaping` specification objects. The window method creates an FIR filter with a windowed impulse response.

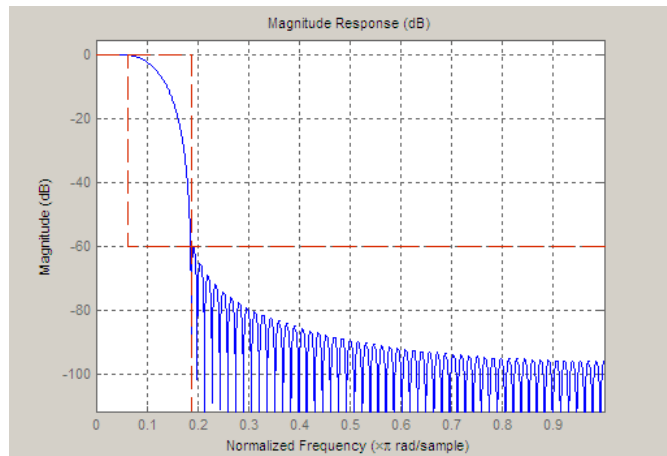
## Examples

Pulse-shaping can be used to change the waveform of transmitted pulses so the signal bandwidth matches that of the communication channel. This helps to reduce distortion and intersymbol interference (ISI).

This example shows how to design a minimum-order raised cosine filter that provides a stop band attenuation of 60 dB, rolloff factor of 0.50, and 8 samples per symbol.

```
h = fdesign.pulseshaping(8,'Raised Cosine','Ast,Beta',60,0.50);  
Hd = design(h);  
fvtool(Hd)
```

This code generates the following figure.

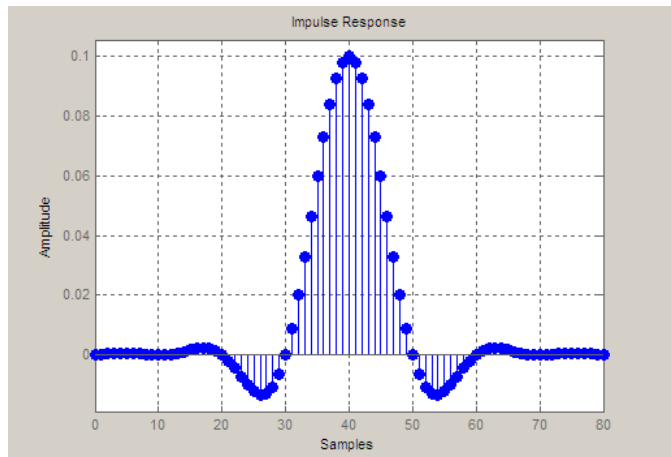


This example shows how to design a raised cosine filter that spans 8 symbol durations (i.e., of order 8 symbols), has a rolloff factor of 0.50, and oversampling factor of 10.

```
h = fdesign.pulseshaping(10,'Raised Cosine','Nsym,Beta',8,0.50);  
Hd = design(h);  
fvtool(Hd, 'impulse')
```

This code generates the following figure.

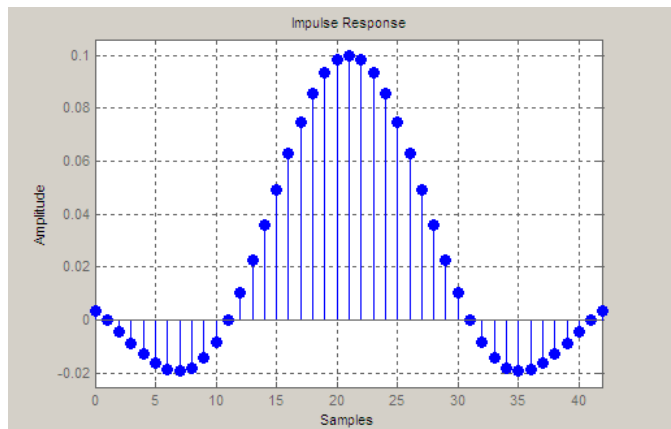




This example shows how to design a square root raised cosine filter of order 42, rolloff factor of 0.25, and 10 samples per symbol.

```
h = fdesign.pulseshaping(10,'Square Root Raised Cosine','N,Beta',42);  
Hd = design(h);  
fvtool(Hd, 'impulse')
```

This code generates the following figure.



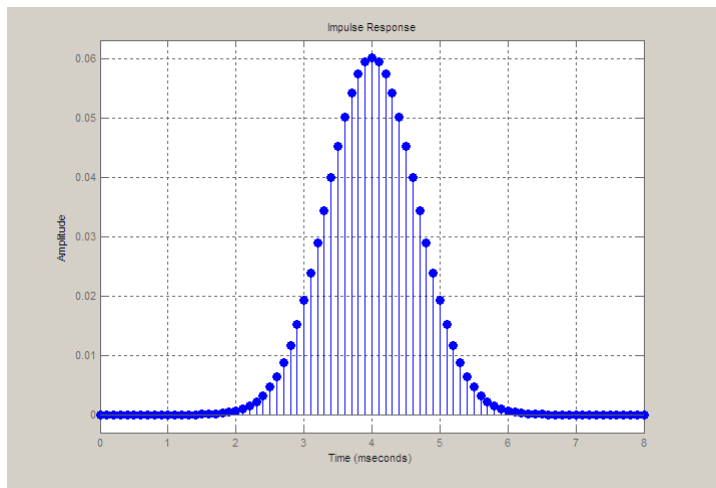
# fdesign.pulseshaping

---

The following example demonstrates how to create a Gaussian pulse-shaping filter with an oversampling factor (sps) of 10, a bandwidth-time symbol product of 0.2, and 8 symbol periods. The sampling frequency is specified as 10 kHz.

```
d=fdesign.pulseshaping(10,'gaussian','nsym,bt',8,0.2,10000);  
Hd=design(d); %note the length of d.Numerator is 8*10+1  
fvtool(Hd,'impulse')
```

The above code generates the following figure. Note the time axis in milliseconds.



For more information, see the “Pulse Shaping Filter Design” demo, [pulseshapingfilterdemo](#)

## See Also

[fdesign](#)  
[design](#)

**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.

<b>design String</b>	<b>Description</b>
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.

<b>design String</b>	<b>Description</b>
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.
Gaussian	Sets the design for the rational-factor sample-rate converter specifications object to Gaussian pulse-shaping.
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.

<b>design String</b>	<b>Description</b>
raised cosine	Sets the design for the rational-factor sample-rate converter specifications object to raised cosine pulse-shaping.
square root raised cosine	Sets the design for the rational-factor sample-rate converter specifications object to square root raised cosine pulse-shaping.

`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.

<b>Design Type</b>	<b>Valid Specification Strings</b>
Arbitrary Magnitude	<ul style="list-style-type: none"> <li>• <code>n,f,a</code> (default string)</li> <li>• <code>n,b,f,a</code></li> </ul>
Arbitrary Magnitude and Phase	<ul style="list-style-type: none"> <li>• <code>n,f,h</code> (default string)</li> <li>• <code>n,b,f,h</code></li> </ul>

Design Type	Valid Specification Strings
Bandpass	<ul style="list-style-type: none"><li>• <code>fst1,fp1,fp2,fst2,ast1,ap,ast2</code> (default string)</li><li>• <code>n,fc1,fc2</code></li><li>• <code>n,fst1,fp1,fp2,fst2</code></li></ul>
Bandstop	<ul style="list-style-type: none"><li>• <code>n,fc1,fc2</code></li><li>• <code>n,fp1,fst1,fst2,fp2</code></li><li>• <code>fp1,fst1,fst2,fp2,ap1,ast,ap2</code> (default string)</li></ul>
CIC	<ul style="list-style-type: none"><li>• <code>fp,ast</code> (default and only string)</li></ul>
CIC Compensator	<ul style="list-style-type: none"><li>• <code>fp,fst,ap,ast</code> (default string)</li><li>• <code>n,fc,ap,ast</code></li><li>• <code>n,fp,ap,ast</code></li><li>• <code>n,fp,fst</code></li><li>• <code>n,fst,ap,ast</code></li></ul>
Gaussian	<p>All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code>. See <code>fdesign.pulseshaping</code></p> <ul style="list-style-type: none"><li>• <code>nsym,bt</code></li></ul>
Halfband	<ul style="list-style-type: none"><li>• <code>tw,ast</code> (default string)</li><li>• <code>n,tw</code></li><li>• <code>n</code></li><li>• <code>n,ast</code></li></ul>

<b>Design Type</b>	<b>Valid Specification Strings</b>
Highpass	<ul style="list-style-type: none"> <li>• fst,fp,ast,ap (default string)</li> <li>• n,fc</li> <li>• n,fc,ast,ap</li> <li>• n,fp,ast,ap</li> <li>• n,fst,fp,ap</li> <li>• n,fst,fp,ast</li> <li>• n,fst,ast,ap</li> <li>• n,fst,fp</li> </ul>
Inverse-Sinc Lowpass	<ul style="list-style-type: none"> <li>• fp,fst,ap,ast (default string)</li> <li>• n,fc,ap,ast</li> <li>• n,fp,fst</li> </ul>
Lowpass	<ul style="list-style-type: none"> <li>• fp,fst,ap,ast (default string)</li> <li>• n,fc</li> <li>• n,fc,ap,ast</li> <li>• n,fp,ap,ast</li> <li>• n,fp,fst</li> <li>• n,fp,fst,ap</li> <li>• n,fp,fst,ast</li> <li>• n,fst,ap,ast</li> </ul>
Nyquist	<ul style="list-style-type: none"> <li>• tw,ast (default string)</li> <li>• n,tw</li> <li>• n</li> <li>• n,ast</li> </ul>

Design Type	Valid Specification Strings
Raised cosine	<p>All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code>. See <code>fdesign.pulseshaping</code></p> <ul style="list-style-type: none"> <li>• <code>ast,beta</code> (default string)</li> <li>• <code>nsym,beta</code></li> <li>• <code>n,beta</code></li> </ul>
Square root raised cosine	<p>All pulse-shaping filter specification strings must be preceded by an integer-valued <code>SamplesPerSymbol</code>. See <code>fdesign.pulseshaping</code></p> <ul style="list-style-type: none"> <li>• <code>ast,beta</code> (default string)</li> <li>• <code>nsym,beta</code></li> <li>• <code>n,beta</code></li> </ul>

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
- **beta** — Rolloff factor for pulse shaping filters (raised cosine and square root raised cosine) expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.
- **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.
- `nsym`— Pulse-shaping filter order in symbols. The length of the impulse response is `nsym*SamplesPerSymbol+1`. The product `nsym*SamplesPerSymbol` must be even.
- `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'); %design with Kaiser window
```

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`

# fftcoeffs

---

**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
```

# fftcoeffs

---

-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” on page 2-663
- “Adaptive Filter Syntaxes” on page 2-664
- “Multirate Filter Syntaxes” on page 2-665

**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`.

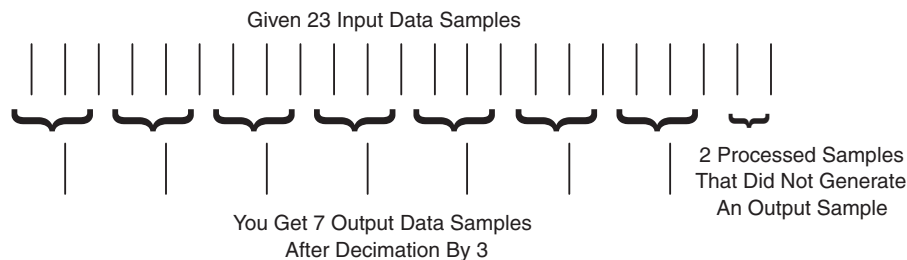
When you are filtering multichannel data, `dim` lets you specify which dimension of the input matrix to filter along — whether a row represents a channel or a column represents a channel. When you provide the `dim` input argument, the filter operates along the dimension specified by `dim`. When your 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.

# filter



## 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.

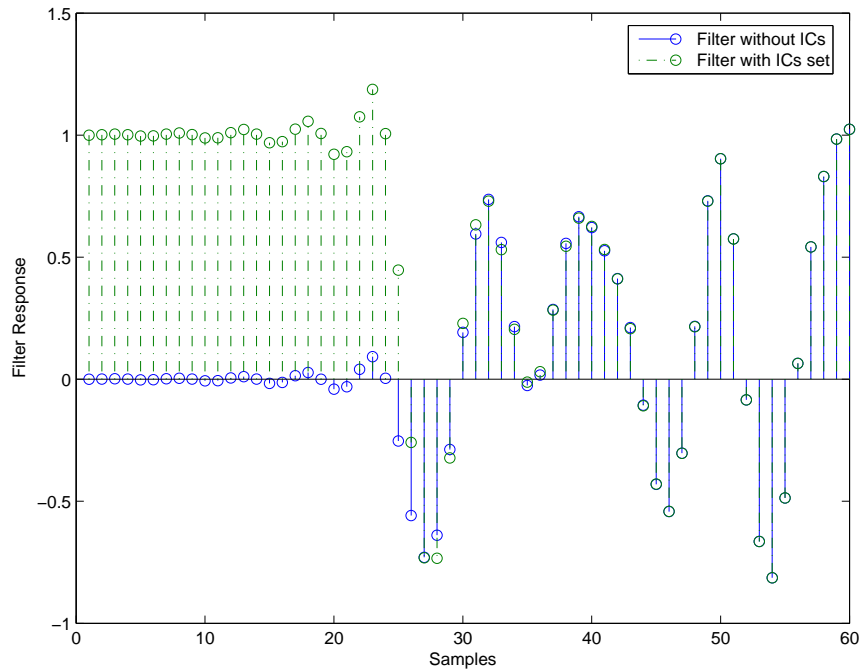
% Do not set specific initial conditions.

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.

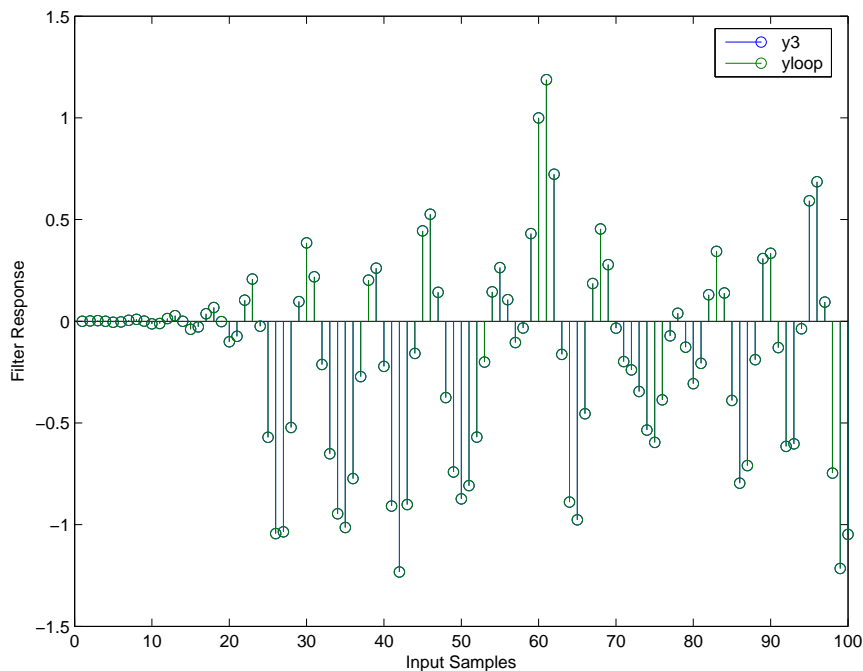
```
reset(hd);           % Clear filter history.
y3 = filter(hd,x);   % Filter entire signal in one block.
```

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 = mfilter.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. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.



**Purpose** GUI-based filter design

**Syntax** `filterbuilder(h)`  
`filterbuilder('response')`

**Description** `filterbuilder` starts a GUI-based tool for building filters. It relies on the `fdesign` object-object oriented filter design paradigm, and is intended to reduce development time during the filter design process. `filterbuilder` uses a specification-centered approach to find the best algorithm for the desired response.

---

**Note** You must have the Signal Processing Toolbox installed to use `fdesign` and `filterbuilder`. Some of the features described below may be unavailable if your installation does not additionally include the Filter Design Toolbox. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

---

The `filterbuilder` GUI contains many features not available in `FDATool`. For more information on how to use `filterbuilder`, see “Designing a Filter in the Filterbuilder GUI”.

To use `filterbuilder`, enter `filterbuilder` at the MATLAB command line using one of three approaches:

- Simply enter `filterbuilder`. MATLAB opens a dialog for you to select a filter response type. After you select a filter response type, `filterbuilder` launches the appropriate filter design dialog box.
- Enter `filterbuilder(h)`, where `h` is an existing filter object. For example, if `h` is a bandpass filter, `filterbuilder(h)` opens the bandpass filter design dialog box. (The `h` object must have been created using `filterbuilder` or must be a `dfilt` or `mfilt` object.)
- Enter `filterbuilder('response')`, replacing `response` with a response string from the following table. MATLAB opens a filter design dialog that corresponds to the response string.

# filterbuilder

---

<b>Response String</b>	<b>Description of Resulting Filter Design</b>
arbmag	Arbitrary response filter (magnitude and phase)
arbmagnphase	Arbitrary response filter (magnitude and phase)
bandpass or bp	Bandpass filter
bandstop or bs	Bandstop filter
cic	CIC filter
ciccomp	CIC compensator
comb	Comb filter
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
parameq	Parametric equalizer filter
peak	Peak filter
pulseshaping	Pulse-shaping filter

---

**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.

**Lowpass Design**

Lowpass Design  
Design a lowpass filter.

Save variable as:

**Main** | Data Types | Code Generation

Filter specifications

Impulse response:    
Order mode:   Order:   
Filter type:

Frequency specifications

Frequency units:   Input Fs:   
Fpass:  Fstop:

Magnitude specifications

Magnitude units:    
Apass:  Astop:

Algorithm

Design method:    
Structure:

▼ Design options

Density factor:   
 Minimum phase  
Minimum order:    
Stopband shape:    
Stopband decay:

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 (`fvttool`) from Signal Processing Toolbox software. For more information about FVTool, refer to Signal Processing Toolbox documentation.

---

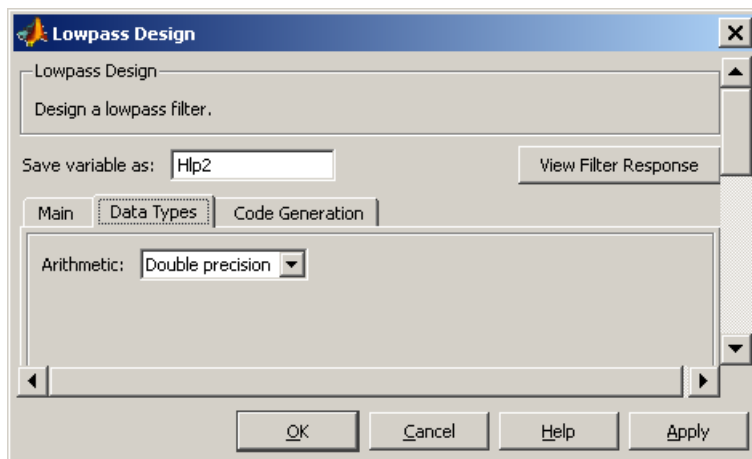
**Note** The `filterbuilder` dialog box includes an **Apply** option. Each time you click **Apply**, `filterbuilder` writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes without overwriting the variable in your 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.

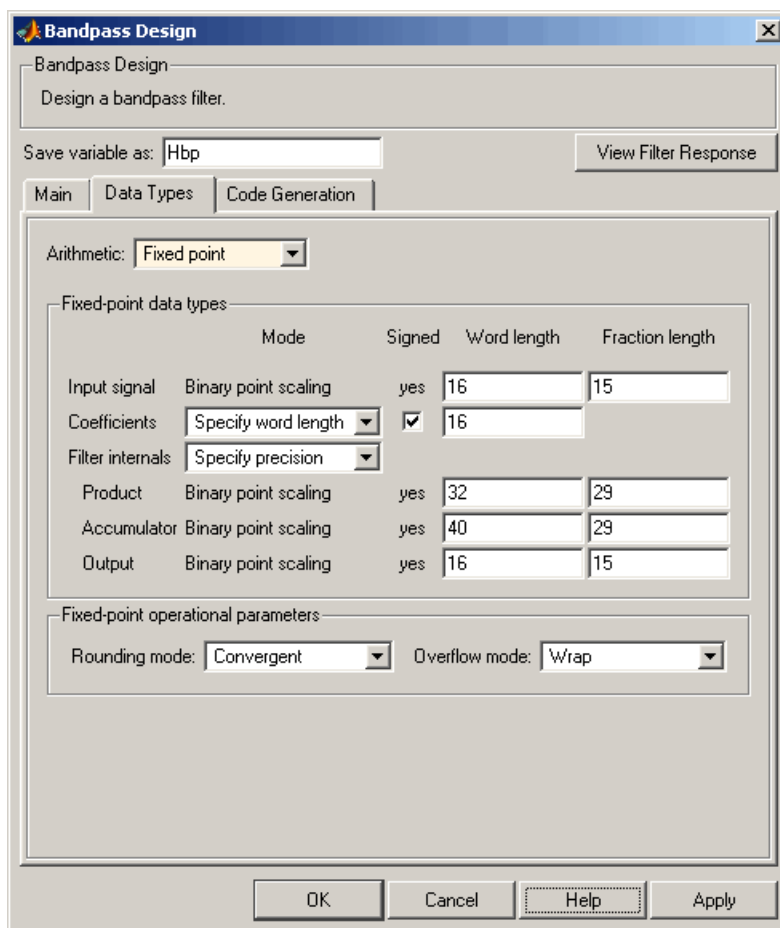


The **Arithmetic** drop down box allows the choice of **Double precision**, **Single precision**, or **Fixed point**. Some of these options may be unavailable depending on the filter parameters. The following table describes these options.

<b>Arithmetic List Entry</b>	<b>Effect on the Filter</b>
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.
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.

<b>Arithmetic List Entry</b>	<b>Effect on the Filter</b>
	This setting allows signed fixed data types only. Fixed-point filter design with <code>filterbuilder</code> is available only when you install Fixed-Point Toolbox software 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).

- **ceil** - Round toward positive infinity.
- **convergent** - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.
- **zero/fix** - Round toward zero.
- **floor** - Round toward negative infinity.
- **nearest** - Round toward nearest. Ties round toward positive infinity.
- **round** - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

### **Overflow mode**

Sets the mode the filter uses to respond to overflow conditions in fixed-point arithmetic. Choose from the following options:

- **Saturate** — Limit the output to the largest positive or negative representable value.

- **Wrap** — Set overflowing values to the nearest representable value using modular arithmetic.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

### **Cast before sum**

Specifies whether to cast numeric data to the appropriate accumulator format before performing sum operations. Selecting **Cast before sum** ensures that the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

If you clear **Cast before sum**, the filter prevents the addends from being cast to the sum format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use. The input format referenced by **Cast before sum** depends on the filter structure you are using.

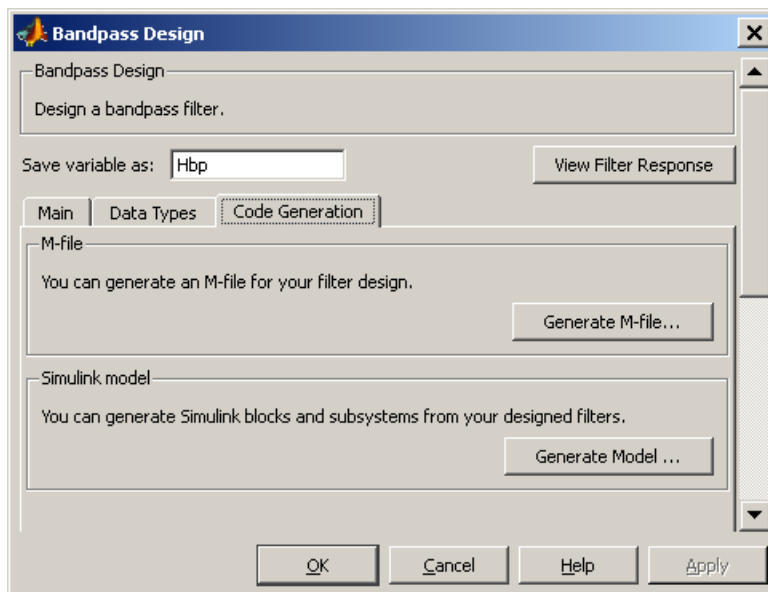
The effect of clearing or selecting **Cast before sum** is as follows:

- **Cleared** — Configures filter summation operations to retain the addends in the format carried from the previous operation.
- **Selected** — Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually, selecting **Cast before sum** generates results from the summation that more closely match those found from digital signal processors.

### **Code Generation Pane**

The code generation pane contains options for various implementations of the completed filter design. Depending on your installation, you can

generate MATLAB, VHDL, and Verilog code from the designed filter. You can also choose to create or update a Simulink model from the designed filter. The following section explains these options.



## HDL

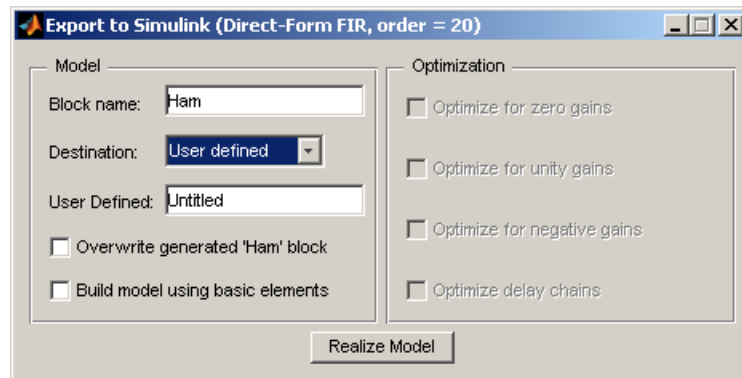
For more information on this option, see “Opening the Generate HDL Dialog Box from the filterbuilder GUI” documentation, where all the parameters on the sub dialog box are explained in detail.

## M-file

Clicking on the **Generate M-file** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable file.

## Simulink Model

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 2-688
- “Bandpass Filter Design Dialog Box — Main Pane” on page 2-692
- “Bandstop Filter Design Dialog Box — Main Pane” on page 2-700
- “CIC Filter Design Dialog Box — Main Pane” on page 2-708
- “CIC Compensator Filter Design Dialog Box — Main Pane” on page 2-711
- “Comb Filter Design Dialog Box—Main Pane” on page 2-717
- “Differentiator Filter Design Dialog Box — Main Pane” on page 2-721
- “Fractional Delay Filter Design Dialog Box — Main Pane” on page 2-728
- “Halfband Filter Design Dialog Box — Main Pane” on page 2-730
- “Highpass Filter Design Dialog Box — Main Pane” on page 2-736
- “Hilbert Filter Design Dialog Box — Main Pane” on page 2-744
- “Inverse Sinc Filter Design Dialog Box — Main Pane” on page 2-750
- “Lowpass Filter Design Dialog Box — Main Pane” on page 2-758



- “Nyquist Filter Design Dialog Box — Main Pane” on page 2-766
- “Notch” on page 2-773
- “Octave Filter Design Dialog Box — Main Pane” on page 2-774
- “Parametric Equalizer Filter Design Dialog Box — Main Pane” on page 2-776
- “Peak/Notch Filter Design Dialog Box — Main Pane” on page 2-782
- “Pulse-shaping Filter Design Dialog Box—Main Pane” on page 2-786

## Arbitrary Response Design Dialog Box – Main Pane

Arbitrary Response Design
✕

Arbitrary Response Design

Design an arbitrary response filter. The constraint can be on the magnitude only, or on the magnitude and the phase.

Save variable as:  View Filter Response

Main

Data Types

Code Generation

Filter specifications

Impulse response:

Order:

Denominator order

Filter type:

---

Response specifications

Number of bands:

Specify response as:

Frequency units:  Input Fs:

---

Band properties

	Frequencies	Amplitudes
1	linspace(0, 1, 30)	[ones(1, 7) zeros(1,8) ones(1,8) zero

---

Algorithm

Design method:

Structure:

▼ Design options

Window:

OK
Cancel
Help
Apply

### 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**

- **Window** — replace the square brackets with the name of a window function or function handle. For example, “hamming” or “@hamming”. If the window function takes parameters other than the length, use a cell array. For example, {'kaiser',3.5} or {@chebwin,60}

## Bandpass Filter Design Dialog Box – Main Pane

**Bandpass Design** [X]

Bandpass Design  
Design a bandpass filter.

Save variable as:

**Main** | Data Types | Code Generation

Filter specifications

Impulse response:

Order mode:  Order:

Filter type:  Interpolation factor:

Decimation factor:

Frequency specifications

Frequency units:  Input Fs:

Fstop1:  Fpass1:

Fpass2:  Fstop2:

Magnitude specifications

Magnitude units:

Astop1:  Apass:

Astop2:

Algorithm

Design method:

Structure:

▼ Design options

Density factor:

Minimum phase

Minimum order:

### 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

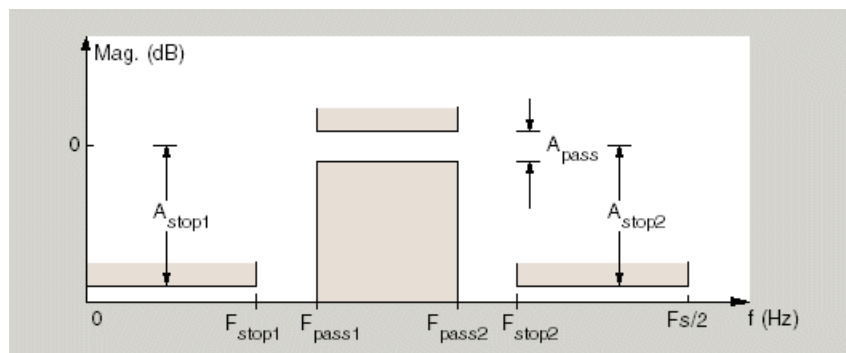
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

## 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  $F_{stop1}$  and  $F_{pass1}$  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

The screenshot shows the 'Bandstop Design' dialog box with the following settings:

- Title Bar:** Bandstop Design
- Instruction:** Design a bandstop filter.
- Save variable as:** Hbs
- Buttons:** View Filter Response
- Tabs:** Main (selected), Data Types, Code Generation
- Filter specifications:**
  - Impulse response: FIR
  - Order mode: Minimum
  - Filter type: Sample-rate converter
  - Order: [empty]
  - Interpolation factor: 2
  - Decimation factor: 3
- Frequency specifications:**
  - Frequency units: Normalized (0 to 1)
  - Input Fs: [empty]
  - Fpass1: .35
  - Fstop1: .45
  - Fstop2: .55
  - Fpass2: .65
- Magnitude specifications:**
  - Magnitude units: dB
  - Apass1: 1
  - Astop: 60
  - Apass2: 1
- Algorithm:**
  - Design method: Equiripple
  - Structure: Direct-form FIR polyphase sample-rate converter
  - Design options:
    - Density factor: 16
    - Minimum phase:
- Buttons:** OK, Cancel, Help, Apply

## 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

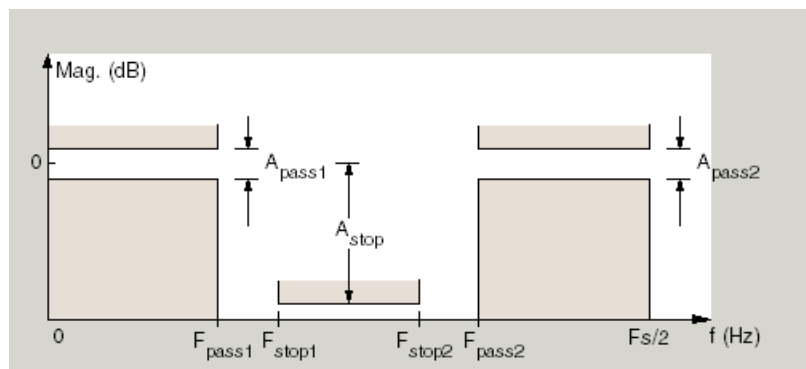
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

## 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:

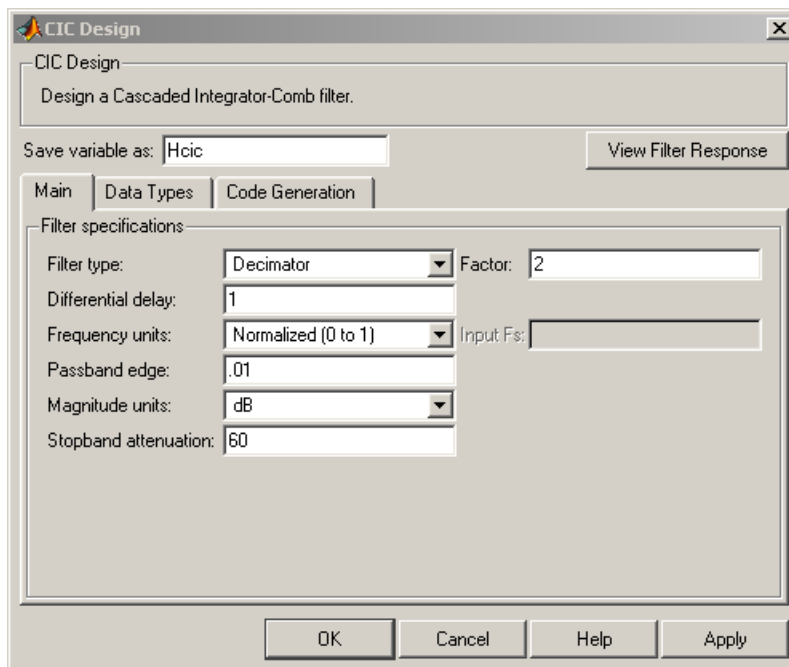
- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that 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

**CIC Compensator Design**

CIC Compensator Design  
Design a CIC compensating filter.

Save variable as:

**Main** | Data Types | Code Generation

Filter specifications

Order mode:  Order:

Filter type:

Number of CIC sections:  Differential delay:

Frequency specifications

Frequency units:  Input Fs:

Fpass:  Fstop:

Magnitude specifications

Magnitude units:

Apass:  Astop:

Algorithm

Design method:

Structure:

▼ Design options

Density factor:

Minimum phase

Minimum order:

Stopband shape:

Stopband decay:

## 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.

## Comb Filter Design Dialog Box—Main Pane

Comb Design

Design a comb filter.

Save variable as:

Main | Data Types | Code Generation

Filter specifications

Comb Type:

Order mode:  Order:

Frequency specifications

Frequency constraints:

Quality factor:

Frequency Units:  Input Fs:

Notch Frequencies:

Magnitude specifications

No magnitude constraints can be specified when specifying a filter order.

Algorithm

Design method:

Structure:

### Filter Specifications

Parameters in this group enable you to specify the type of comb filter and the number of peaks or notches.

### Comb Type

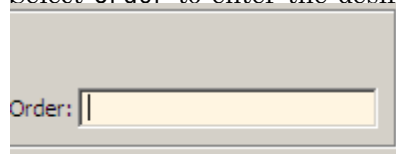
Select either Notch or Peak from the drop-down list. Notch creates a comb filter that attenuates a set of harmonically related

frequencies. **Peak** creates a comb filter that amplifies a set of harmonically related frequencies.

## Order mode

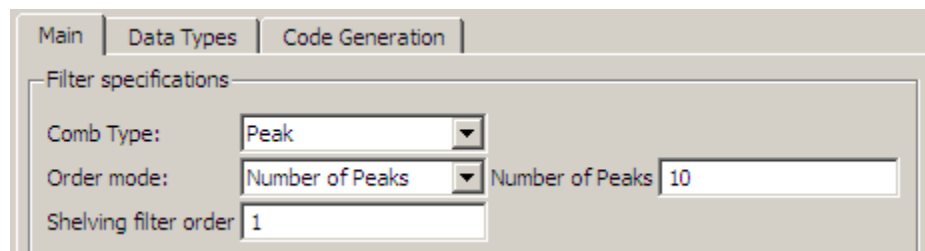
Select either **Order** or **Number of Peaks/Notches** from the drop-down menu.

Select **Order** to enter the desired filter order in the



dialog box. The comb filter has notches or peaks at increments of  $2/Order$  in normalized frequency units.

Select **Number of Peaks** or **Number of Notches** to specify the number of peaks or notches and the **Shelving filter order**



## Shelving filter order

The **Shelving filter order** is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

## Frequency specifications

Parameters in this group enable you to specify the frequency constraints and frequency units.



## Frequency specifications

Select either **Quality factor** or **Bandwidth**.

**Quality factor** is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the  $-3$  dB point.

**Bandwidth** specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the  $-3$  dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

## Frequency Units

Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input Fs** dialog box.

## Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

- dB — Specify the magnitude in decibels (default).
- Squared — Specify the magnitude in squared units.

**Bandwidth gain** — Specify the gain at which the bandwidth is measured. The default is  $-3$  dB.

## Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

## Design Method

The IIR Butterworth design is the only option for peaking or notching comb filters.

## **Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

## **Design Options**

**Differentiator Filter Design Dialog Box – Main Pane**

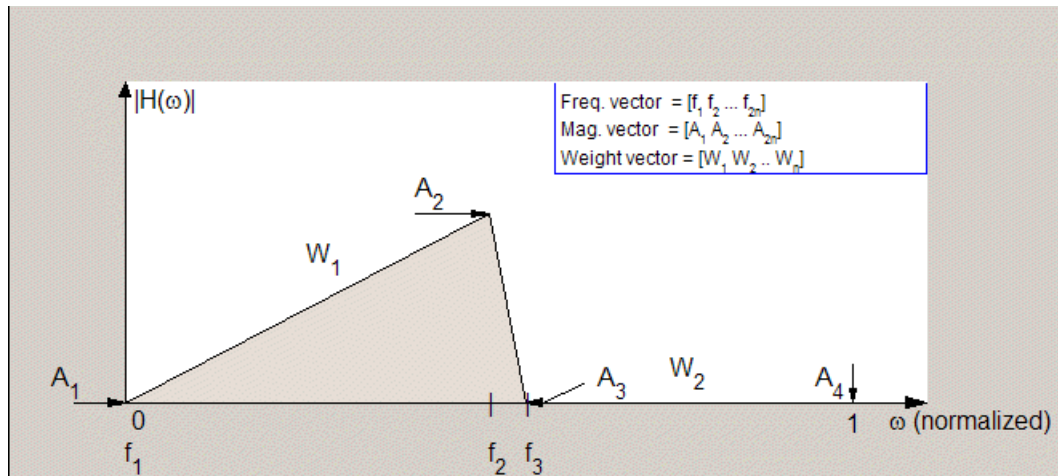
The screenshot shows the 'Differentiator Design' dialog box with the 'Main' tab selected. The dialog is titled 'Differentiator Design' and contains the following sections:

- Differentiator Design:** A text area with the instruction 'Design a differentiator.'
- Save variable as:** A text field containing 'Hdf'.
- View Filter Response:** A button.
- Filter specifications:**
  - Order mode:** A dropdown menu set to 'Minimum'.
  - Order:** An empty text field.
  - Filter type:** A dropdown menu set to 'Single-rate'.
- Frequency specifications:**
  - Frequency units:** A dropdown menu set to 'Normalized (0 to 1)'.
  - Input Fs:** A text field containing '2'.
  - Fpass:** A text field containing '0.45'.
  - Fstop:** A text field containing '0.55'.
- Magnitude specifications:**
  - Magnitude units:** A dropdown menu set to 'dB'.
  - Apass:** A text field containing '1'.
  - Astop:** A text field containing '60'.
- Algorithm:**
  - Design method:** A dropdown menu set to 'Equiripple'.
  - Structure:** A dropdown menu set to 'Direct-form FIR'.
  - Design options:** A collapsed section containing:
    - Density factor:** A text field containing '16'.

At the bottom of the dialog are four buttons: 'OK', 'Cancel', 'Help', and 'Apply'.

**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**

**F<sub>s</sub>**, 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.

**F<sub>pass</sub>**

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**.

**F<sub>stop</sub>**

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**.

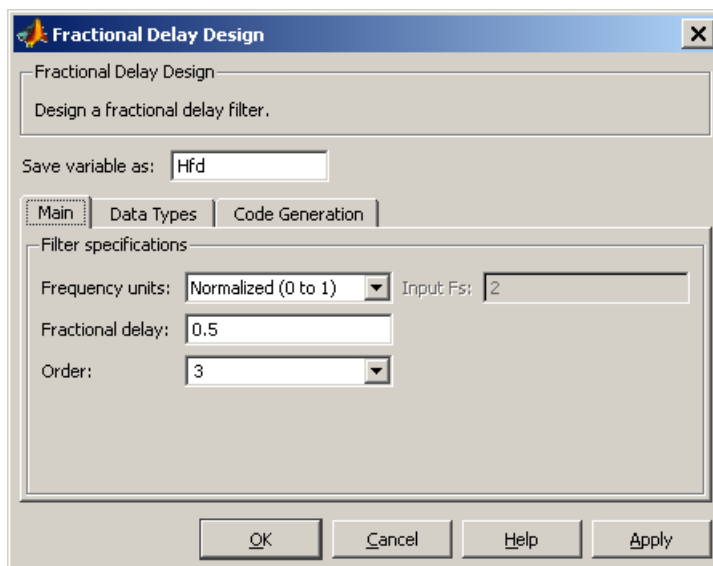
### **Stopband Decay**

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

Halfband Design

Design a halfband filter.

Save variable as:

Main | Data Types | Code Generation

Filter specifications

Impulse response:

Order mode:  Order:

Response type:

Filter type:

Frequency specifications

Frequency units:  Input Fs:

Transition width:

Magnitude specifications

Magnitude units:

Astop:

Algorithm

Design method:

Structure:

Design options

Minimum phase

Stopband shape:

Stopband decay:

### Filter Specifications

Parameters in this group enable you to specify your filter type and 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, or Interpolator. By default, filterbuilder specifies single-rate filters.

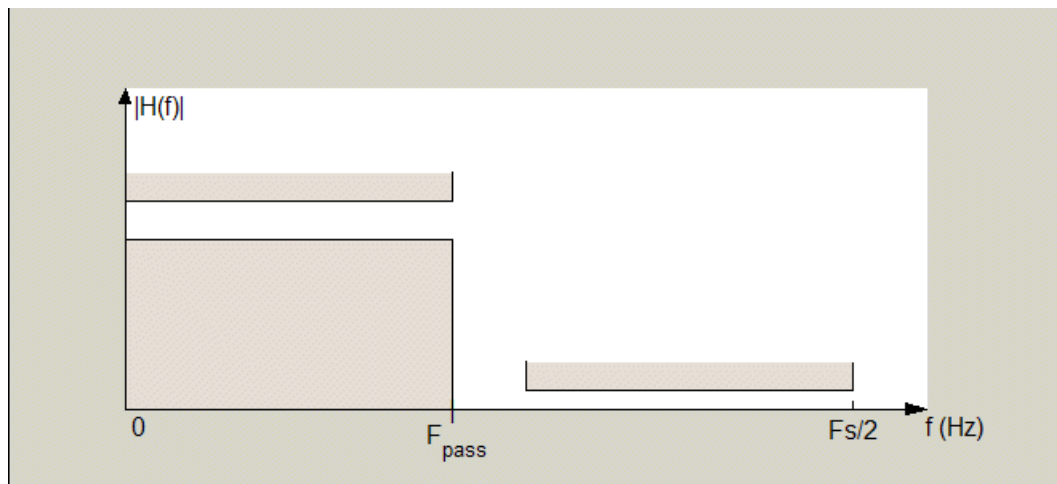
When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that decimates or interpolates your input by a factor of two.

#### Order

Enter the filter order. This option is enabled only if Specify was selected for **Filter order mode**.

### Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.



In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

### Input Fs

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**Transition width**

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

**Magnitude Specifications**

The parameters in this group let you specify the filter response in the passbands and stopbands.

**Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default).

**Astop**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. For FIR halfband filters, the available design options are `equiripple` and `kaiser`. For IIR halfband filters, the available design options are `Butterworth`, `elliptic`, and `IIR quasi-linear phase`.

**Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

## Design Options

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

### Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

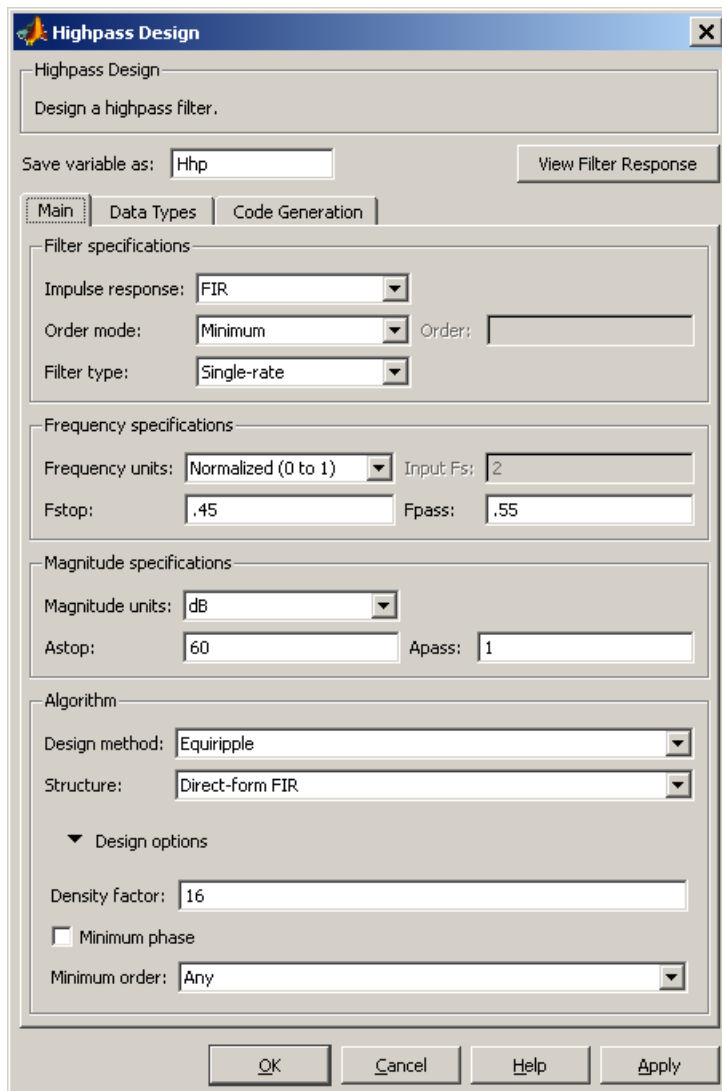
When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to 1/f, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay.



`filterbuilder` applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

## Highpass Filter Design Dialog Box – Main Pane



The image shows a software dialog box titled "Highpass Design" with a close button (X) in the top right corner. The dialog is divided into several sections:

- Highpass Design**: A header section with the text "Design a highpass filter."
- Save variable as:** A text input field containing "Hhp" and a "View Filter Response" button to its right.
- Navigation tabs:** Three tabs labeled "Main", "Data Types", and "Code Generation". The "Main" tab is currently selected.
- Filter specifications:**
  - Impulse response: A dropdown menu set to "FIR".
  - Order mode: A dropdown menu set to "Minimum" and an "Order:" text input field.
  - Filter type: A dropdown menu set to "Single-rate".
- Frequency specifications:**
  - Frequency units: A dropdown menu set to "Normalized (0 to 1)" and an "Input Fs:" text input field containing "2".
  - Fstop: A text input field containing ".45".
  - Fpass: A text input field containing ".55".
- Magnitude specifications:**
  - Magnitude units: A dropdown menu set to "dB".
  - Astop: A text input field containing "60".
  - Apass: A text input field containing "1".
- Algorithm:**
  - Design method: A dropdown menu set to "Equiripple".
  - Structure: A dropdown menu set to "Direct-form FIR".
  - Design options:** A collapsed section containing:
    - Density factor: A text input field containing "16".
    - Minimum phase
    - Minimum order: A dropdown menu set to "Any".

At the bottom of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

## 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

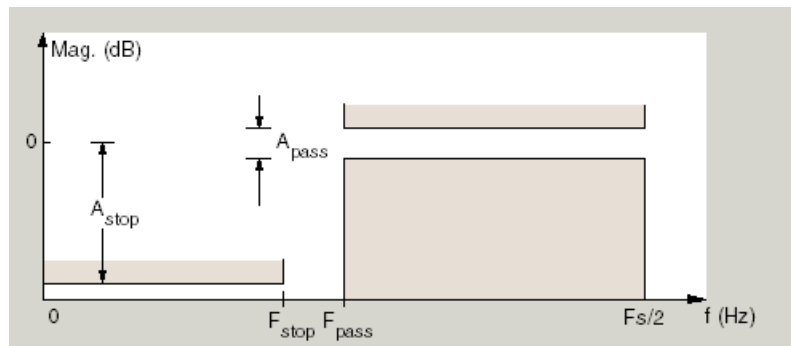
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

## 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  $F_{stop}$  and  $F_{pass}$  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

Hilbert Design

Hilbert Design  
Design a Hilbert filter.

Save variable as:

**Main** | Data Types | Code Generation

Filter specifications

Impulse response:

Order mode:  Order:

Filter type:

Frequency specifications

Frequency units:  Input Fs:

Transition width:

Magnitude specifications

Magnitude units:

Apass:

Algorithm

Design method:

Structure:

▼ Design options

Density factor:

FIR type:

### 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

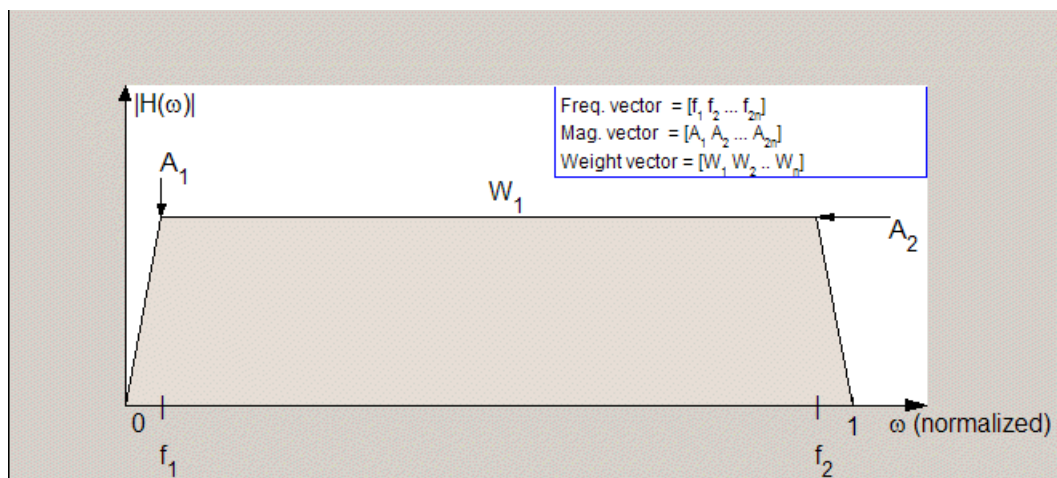
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

## 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

**Inverse Sinc Lowpass Design**

Inverse Sinc Lowpass Design  
Design an inverse-sinc lowpass filter.

Save variable as:

**Main** | Data Types | Code Generation

Filter specifications

Order mode:  Order:

Filter type:

Frequency specifications

Frequency units:  Input Fs:

Fpass:  Fstop:

Magnitude specifications

Magnitude units:

Apass:  Astop:

Algorithm

Design method:

Structure:

▼ Design options

Density factor:

Minimum phase

Minimum order:

Stopband shape:

Stopband decay:



### 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

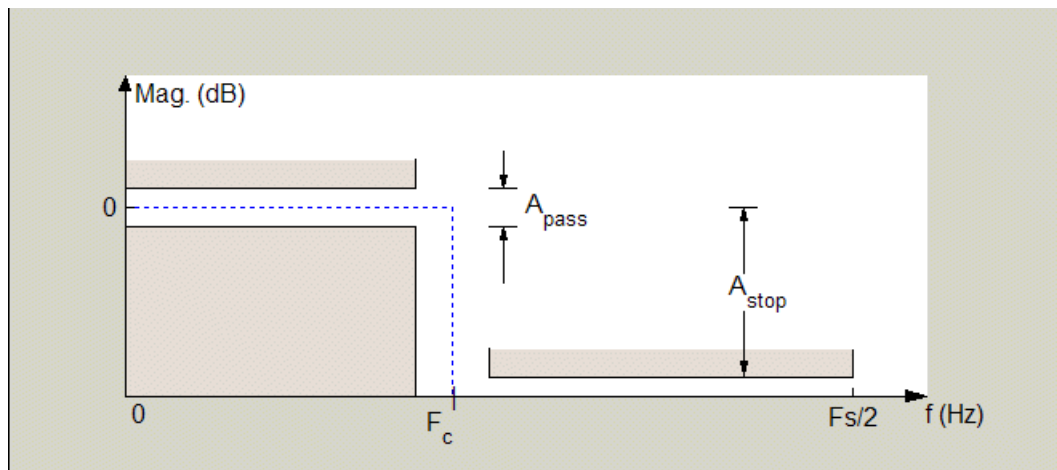
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

#### 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  $F_{pass}$  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 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

Lowpass Design

Design a lowpass filter.

Save variable as:

Main | Data Types | Code Generation

Filter specifications

Impulse response:

Order mode:  Order:

Filter type:  Interpolation factor:

Decimation factor:

Frequency specifications

Frequency units:  Input Fs:

Fpass:  Fstop:

Magnitude specifications

Magnitude units:

Apass:  Astop:

Algorithm

Design method:

Structure:

▼ Design options

Density factor:

Minimum phase

Minimum order:

Stopband shape:

Stopband decay:



### 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

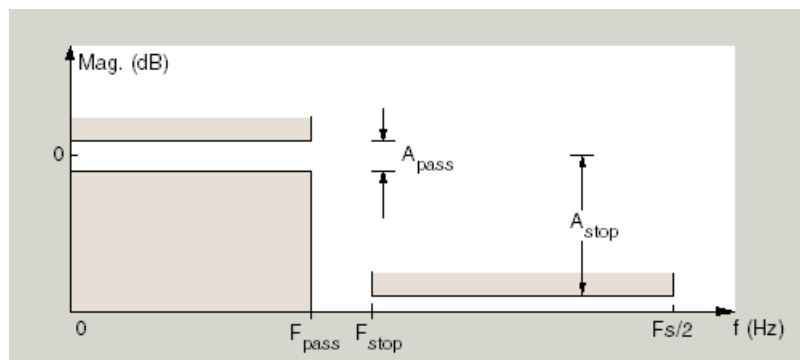
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

## 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  $F_{\text{pass}}$  and  $F_{\text{stop}}$  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.

## Nyquist Filter Design Dialog Box – Main Pane

Nyquist Design

Nyquist Design  
Design a Nyquist filter.

Save variable as:

Main | Data Types | Code Generation

Filter specifications

Band:

Impulse response:

Filter order mode:

Filter type:

Frequency specifications

Frequency units:  Input Fs:

Transition width:

Magnitude specifications

Magnitude units:

Astop:

Algorithm

Design method:

Structure:

### 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

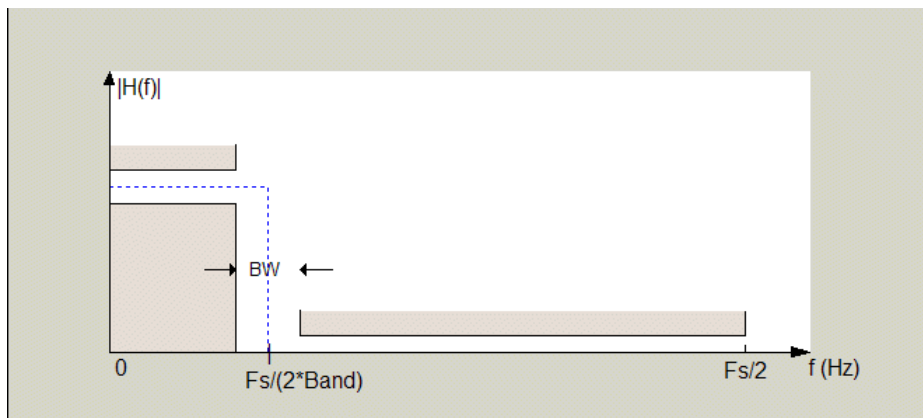
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

## 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.

## Notch

See “Peak/Notch Filter Design Dialog Box — Main Pane” on page 2-782.

## Octave Filter Design Dialog Box – Main Pane

Octave Design

Design an octave filter.

Save variable as:

Main | Data Types | Code Generation

Filter specifications

Order:

Bands per octave:

Frequency units:  Input Fs:

Center frequency:

Algorithm

Design method:

Structure:

Scale SOS filter coefficients to reduce chance of overflow

### 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.

## Parametric Equalizer Filter Design Dialog Box – Main Pane Filter Specifications

The screenshot shows the 'Parametric Equalizer' dialog box with the 'Main' tab selected. The 'Filter specifications' section is active, showing the following settings:

- Save variable as:
- Filter specifications:
  - Order mode:  Order:
- Frequency specifications:
  - Frequency constraints:
  - Frequency units:  Input Fs:
  - Center frequency:  Bandwidth:
  - Passband width:
- Gain specifications:
  - Gain constraints:
  - Gain units:
  - Reference gain:  Center frequency gain:
  - Bandwidth gain:  Passband gain:
- Algorithm:
  - Design method:
  - Structure:
  - Scale SOS filter coefficients to reduce chance of overflow

Buttons at the bottom:

### Filter Specifications

#### Order mode

Select Minimum to design a minimum order filter that meets the design specifications, or Specify to enter a specific filter order. The order mode also affects the possible frequency constraints, which in turn limit the gain specifications. For example, if you specify a Minimum order filter, the available frequency constraints are:

- Center frequency, bandwidth, passband width
- Center frequency, bandwidth, stopband width

If you select **Specify**, the available frequency constraints are:

- Center frequency, bandwidth
- Center frequency, quality factor
- Shelf type, cutoff frequency, quality factor
- Shelf type, cutoff frequency, shelf slope parameter
- Low frequency, high frequency

## **Order**

This parameter is enabled only if the **Order mode** is set to **Specify**. Enter the filter order in this text box.

## **Frequency specifications**

Depending on the filter order, the possible frequency constraints change. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

## **Frequency constraints**

Select the specification to represent the frequency constraints. The following options are available:

- Center frequency, bandwidth, passband width (available for minimum order only)
- Center frequency, bandwidth, stopband width (available for minimum order only)
- Center frequency, bandwidth (available for a specified order only)
- Center frequency, quality factor (available for a specified order only)
- Shelf type, cutoff frequency, quality factor (available for a specified order only)
- Shelf type, cutoff frequency, shelf slope parameter (available for a specified order only)

- Low frequency, high frequency (available for a specified order only)

## Frequency units

Select the frequency units from the available drop down list (Normalized, Hz, kHz, MHz, GHz). If Normalized is selected, then the **Input Fs** box is disabled for input.

## Input Fs

Enter the input sampling frequency. This input box is disabled for input if Normalized is selected in the **Frequency units** input box.

## Center frequency

Enter the center frequency in the units specified by the value in **Frequency units**.

## Bandwidth

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to  $\text{db}(\text{sqrt}(.5))$ , or  $-3$  dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: Center frequency, bandwidth, passband width, Center frequency, bandwidth, stopband width, or Center frequency, bandwidth.

## Passband width

The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is Center frequency, bandwidth, passband width.

## Stopband width

The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency

constraint selected is Center frequency, bandwidth, stopband width.

### **Low frequency**

Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is Low frequency, high frequency. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

### **High frequency**

Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is Low frequency, high frequency. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

### **Gain Specifications**

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

### **Gain constraints**

Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- Reference, center frequency, bandwidth, passband
- Reference, center frequency, bandwidth, stopband
- Reference, center frequency, bandwidth, passband, stopband
- Reference, center frequency, bandwidth

### **Gain units**

Specify the gain units either dB or squared. These units are used for all gain specifications in the dialog box.

## Reference gain

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

## Bandwidth gain

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a bandwidth parameter, or is Low frequency, high frequency.

## Center frequency gain

Specify the center frequency in **Gain units**

## Passband gain

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

## Stopband gain

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constraints** are Center frequency, bandwidth, stopband width.

## Boost/cut gain

The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the **Shelf** type parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

## Algorithm

**Design method**

Select the design method from the drop-down list. Different 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.

## Peak/Notch Filter Design Dialog Box – Main Pane

The screenshot shows the 'Peak/Notch Design' dialog box with the 'Main' tab selected. The dialog is titled 'Peak/Notch Design' and contains the following sections:

- Peak/Notch Design:** A text area with the instruction 'Design a peak or notch filter.'
- Save variable as:** A text field containing 'Hpn' and a 'View Filter Response' button.
- Filter specifications:** A section with a 'Response' dropdown menu set to 'Notch' and an 'Order' text field set to '6'.
- Frequency specifications:** A section with a 'Frequency constraints' dropdown set to 'Center frequency and quality factor', a 'Frequency units' dropdown set to 'Normalized (0 to 1)', an 'Input Fs' text field, a 'Center frequency' text field set to '0.5', and a 'Quality factor' text field set to '2.5'.
- Magnitude specifications:** A section with a 'Magnitude constraints' dropdown set to 'Unconstrained'.
- Algorithm:** A section with a 'Design method' dropdown set to 'Butterworth', a 'Structure' dropdown set to 'Direct-form II SOS', and a checked checkbox labeled 'Scale SOS filter coefficients to reduce chance of overflow'.

At the bottom of the dialog are four buttons: 'OK', 'Cancel', 'Help', and 'Apply'.

### Filter Specifications

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

### Response

Select **Peak** or **Notch** from the drop-down box. 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 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.

## Pulse-shaping Filter Design Dialog Box—Main Pane

Pulse-shaping Design

Design a pulse-shaping filter.

Save variable as:

Main | Data Types | Code Generation

Filter specifications

Pulse shape:

Order mode:  Order:

Samples per symbol:

Frequency specifications

Rolloff factor:

Frequency units:  Input Fs:

Magnitude specifications

Magnitude units:

Astop:

Algorithm

Design method:

Structure:

### Filter Specifications

Parameters in this group enable you to specify the shape and length of the filter.

### Pulse shape

Select the shape of the impulse response from the following options:

- Raised Cosine
- Square Root Raised Cosine
- Gaussian

### Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- **Minimum**— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.
- **Specify order**—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be  $\text{Order}+1$ .
- **Specify symbols**—This option enables the user to specify the length of the impulse response in an alternative manner. If **Specify symbols** is chosen, the **Order** input box changes to the **Number of symbols** input box.

### Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be  $\text{Number of symbols} * \text{Samples per symbol} + 1$ . The product  $\text{Number of symbols} * \text{Samples per symbol}$  must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product  $\text{Number of symbols} * \text{Samples per symbol}$  must be an even number. The filter length will be  $\text{Number of symbols} * \text{Samples per symbol} + 1$ .

## Frequency specifications

Parameters in this group enable you to specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

### Rolloff factor

The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

### Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz

For a Gaussian pulse shape, the available frequency specifications are:

### Bandwidth-time product

This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (BT) must be a positive real number. Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

### Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz

## Magnitude specifications

If the **Order mode** is specified as minimum, the magnitude units may be selected from:

- dB—Specify the magnitude in decibels (default).

- **Linear**—Specify the magnitude in linear units.

### **Algorithm**

The only design method available for FIR pulse-shaping filters is the window method.

# filtstates.cic

---

## 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 =
```

# filtstates.cic

---

```
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

# firband

---

**Purpose** Constrained-band equiripple FIR filter

**Syntax**

```
b = firband(n,f,a,w,c)
b = firband(n,f,a,s)
b = firband(...,'1')
b = firband(...,'minphase')
b = firband(...,'check')
b = firband(...,{lgrid})
[b,err] = firband(...)
[b,err,res] = firband(...)
```

**Description** `firband` 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 = firband(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  $20 \cdot \log(\text{constraint})$  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.

# firband

---

`b = firband(..., 'minphase')` designs a minimum-phase FIR filter. There is also `'maxphase'`.

`b = firband(..., 'check')` produces a warning when there are potential transition-region anomalies in the filter response.

`b = firband(..., {lgrid})`, where `{lgrid}` is a scalar cell array containing an integer, controls the density of the frequency grid.

`[b,err] = firband(...)` returns the unweighted approximation error magnitudes. `err` has one element for each independent approximation error.

`[b,err,res] = firband(...)` returns a structure `res` of optional results computed by `firband`, and contains the following fields:

Structure Field	Contents
<code>res.fgrid</code>	Vector containing the frequency grid used in the filter design optimization
<code>res.des</code>	Desired response on <code>fgrid</code>
<code>res.wt</code>	Weights on <code>fgrid</code>
<code>res.h</code>	Actual frequency response on the frequency grid
<code>res.error</code>	Error at each point (desired response - actual response) on the frequency grid
<code>res.iextr</code>	Vector of indices into <code>fgrid</code> of external frequencies
<code>res.fextr</code>	Vector of extremely frequencies
<code>res.order</code>	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

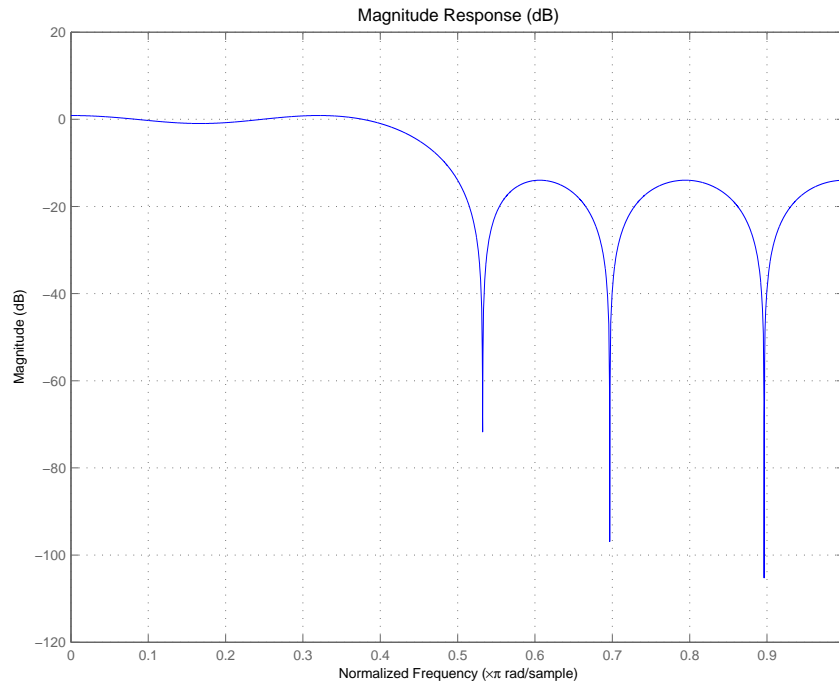
Two examples of designing filters with constrained bands.

### 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.



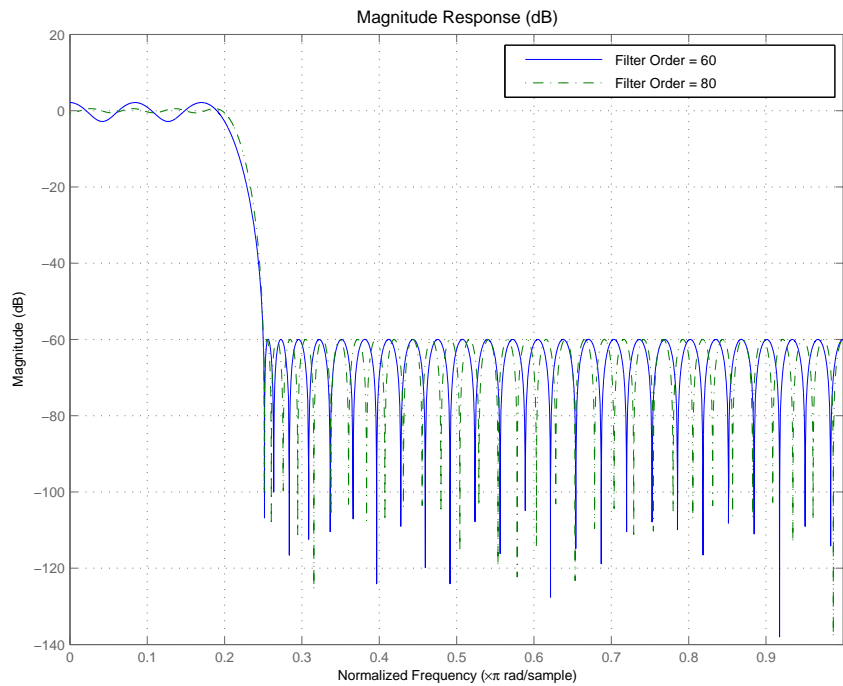
## Example 2

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=firband(60,[0 .2 .25 1],[1 1 0 0],...  
[1 .001],{'w','c'});  
b2=firband(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  $20 \cdot \log(0.001) = -60$ , 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

# fireqint

---

## Purpose

Equiripple FIR interpolators

## Syntax

```
b = fireqint(n,1,alpha)
b = fireqint(n,1,alpha,w)
b = fireqint('minorder',1,alpha,r)
b = fireqint({'minorder',initord},1,alpha,r)
```

## Description

`b = fireqint(n,1,alpha)` designs an FIR equiripple filter useful for interpolating input signals. `n` is the filter order and it must be an integer. `1`, 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 bandwidth 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 bandwidth without affecting the interpolation, resulting in better stopband attenuation for a given `1`. 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  $(\alpha \cdot \pi)$  and  $\pi$ . `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,1,alpha,w)` allows you to specify a vector of weights in `w`. The number of weights required in `w` is given by `1 + floor(1/2)`. 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',1,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  $1 + \text{floor}(1/2)$ .

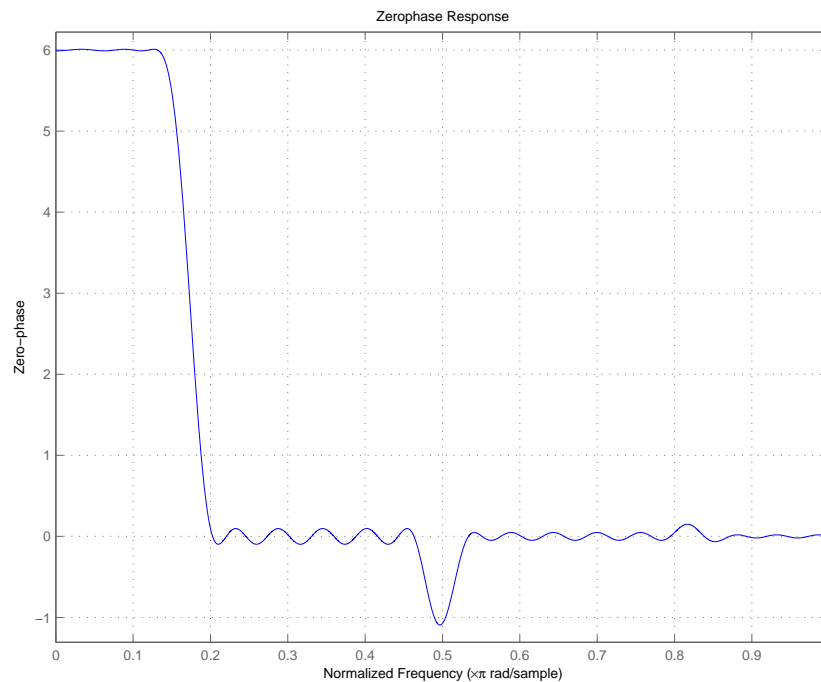
`b = fireqint({'minorder',initord},l,alpha,r)` adds the argument **initord** so you can provide an initial estimate of the filter order. Any positive integer is valid here. Again, you must provide **r**, the vector of maximum deviations or ripples, from the ideal filter magnitude response.

## 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,Fo,DEV)
hd = firceqrip(...,'slope',r)
hd = firceqrip(...,'passedge')
hd = firceqrip(...,'stopedge')
hd = firceqrip(...,'high')
hd = firceqrip(...,'min')
hd = firceqrip(...,'invsinc',C)
```

**Description**

`hd = firceqrip(n,Fo,DEV)` designs 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.

The input argument `Fo` specifies the frequency at the upper edge of the passband in normalized frequency ( $0 < Fo < 1$ ). The two-element vector `dev` specifies the peak or maximum error allowed in the passband and stopbands. Enter `[d1 d2]` for `dev` where `d1` sets the passband error and `d2` sets the stopband error.

`hd = firceqrip(...,'slope',r)` uses the input keyword 'slope' and input argument `r` to design a filter with a nonequiripple stopband. `r` is specified as a positive constant and determines the slope of the stopband attenuation in dB/normalized frequency. Greater values of `r` result in increased stopband attenuation in dB/normalized frequency.

`hd = firceqrip(...,'passedge')` designs a filter where `Fo` specifies the frequency at which the passband starts to rolloff.

`hd = firceqrip(...,'stopedge')` designs a filter where `Fo` 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 a minimum-phase filter.

`hd = firceqrip(...,'invsinc',C)` designs a lowpass filter whose magnitude response has the shape of an inverse sinc function. This may be used to compensate for sinc-like responses in the frequency domain such as the effect of the zero-order hold in a D/A converter. The amount of compensation in the passband is controlled by `C`, which is specified as a scalar or two-element vector. The elements of `C` are specified as follows:

- If `C` is supplied as a real-valued scalar or the first element of a two-element vector, `firceqrip` constructs a filter with a magnitude response of  $1/\text{sinc}(C*\pi*F)$  where  $F$  is the normalized frequency.
- If `C` is supplied as a two-element vector, the inverse-sinc shaped magnitude response is raised to the positive power `C(2)`. If we set  $P=C(2)$ , `firceqrip` constructs a filter with a magnitude response  $1/\text{sinc}(C*\pi*F)^P$ .

If this FIR filter is used with a cascaded integrator-comb (CIC) filter, setting `C(2)` equal to the number of stages compensates for the multiplicative effect of the successive sinc-like responses of the CIC filters.

---

**Note** Since the value of the inverse sinc function becomes unbounded at  $C=1/F$ , the value of `C` should be greater the reciprocal of the passband edge frequency. This can be expressed as  $F_0 < 1/C$ . For users familiar with CIC decimators, `C` is equal to 1/2 the product of the differential delay and decimation factor.

---

## Examples

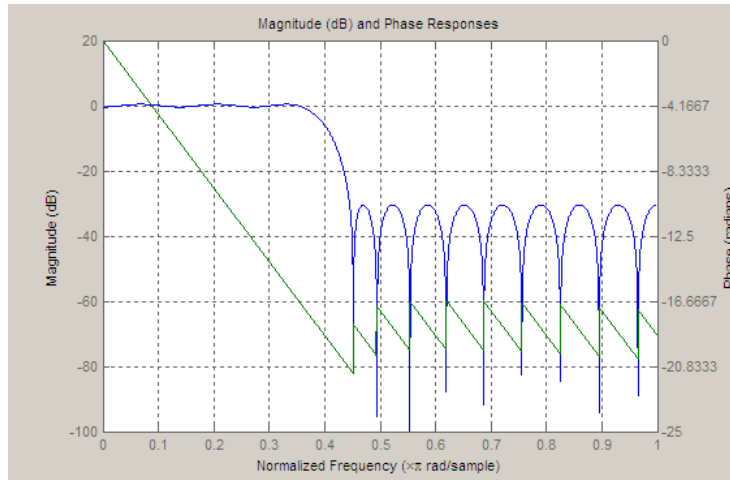
To introduce a few of the variations on FIR filters that you design with `firceqrip`, these five examples cover both the default syntax `hd = firceqrip(n,wo,del)` and some of the optional input arguments. For each example, the input arguments `n`, `wo`, and `del` remain the same.

### Example 1

Design an order = 30 FIR filter.

```
h = firceqrip(30,0.4,[0.05 0.03]); fvtool(h)
```

When the plot appears in the Filter Visualization Tool window, select **Analysis > Overlay Analysis > Phase Response**. Then select **View > Full View**. This displays the following plot.



### Example 2

Design an order = 30 FIR filter with the `stopedge` keyword to define the response at the edge of the filter stopband.

```
h = firceqrip(30,0.4,[0.05 0.03],'stopedge'); fvtool(h)
```

### Example 3

Design an order = 30 FIR filter with the `slope` keyword and `r = 20`.

```
h = firceqrip(30,0.4,[0.05 0.03],'slope',20,'stopedge'); fvtool(h)
```

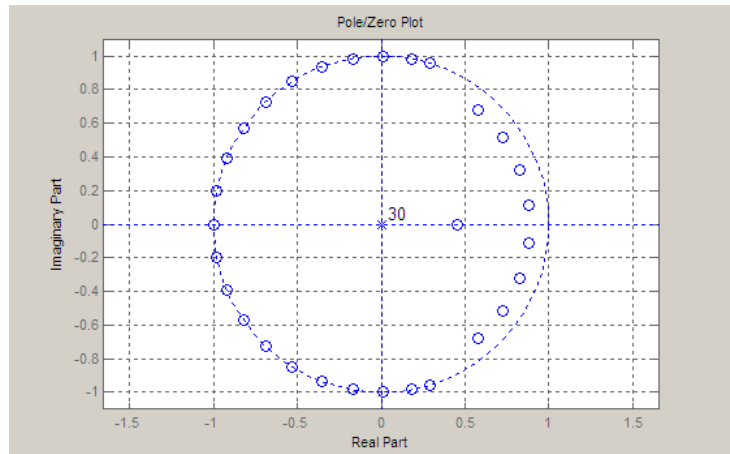
### Example 4

Design an order = 30 FIR filter defining the stopband and specifying that the resulting filter is minimum phase with the `min` keyword.

```
h = firceqrip(30,0.4,[0.05 0.03],'stopedge','min'); fvtool(h)
```

Comparing this filter to the filter in Example 1, the cutoff frequency  $\omega_0 = 0.4$  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,  $z = 1$ .



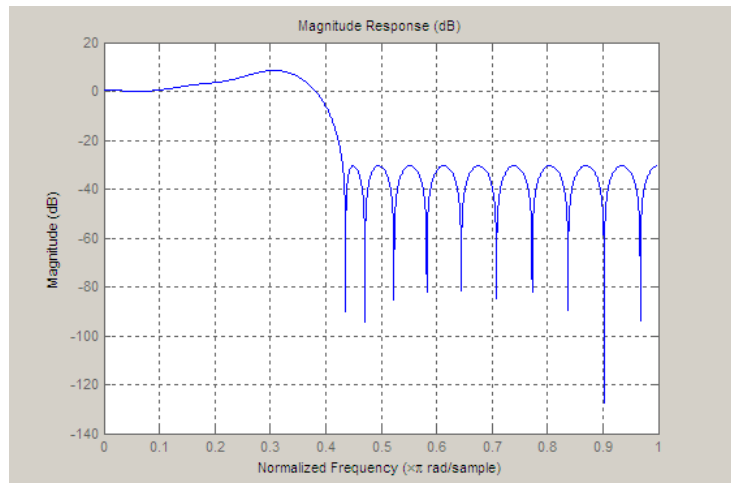
## Example 5

Design an order = 30 FIR filter with the `invsinc` keyword to shape the filter passband with an inverse sinc function.

```
h = firceqrip(30,0.4,[0.05 0.03],'invsinc',[2 1.5]); fvtool(h)
```

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**

`fdesign.decimator`, `firhalfband`, `firnyquist`, `firgr`, `ifir`,  
`iirgrpdelay`, `iirlpnorm`, `iirlpnormc`  
`fircls`, `firls`, `firpm` in Signal Processing Toolbox documentation

# fircls

---

**Purpose** FIR Constrained Least Squares filter

**Syntax**

```
hd = design(d, 'fircls')
hd = design(d, 'fircls', 'FilterStructure', value)
hd = design(d, 'fircls', 'PassbandOffset', value)
hd = design(d, 'fircls', 'zerophase', value)
```

**Description** `hd = design(d, 'fircls')` designs a FIR Constrained Least Squares (CLS) filter, `hd`, from a filter specifications object, `d`.

`hd = design(d, 'fircls', 'FilterStructure', value)` where `value` is one of the following filter structures:

- 'dffir', a discrete-time, direct-form FIR filter (the default value)
- 'dffirt', a discrete-time, direct-form FIR transposed filter
- 'dfsymfir', a discrete-time, direct-form symmetric FIR filter
- 'fftfir', a discrete-time, overlap-add, FIR filter

`hd = design(d, 'fircls', 'PassbandOffset', value)` where `value` sets the passband band gain in dB. The `PassbandOffset` and `Ap` values affect the upper and lower approximation bound in the passband as follows:

- Lower bound =  $(\text{PassbandOffset} - A_p / 2)$
- Upper bound =  $(\text{PassbandOffset} + A_p / 2)$

For bandstop filters, the `PassbandOffset` is a vector of length two that specifies the first and second passband gains. The `PassbandOffset` value defaults to 0 for lowpass, highpass and bandpass filters. The `PassbandOffset` value defaults to [0 0] for bandstop filters.

`hd = design(d, 'fircls', 'zerophase', value)` where `value` is either 'true' ('1') or 'false' ('0'). If `zerophase` is true, the lower approximation bound in the stopband is forced to zero (i.e., the filter has a zero phase response). `zerophase` is false (0) by default.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d, 'fircls')
```

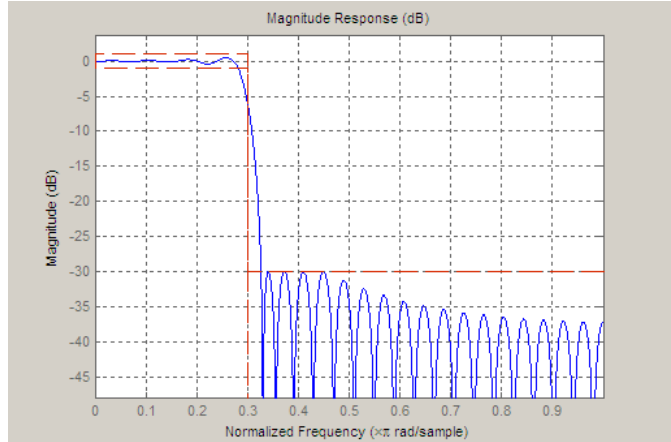
For complete help about using `fircls`, refer to the command line help system. For example, to get specific information about using `fircls` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d, 'fircls')
```

## Examples

The following example designs a constrained least-squares FIR lowpass filter.

```
h = fdesign.lowpass('n,fc,ap,ast', 50, 0.3, 2, 30);
Hd = design(h, 'fircls');
fvtool(Hd)
```

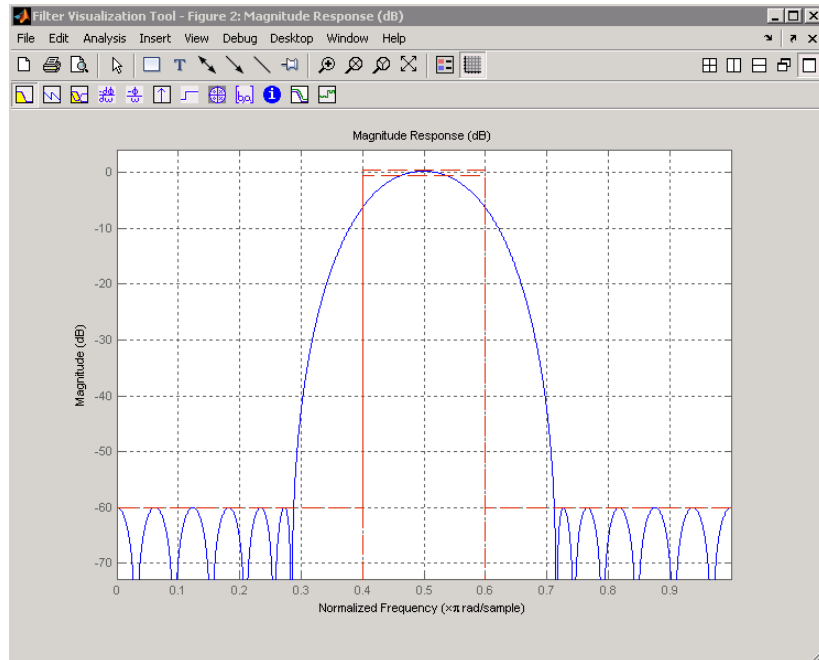


The following example constructs a constrained least-squares FIR bandpass filter.

```
d = fdesign.bandpass('N,Fc1,Fc2,Ast1,Ap,Ast2', ...
```

# fircls

```
30,0.4,0.6,60,1,60);  
Hd = design(d, 'fircls');  
fvtool(Hd)
```



**See Also** [cheby1](#), [cheby2](#), [ellip](#)

**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  $\{ 'e1', 'e1', 'e1', \dots \}$ , it is equivalent to omitting  $e$ , as in  $b = \text{firgr}(n, f, a, w)$ .

$b = \text{firgr}(n, f, a, s)$  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'})
```

$b = \text{firgr}(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],\dots\{ 'n' 'i' 'f' 'n' 'n' 'n' \})$  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.

$b = \text{firgr}(n, f, a, s, w, e)$  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
<code>res.fgrid</code>	Vector containing the frequency grid used in the filter design optimization
<code>res.des</code>	Desired response on <code>fgrid</code>
<code>res.wt</code>	Weights on <code>fgrid</code>
<code>res.h</code>	Actual frequency response on the frequency grid
<code>res.error</code>	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  $p1, p2, \dots, pn$  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 be about 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

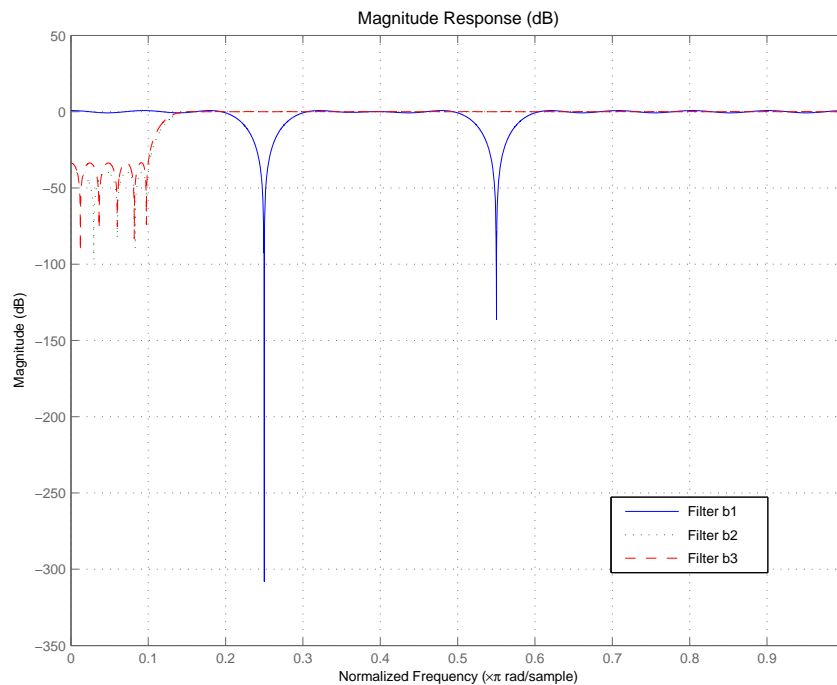
design a second highpass filter with forced values and independent approximation errors.

```
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, fir1s, 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.

**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,flip1r(b))` is a halfband filter). This can be useful for designing perfect reconstruction, two-channel FIR filter

# firhalfband

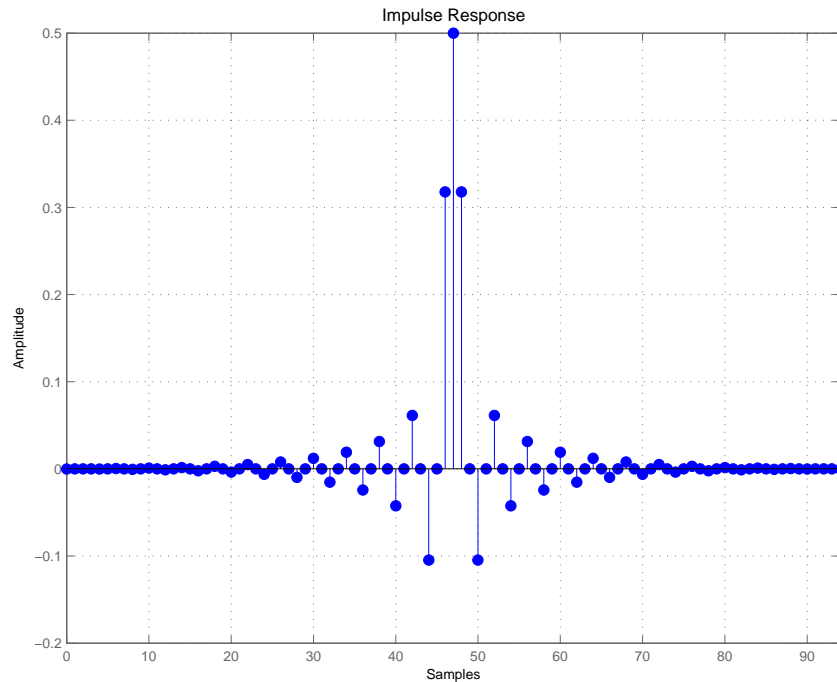
banks. The **minphase** option for `firhalfband` is not available for the window-based halfband filter designs — `b = firhalfband(n,win)` and `b = firhalfband('minorder',fp,dev,'kaiser')`.

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, fir1s, 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.

# firlp2lp

---

**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  $H_r(w)$  to a Type I lowpass FIR filter `g` with zero-phase response  $[1 - H_r(\pi-w)]$ .

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.



**Purpose**

Convert FIR lowpass filter to Type I FIR highpass filter

**Syntax**

```
g = fir1p2hp(b)
g = fir1p2hp(b, 'narrow')
g = fir1p2hp(b, 'wide')
```

**Description**

`g = fir1p2hp(b)` transforms the lowpass FIR filter `b` into a Type I highpass FIR filter `g` with zero-phase response  $H_r(\pi-w)$ . 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 = fir1p2hp(b, 'narrow')` transforms the lowpass FIR filter `b` into a Type I narrow band highpass FIR filter `g` with zero-phase response  $H_r(\pi-w)$ . `b` can be any FIR filter, including a nonlinear-phase filter.

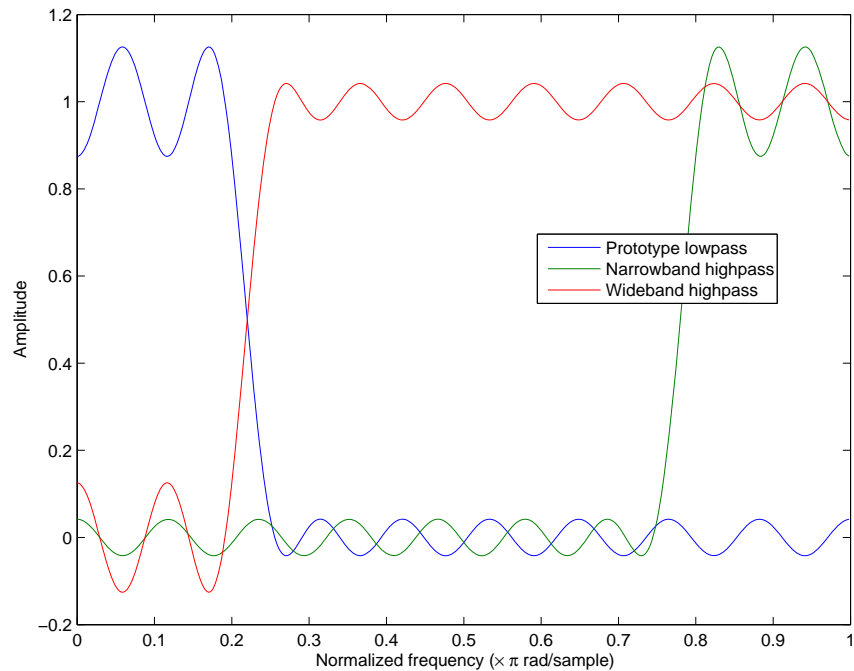
`g = fir1p2hp(b, 'wide')` transforms the Type I lowpass FIR filter `b` with zero-phase response  $H_r(w)$  into a Type I wide band highpass FIR filter `g` with zero-phase response  $1 - H_r(w)$ . Note the restriction that `b` must be a Type I linear-phase filter.

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 = fir1p2hp(b);
zerophase(h);
g = fir1p2hp(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.

**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 is 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- $p$ th 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- $p$ th 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],...
```

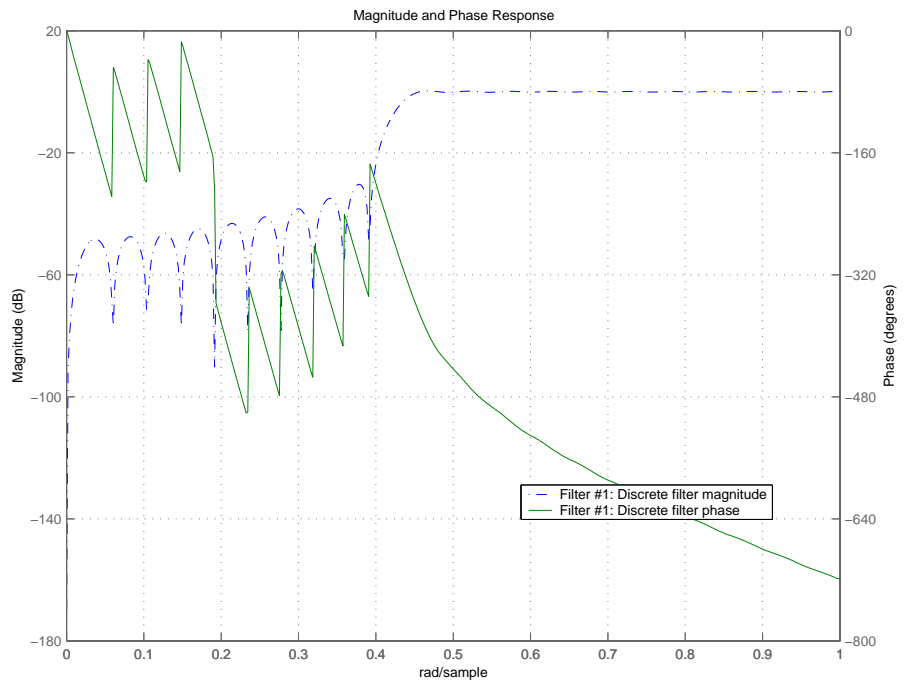
```
[1 1 1 2 2]);
fvtool(b)
```

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).

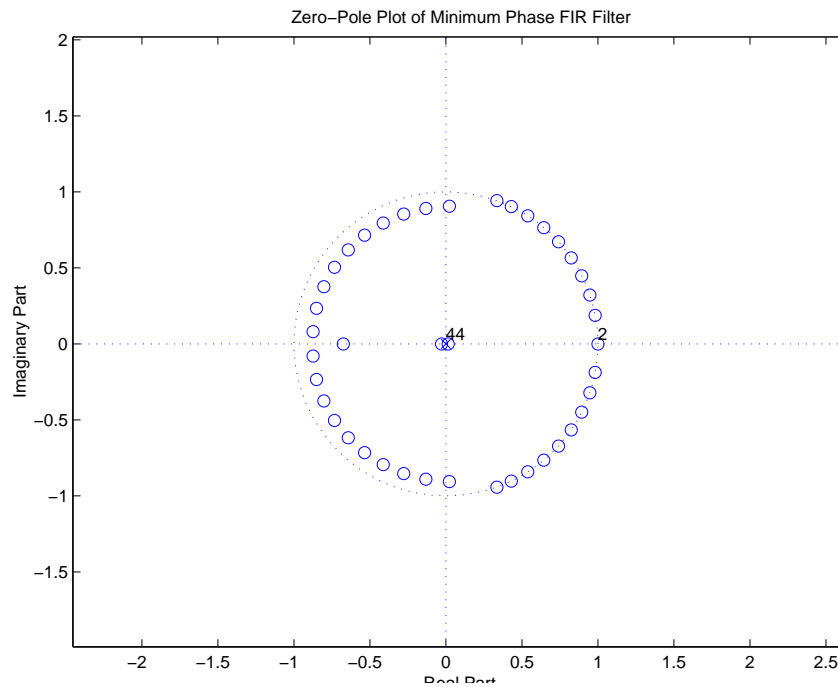
Now for the minimum-phase filter.

```
% 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**

Least square linear-phase FIR filter design

**Syntax**

```
b = firls(n,f,a)
b = firls(n,f,a,w)
b = firls(n,f,a,'ftype')
b = firls(n,f,a,w,'ftype')
```

**Description**

`firls` designs a linear-phase FIR filter that minimizes the weighted, integrated squared error between an ideal piecewise linear function and the magnitude response of the filter over a set of desired frequency bands.

`b = firls(n,f,a)` returns row vector `b` containing the  $n+1$  coefficients of the order  $n$  FIR filter whose frequency-amplitude characteristics approximately match those given by vectors `f` and `a`. The output filter coefficients, or “taps,” in `b` obey the symmetry relation.

$$b(k) = b(n+2-k), \quad k = 1, \dots, n+1$$

These are type I ( $n$  odd) and type II ( $n$  even) linear-phase filters. Vectors `f` and `a` specify the frequency-amplitude characteristics of the filter:

- `f` is a vector of pairs of frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order. Duplicate frequency points are allowed and, in fact, can be used to design a filter exactly the same as those returned by the `fir1` and `fir2` functions with a rectangular (`rectwin`) window.
- `a` is a vector containing the desired amplitude at the points specified in `f`.

The desired amplitude function at frequencies between pairs of points  $(f(k), f(k+1))$  for  $k$  odd is the line segment connecting the points  $(f(k), a(k))$  and  $(f(k+1), a(k+1))$ .

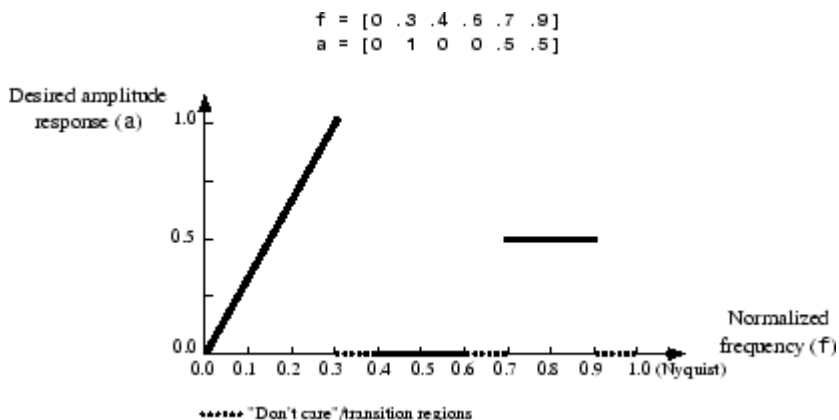
The desired amplitude function at frequencies between pairs of points  $(f(k), f(k+1))$  for  $k$  even is unspecified. These are transition or “don’t care” regions.

# firls

- `f` and `a` are the same length. This length must be an even number.

`firls` always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `firls` increments it by 1.

The figure below illustrates the relationship between the `f` and `a` vectors in defining a desired amplitude response.



`b = firls(n, f, a, w)` uses the weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

`b = firls(n, f, a, 'ftype')` and

`b = firls(n, f, a, w, 'ftype')` specify a filter type, where `'ftype'` is:

- `'hilbert'` for linear-phase filters with odd symmetry (type III and type IV). The output coefficients in `b` obey the relation

$$b(k) = -b(n+2-k), \quad k = 1, \dots, n+1$$

. This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.



- 'differentiator' for type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of  $(1/f)^2$  so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, the filters minimize the relative integrated squared error (the integral of the square of the ratio of the error to the desired amplitude).

## Examples

### Example 1

Design an order 255 lowpass filter with transition band:

```
b = firls(255,[0 0.25 0.3 1],[1 1 0 0]);
```

### Example 2

Design a 31 coefficient differentiator:

```
b = firls(30,[0 0.9],[0 0.9*pi],'differentiator');
```

An ideal differentiator has the response

$$D(\omega) = j\omega$$

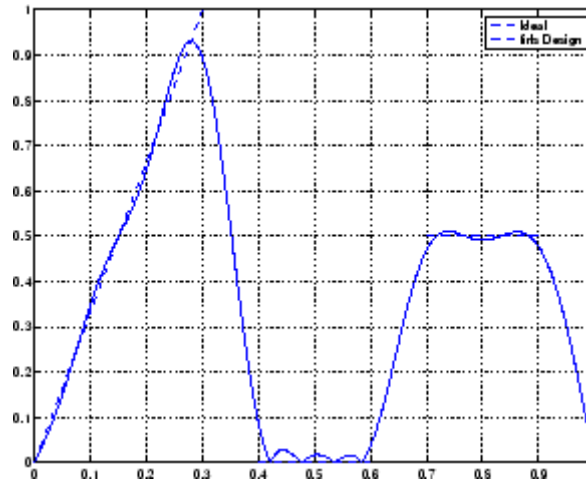
The amplitudes include a pi multiplier because the frequencies are normalized by pi.

### Example 3

Design a 24th-order anti-symmetric filter with piecewise linear passbands and plot the desired and actual frequency response:

```
F = [0 0.3 0.4 0.6 0.7 0.9];
A = [0 1 0 0 0.5 0.5];
b = firls(24,F,A,'hilbert');
for i=1:2:6,
    plot([F(i) F(i+1)],[A(i) A(i+1)],'--'), hold on
end
[H,f] = freqz(b,1,512,2);
plot(f,abs(H)), grid on, hold off
```

```
legend('Ideal','firls Design')
```



## Algorithm

Reference [1] describes the theoretical approach behind `firls`. The function solves a system of linear equations involving an inner product matrix of size roughly  $n/2$  using the MATLAB `\` operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for  $n$  even and odd respectively, while the `'hilbert'` and `'differentiator'` flags produce type III ( $n$  even) and IV ( $n$  odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response $H(f)$ , $f = 0$	Response $H(f)$ , $f = 1$ (Nyquist)
Type I	Even	$b(k) = b(n+2-k)$ , $k=1, \dots, n+1$	No restriction	No restriction
Type II	Even	$b(k) = b(n+2-k)$ , $k=1, \dots, n+1$	No restriction	$H(1) = 0$
Type III	Odd	$b(k) = -b(n+2-k)$ , $k=1, \dots, n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd	$b(k) = -b(n+2-k)$ , $k=1, \dots, n+1$	$H(0) = 0$	No restriction

## Diagnostics

One of the following diagnostic messages is displayed when an incorrect argument is used:

```
F must be even length.
F and A must be equal lengths.
Requires symmetry to be 'hilbert' or 'differentiator'.
Requires one weight per band.
Frequencies in F must be nondecreasing.
Frequencies in F must be in range [0,1].
```

A more serious warning message is

```
Warning: Matrix is close to singular or badly scaled.
```

This tends to happen when the product of the filter length and transition width grows large. In this case, the filter coefficients  $b$  might not represent the desired filter. You can check the filter by looking at its frequency response.

## See Also

`fir1`, `fir2`, `firrcos`, `firpm`

## References

[1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 54-83.

[2] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 256-266.

**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.

<b>Purpose</b>	Lowpass Nyquist (Lth-band) FIR filter
<b>Syntax</b>	<pre> 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') </pre>
<b>Description</b>	<p><code>b = firnyquist(n,l,r)</code> designs an Nth order, Lth band, Nyquist FIR filter with a rolloff factor <math>r</math> and an equiripple characteristic.</p> <p>The rolloff factor <math>r</math> is related to the normalized transition width <math>tw</math> by <math>tw = 2\pi(r/l)</math> (rad/sample). The order, <math>n</math>, must be even. <math>l</math> must be an integer greater than one. If <math>l</math> is not specified, it defaults to 4. <math>r</math> must satisfy <math>0 &lt; r &lt; 1</math>. If <math>r</math> is not specified, it defaults to 0.5.</p> <p><code>b = firnyquist('minorder',l,r,dev)</code> designs a minimum-order, Lth band Nyquist FIR filter with a rolloff factor <math>r</math> using the Kaiser window. The peak ripple is constrained by the scalar <code>dev</code>.</p> <p><code>b = firnyquist(n,l,r,decay)</code> designs an Nth order (<math>n</math>), Lth band (<math>l</math>), Nyquist FIR filter where the scalar <code>decay</code>, specifies the rate of decay in the stopband. <code>decay</code> must be nonnegative. If you omit or leave it empty, <code>decay</code> defaults to 0 which yields an equiripple stopband. A nonequiripple stopband (<code>decay</code> <math>\neq</math> 0) may be desirable for decimation purposes.</p> <p><code>b = firnyquist(n,l,r,'nonnegative')</code> 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.</p> <p><code>b = firnyquist(n,l,r,'minphase')</code> returns the minimum-phase spectral factor <code>bmin</code> of order <math>n</math>. <code>bmin</code> meets the condition <code>b=conv(bmin,bmax)</code> so that <code>b</code> is an Lth band FIR Nyquist filter of order <math>2n</math> with filter rolloff factor <math>r</math>. Obtain <code>bmax</code>, the maximum phase spectral factor by reversing the coefficients of <code>bmin</code>. For example, <code>bmax = bmin(end:-1:1)</code>.</p>

## 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  $(2*dev)$ .

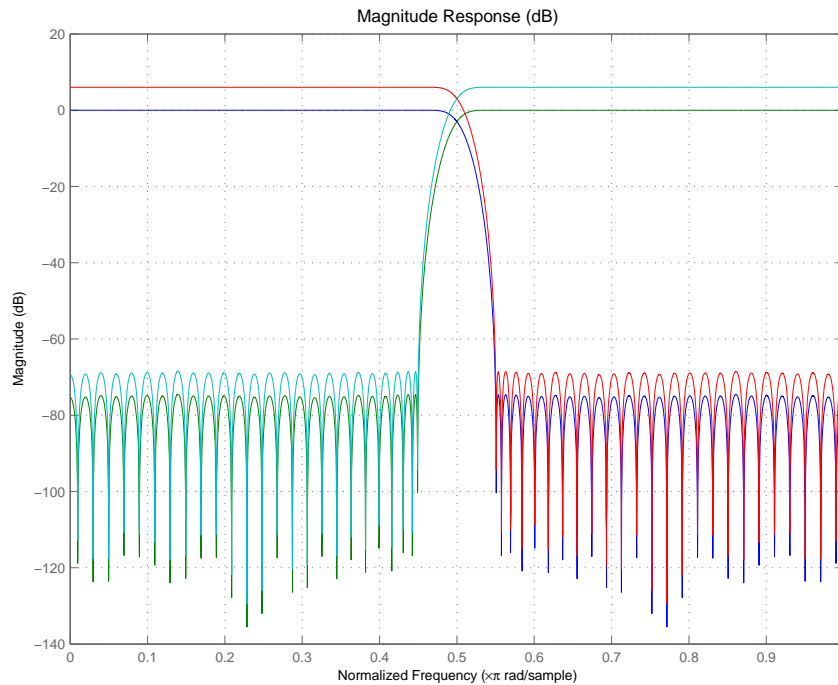
`[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)` designs the four filters such that `h0` meets the passband-edge specification `fp` and the stopband-ripple `dev` using minimum order filters to meet the specification.

**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

# firtype

---

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].

# freqrespest

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`.

### Example 1

Start by estimating the frequency response of a fixed-point FIR filter that has filter internals set to full precision.

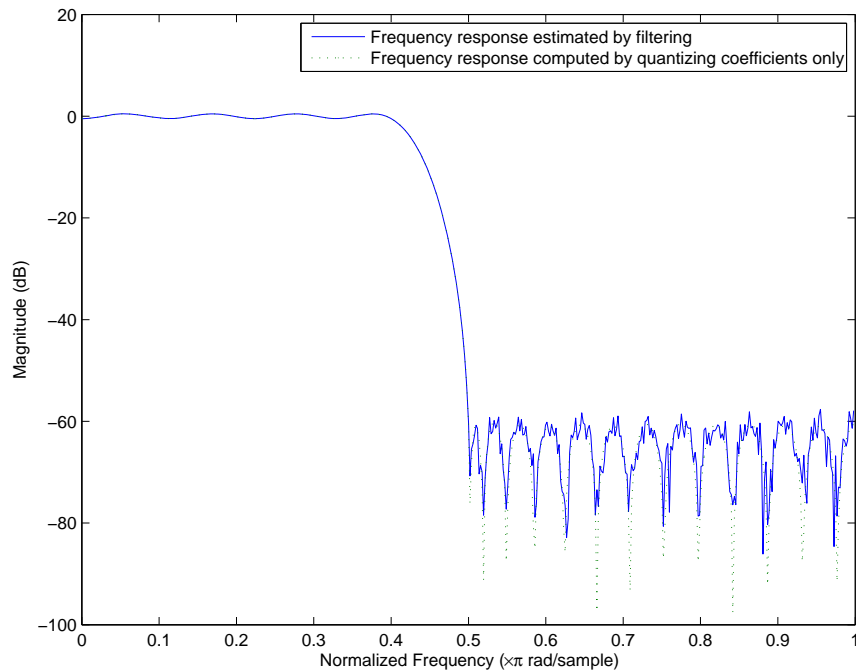
```
hd = design(fdesign.lowpass(.4,.5,1,60),'equiripple');
hd.arithmetic = 'fixed';
[h,w] = freqrespest(hd); % This should be about the same as freqz.
```

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;
```

# freqrespest

```
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)')
```



Example 2



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, noise PSD, 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
<code>dffir</code>	Direct-form FIR filter
<code>dffirt</code>	Transposed direct-form FIR filter
<code>dfsymfir</code>	Symmetrical direct-form FIR filter
<code>dfasymfir</code>	Asymmetrical direct-form FIR filter
<code>fftfir</code>	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

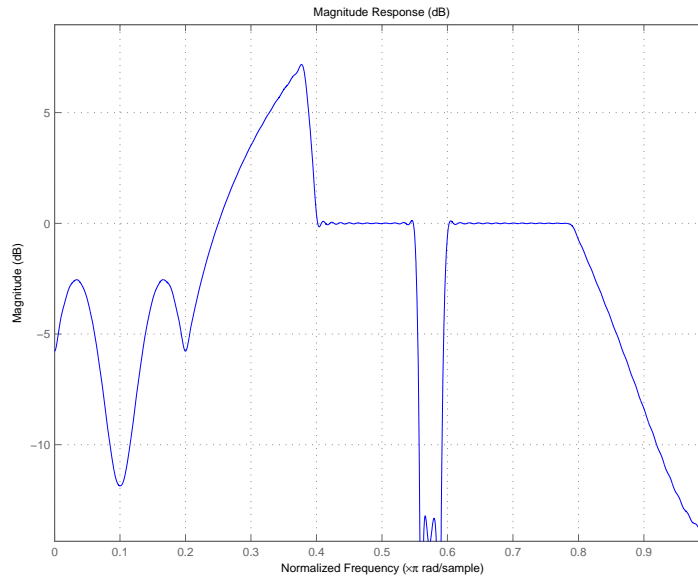
These examples design FIR filters that have arbitrary magnitude responses. In the first filter, the response has three distinct sections and the resulting filter is real.

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.

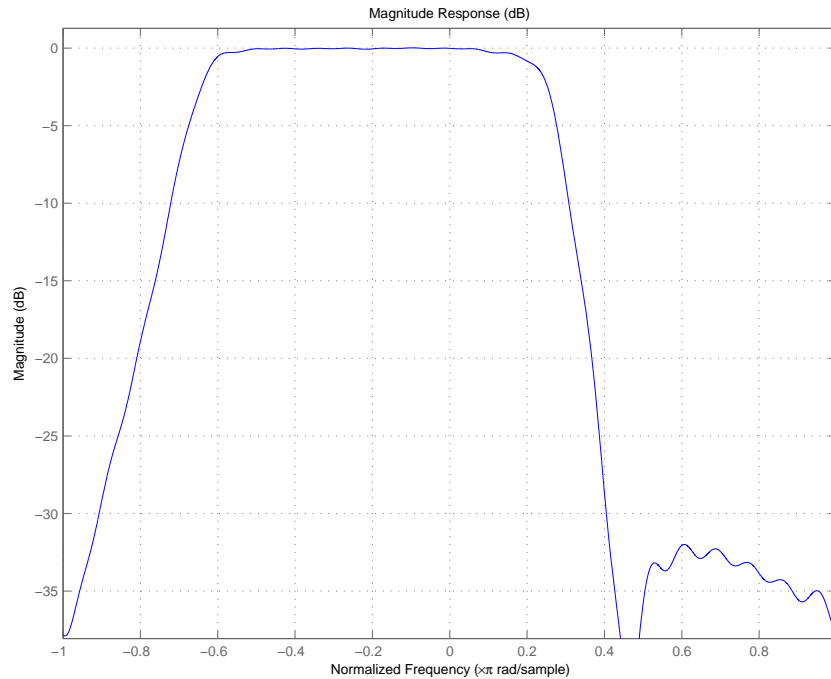
# freqsamp



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

# freqz

---

**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(hm,n)
freqz(hm)
```

**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  $\pi$  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  $\pi$  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 `hm`. When `hm` is a vector of multirate filters, `freqz` returns the matrix `h`. Each column of `h` corresponds to one filter in the vector `hm`.

`[h,w] = freqz(hm,n)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the multirate filter `hm`. `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  $\pi$  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(hm)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the adaptive filter `hm`. If `hm` 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`.

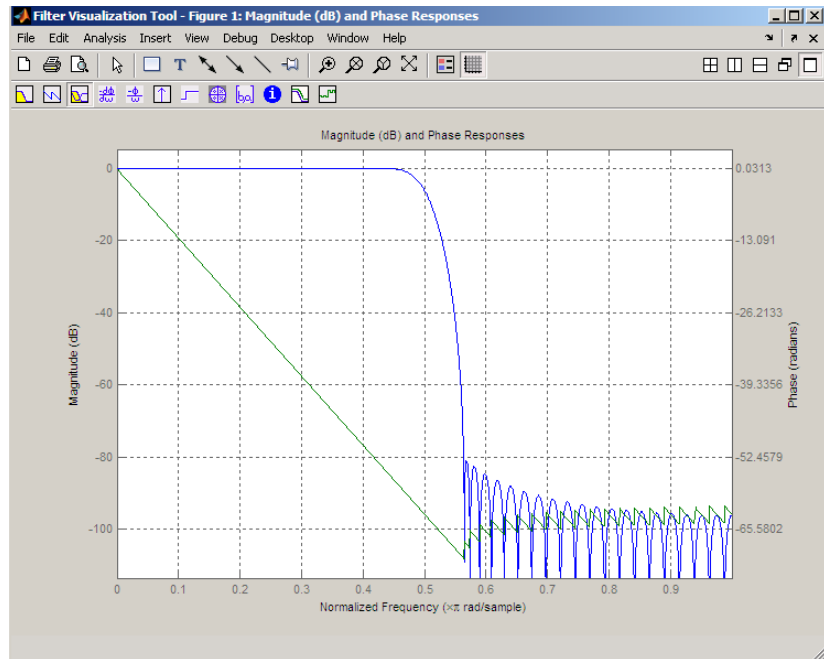
## Algorithm

`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  $\omega$  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  $\pi$  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

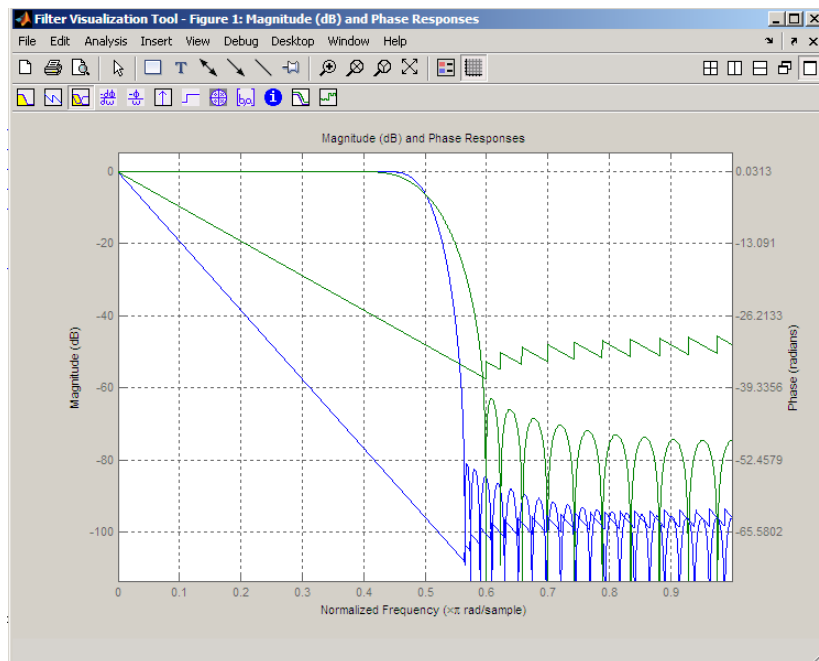
Plot the estimated frequency response of a filter. This example uses discrete-time filters, but any `adaptfilt`, `dfilt`, or `mfilt` object would work. First plot the results for one filter.

```
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 jth 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.

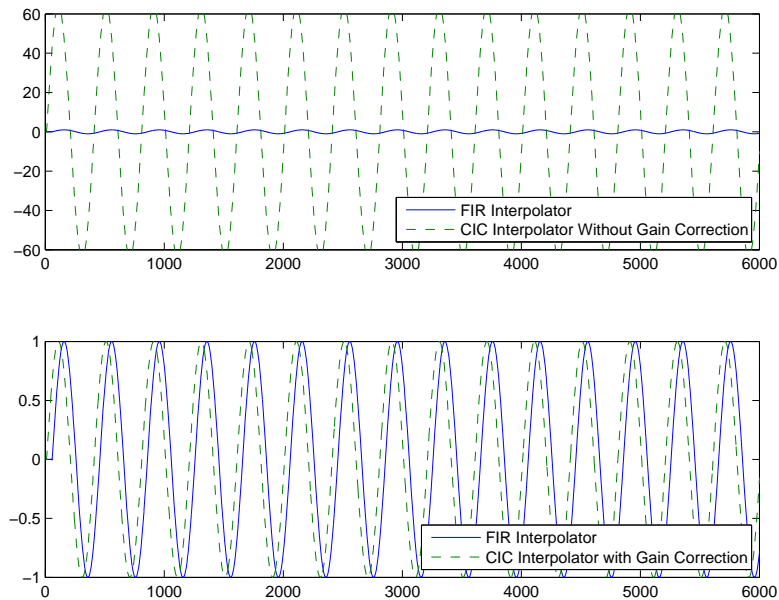
l = 4; % Interpolation factor for FIR filter.
d = fdesign.interpolator(l);
hm = design(d,'multistage');
ym = filter(hm,x);

r = 4; % Interpolation factor for the CIC filter.
```

# gain

```
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  $\pi$  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

$$-\frac{d}{dw}(\text{angle}(w))$$

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  $\pi$  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

$$-\frac{d}{dw}(\text{angle}(w))$$



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  $\pi$  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

$$-\frac{d}{dw}(\text{angle}(w))$$

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

---

`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.

---

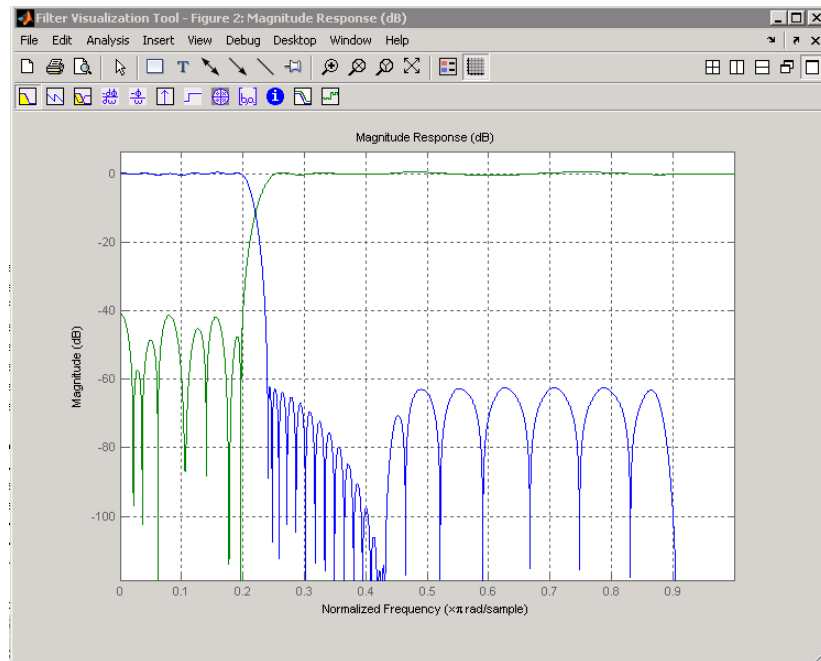
## Examples

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, fir1s, firpm in Signal Processing Toolbox documentation



<b>Purpose</b>	Transform IIR complex bandpass filter to IIR complex bandpass filter with different characteristics
<b>Syntax</b>	<code>[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)</code>
<b>Description</b>	<p><code>[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)</code> 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.</p> <p>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.</p> <p>This transformation effectively places two features of an original filter, located at frequencies <math>W_{o1}</math> and <math>W_{o2}</math>, at the required target frequency locations, <math>W_{t1}</math>, and <math>W_{t2}</math> respectively. It is assumed that <math>W_{t2}</math> is greater than <math>W_{t1}</math>. 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.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>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.</p>
<b>Examples</b>	<p>Design a prototype real IIR halfband filter using a standard elliptic approach:</p> <pre>[b, a] = ellip(3, 0.1, 30, 0.409);</pre>

# iirbpc2bpc

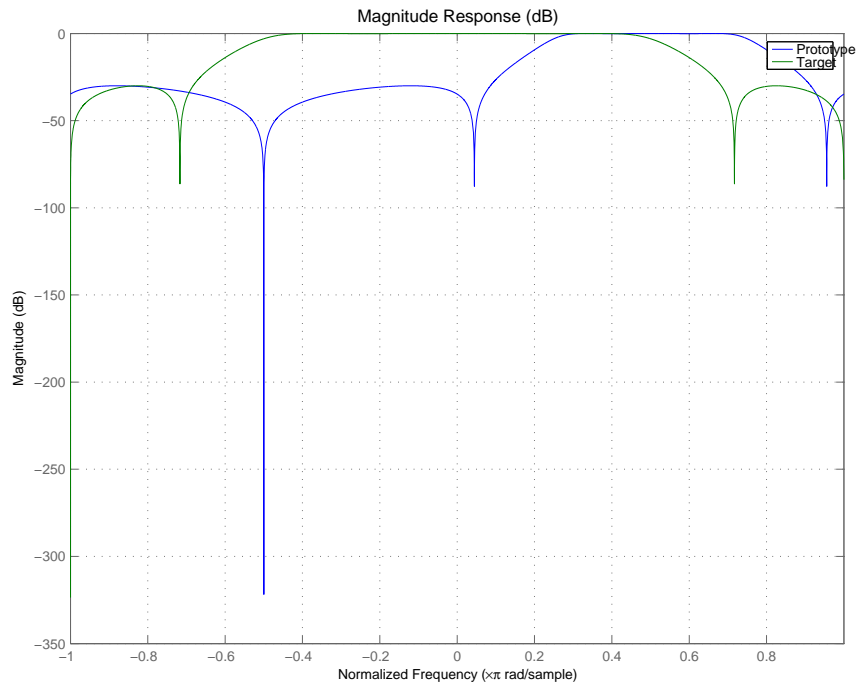
Create a complex passband from 0.25 to 0.75:

```
[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**

<b>Variable</b>	<b>Description</b>
<i>B</i>	Numerator of the prototype lowpass filter
<i>A</i>	Denominator of the prototype lowpass filter
<i>Wo</i>	Frequency values to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>Num</i>	Numerator of the target filter
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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

# iircomb

---

**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  $2\pi$  — the number of notches equals  $n+1$ .

For the notching filter, the transfer function takes the form

$$H(z) = b \times \frac{1 - z^{-n}}{1 - \alpha z^{-n}}$$

where  $\alpha$  and  $b$  are the positive scalars 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  $q = \omega_o/bw$  where  $\omega_o$  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

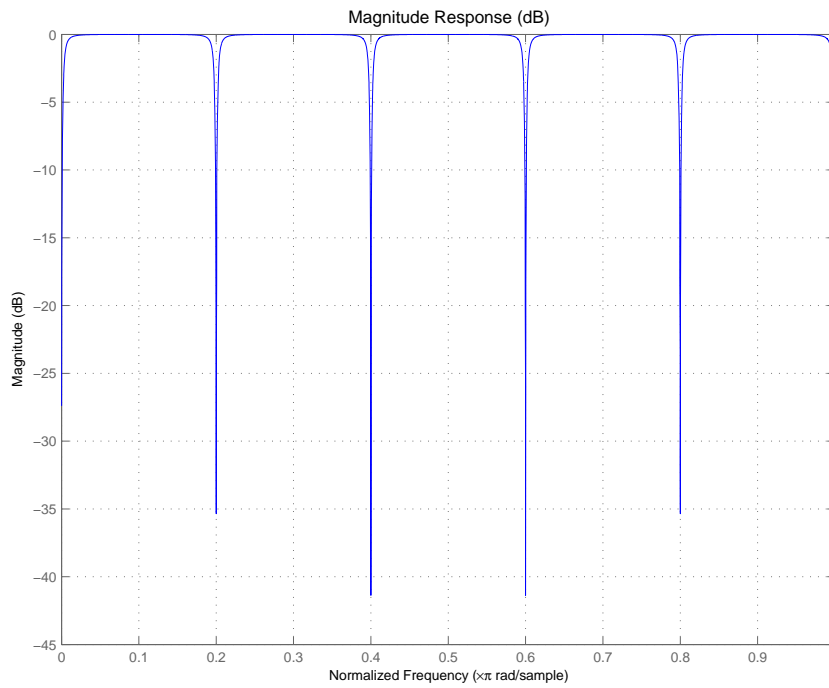
Design and plot an IIR notch filter with 11 notches (equal to filter order plus 1) that removes a 60 Hz tone ( $f_0$ ) from a signal at 600 Hz ( $f_s$ ). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth.

```
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.

# iircomb

---



**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:

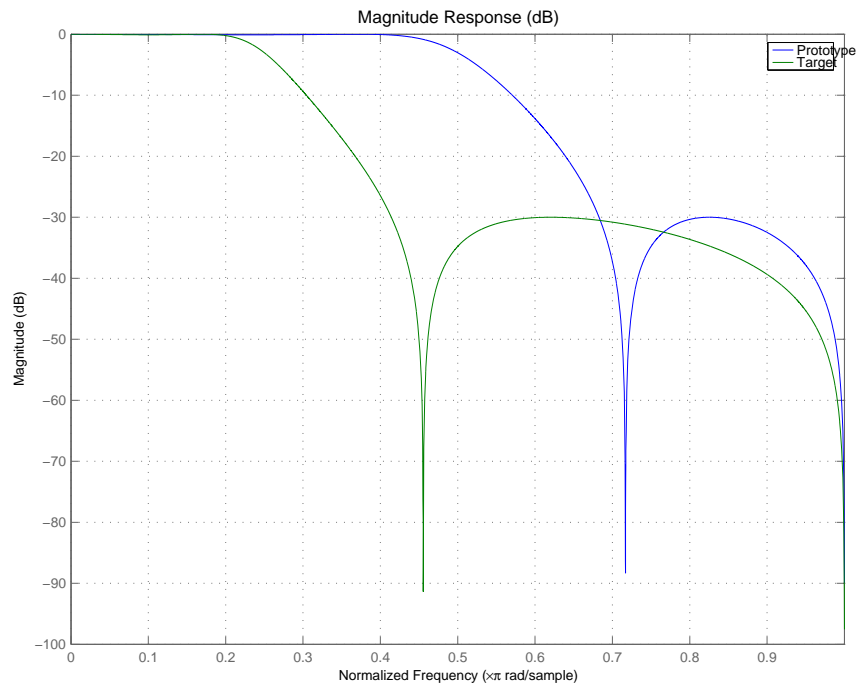
```
[b, a] = ellip(3, 0.1, 30, 0.409);  
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);  
[num, den] = iirftransf(b, a, AlpNum, AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Here's the comparison between the filters.

# iirtransf



## Arguments

Variable	Description
<i>OrigNum</i>	Numerator of the prototype lowpass filter
<i>OrigDen</i>	Denominator of the prototype lowpass filter
<i>FTFNum</i>	Numerator of the mapping filter
<i>FTFDen</i>	Denominator of the mapping filter
<i>OutNum</i>	Numerator of the target filter
<i>OutDen</i>	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- $p$ th 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 contain 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- $p$ th 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
<code>df1sos</code>	Direct-form I IIR filter with second-order sections
<code>df2sos</code>	Direct-form II IIR filter with second-order sections
<code>df1tsos</code>	Transposed direct-form I IIR filter with second-order sections
<code>df2tsos</code>	Transposed direct-form II IIR filter with second-order sections

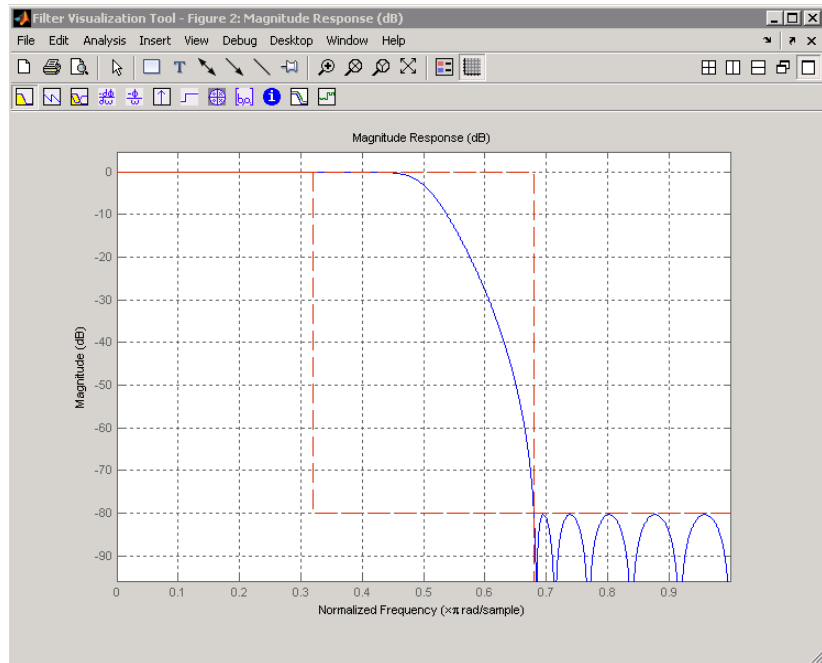
## Examples

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)



**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  $W_t$ s.

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] = iir1p2bp(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);
```

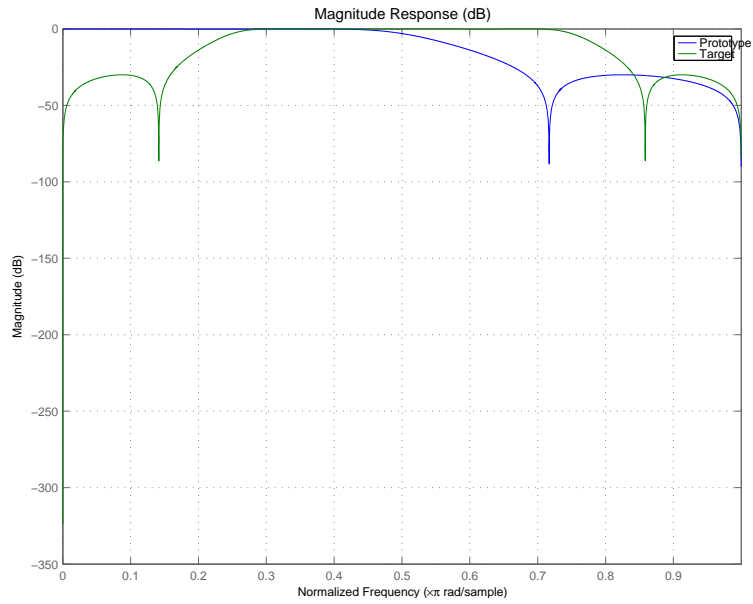
Create the real bandpass filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies  $W_{t1}=0.25$  and  $W_{t2}=0.75$ :

```
[num,den] = iir1p2bp(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
$W_0$	Frequency value to be transformed from the prototype filter
$W_t$	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] = iir1p2bpc(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);
```

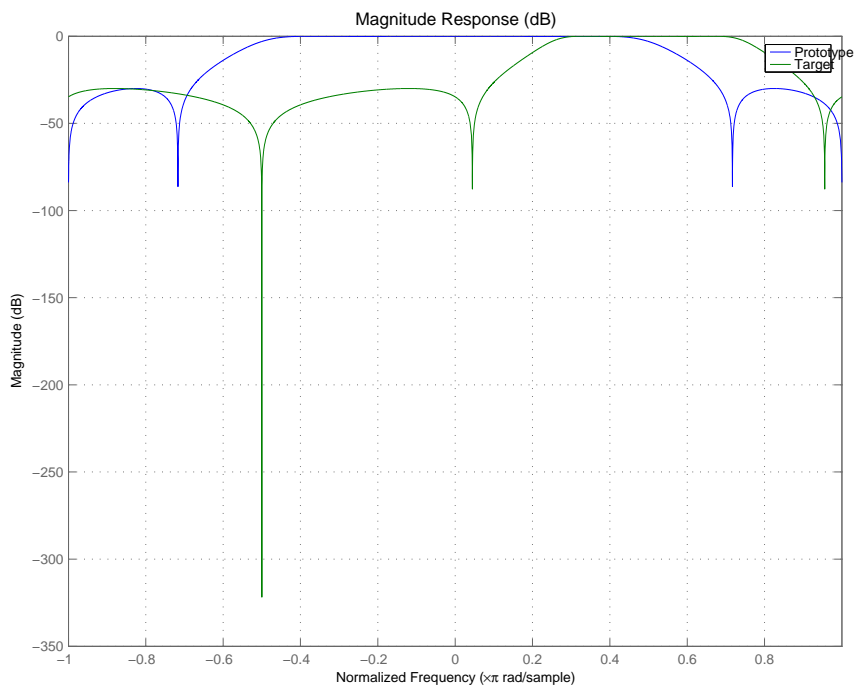
Move the cutoffs of the prototype filter to the new locations  $W_{t1}=0.25$  and  $W_{t2}=0.75$  creating a complex bandpass filter:

```
[num, den] = iir1p2bpc(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
$W_0$	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.
$W_t$	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

<b>Variable</b>	<b>Description</b>
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

## See Also

iirftransf, allpasslp2bpc, zpk1p2bpc

**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  $W_t$ s.

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  $\omega_0$  and the target frequencies vector  $\omega_t$ . 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:

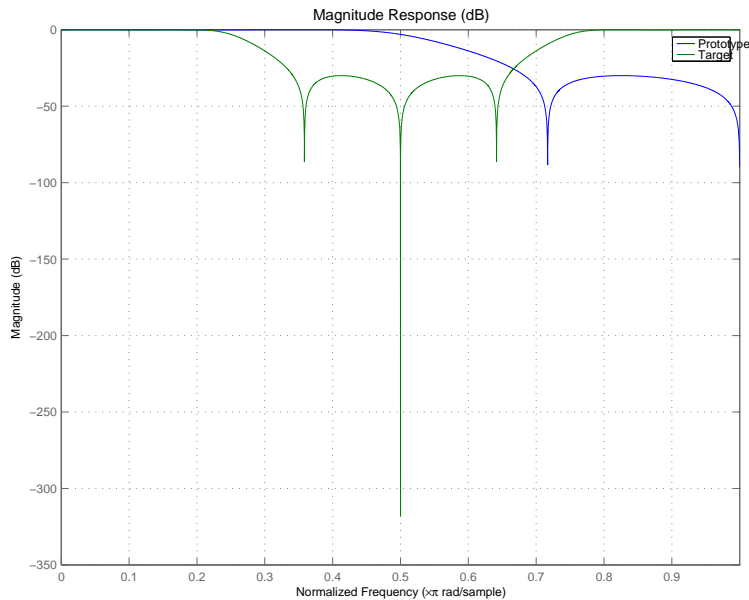
```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create the real bandstop filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies  $\omega_{t1}=0.25$  and  $\omega_{t2}=0.75$ :

```
[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
<i>B</i>	Numerator of the prototype lowpass filter
<i>A</i>	Denominator of the prototype lowpass filter
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>Num</i>	Numerator of the target filter
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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.

<b>Purpose</b>	Transform IIR lowpass filter to IIR complex bandstop filter
<b>Syntax</b>	<pre>[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt) [G,AllpassNum,AllpassDen] = iirlp2bsc(Hd,Wo,Wt)</pre> where Hd is a <code>dfilt</code> object
<b>Description</b>	<p><code>[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)</code> 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.</p> <p>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.</p> <p>This transformation effectively places one feature of an original filter, located at frequency <math>-W_o</math>, at the required target frequency location, <math>W_{t1}</math>, and the second feature, originally at <math>+W_o</math>, at the new location, <math>W_{t2}</math>. It is assumed that <math>W_{t2}</math> is greater than <math>W_{t1}</math>. Additionally the transformation swaps passbands with stopbands in the target filter.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>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.</p> <p>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</p>

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);
```

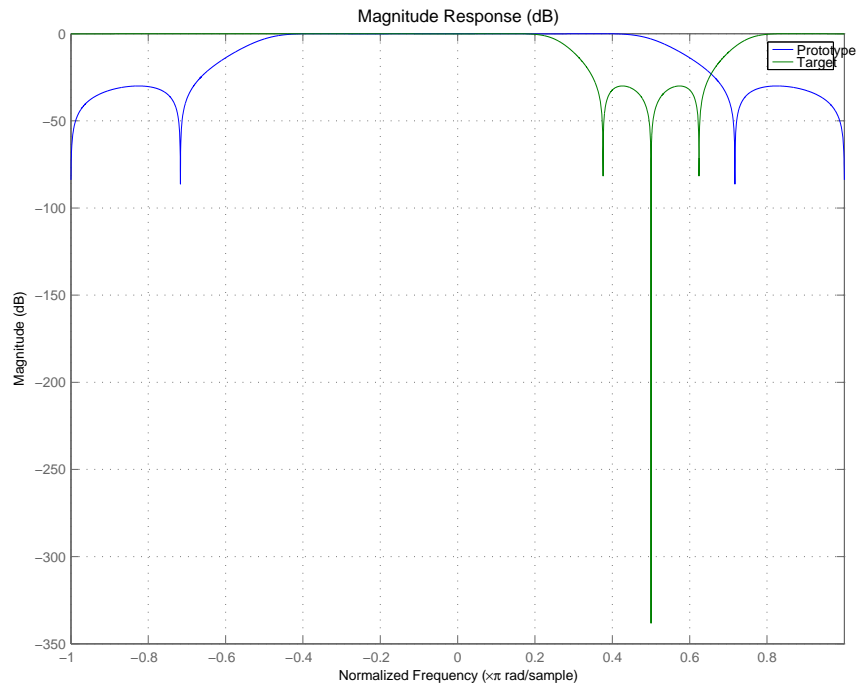
Move the cutoffs of the prototype filter to the new locations  $W_{t1}=0.25$  and  $W_{t2}=0.75$  creating a complex bandstop filter:

```
[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
$W_0$	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.
$W_t$	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
<i>Num</i>	Numerator of the target filter
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

**See Also**

iirftransf, allpasslp2bsc, zpklp2bsc.

# iirlp2hp

---

**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, `iirlp2hp` 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.

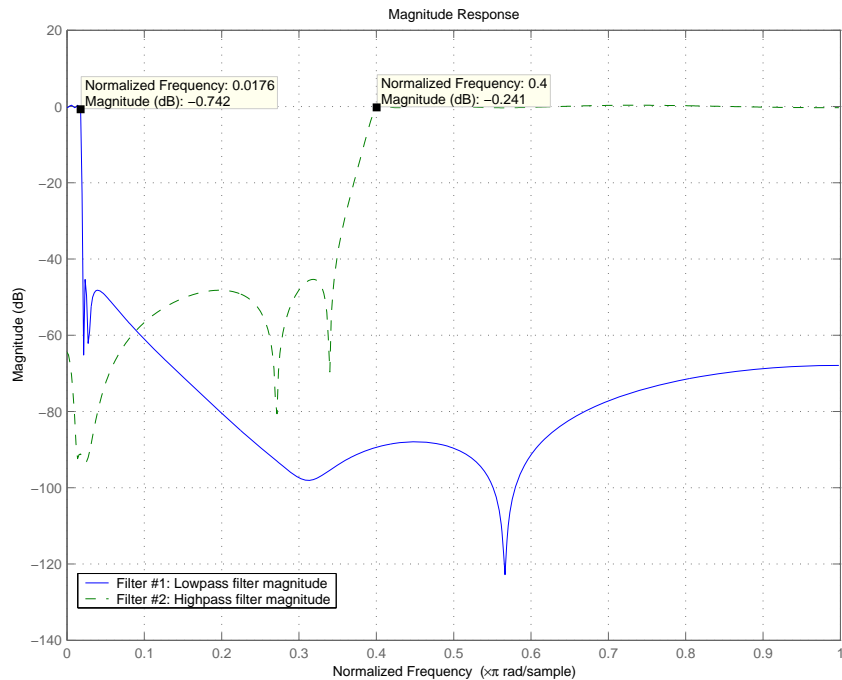
In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

`[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](#), [fir1p2lp](#), [fir1p2hp](#)

## References

Mitra, Sanjit K., *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.



**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, `iirlp2hp` 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.

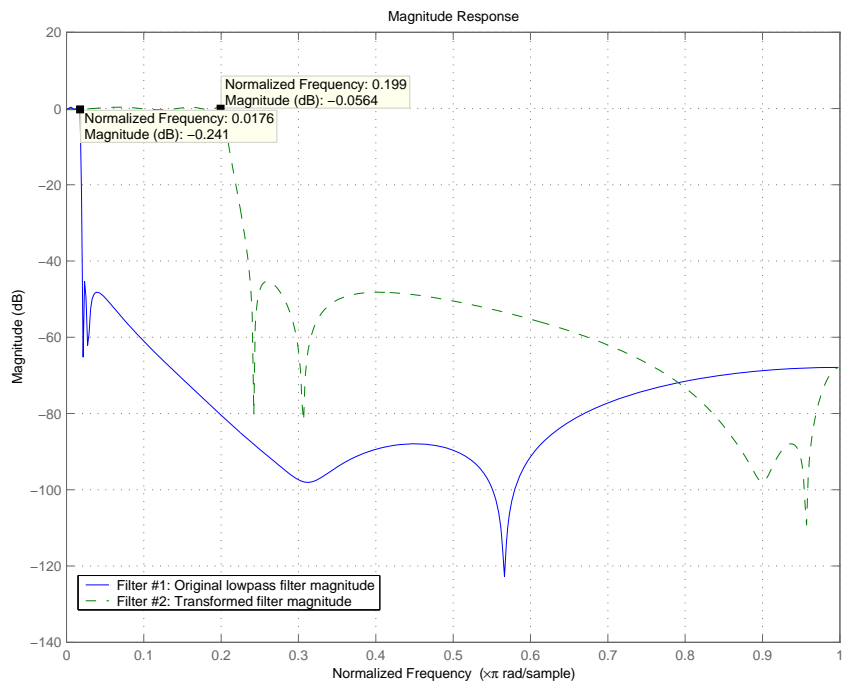
`[G,AllpassNum,AllpassDen] = iirlp2lp(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a lowpass magnitude response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that 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, fir1p2lp, fir1p2hp

## References

Mitra, Sanjit K, *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

# iirlp2mb

---

**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 Mth-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,  $W_{t1}, \dots, W_{tM}$ .

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

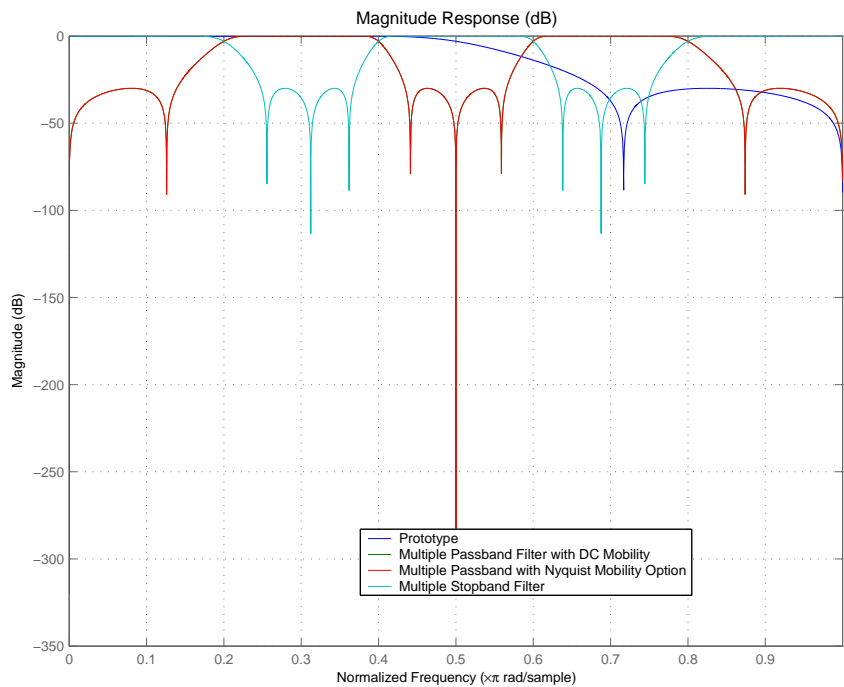
Create the real multiband filter with two stopbands:

```
[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
<i>B</i>	Numerator of the prototype lowpass filter
<i>A</i>	Denominator of the prototype lowpass filter
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>Pass</i>	Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default
<i>Num</i>	Numerator of the target filter

Variable	Description
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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`, `allpass1p2mb`, `zpk1p2mb`

## 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.



**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,  $W_{t1}, \dots, W_{tM}$ .

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:

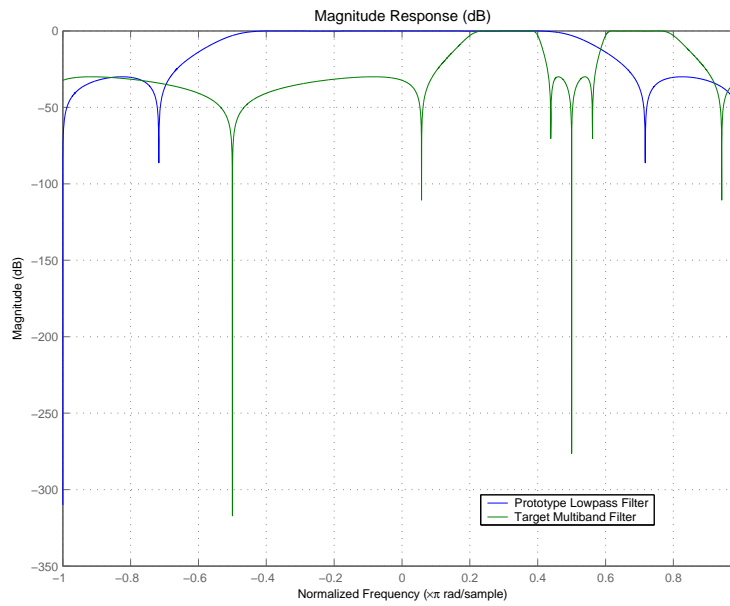
```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

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
<i>B</i>	Numerator of the prototype lowpass filter.
<i>A</i>	Denominator of the prototype lowpass filter.
<i>W<sub>0</sub></i>	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.
<i>W<sub>c</sub></i>	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.
<i>Num</i>	Numerator of the target filter.

# iirlp2mbc

---

<b>Variable</b>	<b>Description</b>
<i>Den</i>	Denominator of the target filter.
<i>AllpassNum</i>	Numerator of the mapping filter.
<i>AllpassDen</i>	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 Nth-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  $W_{o1}, \dots, W_{oN}$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

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);
```

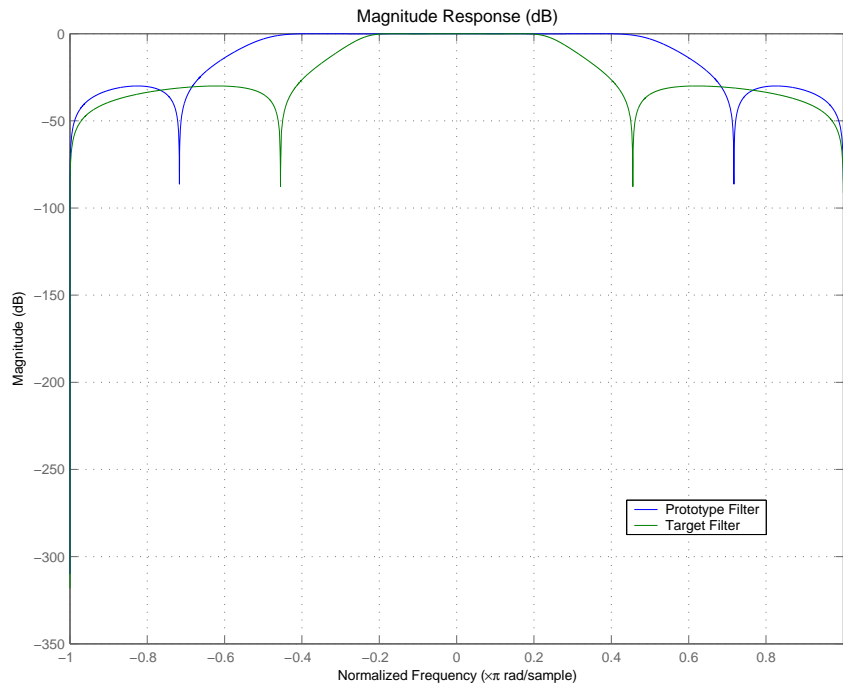
Create the complex bandpass filter from the real lowpass filter:

```
[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.
$W_o$	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.
$W_t$	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.

# iirlp2xc

---

<b>Variable</b>	<b>Description</b>
<i>Den</i>	Denominator of the target filter.
<i>AllpassNum</i>	Numerator of the mapping filter.
<i>AllpassDen</i>	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 Nth-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  $W_{o1}, \dots, W_{oN}$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

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);
```

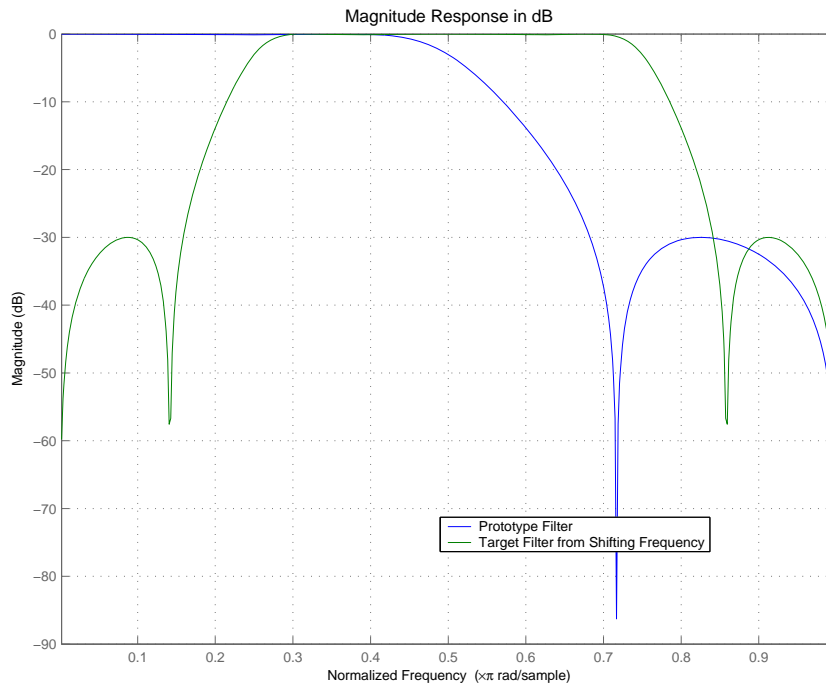
Move the cutoffs of the prototype filter to the new locations  $W_{t1}=0.25$  and  $W_{t2}=0.75$  creating a real bandpass filter:

```
[num, den] = iir1p2xn(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);
```

`iir1p2xn` has created the desired bandpass filter with the cutoff locations specified in the command.



## Arguments

Variable	Description
<i>B</i>	Numerator of the prototype lowpass filter
<i>A</i>	Denominator of the prototype lowpass filter
<i>Wo</i>	Frequency values to be transformed from the prototype filter
<i>Wt</i>	Desired frequency locations in the transformed target filter
<i>Pass</i>	Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default
<i>Num</i>	Numerator of the target filter

Variable	Description
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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, allpass1p2xn, zpk1p2xn

### 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.

# iirlpnorm

---

**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-`p`th 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 2-925 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-`p`th 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  $0 < \text{radius} < 1$ . `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:

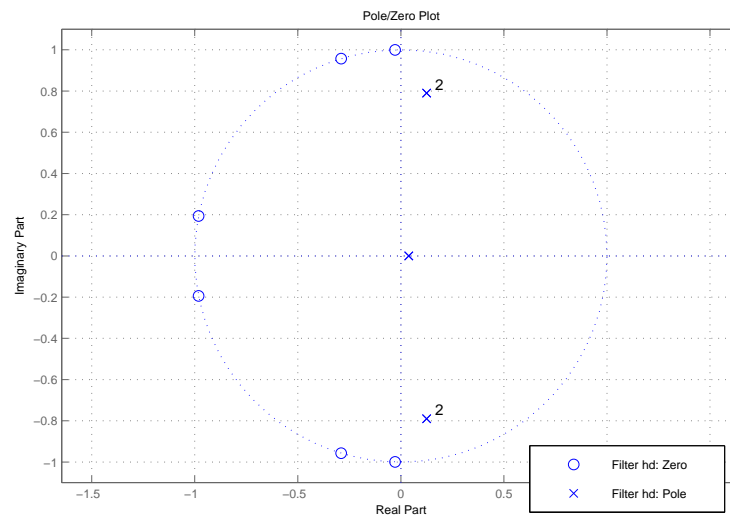
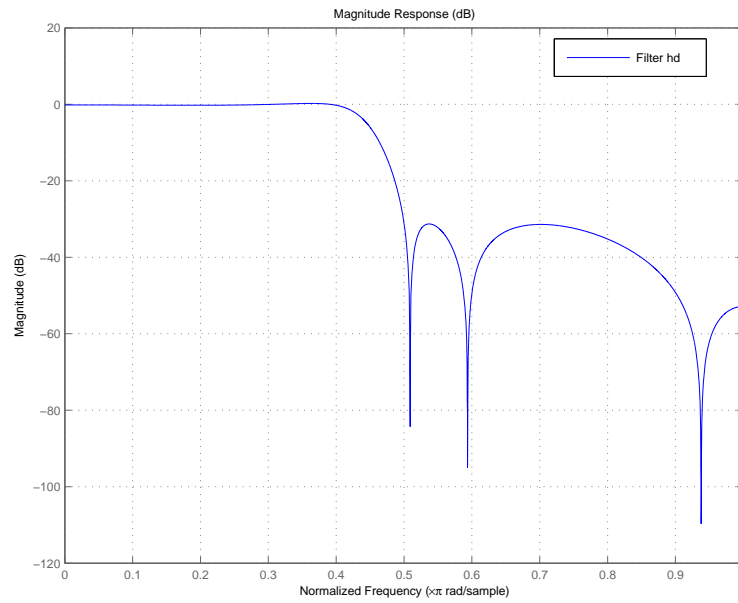
- 1 The approximation error is extremely small (try reducing the number of poles or zeros — refer to the hints above).
- 2 The filter specifications have huge variation, such as `a=[1 1e9 0 0]`.

## 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.



# iirlpnormc

---

## See Also

freqz, filter, iirgrpdelay, iirlpnorm, zplane

## References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

---

<b>Purpose</b>	RLS IIR filter from specification object
<b>Syntax</b>	<pre>hd = design(d,'iirls') hd = design(d,'iirls',designoption,value,designoption,value, ...)</pre>
<b>Description</b>	<code>hd = design(d,'iirls')</code> designs a least-squares filter specified by the filter specification object <code>d</code> .

---

**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.

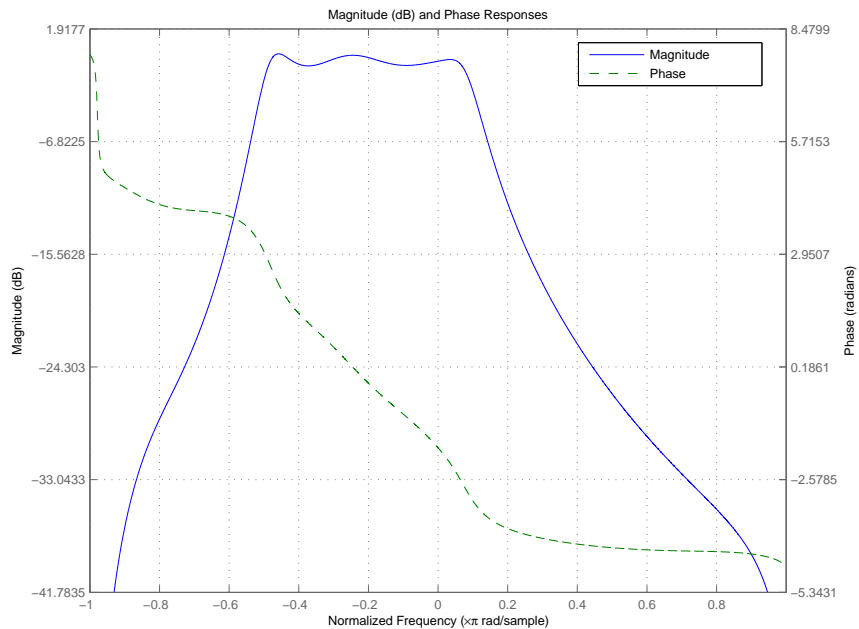
```
help(d,'iirls')
```

## 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, fir1s`

# iirnotch

---

**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  $w_0$ , and with the bandwidth at the -3 dB point set to `bw`. To design the filter,  $w_0$  must meet the condition  $0.0 < w_0 < 1.0$ , where 1.0 corresponds to  $\pi$  radians per sample in the frequency range.

The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = w_0/bw$  where  $\omega_0$  is  $w_0$ , the frequency to remove from the signal.

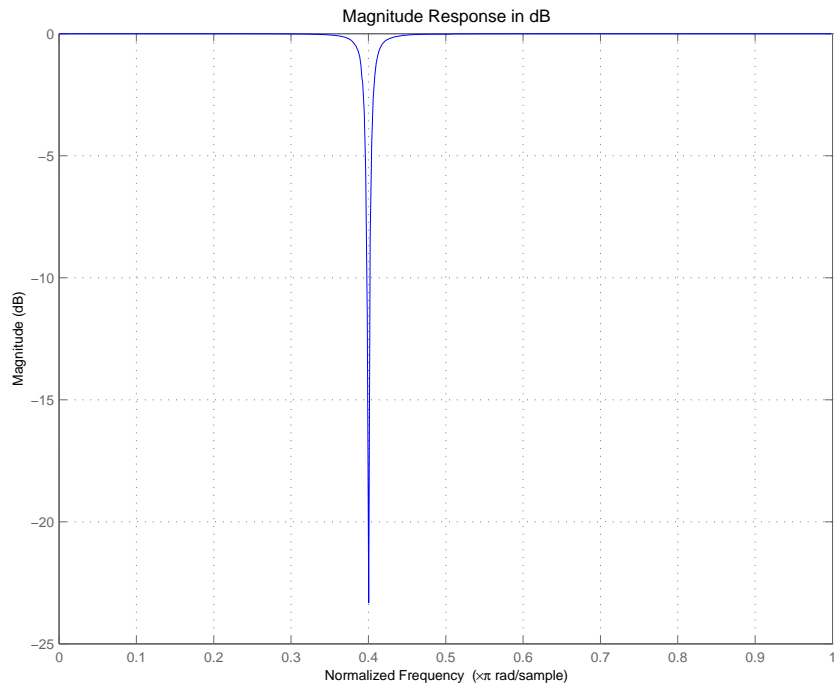
`[num,den] = iirnotch(w0,bw,ab)` returns a digital notching filter whose bandwidth, `bw`, is specified at a level of `-ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default -3 dB point, such as -6 dB or 0 dB.

**Examples** Design and plot an IIR notch filter that removes a 60 Hz tone ( $f_0$ ) from a signal at 300 Hz ( $f_s$ ). 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  $0.4\pi$  radians per sample. Compare this plot to the comb filter plot shown on the reference page for `iircomb`.





**See Also** `firgr`, `iircomb`, `iirpeak`

# 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  $w_0$ , and with the bandwidth at the +3 dB point set to  $bw$ . To design the filter,  $w_0$  must meet the condition  $0.0 < w_0 < 1.0$ , where 1.0 corresponds to  $\pi$  radians per sample in the frequency range.

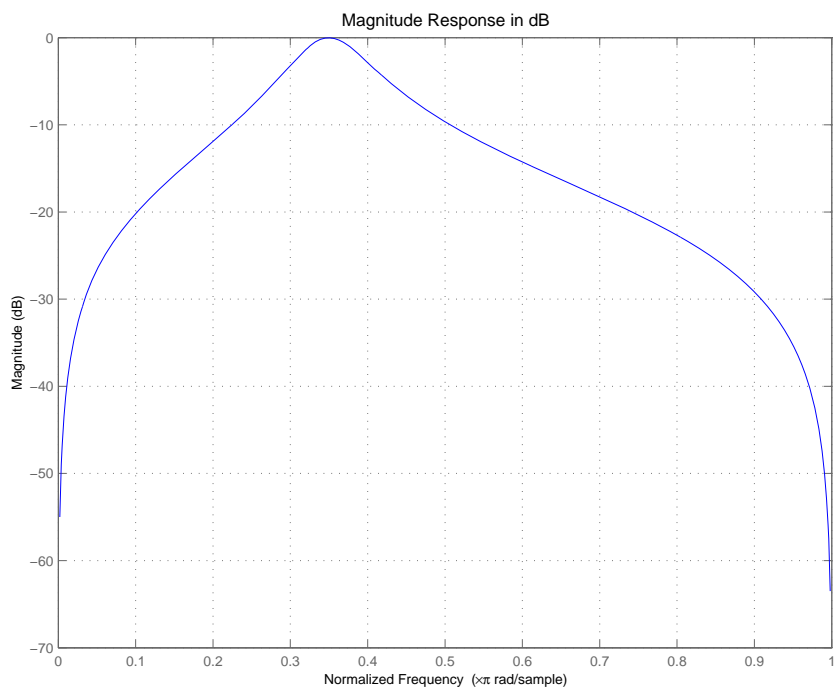
The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = w_0/bw$  where  $\omega_0$  is  $w_0$  the signal frequency to boost.

`[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`

# iirpowcomp

---

**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  $\text{conj}(bp(\text{end}:-1:1)) = c*bp$ .

When `c` is omitted, it is chosen as follows:

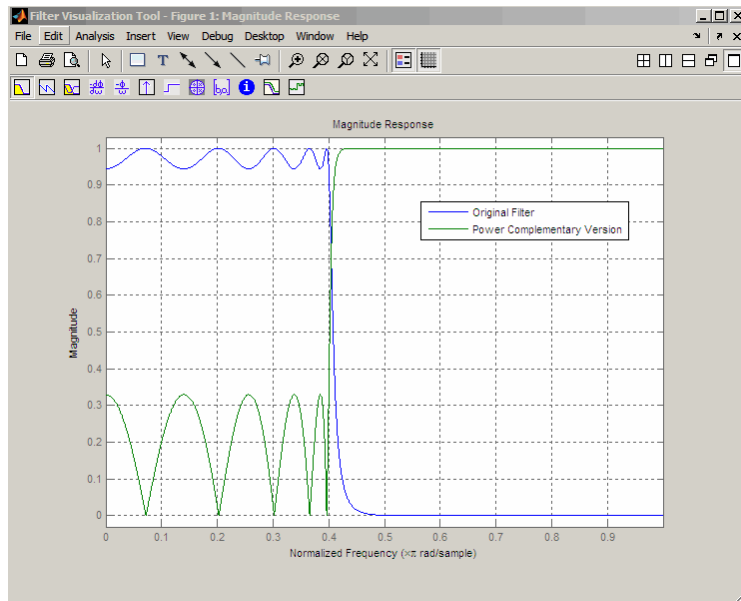
- When `b` is real, chooses `C` as 1 or -1, whichever yields `bp` real
- When `b` is complex, `C` defaults to 1

`ap` is always equal to `a`.

## Examples

```
[b,a]=cheby1(10,.5,.4);  
[bp,ap]=iirpowcomp(b,a);  
Hd1 = dfilt.df2(b,a);  
Hd2 = dfilt.df2(bp,ap);  
hfvt = fvtool([Hd1,Hd2], 'MagnitudeDisplay', 'Magnitude');  
legend(hfvt, 'Original Filter', 'Power Complementary Version');
```

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](#)

# iirrateup

---

**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 Nth-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

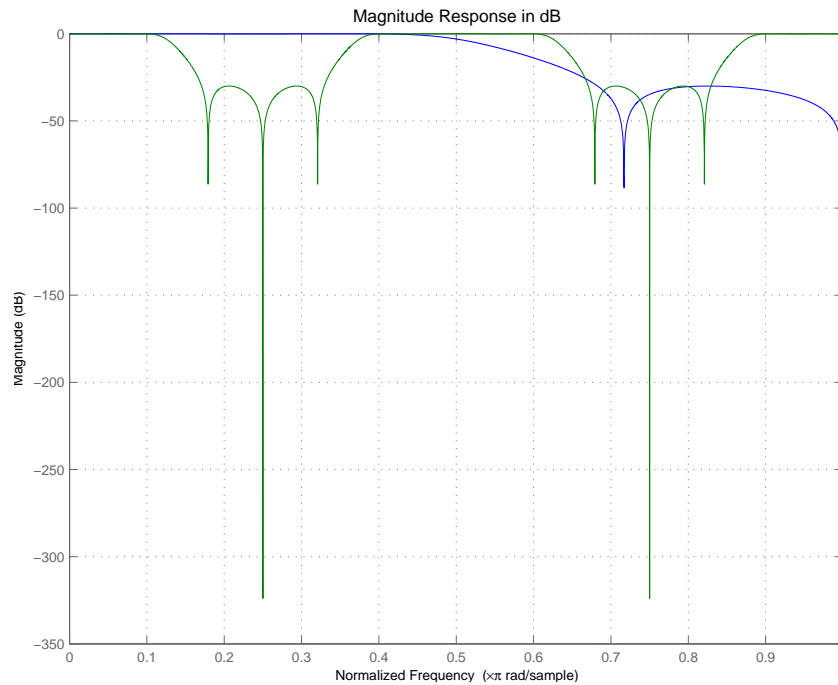
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[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
<i>B</i>	Numerator of the prototype lowpass filter
<i>A</i>	Denominator of the prototype lowpass filter
<i>N</i>	Frequency multiplication ratio
<i>Num</i>	Numerator of the target filter
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

# iirateup

---

## **See Also**

iirftransf, allpassrateup, zpkrateup



**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);
```

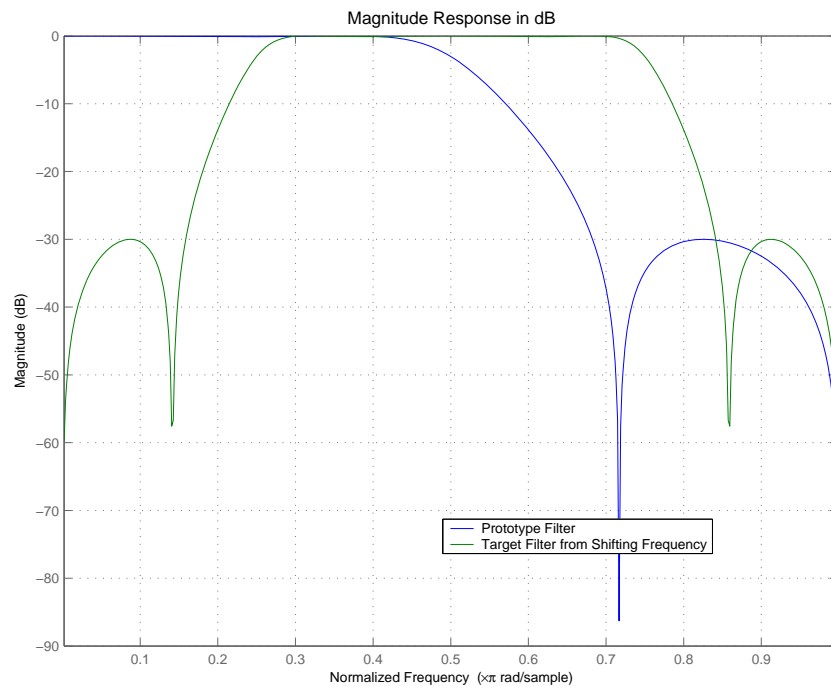
Perform the real frequency shift by defining where the selected feature of the prototype filter, originally at  $W_0=0.5$ , should be placed in the target filter,  $W_t=0.75$ :

```
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**

<b>Variable</b>	<b>Description</b>
<i>B</i>	Numerator of the prototype lowpass filter
<i>A</i>	Denominator of the prototype lowpass filter
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency location in the transformed target filter
<i>Num</i>	Numerator of the target filter
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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**

iirfttransf, allpassshift, zpkshift.

# iirshiftc

---

**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);
```

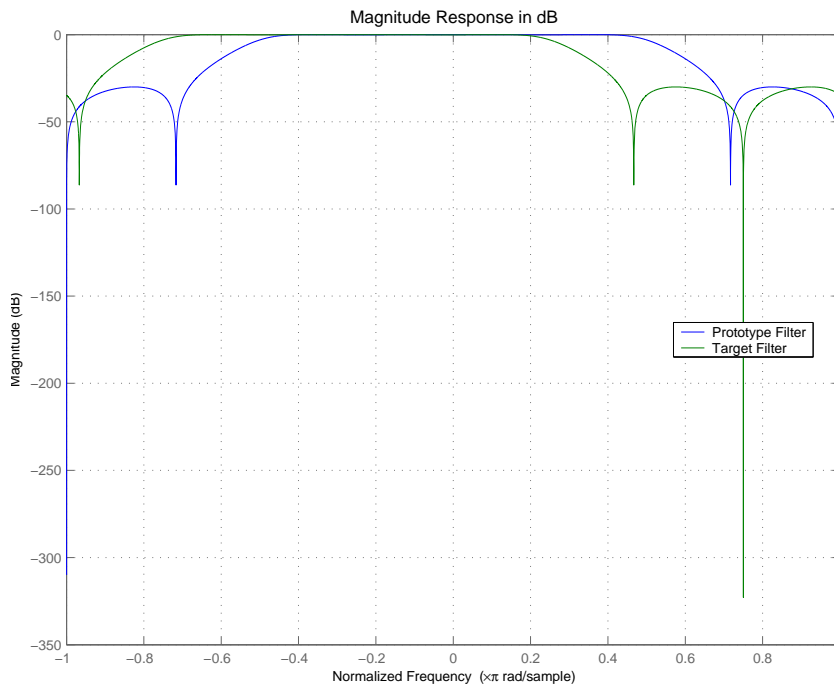
Rotate all features of the prototype filter in the frequency domain by the same amount by specifying where the selected feature of an original filter,  $W_o = 0.5$ , should appear in the target filter,  $W_t = 0.25$ :

```
[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  $\omega_t = 0.25$ .



## Arguments

Variable	Description
$B$	Numerator of the prototype lowpass filter
$A$	Denominator of the prototype lowpass filter

Variable	Description
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency location in the transformed target filter
<i>Num</i>	Numerator of the target filter
<i>Den</i>	Denominator of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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. Schaffer 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**

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.

- “Adaptive Filters” on page 2-949
- “Discrete-Time Filters” on page 2-950
- “Multirate Filters” on page 2-950

**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

$[h, t] = \text{impz}(hd)$  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 ( $t = [0 \ 1 \ 2 \dots]$ ).  $\text{impz}$  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,  $\text{impz}$  returns the matrix  $h$ . Each column of  $h$  corresponds to one filter in the vector  $hd$ .

$\text{impz}(hd)$  uses FVTool to plot the impulse response of the discrete-time filter  $hd$ . If  $hd$  is a vector of filters,  $\text{impz}$  plots the response and for each filter in the vector.

## Multirate Filters

$[h, t] = \text{impz}(hm)$  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 ( $t = [0 \ 1 \ 2 \dots]$ ).  $[h, t] = \text{impz}(hm)$  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,  $\text{impz}$  returns the matrix  $h$ . Each column of  $h$  corresponds to one filter in the vector  $ha$ .

$\text{impz}(hm)$  uses FVTool to plot the impulse response of the multirate filter  $hm$ . If  $ha$  is a vector of filters,  $\text{impz}$  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,  $\text{impz}$  assumes the filter is running at that rate.

For multistage cascades,  $\text{impz}$  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.  $\text{impz}$  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, lowpass 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.

```
d = fdesign.lowpass(.4,.5,1,80);  
% Create a design object for the prototype filter.
```

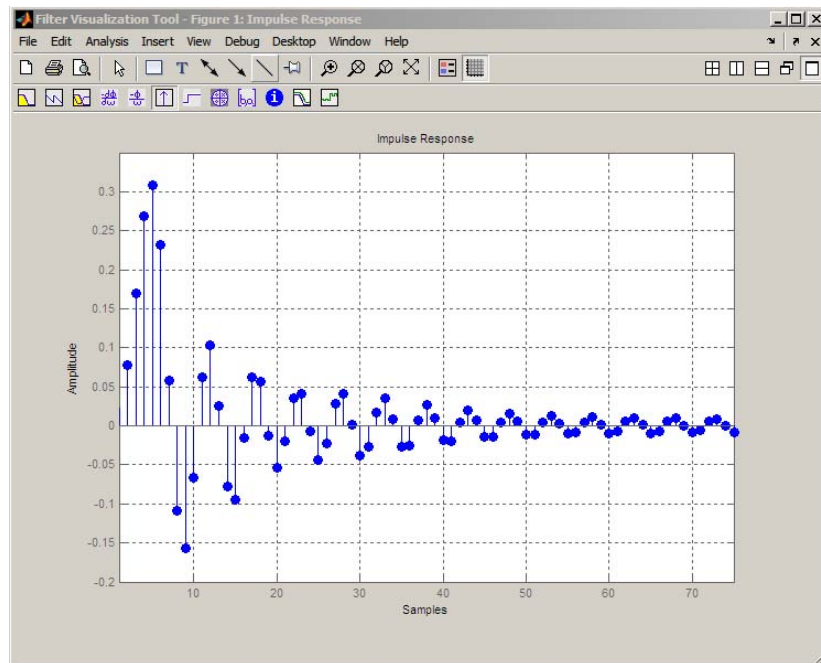
Use `ellip` to design a minimum order discrete-time filter in second-order section form.

```
hd=design(d, 'ellip');
```

Convert `hd` to fixed-point and plot the impulse response:

```
impz(hd);  
axis([1 75 -0.2 0.35])
```

# impz



**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

# int

---

**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`

# isallpass

---

**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

$$H(z) = \frac{\alpha_n + \dots + \alpha_1 z^{-(n-1)} + z^{-n}}{1 + \alpha_1 z^{-1} + \dots + \alpha_n z^{-n}}$$

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);  
isallpass(hd)  
% Returns a 1 indicated hd is all pass.
```

**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)
```

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)  
h=design(d) % Use the default design method iirlpnorm.  
islinphase(h)
```

**See Also** `isallpass`, `isfir`, `ismaxphase`, `isminphase`, `isreal`, `issos`, `isstable`

**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= 60;  
d = fdesign.lowpass('fp,fst,ap,ast',fp,fst,ap,ast,fs);  
h = design(d,'equiripple','maxphase',true);  
ismaxphase(h)
```

To make this a minimum phase filter, use `fliplr` to change the coefficient order. Reordering the coefficients this way changes the phase from maximum to minimum.

```
h.numerator=fliplr(h.numerator);  
isminphase(h)
```

# ismaxphase

---

returns 1 so this is a minimum phase filter.

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)
```

## See Also

isallpass, isfir, islinphase, isminphase, isreal, issos, isstable

**Purpose** Determine whether filter is minimum phase

**Syntax** `isminphase(h)`  
`isminphase(h,tolerance)`

**Description** `isminphase(h)` determines if the filter object `h` is minimum phase, and returns 1 if true and 0 if false. `isminphase` works with `adapfilt`, `dfilt`, and `mfilt` objects.

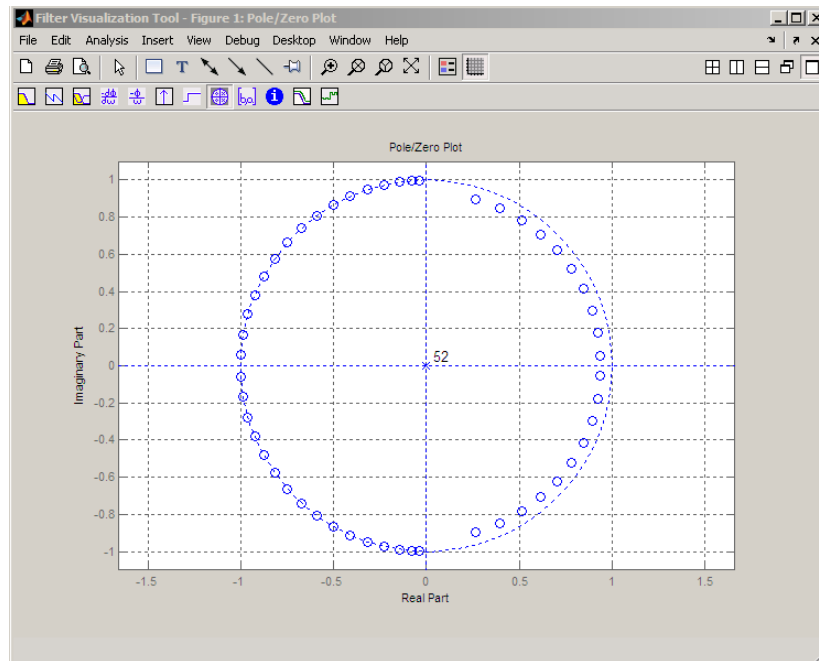
`isminphase(h,tolerance)` uses the input argument `tolerance` to determine whether the filter transfer function zeros are outside the unit circle by more than `tolerance`. If the zeros do not lie outside the unit circle by more than `tolerance`, `isminphase` returns a 1. If any zeros are more than `tolerance` outside the unit circle, `isminphase` returns a 0.

A filter is *minimum phase* when all the zeros of its transfer function are on or inside the unit circle, or the numerator is a scalar. An equivalent definition for a minimum phase filter is a causal and stable system with a causal and stable inverse.

**Examples** This example creates a minimum-phase filter.

```
fp = 200;  
fst= 230;  
fs = 900;  
ap = 1;  
ast= 60;  
d = fdesign.lowpass('fp,fst,ap,ast',fp,fst,ap,ast,fs);  
h = design(d,'equiripple','minphase',true);  
isminphase(h)  
% Returns a 1  
fvtool(h,'analysis','polezero')
```

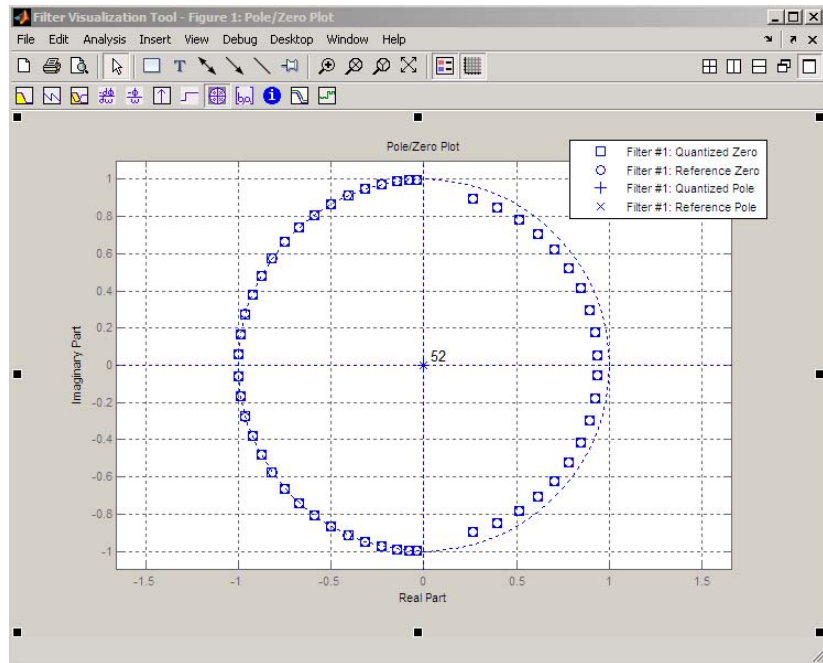
# isminphase



Note that the zeros of the FIR filter are all on or inside the unit circle.

To illustrate the effect of coefficient quantization, make  $h$  a fixed-point filter. The quantization process results in the filter no longer being minimum phase. However, if the tolerance is increased to 0.01, the quantized design satisfies the definition of a minimum phase filter.

```
h.arithmetic='fixed';
isminphase(h)
% Returns a 0
fvtool(h,'analysis','polezero');
isminphase(h,0.01)
% Returns a 1
```



## See Also

isallpass, isfir, islinphase, ismaxphase, isreal, issos, isstable,

# isreal

---

**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.arithmetic='fixed';
isreal(hd)
% Returns a 1
```

**See Also** `isfir`, `islinphase`, `ismaxphase`, `isminphase`, `issos`, `isstable`, `isallpass`



---

<b>Purpose</b>	Determine whether filter is SOS form
<b>Syntax</b>	<code>issos(hd)</code>
<b>Description</b>	<code>issos(hd)</code> determines whether quantized filter <code>hq</code> consists of second-order sections. Returns 1 if all sections of quantized filter <code>hq</code> have order less than or equal to two, and 0 otherwise.
<b>Examples</b>	<p>By default, <code>fdesign</code> and <code>design</code> return SOS filters when possible. This example designs a lowpass SOS filter that uses fixed-point arithmetic.</p> <pre>d=fdesign.lowpass('n,fp,ap,ast',40,0.55,0.1,60) designmethods(d) hd=design(d,'ellip'); hd.arithmetic='fixed'; issos(hd)</pre> <p>The fixed-point filter <code>hd</code> is in second-order section form, as is the double-precision version.</p>
<b>See Also</b>	<code>isallpass</code> , <code>isfir</code> , <code>islinphase</code> , <code>ismaxphase</code> , <code>isminphase</code> , <code>isreal</code> , <code>isstable</code>

# 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')
isstable(hd)
hd2=design(d,'equiripple');
isstable(hd2)
```

**See Also** `isallpass`, `isfir`, `islinphase`, `ismaxphase`, `isminphase`, `isreal`, `issos`, `zplane`

**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',200,0.01)  
hd= design(d,'kaiserwin','filterstructure','dffir')
```

# kaiserwin

---

In this example, `kaiserwin` uses an interpolating filter specification object:

```
d=fdesign.interpolator(4,'lowpass');  
h = design(d,'kaiserwin');
```

## See Also

`equiripple`, `firls`

**Purpose** Fractional delay filter from `fdesign.fracdelay` specification object

**Syntax** `Hd = design(d, 'lagrange')`  
`hd = design(d, 'lagrange', FilterStructure, structure)`

**Description** `Hd = design(d, 'lagrange')` designs a fractional delay filter using the Lagrange method based on the specifications in `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')
```

# 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`

# limitcycle

---

**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: <ul style="list-style-type: none"><li>• <b>Granular</b> — indicates that a granular overflow occurred.</li><li>• <b>Overflow</b> — indicates that an overflow limit cycle occurred.</li><li>• <b>None</b> — indicates that the test did not find any limit cycles.</li></ul>
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, <code>Trial = 10</code> 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)
```

# limitcycle

---

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.

<b>stopcriterion Setting</b>	<b>Description</b>
<b>either</b>	Stop the Monte Carlo trials when <code>limitcycle</code> detects either a granular or overflow limit cycle.
<b>granular</b>	Stop the Monte Carlo trials when <code>limitcycle</code> detects a granular limit cycle.
<b>overflow</b>	Stop the Monte Carlo trials when <code>limitcycle</code> 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

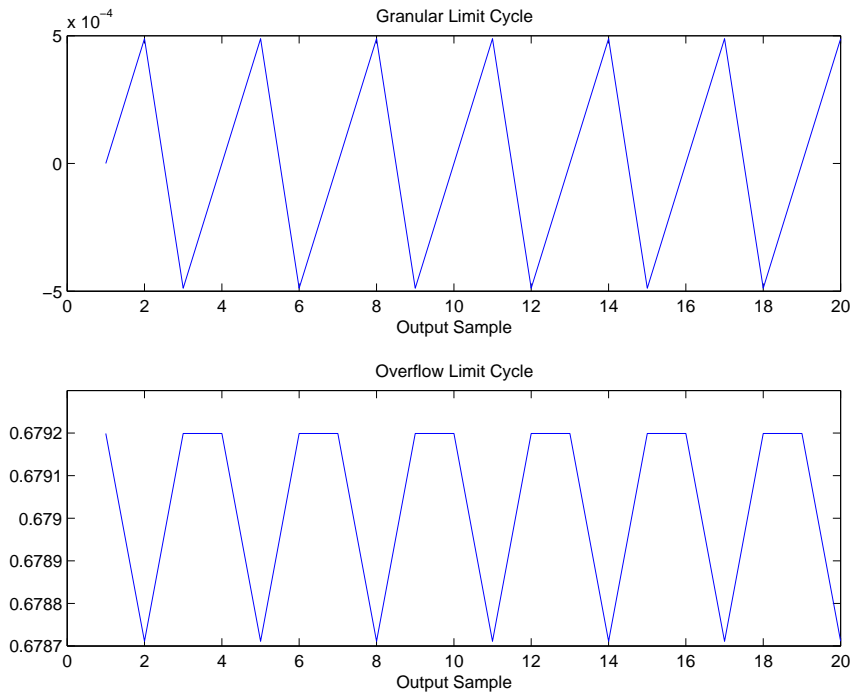
oreport =

    LimitCycle: 'overflow'
             Zi: [2x1 double]
             Output: [1303x1 embedded.fi]
```

# limitcycle

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`

<b>Purpose</b>	Maxflat FIR filter
<b>Syntax</b>	<pre>hd = design(d,'maxflat') hd = design(d,'maxflat','FilterStructure',structure)</pre>
<b>Description</b>	<p><code>hd = design(d,'maxflat')</code> designs a maximally flat filter, <code>hd</code>, from a filter specification object, <code>d</code>.</p> <p><code>hd = design(d,'maxflat','FilterStructure',structure)</code> designs a maximally flat filter where <code>structure</code> is one of the following:</p> <ul style="list-style-type: none"><li>• 'dffir', a discrete-time, direct-form FIR filter (the default value)</li><li>• 'dffirt', a discrete-time, direct-form FIR transposed filter</li><li>• 'dfsymfir', a discrete-time, direct-form symmetric FIR filter</li><li>• 'fftfir', a discrete-time, overlap-add, FIR filter</li></ul>
<b>Examples</b>	<p>Example 1: Design a lowpass filter with a maximally flat FIR structure.</p> <pre>d = fdesign.lowpass('N,F3dB', 50, 0.3); Hd = design(d, 'maxflat');</pre> <p>Example 2: Design a highpass filter with a maximally flat overlap-add FIR structure.</p> <pre>d = fdesign.highpass('N,F3dB', 50, 0.7); Hd=design(d,'maxflat','FilterStructure','fftfir');</pre>

# maximizestopband

---

**Purpose** Maximize stopband attenuation of fixed-point filter

**Syntax** `Hq = maximizestopband(Hd,Wordlength)`  
`Hq = maximizestopband(Hd,Wordlength,'Ntrials',N)`

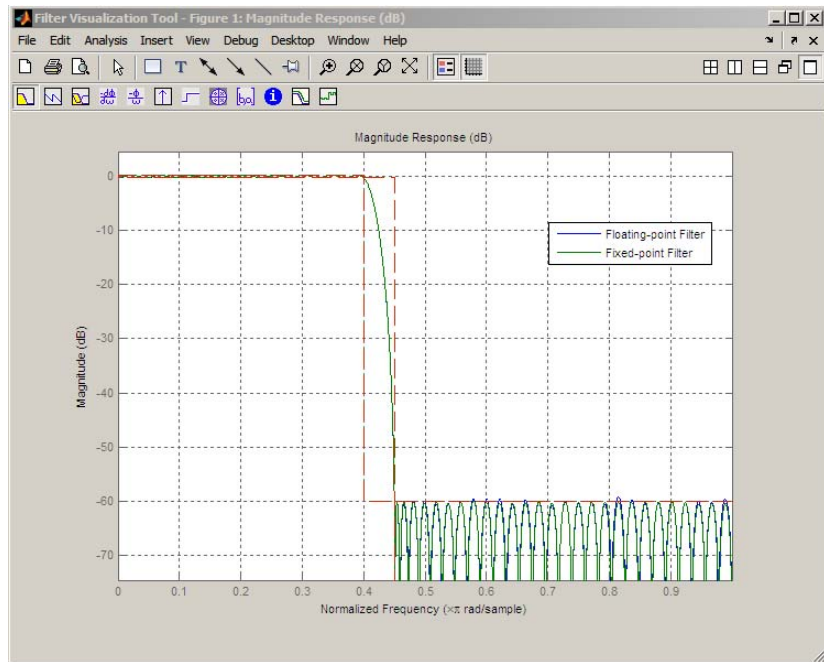
**Description** `Hq = maximizestopband(Hd,Wordlength)` quantizes the single-stage or multistage FIR filter `Hd` and returns the fixed-point filter `Hq` with wordlength `wordlength` that maximizes the stopband attenuation. `Hd` must be generated using `fdesign` and `design`. For multistage filters, `wordlength` can either be a scalar or vector. If `wordlength` is a scalar, the same wordlength is used for all stages. If `wordlength` is a vector, each stage uses the corresponding element in the vector. The vector length must equal the number of stages. `maximizestopband` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`

`Hq = maximizestopband(Hd,Wordlength,'Ntrials',N)` specifies the number of Monte Carlo trials to use in the maximization. `Hq` is the fixed-point filter with the largest stopband attenuation among the trials. The number of Monte Carlo trials `N` defaults to 1.

You must have the Fixed-Point Toolbox software installed to use this function.

**Examples** Maximize stopband attenuation for 16-bit fixed-point filter.

```
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',0.4,0.45,0.5,60);
Hd = design(Hf,'equiripple');
WL = 16; % use 16 bits to represent coefficients
Hq = maximizestopband(Hd,WL);
% Compare stopband attenuation
md = measure(Hd);
mq = measure(Hq);
hfvf=fvtool(Hd,Hq,'showreference','off');
legend(hfvf,'Floating-point Filter','Fixed-point Filter');
```



## See Also

`constraincoeffw1` | `design` | `fdesign` | `minimizecoeffw1` | `measure` | `rand`

## Tutorials

- “Fixed-Point Concepts”

# maxstep

---

**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  $0 < \text{ha.stepsize} < \text{mumaxmse}/2$ .

`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
```

## maxstep

---

### **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.

<b>Lowpass Filter Specification</b>	<b>Description</b>
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 $F_{pass}$ and $F_{stop}$ .

In contrast, when you use a bandstop design object, `measure(hd)` returns these specifications for the resulting bandstop filter.

<b>Bandstop Filter Specification</b>	<b>Description</b>
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.

<b>Bandstop Filter Specification</b>	<b>Description</b>
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.
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 $F_{pass}$ and $F_{stop}$ .

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  $F_s$ , 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'
```

## measure

---

```
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  $F_s$  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	Coefficient Mapping Support in <code>realizemdl</code>
<code>mfilt.cascade</code>	Cascade multirate filters to form another filter	Supported
<code>mfilt.cicdecim</code>	Cascaded integrator-comb decimator	Not supported
<code>mfilt.cicinterp</code>	Cascaded integrator-comb interpolator	Not supported
<code>mfilt.farrowsrc</code>	Multirate Farrow filter	Supported.
<code>mfilt.fftfirinterp</code>	Overlap-add FIR polyphase interpolator	Not supported

<b>Filter Structure String</b>	<b>Description of Resulting Multirate Filter</b>	<b>Coefficient Mapping Support in realizedml</b>
<code>mfilt.firdecim</code>	Direct-form FIR polyphase decimator	Supported
<code>mfilt.firfracdecim</code>	Direct-form FIR polyphase fractional decimator	Not supported
<code>mfilt.firfracinterp</code>	Direct-form FIR polyphase fractional interpolator	Not supported
<code>mfilt.firinterp</code>	Direct-form FIR polyphase interpolator	Supported
<code>mfilt.firsrc</code>	Direct-form FIR polyphase sample rate converter	Supported
<code>mfilt.firtdecim</code>	Direct-form transposed FIR polyphase decimator	Supported
<code>mfilt.holdinterp</code>	FIR hold interpolator	Not supported
<code>mfilt.iirdecim</code>	IIR decimator	Supported
<code>mfilt.iirinterp</code>	IIR interpolator	Supported
<code>mfilt.linearinterp</code>	FIR Linear interpolator	Supported
<code>mfilt.iirwdfdecim</code>	IIR wave digital filter decimator	Supported
<code>mfilt.iirwdfinterp</code>	IIR wave digital filter interpolator	Supported

### 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.

---

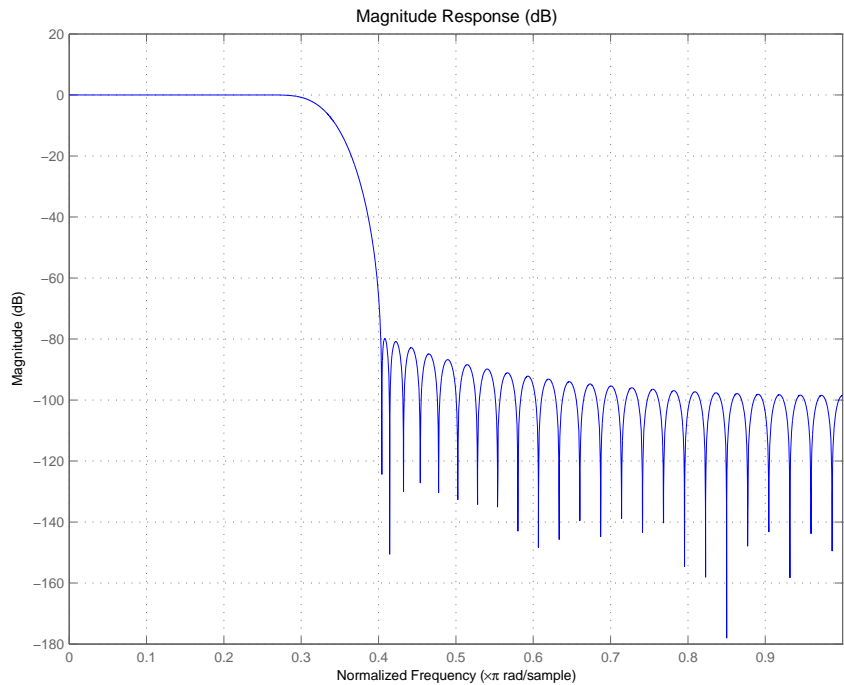
## Examples

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);  
fvtool(hm);
```

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.

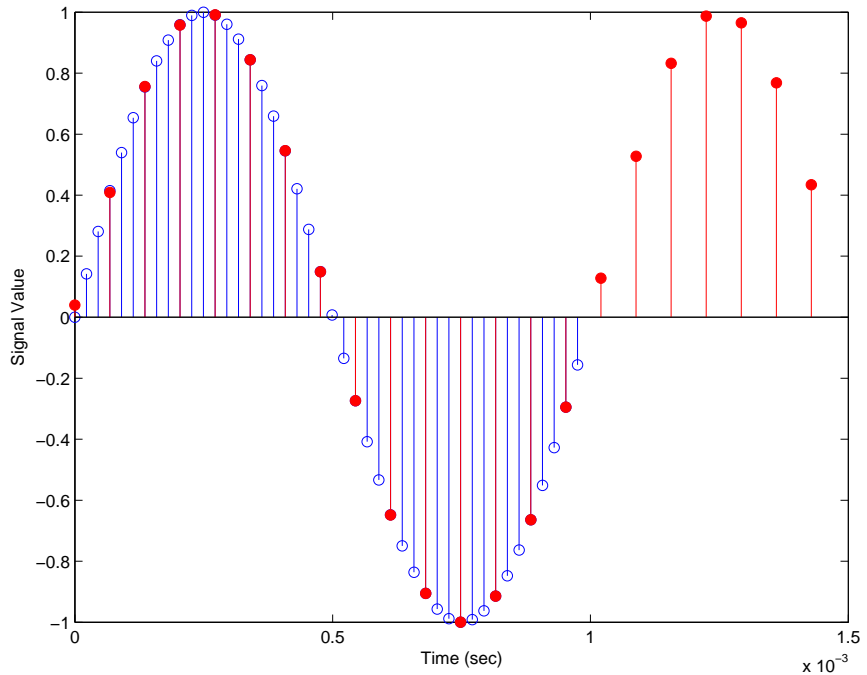
```
isstable(hm)
% Returns a 1 for true
```

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.



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. 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
<code>r</code>	Decimation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. Default value is 2.
<code>m</code>	Differential delay. Changes both the shape and number of nulls in the filter response. Also affects the null locations. Increasing <code>m</code> increases the number and sharpness of the nulls and response between nulls. Generally, one or two work best as values for <code>m</code> . Default is 1.
<code>n</code>	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.
<code>iwl</code>	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	<p>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.</p> <p>When you elect to specify wlps as an input argument, the SectionWordLengthMode property automatically switches from the default value of MinWordLengths to SpecifyWordLengths.</p>

## 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 2-1014 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 2-1014.

**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.

<b>Name</b>	<b>Values</b>	<b>Default</b>	<b>Description</b>
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 2-1006 below for details.

## mfilt.cicdecim

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.

<b>Name</b>	<b>Values</b>	<b>Default</b>	<b>Description</b>
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. <b>PersistentMemory</b> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When <b>PersistentMemory</b> is <b>false</b> , you cannot access the filter states. Setting <b>PersistentMemory</b> to <b>true</b> reveals the <b>States</b> property so you can modify the filter states.

# mfilt.cicdecim

Name	Values	Default	Description
SectionWordLengths	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 SectionWordLengthMode is SpecifyWordLengths.
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

Name	Values	Default	Description
			<p>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 <code>SpecifyWordLengths</code>, you provide the word length for each section. In addition, choosing <code>SpecifyWordLengths</code> exposes the <code>SectionWordLengths</code> property for you to modify as needed.</p>
States	filtstates.cic object	m+1-by-n matrix of zeros, after you call function <code>int</code> .	<p>Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. <code>m</code> is the differential delay of the comb section and <code>n</code> 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 <code>PersistentMemory</code> is true. Refer to the</p>

Name	Values	Default	Description
			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

$$B_{accum} = \text{ceil}(N_{secs}(\text{Log}_2(D \times M)) + \text{InputWordLength})$$

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 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  $2*(\text{NumberOfSections})$ , 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 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)
set(hm,'InputFracLength',0); % Integer input specified.
y=filter(hm,x);
sts=int(hm.states)
```

`STS` is an integer matrix that `int` returns from the contents of the `filtstates.cic` object in `hm`.

## Design Considerations

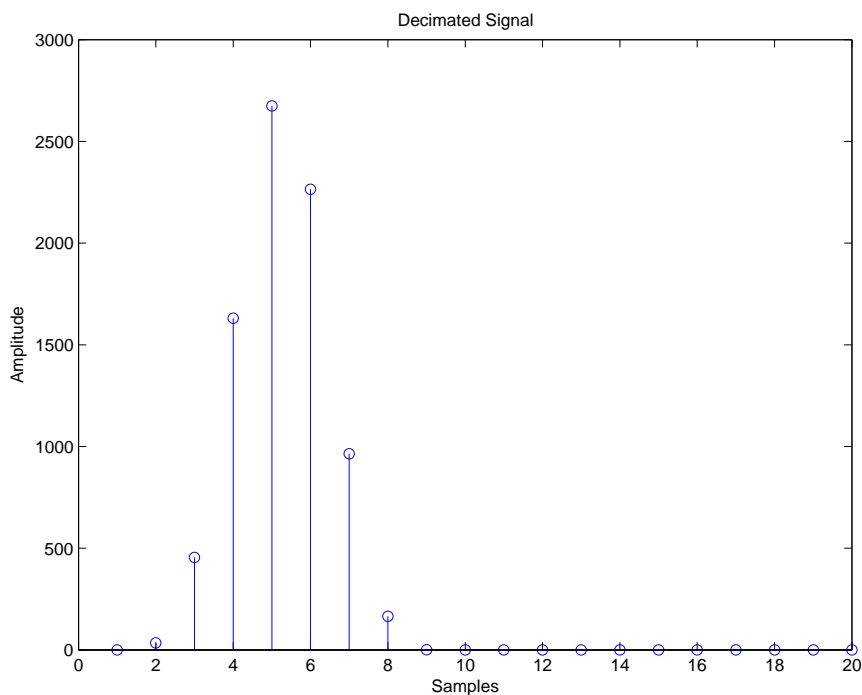
When you design your CIC decimation filter, remember the following general points:

- The filter output spectrum has nulls at  $\omega = k * 2\pi/rm$  radians,  $k = 1,2,3\dots$
- 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:

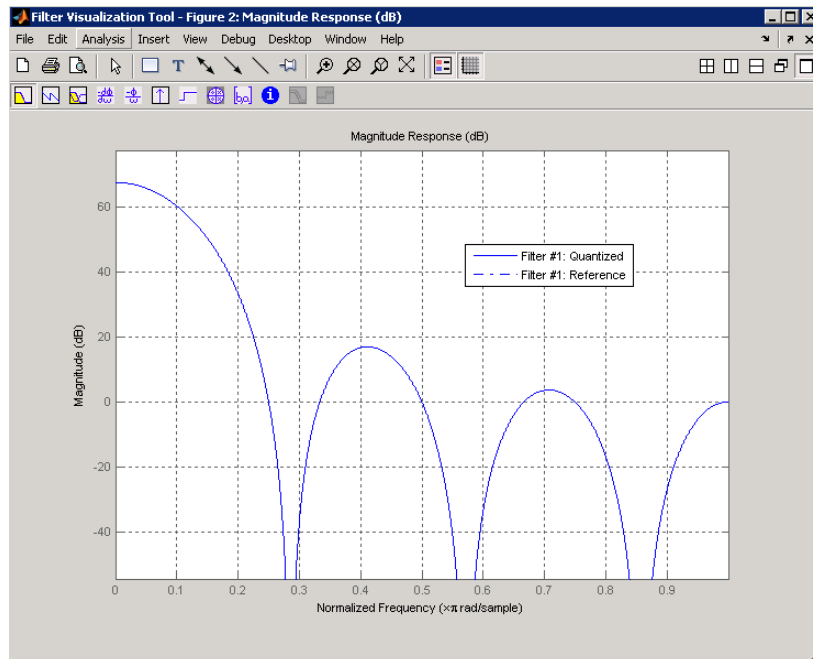
```
m = 2; % Differential delays in the filter.
n = 4; % Filter sections
r = 8; % Decimation factor
x = int16(zeros(160,1)); x(1) = 1; % Create a 160-point
                                     % impulse signal.
hm = mfilt.cicdecim(r,m,n); % Expects 16-bit input
                               % by default.
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.

```

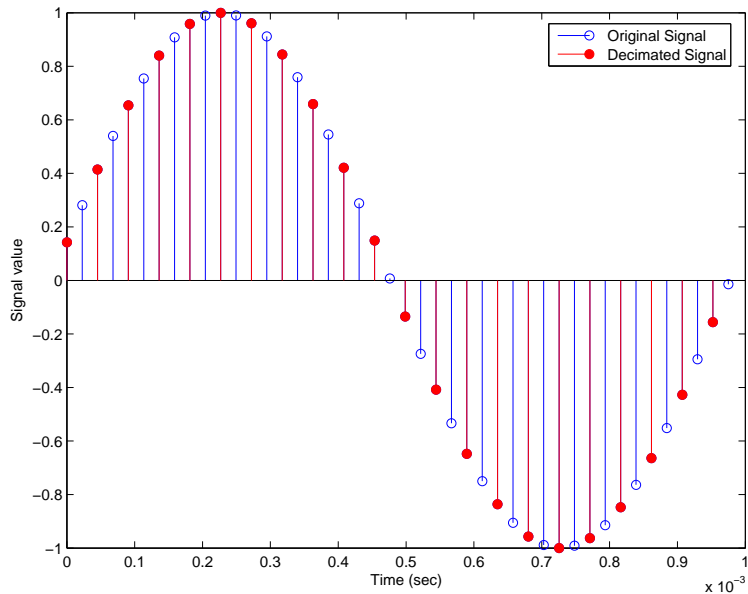
r = 2; % Decimation factor.
hm = mfilt.cicdecim(r); % Use default NumberOfSections &
% DifferentialDelay property values.
fs = 44.1e3; % Original sampling frequency: 44.1kHz.
n = 0:10239; % 10240 samples, 0.232 second long signal.
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1kHz.

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');
```

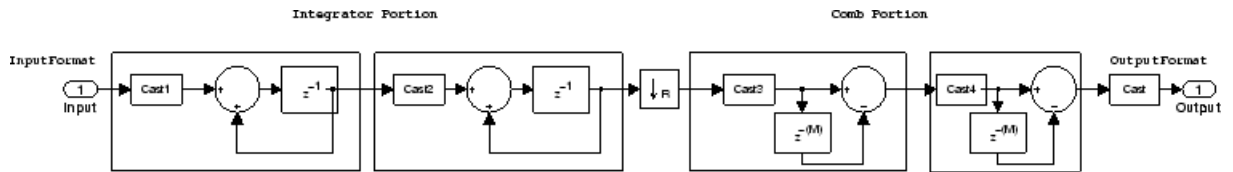


## Algorithm

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

# mfilt.cicinterp

---

**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`.

<b>Input Arguments</b>	<b>Description</b>
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. The default value 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. 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	<p>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.</p> <p>When you elect to specify WLPS as an input argument, the SectionWordLengthMode property automatically switches from the default value of MinWordLengths to SpecifyWordLengths.</p>

`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  $j$ th section, is given by

$$G_j = \begin{cases} 2^j, & j = 1, 2, \dots, N \\ \frac{2^{2N-j}(RM)^{j-N}}{R}, & j = N + 1, \dots, 2N \end{cases}$$

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

$$W_N = B_{in} + N - 1 \text{ if } M = 1$$

The conversions denoted by the cast blocks in the integrator diagrams in “Algorithm” on page 2-1026 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**

The following table lists the properties for the filter with a description of each.

<b>Name</b>	<b>Values</b>	<b>Default</b>	<b>Description</b>
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, SpecifyWordLengths, SpecifyPrecision	FullPrecision	Set the usage mode for the filter. Refer to “Usage Modes” on page 2-1022 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.

# mfilt.cicinterp

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.
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.

Name	Values	Default	Description
			States that the filter does not change are not affected. When <code>PersistentMemory</code> is <code>false</code> , you cannot access the filter states. Setting <code>PersistentMemory</code> to <code>true</code> reveals the <code>States</code> property so you can modify the filter states.
SectionWordLengths	Any integer or a vector of length $2^N$ , where $N$ is a positive integer.  This property only applies when the <code>FilterInternals</code> is <code>SpecifyWordLengths</code> .	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 <code>SectionWordLengths</code> as a scalar or vector of length $2*n$ , where $n$ is the number of sections. When <code>SectionWordLengths</code> is a scalar, the scalar value is applied to each filter section. When <code>SectionWordLengths</code> 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

# mfilt.cicinterp

Name	Values	Default	Description
			SectionWordLengthMode is SpecifyWordLengths.
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 usage modes which are set using the FilterInternals property:

- **FullPrecision** — In this mode, the word and fraction lengths of the filter sections and outputs are automatically selected for you. The output and last section word lengths are set to:

$$\text{wordlength} = \text{ceil}(\log_2((RM)^N / R)) + I,$$



where  $R$  is the interpolation factor,  $M$  is the differential delay,  $N$  is the number of filter sections, and  $I$  denotes the input word length.

- **MinWordLengths** — In this mode, you specify the word length of the filter output in the `OutputWordLength` property. The word lengths of the filter sections are automatically set in the same way as in the `FullPrecision` mode. The section fraction lengths are set to the input fraction length. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.
- **SpecifyWordLengths** — In this mode, you specify the word lengths of the filter sections and output in the `SectionWordLengths` and `OutputWordLength` properties. The fraction lengths of the filter sections are set such that the spread between word length and fraction length is the same as in full-precision mode. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.
- **SpecifyPrecision** — In this mode, you specify the word and fraction lengths of the filter sections and output in the `SectionWordLengths`, `SectionFracLengths`, `OutputWordLength`, and `OutputFracLength` properties.

### 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 the `int` method to assign the states. As an example, here are the states for a CIC interpolator `hm` before and after filtering data:

```
x = fi(cos(pi/4*[0:99]),true,16,0); % Fixed-point input data
hm = mfilt.cicinterp(2,1,2,16,16,16);
% get initial states-all zero
sts=int(hm.states)
set(hm,'InputFracLength',0); % Integer input specified
y=filter(hm,x);
sts=int(hm.states)
%sts =
%
%      -1      -1
%      -1      -1
```

## 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,  $k = 1,2,3,\dots$
- 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.

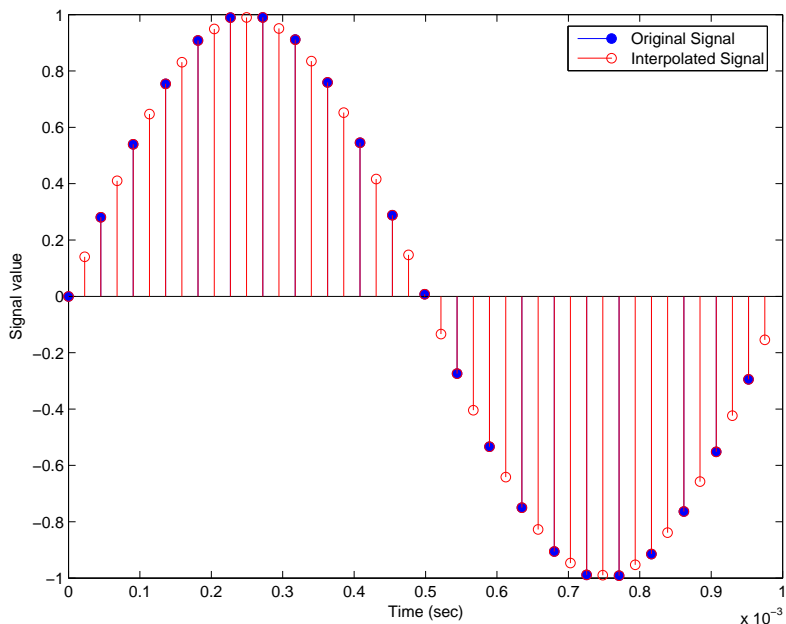
y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% 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.



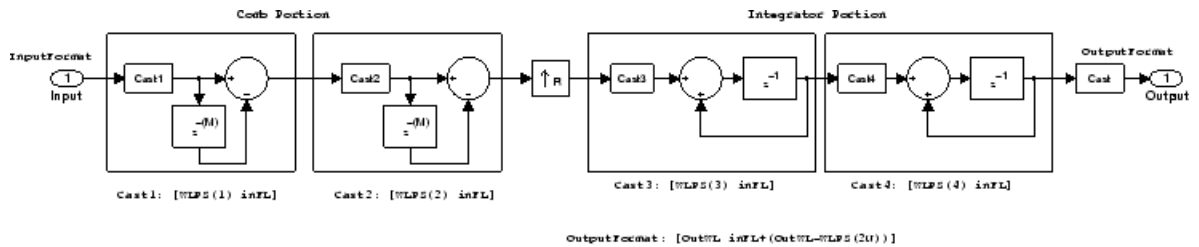
Use the filter visualization tool (FVTool) to plot the response of the interpolator object. For example, to plot the response of an interpolator with an interpolation factor of 7, 4 sections, and 1 differential delay, do something like the following:

```
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 ( $n = 2$ ).  $f_s$  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

# mfilt.farrowsrc

---

**Purpose** Sample rate converter with arbitrary conversion factor

**Syntax**

```
hm = mfilt.farrowsrc(L,M,C)
hm = mfilt.farrowsrc
hm = mfilt.farrowsrc(1,...)
```

**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(1,...)` 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.

<b>Name</b>	<b>Values</b>	<b>Description</b>
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object.
Arithmetic	String	Reports the arithmetic 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.

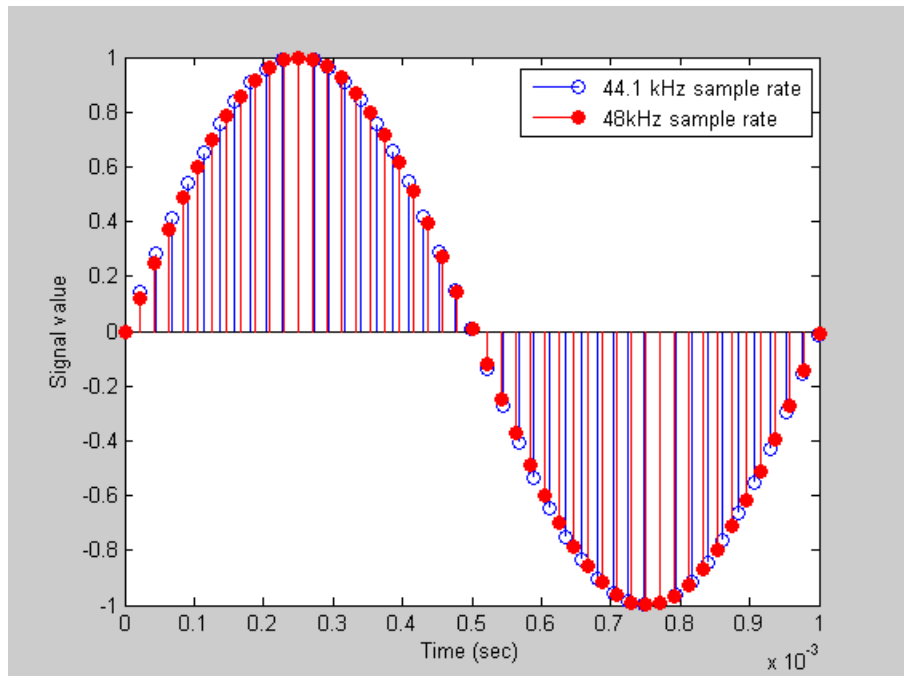
## Example

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:





# mfilt.fftfirinterp

---

**Purpose** Overlap-add FIR polyphase interpolator

**Syntax**  
`hm = mfilt.fftfirinterp(1,num,b1)`  
`hm = mfilt.fftfirinterp`  
`hm = mfilt.fftfirinterp(1,...)`

**Description** `hm = mfilt.fftfirinterp(1,num,b1)` 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  $[b1 + \text{ceil}(\text{length}(\text{num})/1) - 1]$ . It is to your advantage to choose `b1` 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
<code>1</code>	Interpolation factor for the filter. <code>1</code> specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for <code>1</code> it defaults to <code>2</code> .
<code>num</code>	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When <code>num</code> is not provided as an input, <code>fftfirinterp</code> uses a lowpass Nyquist filter with gain equal to <code>1</code> and cutoff frequency equal to $\pi/1$ by default.
<code>b1</code>	Length of each block of input data used in the filtering. <code>b1</code> must be an integer. When you omit input <code>b1</code> , it defaults to <code>100</code> .

`hm = mfilt.fffirinterp` constructs the filter using the default values for `l`, `num`, and `bl`.

`hm = mfilt.fffirinterp(l,...)` constructs the filter using the input arguments you provide and defaults for the argument you omit.

## **mfilt.fffirinterp 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.fffirinterp` objects. The next table describes each property for an `mfilt.fffirinterp` filter object.

<b>Name</b>	<b>Values</b>	<b>Description</b>
FilterStructure		Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> 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.

# mfilt.fftfirinterp

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.

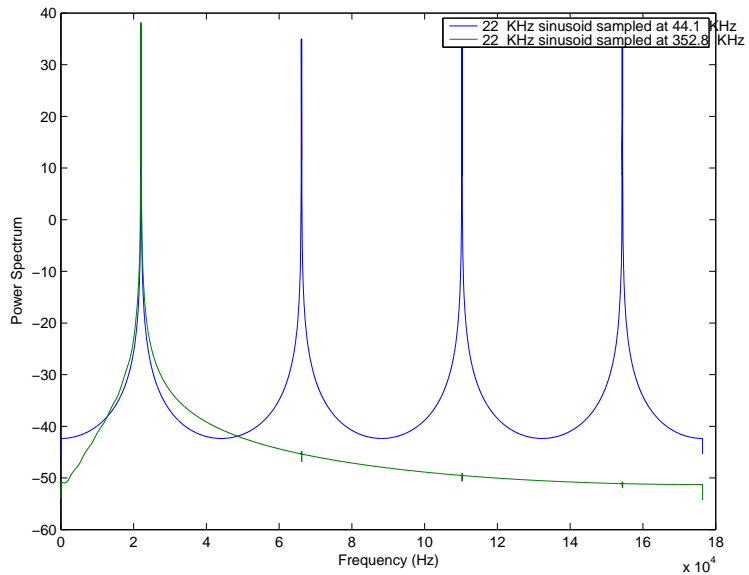
```
l = 8; % Interpolation factor
hm = mfilt.fftfirinterp(l); % We use the default filter
n = 8192; % Number of points
hm.blocklength = n; % Set block length to number of points
fs = 44.1e3; % Original sample freq: 44.1 kHz.
n = 0:n-1; % 0.1858 secs of data
x = sin(2*pi*n*22e3/fs); % Original signal, sinusoid at 22 kHz
y = filter(hm,x); % Interpolated sinusoid
xu = 1*upsample(x,8); % Upsample to compare--the spectrum
% does not change
[px,f]=periodogram(xu,[],65536,1*fs); % Power spectrum of original
% signal
[py,f]=periodogram(y,[],65536,1*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  $\pi/m$  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`.

<b>Input Argument</b>	<b>Description</b>
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 $\pi/m$ 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.

<b>Name</b>	<b>Values</b>	<b>Description</b>
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.
DecimationFactor	Integer	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of

## mfilt.firdecim

---

Name	Values	Description
		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 <code>mfilt</code> 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 — $\text{InputOffset} = \text{mod}(\text{length}(nx), m)$ 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. <code>PersistentMemory</code> set to <code>false</code> returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to



Name	Values	Description
		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” on page 3-116.

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 <code>fixed</code> 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 <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

<b>Name</b>	<b>Values</b>	<b>Description</b>
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.
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.

# mfilt.firdecim

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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.
RoundMode	[convergent], ceil, fix, floor, nearest, 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). <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul>

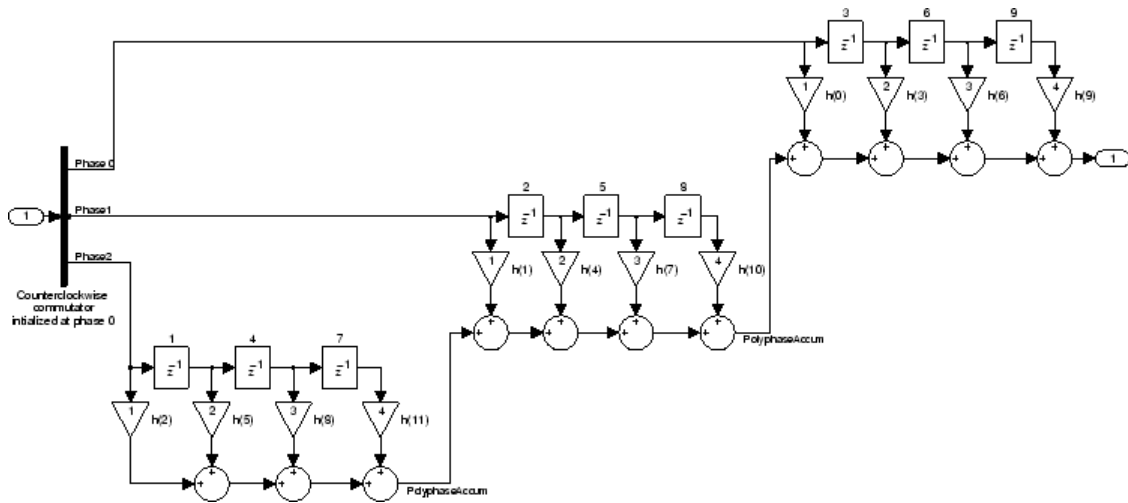
Name	Values	Description
		The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. 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 Fixed-Point Toolbox documentation. 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`.

# 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

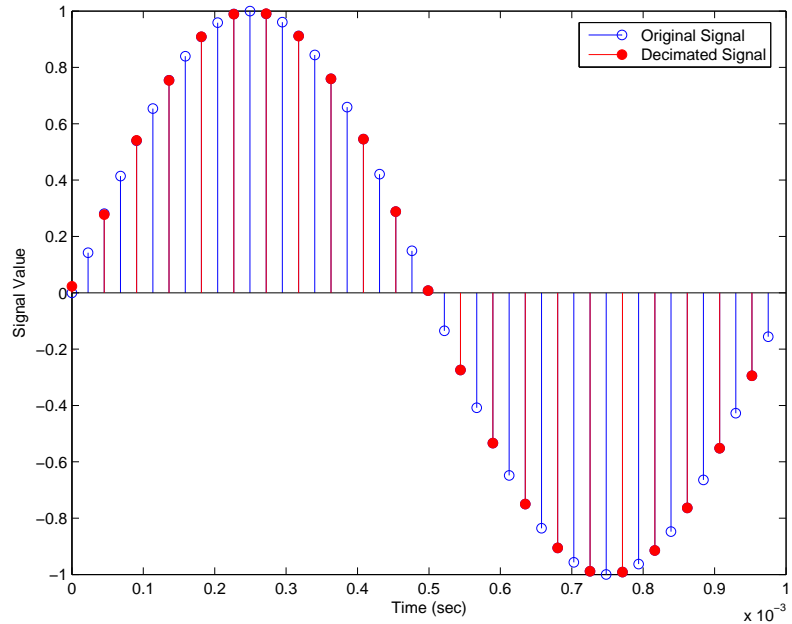
Convert an input signal from 44.1 kHz to 22.05 kHz using decimation by a factor of 2. In the figure that appears after the example code, you see the results of the decimation.

```
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`

# mfilt.firfracdecim

---

**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 1 and cutoff frequency of  $\pi/\max(l, m)$  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  $1/m$  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
<code>l</code>	Interpolation factor for the filter. It must be an integer. When you do not specify a value for <code>l</code> it defaults to 2.
<code>num</code>	Vector containing the coefficients of the FIR lowpass filter used for decimation. When <code>num</code> is not provided as an input, <code>firfracdecim</code> uses a lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to $\pi/\max(l, m)$ by default.



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 1 + 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 <code>mfilt</code> object.
InputOffset	Integers	Contains the number of input data samples processed without generating an output sample. The default value is 0.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for interpolation.
RateChangeFactors	[1,m]	Reports the decimation (m) and interpolation (1) factors for the filter object. Combining these factors results in the final rate change for the signal.

## mfilt.firfracdecim

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 $(1h-1)*m+(l-1)*(1o+mo)$ where $1h$ is the length of each subfilter, and $l$ and $m$ are the interpolation and decimation factors. $1o$ and $mo$ , the input and output delays between each interpolation phase, are integers from Euclid's theorem such that $1o*1-mo*m = -1$ (refer to the reference for more details). Use <code>euclidfactors</code> to get $1o$ and $mo$ for an <code>mfilt.firfracdecim</code> object

### Example

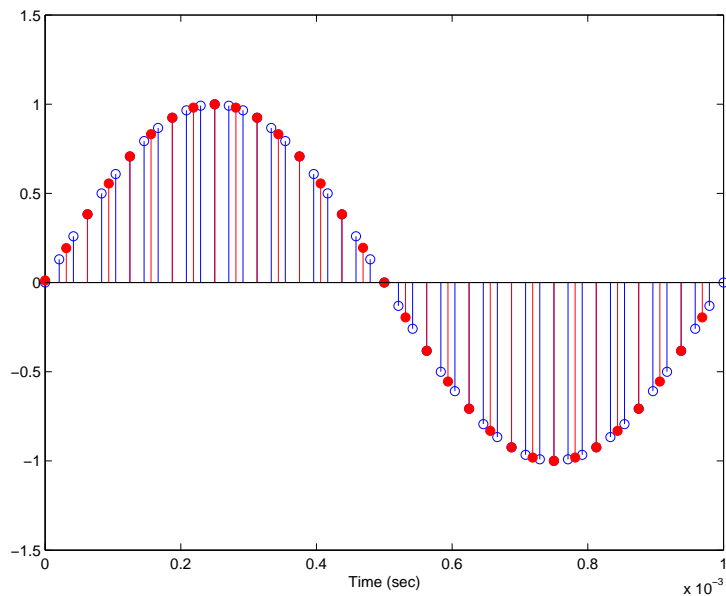
To demonstrate `firfracdecim`, perform a fractional decimation by a factor of  $2/3$ . This is one way to downsample a 48 kHz signal to 32 kHz, commonly done in audio processing.

```

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.



# **mfilt.firfracdecim**

---

## **See Also**

`mfilt.firsrc`, `mfilt.firfracinterp`, `mfilt.firinterp`,  
`mfilt.firdecim`

## **References**

Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley & Sons, Ltd., 1994

**Purpose** Direct-form FIR polyphase fractional interpolator

**Syntax** `hm = mfilter.firfracinterp(1,m,num)`

**Description** `hm = mfilter.firfracinterp(1,m,num)` returns a direct-form FIR polyphase fractional interpolator `mfilter` object. `1` is the interpolation factor. It must be an integer. If not specified, `1` defaults to `3`.

`m` is the decimation factor. Like `1`, 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 `1`, `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 `1` and cutoff frequency of  $\pi/\max(1,m)$  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  $1/m$  of the input rate. For proper interpolation, you specify `1` to be greater than `m`. By default, the resulting interpolation factor is  $3/2$  when you do not provide `1` and `m` in the calling syntax.

### Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
<code>1</code>	Interpolation factor for the filter. <code>1</code> specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for <code>1</code> it defaults to <code>3</code> .
<code>num</code>	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When <code>num</code> is not provided as an input, <code>firfracinterp</code> uses a

# mfilt.firfracinterp

Input Argument	Description
	lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to $\pi/\max(1,m)$ 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 1 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 <code>mfilt</code> object.
Numerator		Vector containing the coefficients of the FIR lowpass filter used for interpolation.
RateChangeFactors	[1,m]	Reports the decimation (m) and interpolation (1) 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 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.

```

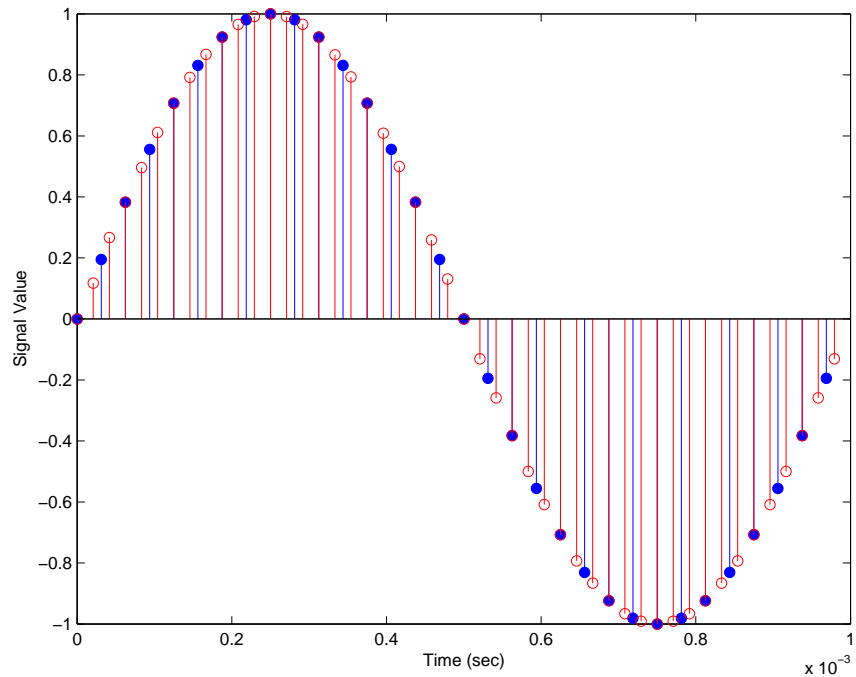
l = 3; m = 2; % Interpolation/decimation factors.
hm = mfilt.firfracinterp(l,m); % We use the default filter
fs = 32e3; % Original sample freq: 32 kHz.
n = 0:6799; % 6800 samples, 0.212 second long signal
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz
y = filter(hm,x); % 10200 samples, 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')
```

# mfilt.firfracinterp

```
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`



**Purpose** FIR filter-based interpolator

**Syntax**  
`Hm = mfilt.firinterp(L)`  
`Hm = mfilt.firinterp(L,num)`

**Description** `Hm = mfilt.firinterp(L)` returns a FIR polyphase interpolator object `Hm` with an interpolation factor of `L` and gain equal to `L`. `L` defaults to 2 if unspecified.

`Hm = mfilt.firinterp(L,num)` uses the values in the vector `num` as the 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
1	Interpolation factor for the filter. 1 specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for 1 it defaults to 2.
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When <code>num</code> is not provided as an input, <code>firinterp</code> uses a lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to $\pi/1$ by default. The default length for the Nyquist filter is $24 \times 1$ . 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 <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>mfilt</code> object. Describes the signal flow for the filter object.
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.

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. <code>PersistentMemory</code> set to <code>false</code> returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to <code>true</code> allows you to modify the <code>States</code> 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” on page 3-116.

## mfilt.firinterp

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.

<b>Name</b>	<b>Values</b>	<b>Description</b>
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.

# mfilt.firinterp

Name	Values	Description
OverflowMode	saturate, [wrap]	<p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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.</p>
RoundMode	[convergent], <code>ceil</code> , <code>fix</code> , <code>floor</code> , <code>nearest</code> , <code>round</code>	<p>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).</p> <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul>

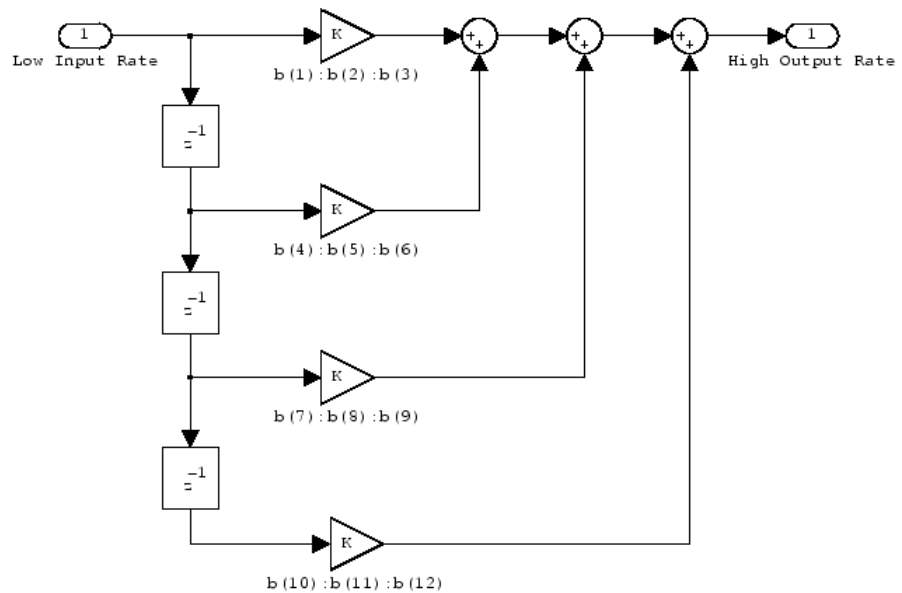
Name	Values	Description
		The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. 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 Fixed-Point Toolbox documentation.

**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.

# mfilt.firinterp



## 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.

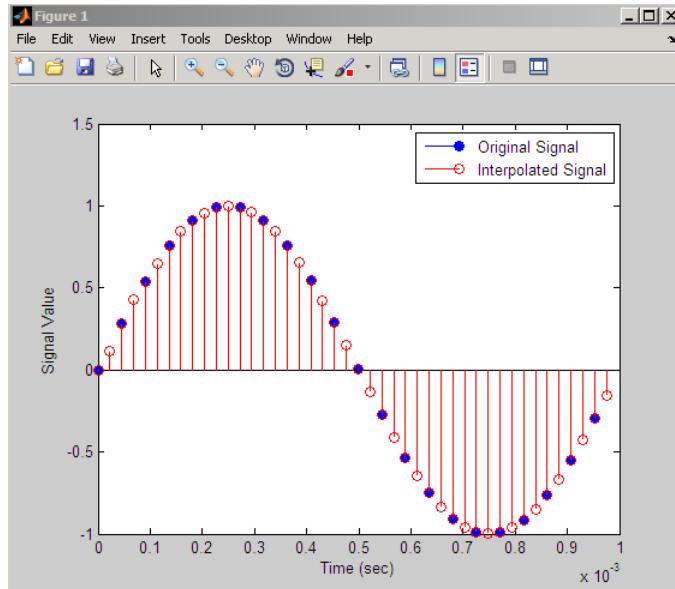
hold on;

% Plot interpolated signal (44.1 kHz) in red
stem(n(1:44)/(fs*L),y(25:68),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```



```
legend('Original Signal','Interpolated Signal');
```

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(1,m,num)`

**Description** `hm = mfilt.firsrc(1,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 1 and cutoff frequency of  $\pi/\max(1,m)$  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  $1/m$  to the input signal. For proper rate changing to occur, `l` and `m` must be relatively prime — meaning the ratio  $1/m$  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
1	Interpolation factor for the filter. 1 specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for 1, 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, <code>mfilt.firsrc</code> uses a lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to $\pi/\max(1,m)$ 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 1 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

## **mfilt.firsrc**

---

input arguments for creating `mfilt.firsrc` objects. The next table describes each property for an `mfilt.firsrc` filter object.

<b>Name</b>	<b>Values</b>	<b>Description</b>
Arithmetic	[Double], single, fixed	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . 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 <code>mfilt</code> 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 — $\text{InputOffset} = \text{mod}(\text{length}(nx), m)$ 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. <code>PersistentMemory</code> set to <code>false</code> returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to <code>true</code> allows you to modify the <code>States</code> , <code>InputOffset</code> , and <code>PolyphaseAccum</code> properties.

Name	Values	Description
RateChangeFactors	Positive integers. [2 3]	Specifies the interpolation and decimation factors [1 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” on page 3-116.

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.

Name	Values	Description
Arithmetic	fixed for fixed-point filters	Setting this to <b>fixed</b> 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 <b>false</b> 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, <b>FullPrecision</b> , sets automatic word and fraction length determination by the filter. <b>SpecifyPrecision</b> 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.

Name	Values	Description
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 [1 m] (the rate change factors) for changing the input sample rate by nonintegral amounts.

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	<p>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).</p> <ul style="list-style-type: none"><li>• <code>ceil</code> - Round toward positive infinity.</li><li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <code>fix</code> - Round toward zero.</li><li>• <code>floor</code> - Round toward negative infinity.</li><li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li><li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>



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 Fixed-Point Toolbox documentation. 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  $m = 160$ .

```

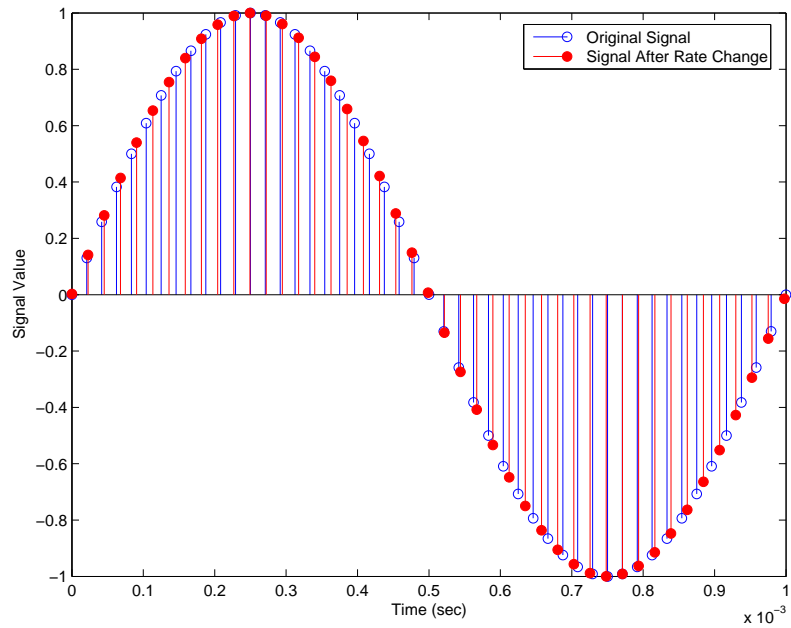
l = 147; m = 160;           % Interpolation/decimation factors.
hm = mfilt.firsrc(l,m);    % Use the default FIR filter.
fs = 48e3;                 % Original sample freq: 48 kHz.
n = 0:10239;               % 10240 samples, 0.213 seconds long.
x = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 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  $\pi/m$  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  $\pi/m$  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`.

# mfilt.firtdecim

---

Input Argument	Description
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, <code>firtdecim</code> uses a lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to $\pi/m$ 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. <code>m</code> 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 <code>m</code> 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. <code>m</code> 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 — $\text{InputOffset} = \text{mod}(\text{length}(nx), m)$ 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” on page 3-116.

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.

## mfilt.firtdecim

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 <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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 ( <code>PolyphaseAccum</code> ) and the adders in the filter that the user controls and that may introduce quantization effects when <code>FilterInternals</code> is set to <code>SpecifyPrecision</code> .
RoundMode	[convergent], ceil, fix, floor, nearest, 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). <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> </ul>

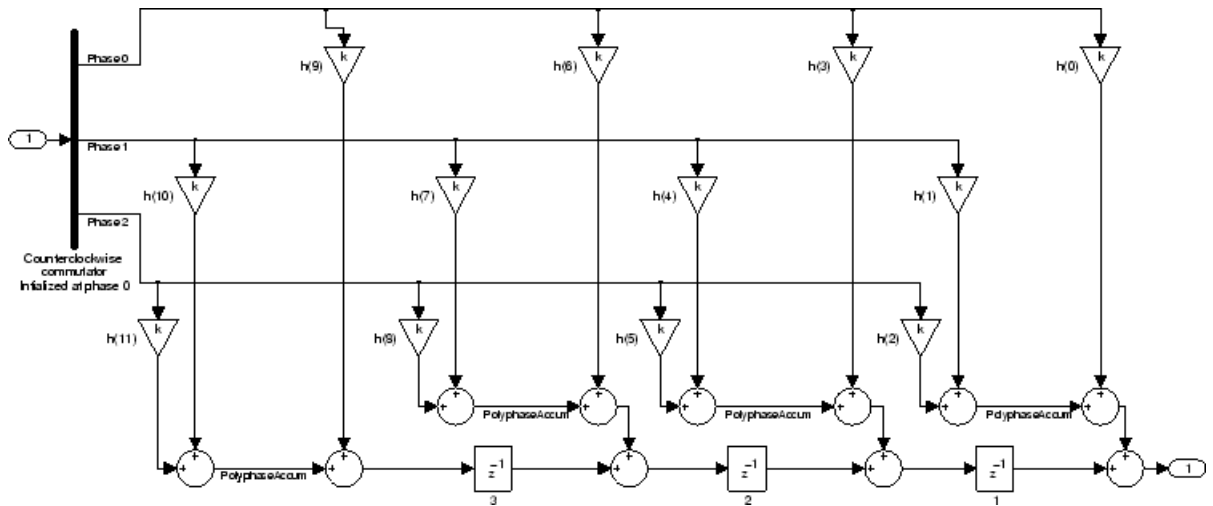
# mfilt.firtdecim

Name	Values	Description
		<ul style="list-style-type: none"><li>• round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>
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 Fixed-Point Toolbox documentation. 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];
```

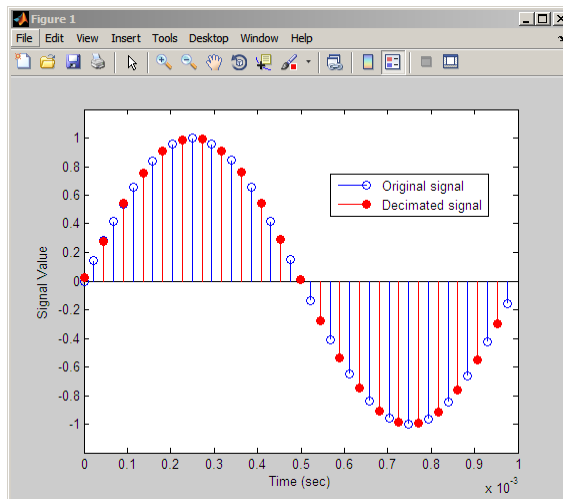
## Examples

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.
```

# mfilt.firtdecim

```
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1 kHz.  
axis([0 0.001 -1.2 1.2]);  
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');  
legend('Original signal','Decimated signal','location','best');
```



## See Also

`mfilt.firdecim`, `mfilt.firfracdecim`, `mfilt.cicdecim`

**Purpose** FIR hold interpolator

**Syntax** `hm = mfilt.holdinterp(1)`

**Description** `hm = mfilt.holdinterp(1)` returns the object `hm` that represents a hold interpolator with the interpolation factor 1. To work, 1 must be an integer. When you do not include 1 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`.

# **mfilt.holdinterp**

---

<b>Input Argument</b>	<b>Description</b>
1	Interpolation factor for the filter. 1 specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for 1 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.

<b>Name</b>	<b>Values</b>	<b>Description</b>
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 <code>mfilt</code> object.
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'	Determines whether the filter states are restored to zero for each filtering operation.
States	Double or single array	Filter states. <code>states</code> defaults to a vector of zeros that has length equal to <code>nstates</code> ( <code>hm</code> ). Always available, but visible in the display only when <code>PersistentMemory</code> 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” on page 3-116.

## mfilt.holdinterp

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 <code>mfilt</code> 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.
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 <code>fi</code> objects, with the



Name	Values	Description
		associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation.

## Filter Structure

Hold interpolators do not have filter coefficients and their filter structure is trivial.

## 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.

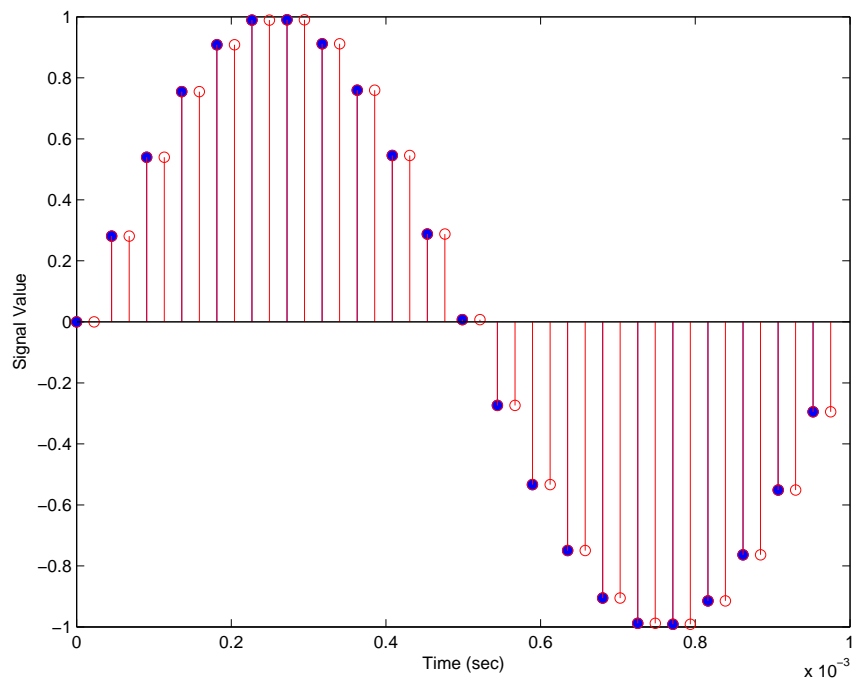
```

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`.

# mfilt.holdinterp



## 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.

Although the first example shows how to construct an IIR decimators explicitly, one usually constructs an IIR decimators filter as a result of designing an decimators, as shown in the subsequent examples.

## Examples

When the coefficients are known, you can construct the IIR decimator directly using `mfilt.iirdecim`. For example, if the filter's coefficients are `[0.6 0.5]` for the first phase in the first stage, `0.7` for the second phase in the first stage and `0.8` for the third phase in the first stage; as well as `0.5` for the first phase in the second stage and `0.4` for the second phase in the second stage, construct the filter as shown here.

```
Hm = mfilt.iirdecim({[0.6 0.5] 0.7 0.8},{0.5 0.4})
```

Also refer to the “Quasi-Linear Phase Halfband and Dyadic Halfband Designs” section of the “IIR Polyphase Filter Design” demo, `iirallpassdemo` demo.

When the coefficients are not known, use the approach given by the following set of examples. Start by designing 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.
hm = fdesign.decimator(m, 'halfband', 'tw,ast', tw, ast, fs);
```

`hm` contains the specifications for a decimator defined by `tw`, `ast`, `m`, and `fs`.

Use the specification object `hm` to design a `mfilt.iirdecim` filter object.

```
d = design(hm, 'ellip', 'filterstructure', 'iirdecim');
% Note that realizemdl requires Simulink
realizemdl(d) % 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.

```
hm = fdesign.decimator(m, 'halfband', 'tw,ast', tw, ast, fs);
```

Finally, design the filter `d`.

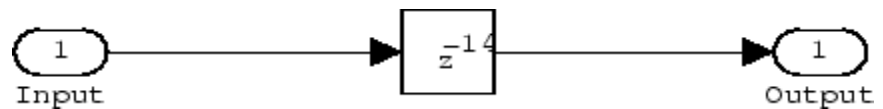
```
d = design(hm, 'iirlinphase', 'filterstructure', 'iirdecim');
```

```
% Note that realizemd1 requires Simulink  
realizemd1(d) % Build model of the filter.
```

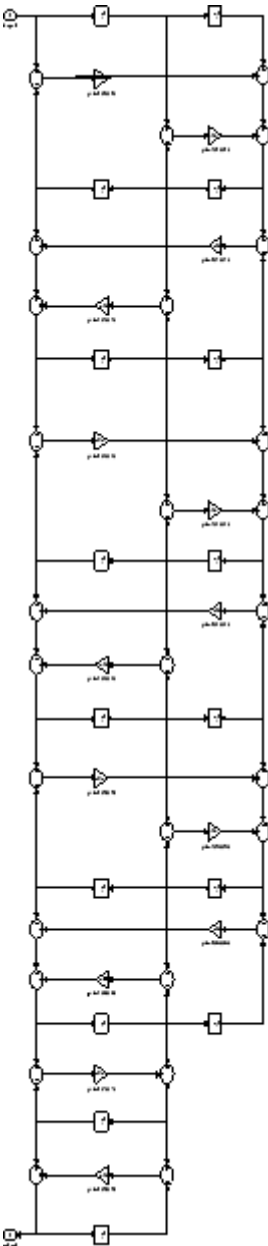
The filter implementation appears in this model, generated by realizemd1 and Simulink.

Given the design specifications shown here

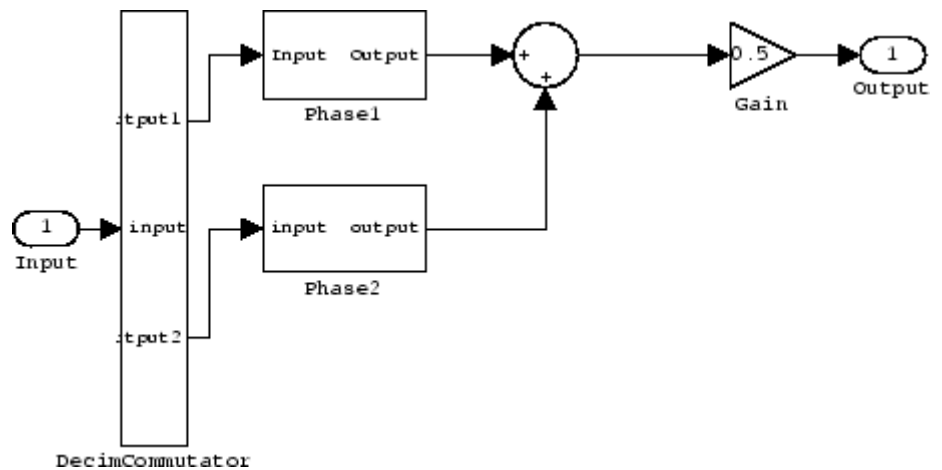
the first phase is a delay section with 0s and a 1 for coefficients and the second phase is a linear phase decimator, shown in the next models.



**Phase 1 model**



**Phase 2 model**



### Overall model

For more information about Multirate Filter Constructors see the “Getting Started with Multirate Filter (MFILT) Objects” demo, `mfiltgettingstarteddemo`.

### See Also

`dfilt.cascadeallpass`, `mfilt`, `mfilt.iirinterp`, `mfilt.iirwdfdecim`

# mfilt.iirinterp

---

**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.

Although the first example shows how to construct an IIR interpolator explicitly, one usually constructs an IIR interpolator filter as a result of designing an interpolator, as shown in the subsequent examples.

## Examples

When the coefficients are known, you can construct the IIR interpolator directly using `mfilt.iirinterp`. In the following example, a cascaded polyphase IIR interpolator filter is constructed using 2 phases for each of three stages. The coefficients are given below:

```
Phase1Sect1=0.0603;Phase1Sect2=0.4126; Phase1Sect3=0.7727;  
Phase2Sect1=0.2160; Phase2Sect2=0.6044; Phase2Sect3=0.9239;
```



Next the filter is implemented by passing the above coefficients to `mfilt.iirinterp` as cell arrays, where each cell array represents a different phase.

```
Hm = mfilt.iirinterp({Phase1Sect1,Phase1Sect2,Phase1Sect3},...
{Phase2Sect1,Phase2Sect2,Phase2Sect3})
```

```
Hm =
```

```

    FilterStructure: 'IIR Polyphase Interpolator'
      Polyphase: Phase1: Section1: 0.0603
                  Section2: 0.4126
                  Section3: 0.7727
                Phase2: Section1: 0.216
                  Section2: 0.6044
                  Section3: 0.9239
  InterpolationFactor: 2
    PersistentMemory: false
```

Also refer to the “Quasi-Linear Phase Halfband and Dyadic Halfband Designs” section of the “IIR Polyphase Filter Design” demo, `iirallpassdemo` demo.

When the coefficients are not known, use the approach given by the following set of examples. Start by designing 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');
```

# mfilt.iirinterp

---

```
% Note that realizemdl requires Simulink
realizemdl(hm) % Build model of the filter.
```

Designing a linear phase halfband interpolator follows the same pattern.

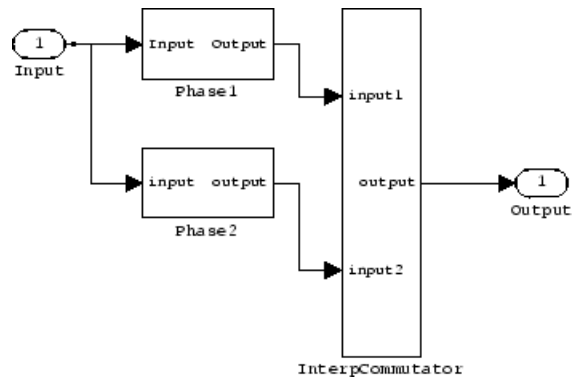
```
tw = 100; % Transition width of filter.
ast= 60; % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l = 2; % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

fdesign.interpolator returns a specification object that stores the design features for an interpolator.

Now perform the actual design that results in an mfilter.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.



For more information about Multirate Filter Constructors see the “Getting Started with Multirate Filter (MFILT) Objects” demo, `mfiltgettingstarteddemo`.

### **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.

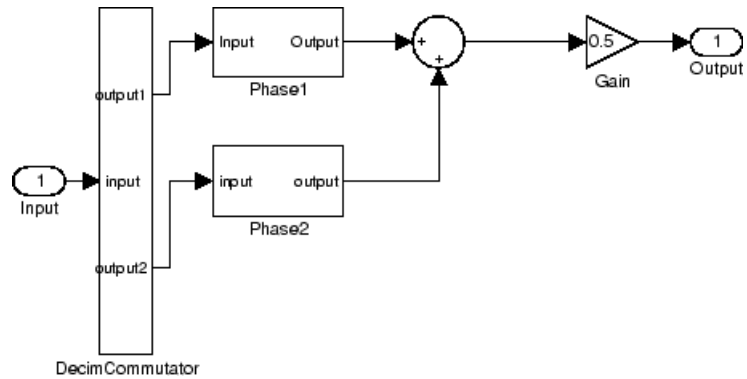
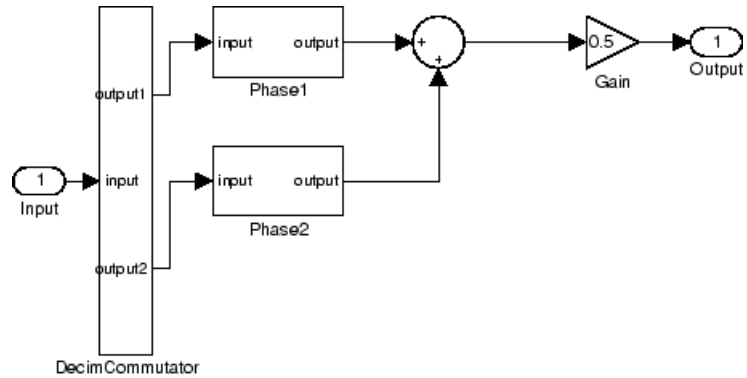
```
tw = 100; % Transition width of filter, 100 Hz.  
ast = 60; % Filter stopband attenuation = 80 dB  
fs = 2000; % Input signal sampling frequency.  
m = 2; % Decimation factor.  
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Use d to design the final filter hm, an `mfilt.iirwdfdecim` object.

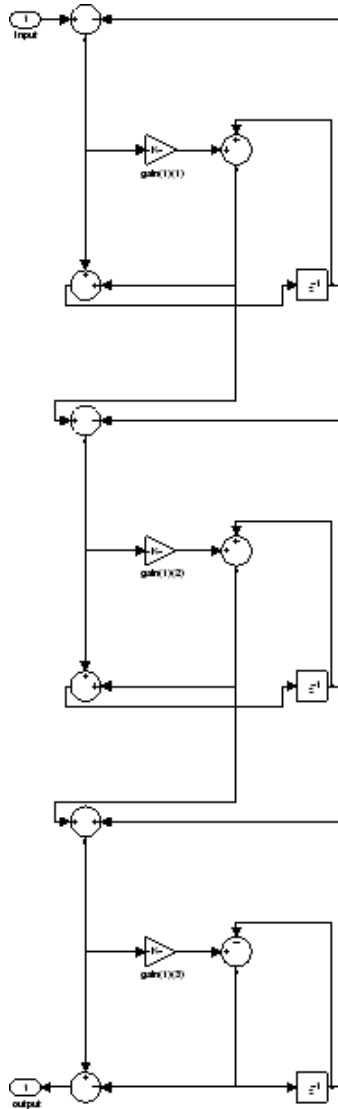
```
hm = design(d,'iirlinphase','filterstructure',...  
'iirwdfdecim');  
% Note that realizemdl requires Simulink  
realizemdl(hm) % Build model of the filter.
```

# mfilt.iirwdfdecim

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.



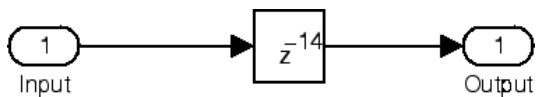
This is the Phase1 subsystem from the halfband model.



## mfilt.iirwdfdecim

---

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 = mfilter.iirwdfinterp(c1,c2,...)
```

## Description

`hm = mfilter.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 `dfilter.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilter.iirwdfinterp` interprets them same way as `mfilter.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 `dfilter.delay` section with latency equal to the number of zeros, rather than a `dfilter.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);
```

## mfilt.iirwdfinterp

---

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(1)`

**Description** `hm = mfilt.linearinterp(1)` returns an FIR linear interpolator `hm` with an integer interpolation factor `1`. Provide `1` as a positive integer. The default value for the interpolation factor is `2` when you do not include the input argument `1`.

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
1	Interpolation factor for the filter. 1 specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for 1 it defaults to 2.

# mfilt.linearinterp

## 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 <code>mfilt</code> object.
InterpolationFactor	Integer	Interpolation factor for the filter. 1 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. <code>states</code> defaults to a vector of zeros that has length equal to <code>nstates(hm)</code> . Always available, but visible in the display only when <code>PersistentMemory</code> 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” on page 3-116.

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 <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.

## mfilt.linearinterp

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.

# mfilt.linearinterp

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	<p>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).</p> <ul style="list-style-type: none"><li>• <b>ceil</b> - Round toward positive infinity.</li><li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li><li>• <b>fix</b> - Round toward zero.</li><li>• <b>floor</b> - Round toward negative infinity.</li><li>• <b>nearest</b> - Round toward nearest. Ties round toward positive infinity.</li><li>• <b>round</b> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li></ul> <p>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.</p>

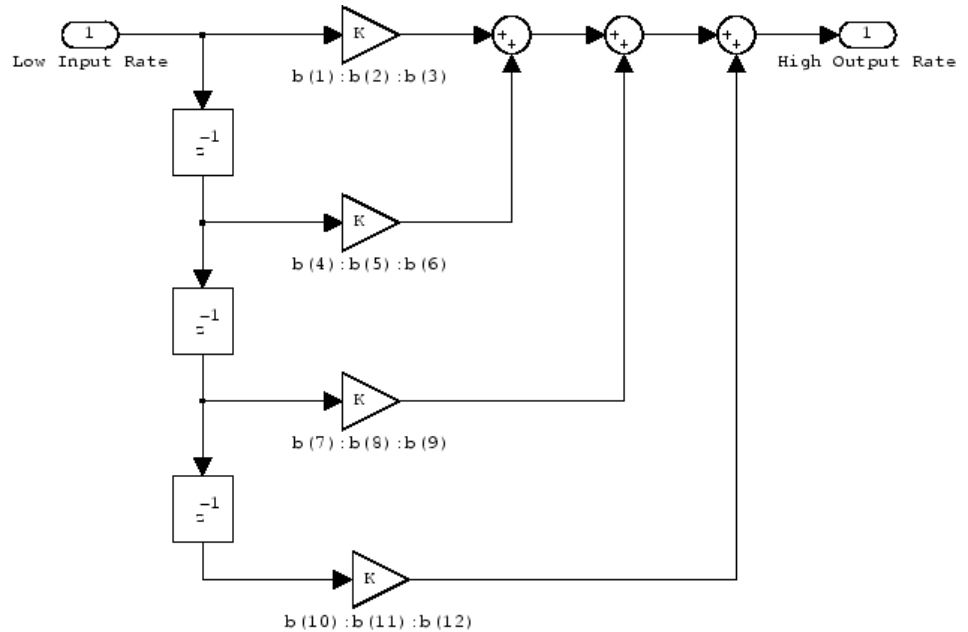


Name	Values	Description
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation. For information about the ordering of the states, refer to the filter structure in the following section.

# mfilt.linearinterp

## Filter Structure

Linear interpolator structures depend on the FIR filter you use to implement the filter. By default, the structure is direct-form FIR.



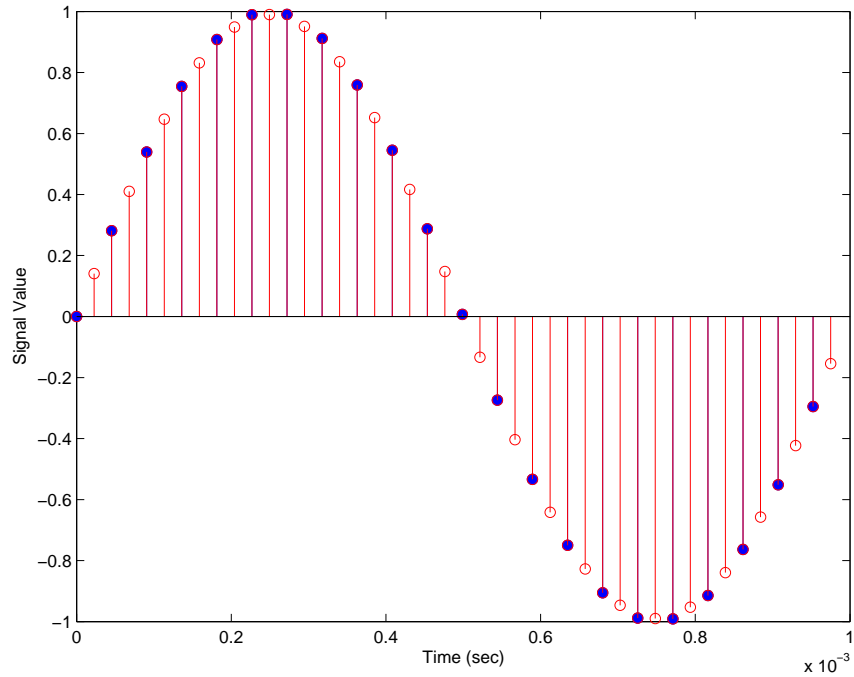
## Examples

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*1),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`

# minimizecoeffwl

---

**Purpose** Minimum wordlength fixed-point filter

**Syntax**

```
Hq = minimizecoeffwl(Hd)
Hq = minimizecoeffwl(Hd,...,'NoiseShaping',NSFlag)
Hq = minimizecoeffwl(Hd,...,'NTrials',N)
Hq = minimizecoeffwl(Hd,...,'Apasstol',Apasstol)
Hq = minimizecoeffwl(Hd,...,'Astoptol',Astoptol)
Hq = minimizecoeffwl(Hd,...,'MatchrefFilter',RefFiltFlag)
```

**Description**

Hq = minimizecoeffwl(Hd) returns the minimum wordlength fixed-point filter object Hq that meets the design specifications of the single-stage or multistage FIR filter object Hd. Hd must be generated using fdesign and design. If Hd is a multistage filter object, the procedure minimizes the wordlength for each stage separately. minimizecoeffwl uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator rand.

Hq = minimizecoeffwl(Hd,...,'NoiseShaping',NSFlag) enables or disables the stochastic noise-shaping procedure in the minimization of the wordlength. By default NSFlag is true. Setting NSFlag to false minimizes the wordlength without using noise-shaping.

Hq = minimizecoeffwl(Hd,...,'NTrials',N) specifies the number of Monte Carlo trials to use in the minimization. Hq is the filter with the minimum wordlength among the N trials that meets the specifications in Hd. 'NTrials' defaults to one.

Hq = minimizecoeffwl(Hd,...,'Apasstol',Apasstol) specifies the passband ripple tolerance in dB. 'Apasstol' defaults to 1e-4.

Hq = minimizecoeffwl(Hd,...,'Astoptol',Astoptol) specifies the stopband tolerance in dB. 'Astoptol' defaults to 1e-2.

Hq = minimizecoeffwl(Hd,...,'MatchrefFilter',RefFiltFlag) determines whether the fixed-point filter matches the filter order and transition width of the floating-point design. Setting 'MatchRefFilter' to true returns a fixed-point filter with the same order and transition width as Hd. The 'MatchRefFilter' property defaults to false and the

resulting fixed-point filter may have a different order and transition width than the floating-point design Hd.

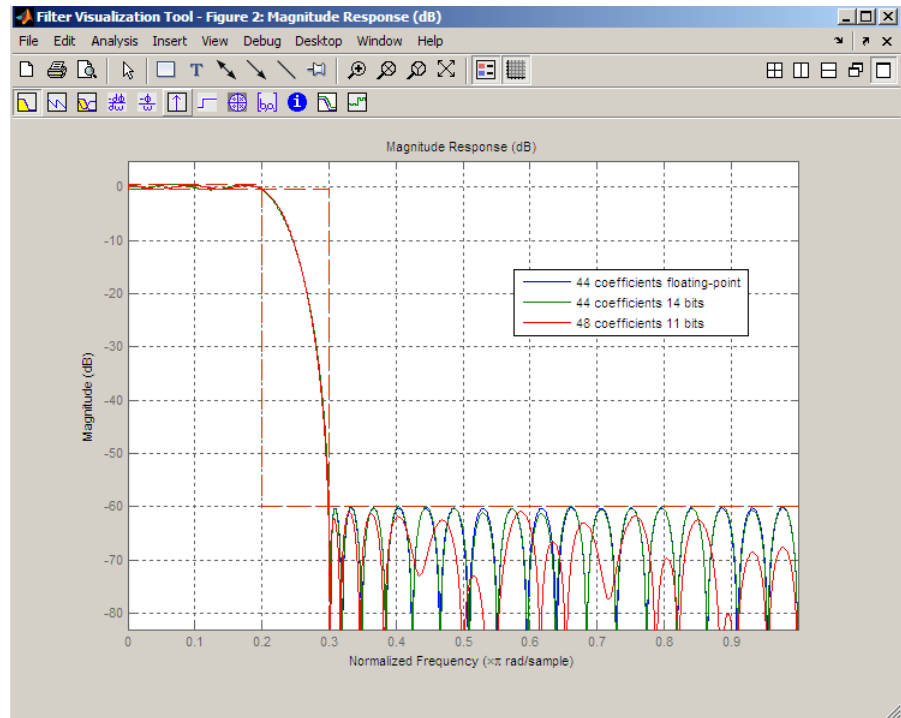
You must have the Fixed-Point Toolbox software installed to use this function.

## Examples

Minimize wordlength for lowpass FIR equiripple filter:

```
f=fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.3,1,60);
% Design filter with double-precision floating point
Hd=design(f,'equiripple');
% Find minimum wordlength fixed-point filter
% with 0.15 dB stopband tolerance
Hq=minimizecoeffwl(Hd,'MatchRefFilter',true,'Astoptol',0.15);
Hq1=minimizecoeffwl(Hd,'Astoptol',0.15);
% Hq.coeffwordlength is 14 bits.
% Hq1.coeffwordlength is 11 bits
hfvtool(Hd,Hq,Hq1,'showreference','off');
legend(hfvtool,'44 coefficients floating-point',...
'44 coefficients 14 bits','48 coefficients 11 bits');
```

# minimizecoeffwl



## See Also

`constraincoeffwl` | `design` | `fdesign` | `maximizestopband` | `measure` | `rand`

## Tutorials

- “Fixed-Point Concepts”

**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] = msessim(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');
```

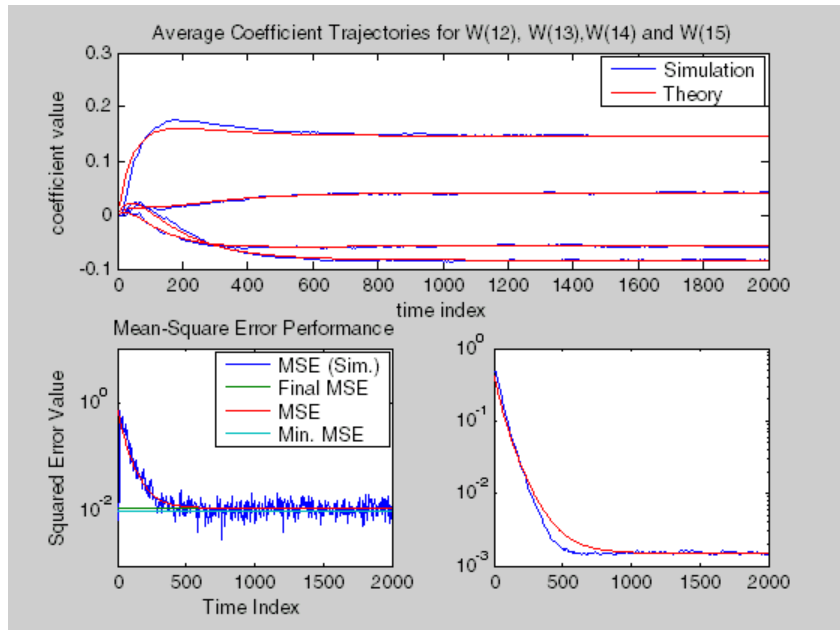


```

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`

# mstesim

---

**Purpose** Measured mean-squared error for adaptive filter

**Syntax**

```
mse = mstesim(ha,x,d)
[mse,meanw,w,tracek] = mstesim(ha,x,d)
[mse,meanw,w,tracek] = mstesim(ha,x,d,m)
```

**Description** `mse = mstesim(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] = mstesim(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] = mstesim(ha,x,d,m)` specifies an optional input argument `m` that is the decimation factor for computing `meanw`, `mse`, and `tracek`. When `m > 1`, `mstesim` 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.

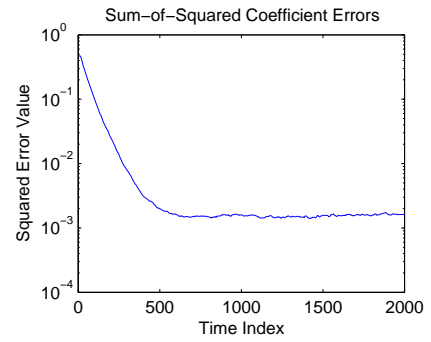
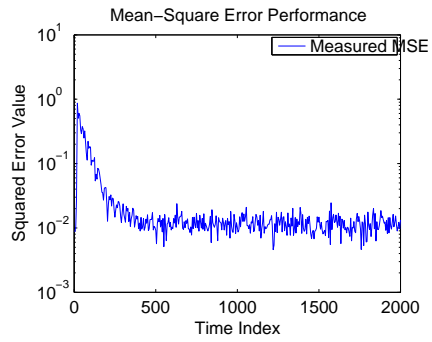
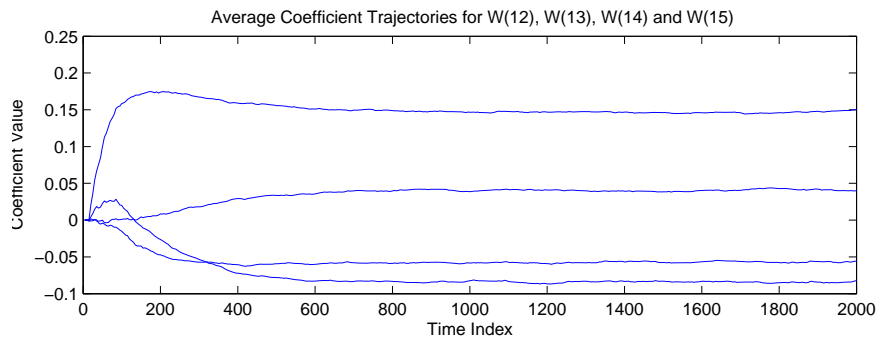
```

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

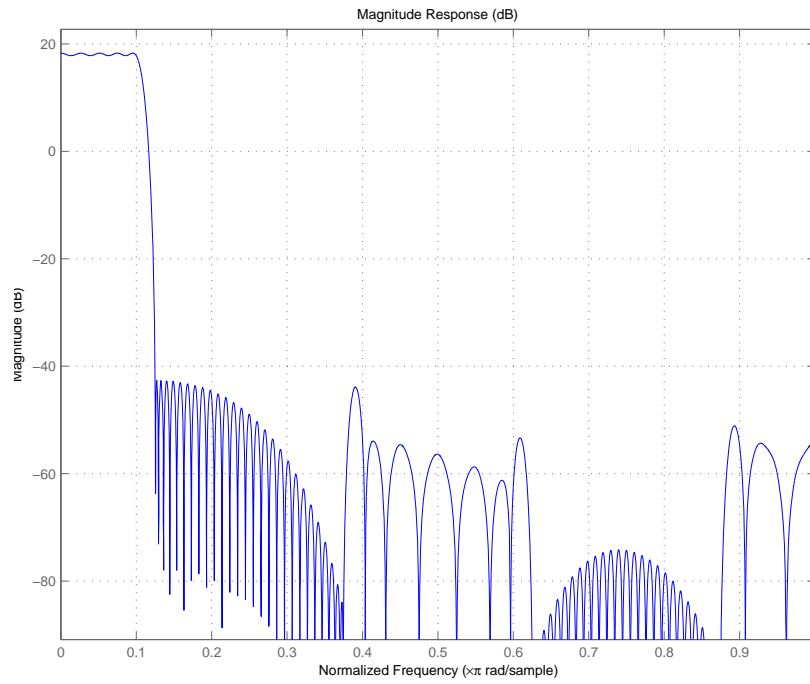
Design a minimum-order, multistage Nyquist interpolator.

```
l = 15; % Interpolation factor. Also the Nyquist band.
tw = 0.05; % Normalized transition width
ast = 40; % Minimum stopband attenuation in dB
d = fdesign.interpolator(1, 'nyquist', l, 'tw,ast', 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');
% Use halfband filters if possible.
hm = design(d, 'multistage', 'Usehalfbands', true);
fvtool(hm);
```

This figure shows the response for `hm`.



**See Also** design, designopts

# noisepsd

---

**Purpose** Power spectral density of filter output

**Syntax**  
`hpsd = noisepsd(hd,1)`  
`hpsd = noisepsd(hd,1,p1,v1,p2,v2,...)`  
`noisepsd(hd,1,opts)`

**Description** `hpsd = noisepsd(hd,1)` computes the power spectral density (PSD) at the output of filter `hd` due to roundoff noise produced by quantization errors within the filter. `1` is the number of trials used to compute the average. The PSD is computed from the average over the `1` trials. The more trials you specify, the better the estimate, but at the expense of longer computation time. When you do not explicitly set `1`, 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,1,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.
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.



Property Name	Default Value	Description and Valid Entries
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.
SpectrumType	onesided	<p>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.</p> <ul style="list-style-type: none"> <li>onesided converts the spectrum to a spectrum calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically.</li> <li>twosided converts the spectrum to a spectrum calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.</li> </ul>
CenterDC	false	<p>Shifts the zero-frequency component to the center of a two-sided spectrum.</p> <ul style="list-style-type: none"> <li>When you set SpectrumType to onesided, it is changed to twosided and the data is converted to a two-sided spectrum.</li> <li>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.</li> </ul>

---

**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`:

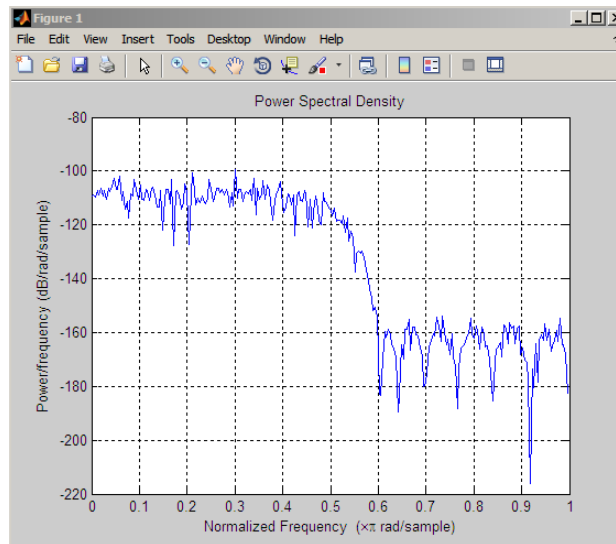
```
set(opts, 'fs', 48e3); % Set Fs to 48 kHz.
```

## 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`.

**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.

# noisepsdopts

---

**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
<code>Nfft</code>	512	Specifies the number of FFT points to use to calculate the PSD.
<code>NormalizedFrequency</code>	<code>true</code>	Determines whether to use normalized frequency. Enter one of the logical <code>true</code> or <code>false</code> . Note that you do not use single quotations around this property value because it is a logical value, not a string.
<code>Fs</code>	<code>normalized</code>	Specifies the sampling frequency to use when you set <code>NormalizedFrequency</code> to <code>false</code> . Any integer value greater than 1 works. Enter the value in Hz.

Property Name	Default Value	Description and Valid Entries
SpectrumType	onesided	<p>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.</p> <ul style="list-style-type: none"> <li>onesided converts the spectrum to a spectrum calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically.</li> <li>twosided converts the spectrum to a spectrum calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.</li> </ul>
CenterDC	false	<p>Shifts the zero-frequency component to the center of a two-sided spectrum.</p> <ul style="list-style-type: none"> <li>When you set SpectrumType to onesided, it is changed to twosided and the data is converted to a two-sided spectrum.</li> <li>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.</li> </ul>

# noisepsopts

---

## See Also

`noisepso`

**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 polyphase interpolator, which is an `mfilt` object.

```
hm = mfilt.firinterp;  
linf = norm(hm,'linf');  
linf =
```

```
    1
```

## See Also

`reorder`, `scale`, `scalecheck`

# normalize

---

**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=design(d,'ellip');  
hd.arithmetic='fixed'
```

Check the filter coefficients. Note that `hd.sosMatrix(1,2)>1`

```
hd.sosMatrix
```

Use `normalize` to modify the coefficients into the range between -1 and 1. The output `g` contains the gains applied to each section of the SOS filter.

```
g = normalize(hd)  
hd.sosMatrix
```

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 `normalizefreq` to convert to absolute frequency specifications, with a sampling frequency of 1000 Hz.

```
normalizefreq(d, false, 1e3)
```

```
d
```

```
d =
```

# normalizefreq

---

```
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  $F_s/2$ , 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  $F_s$  as an input argument and you are converting to absolute frequency specifications, `normalizefreq` sets  $F_s$  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

# nstates

---

**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=fir1s(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');

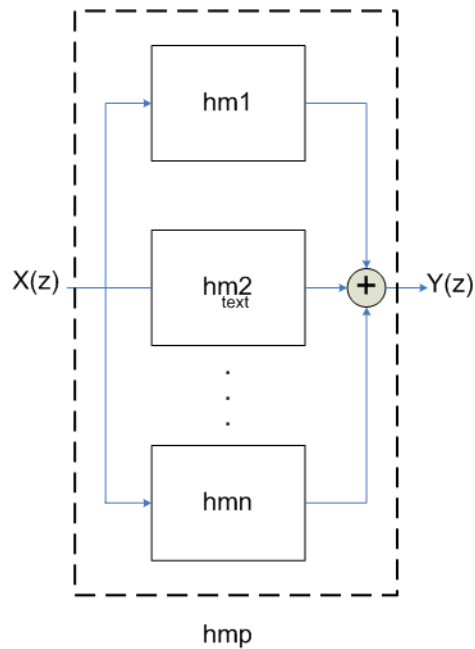
n=order(hq) % Check the order of the overall filter.
n = 4
```

**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

**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

# phasez

---

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
```

# polyphase

---

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

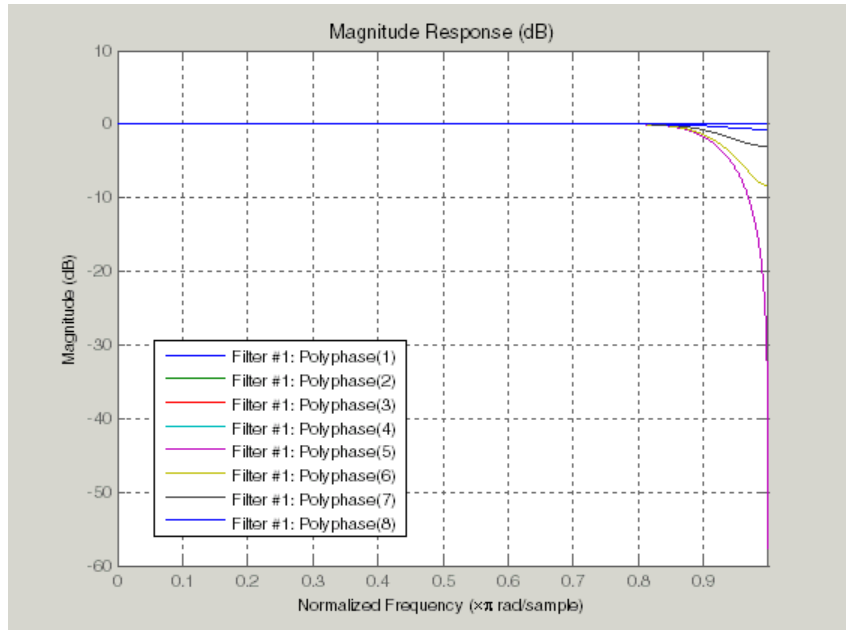
Columns 17 through 24

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`

# qreport

---

**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 Fixed-Point Toolbox 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 Fixed-Point Toolbox documentation.

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 for `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.

# realizemdl

Property Name	Property Values	Description
Destination	'current' (default) or 'new', <i>Subsystemname</i>	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
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 ' <i>blockname</i> '.
MapCoeffstoPorts	'off' (default) or 'on'	Specify whether to map the coefficients of the filter to the ports of the block.
MapCoeffstoPorts	'off' (default) or 'on'	Specifies whether to apply the current filter states to the realized model. This lets you save states from a filter object you may have used or configured in a specific way. The default setting of 'off' means the states are not transferred to the model. Setting the property to 'on' preserves the current filter states in the realized model.
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.



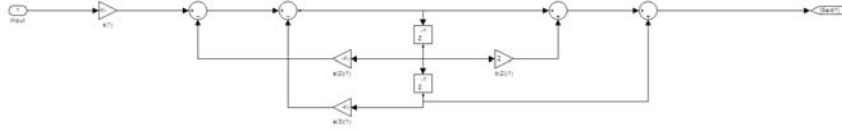
Property Name	Property Values	Description
OptimizeOnes	'off' (default) or 'on'	Specify whether to replace unity-gain blocks with direct connections.
OptimizeNegOnes	'off' (default) or 'on'	Specify whether to replace negative unity-gain blocks with
OptimizeDelayChains	'off' (default) or 'on'	Specify whether to replace cascaded chains of delay blocks with a single integer delay block to provide an equivalent delay.
CoeffNames	{ 'Num' } (default FIR) { 'Num', 'Den' } (default direct form IIR) { 'Num', 'Den', 'g' } (default IIR SOS), { 'Num_1', 'Num_2', 'Num_3' ... } (default multistage) { 'K' } (default form lattice)	Specify the coefficient variable names as string variables in a cell array.  MapCoeffsToPorts must be set to 'on' for this property to apply.

## Examples

Realize Simulink model of lowpass Butterworth filter:

```
d = fdesign.lowpass('N,F3dB',4,0.25);
Hd = design(d,'butter');
realizemdl(Hd);
```

The realized model is shown in the figure:



Realize Simulink model with coefficients mapped to ports:

```
d = fdesign.lowpass('N,F3dB',4,0.25);  
Hd = design(d,'butter');  
%Realize Simulink model and export coefficients  
realizemdl(Hd,'MapCoeffsToPorts','on');
```

In this case, the filter is an IIR filter with a direct form II second-order sections structure. Setting `MapCoeffsToPorts` to 'on' exports the numerator coefficients, the denominator coefficients, and the gains to the MATLAB workspace using the default variable names `Num`, `Den`, and `g`. Each column of `Num` and `Den` represents one second-order section. You can modify the filter coefficients directly in the MATLAB workspace providing tunability to the realized Simulink model.

## See Also

`design` | `fdesign`

## How To

- *Simulink Getting Started Guide*

**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.

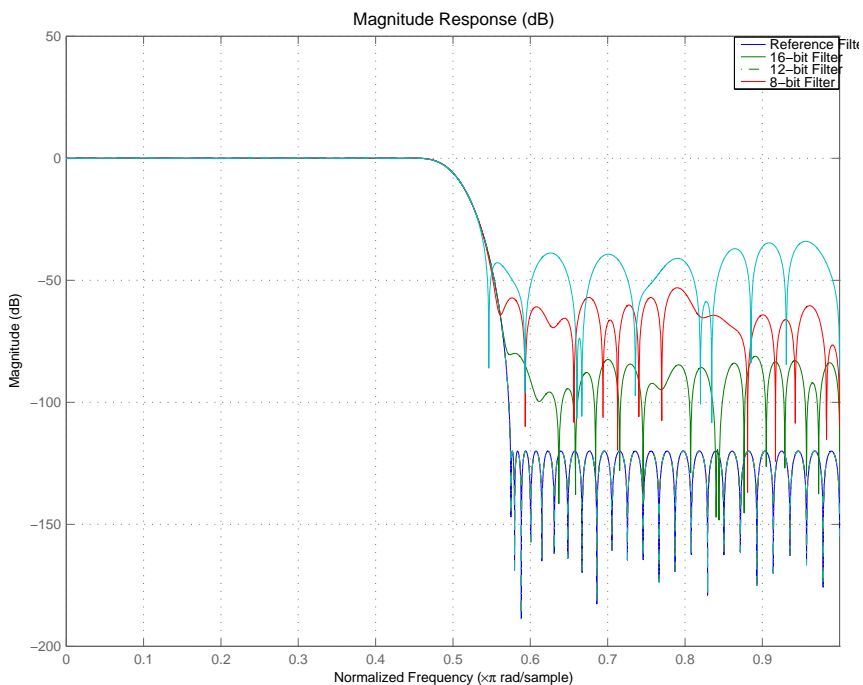
**Examples**

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

**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

---

`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

hdreorder=reorder(hd,'auto');
```

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);

```

# reorder

---

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

# scale

---

**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 the 'L2' norm is equal to the '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'), '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. The default scaling 'Linf' 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.

<b>Parameter</b>	<b>Default</b>	<b>Description and Valid Value</b>
MaxNumerator	2	Maximum allowed value for numerator coefficients.
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.

# scale

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)
```

## References

Dehner, G.F. “Noise Optimized Digital Filter Design: Tutorial and Some New Aspects.” *Signal Processing*. Vol. 83, Number 8, 2003, pp. 1565–1582.

## 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.df1ir]

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`

# 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 <code>MaxScaleValue</code> limit only when you set <code>ScaleValueConstraint</code> to a value other than <code>unit</code> . Setting <code>MaxScaleValue</code> to a numerical value automatically changes the <code>ScaleValueConstraint</code> setting to <code>none</code> .
NumeratorConstraint	none	Specifies whether and how to constrain numerator coefficient values. Options are <code>none</code> , <code>normalize</code> , <code>po2</code> , and <code>unit</code> ,



Parameter	Default	Description and Valid Value
OverflowMode	wrap	Sets the way the filter handles arithmetic overflow situations during scaling. Choose one of wrap 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`

# set2int

---

**Purpose** Configure filter for integer filtering

**Syntax**

```
set2int(h)
set2int(h,coeffw1)
set2int(...,inw1)
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,coeffw1)` uses the number of bits specified by `coeffw1` as the word length it uses to represent the filter coefficients.

`set2int(...,inw1)` uses the number of bits specified by `coeffw1` as the word length it uses to represent the filter coefficients and the number of bits specified by `inw1` 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`

# setspecs

---

## 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')
setspecs(d, 20, .4);
d
```

### Example 2

Now specify a sampling frequency after you make `d`.

```
d = fdesign.lowpass('n,fc')
setspecs(d, 20, 4, 20);
```

### 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;
```

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')
```

### Example 4

Finally, use `setspecs` to change the Specification string and apply new filter specifications to `d`.

# setspecs

---

```
d = fdesign.lowpass;  
setspecs(d, 'N,Fc',30,0.2)  
d
```

## 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 ( $L_\infty$  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)

Hq1 =

    FilterStructure: 'Direct-Form II Transposed, Second-Order Sections'
    Arithmetic: 'double'
    sosMatrix: [4x6 double]
    ScaleValues: [0.00927734375;1;1;1;1]
    OptimizeScaleValues: true
    PersistentMemory: false
```

## 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 <code>specifyall</code>
<code>CoeffAutoScale</code>	<code>true</code>	<code>false</code>
<code>OutputMode</code>	<code>AvoidOverflow</code>	<code>SpecifyPrecision</code>
<code>ProductMode</code>	<code>FullPrecision</code>	<code>SpecifyPrecision</code>
<code>AccumMode</code>	<code>KeepMSB</code>	<code>SpecifyPrecision</code>
<code>RoundMode</code>	<code>convergent</code>	<code>convergent</code>
<code>OverflowMode</code>	<code>wrap</code>	<code>wrap</code>

`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

# specifyall

---

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

```

# specifyall

---

```
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`, `refilter`

`fi`, `fimath` in Fixed-Point Toolbox 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  $|\text{beta}| = 1$ , such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[\bar{\beta} \cdot H1(z) + \beta \cdot H2(z)]$$

representing the generalized allpass decomposition.

In the above equations,  $H1(z)$  and  $H2(z)$  are real or complex allpass IIR filters given by

$$H1(z) = \frac{\text{fliplr}(\overline{D1(z)})}{D1(z)}, H2(z) = \frac{\text{fliplr}(\overline{D2(z)})}{D2(z)}$$

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 denominator.  
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**  
 $[k1, k2] = \text{tf2cl}(b, a)$   
 $[k1, k2] = \text{tf2cl}(b, a)$

**Description**  $[k1, k2] = \text{tf2cl}(b, a)$  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  $k2$ .

$[k1, k2] = \text{tf2cl}(b, a)$  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

$$[k1, k2, \text{beta}] = \text{tf2cl}(b, a)$$

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

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right) [\bar{\beta} \cdot H1(z) + \beta \cdot H2(z)]$$

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 denominator.  
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

## 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.interpolator(3, 'lowpass', .45,0.55,.1,60);
designmethods(d)
% FIR Design Methods for class fdesign.interpolator (Fp,Fst,Ap,Ast):
% equiripple
% ifir
% kaiserwin
% multistage
```

# validstructures

---

```
validstructures(d, 'kaiserwin')
% 'firinterp'    'fftfirinterp'
```

Now you can specify the filter structure when you design the filter `hm`.

```
hm=design(d, 'kaiserwin', 'FilterStructure', 'firinterp');
```

## 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)
designmethods(d)
% FIR Design Methods for class fdesign.cicdecim (Fp,Ast):
% multisection
c = validstructures(d, 'multirate')
% '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)
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')
```

## Example 4

Multirate filters support `validstructures`.

```
d=fdesign.rsrc(4,5);  
designmethods(d);  
validstructures(d,'kaiserwin')  
% 'firsrc'      'firfracdecim'
```

**See Also**

design, designmethods, designopts, fdesign

# window

---

**Purpose** FIR filter using windowed impulse response

**Syntax**  
`h = window(d,fcnhd1,fcarg)`  
`h = window(d,win)`

**description** `h = window(d,fcnhd1,fcarg)` 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`.

`fcnhd1` is a handle to a filter design function that returns a window vector, such as the `hamming` or `blackman` functions. `fcarg` 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

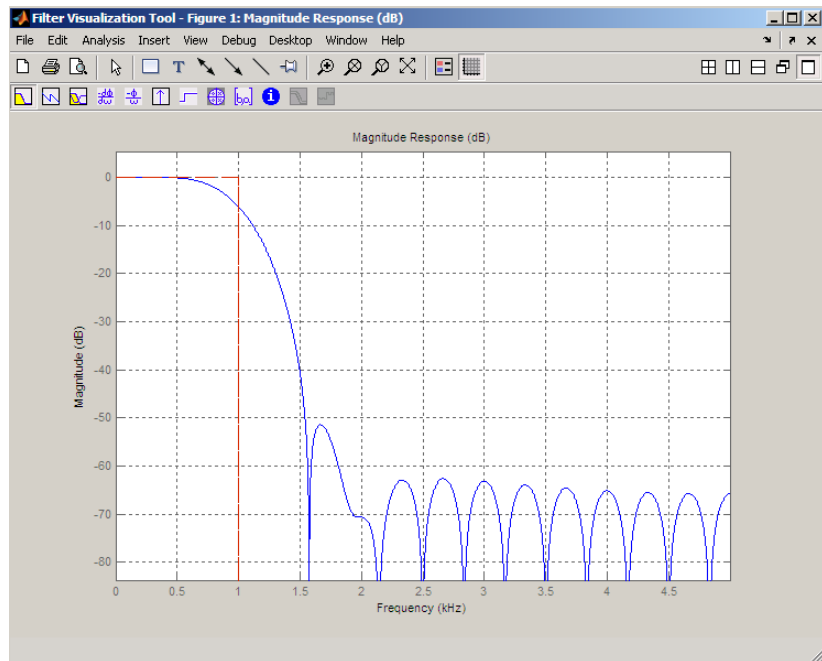
These examples design filters using the two design techniques of specifying a function handle or passing a window vector as an input argument.

### Example 1

Use a function handle and optional input arguments to design a multirate filter. We use a function handle to `hamming` to provide the window. Since this example creates a decimator filter specifications object, `window` returns a multirate filter.

```
d = fdesign.decimator(4,'lowpass','N,Fc',30,1000,10000);  
% Lowpass decimator with a 6-dB down frequency of 1 kHz  
% Order equal to 30 and sampling frequency 10 kHz  
Hd =window(d,'window',@hamming);  
fvtool(Hd)
```



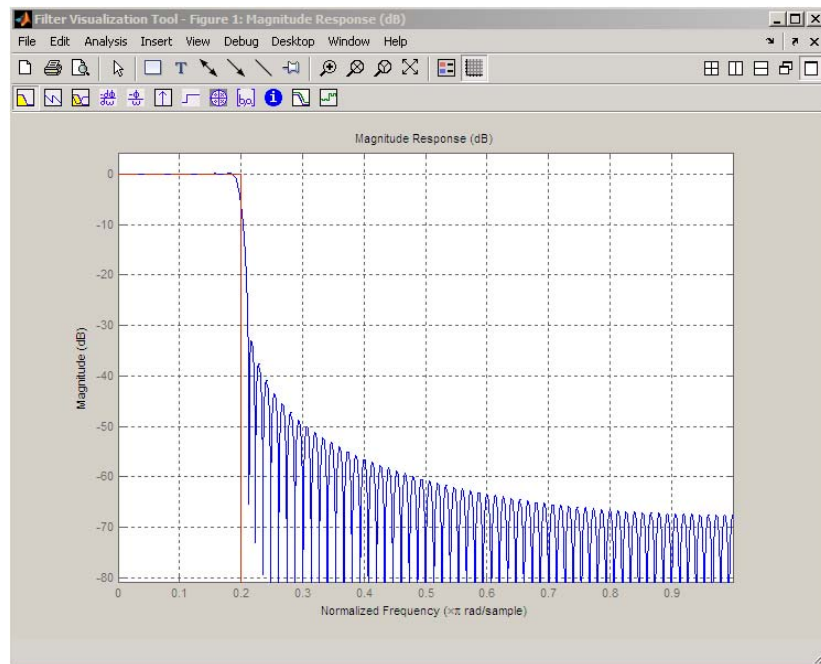


## Example 2

Use a window vector provided by the kaiser window function to design a Nyquist filter. The window length must be the filter order plus one. .

```
d = fdesign.nyquist(5,'n',150);  
% Kaiser window with beta parameter 2.5  
Hd = window(d,'window',kaiser(151,2.5));  
fvtool(Hd)
```

# window



**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

---

<b>Purpose</b>	Zero-pole-gain complex bandpass frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)</code> returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.</p> <p>It also returns the numerator, <code>AllpassNum</code>, and the denominator, <code>AllpassDen</code>, of the allpass mapping filter. The original lowpass filter is given with zeros, <math>Z</math>, poles, <math>P</math>, and gain factor, <math>K</math>.</p> <p>This transformation effectively places two features of an original filter, located at frequencies <math>W_{o1}</math> and <math>W_{o2}</math>, at the required target frequency locations, <math>W_{t1}</math>, and <math>W_{t2}</math> respectively. It is assumed that <math>W_{t2}</math> is greater than <math>W_{t1}</math>. 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.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>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.</p>

## Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

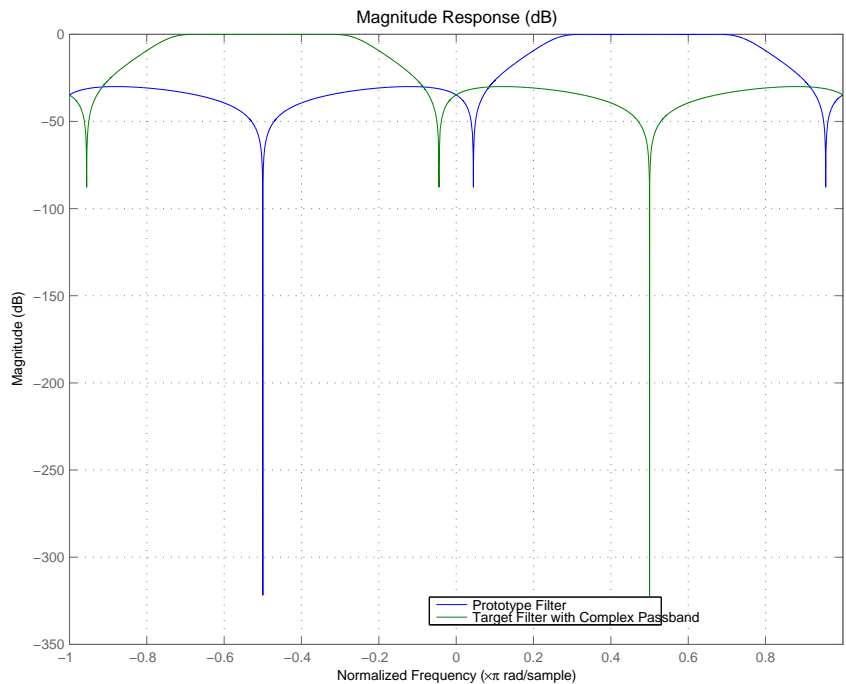
Create a complex passband from 0.25 to 0.75:

```
[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.



# zpkbpc2bpc

---

## Arguments

Variable	Description
<i>Z</i>	Zeros of the prototype lowpass filter
<i>P</i>	Poles of the prototype lowpass filter
<i>K</i>	Gain factor of the prototype lowpass filter
<i>Wo</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency location in the transformed target filter
<i>Z2</i>	Zeros of the target filter
<i>P2</i>	Poles of the target filter
<i>K2</i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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:

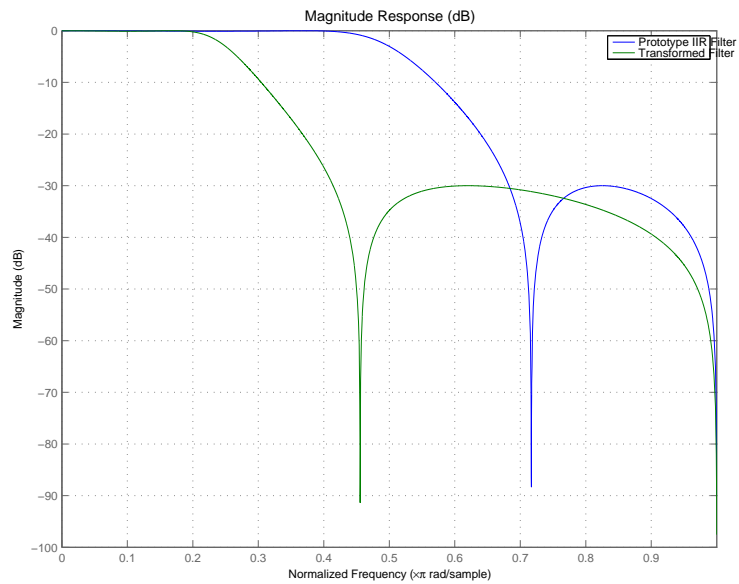
```
[b, a] = ellip(3,0.1,30,0.409);  
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);  
[z2, p2, k2] = zpkftransf(roots(b),roots(a),b(1),AlpNum,AlpDen);
```

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.

# zpkftransf



## 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

<b>Purpose</b>	Zero-pole-gain lowpass to bandpass frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bp(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bp(Z,P,K,Wo,Wt)</code> returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.</p> <p>It also returns the numerator, <code>AllpassNum</code>, and the denominator <code>AllpassDen</code>, of the allpass mapping filter. The prototype lowpass filter is given with zeros, <math>Z</math>, poles, <math>P</math>, and gain factor, <math>K</math>.</p> <p>This transformation effectively places one feature of an original filter, located at frequency <math>-W_o</math>, at the required target frequency location, <math>W_{t1}</math>, and the second feature, originally at <math>+W_o</math>, at the new location, <math>W_{t2}</math>. It is assumed that <math>W_{t2}</math> is greater than <math>W_{t1}</math>. 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 <math>W_t</math>.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>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.</p> <p>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.</p>
<b>Examples</b>	Design a prototype real IIR halfband filter using a standard elliptic approach:

# zpk1p2bp

```
[b, a] = ellip(3,0.1,30,0.409);  
z = roots(b);  
p = roots(a);  
k = b(1);  
[z2,p2,k2] = zpk1p2bp(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
<i>Z</i>	Zeros of the prototype lowpass filter
<i>P</i>	Poles of the prototype lowpass filter
<i>K</i>	Gain factor of the prototype lowpass filter
<i>W0</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency location in the transformed target filter
<i>Z2</i>	Zeros of the target filter
<i>P2</i>	Poles of the target filter
<i>K2</i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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, allpass1p2bp, iir1p2bp

## 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.

# zpk1p2bpc

---

**Purpose** Zero-pole-gain lowpass to complex bandpass frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bpc(Z,P,K,Wo,Wt)`

**Description** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bpc(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] = zpk1p2bpc(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
<i>Z</i>	Zeros of the prototype lowpass filter
<i>P</i>	Poles of the prototype lowpass filter
<i>K</i>	Gain factor of the prototype lowpass filter
<i>W0</i>	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.
<i>Wt</i>	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.
<i>Z2</i>	Zeros of the target filter
<i>P2</i>	Poles of the target filter
<i>K2</i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

## See Also

zpkftransf, allpass1p2bpc, iir1p2bpc

# zpk1p2bs

---

<b>Purpose</b>	Zero-pole-gain lowpass to bandstop frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bs(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bs(Z,P,K,Wo,Wt)</code> returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.</p> <p>It also returns the numerator, <code>AllpassNum</code>, and the denominator, <code>AllpassDen</code>, of the allpass mapping filter. The prototype lowpass filter is given with zeros, <math>Z</math>, poles, <math>P</math>, and gain factor, <math>K</math>.</p> <p>This transformation effectively places one feature of an original filter, located at frequency <math>-W_o</math>, at the required target frequency location, <math>W_{t1}</math>, and the second feature, originally at <math>+W_o</math>, at the new location, <math>W_{t2}</math>. It is assumed that <math>W_{t2}</math> is greater than <math>W_{t1}</math>. 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 <math>W_o</math> and <math>W_t</math>s.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. After the transformation feature <math>F_2</math> will precede <math>F_1</math> in the target filter. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>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.</p>

## 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] = zpk1p2bs(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
<i>Z</i>	Zeros of the prototype lowpass filter
<i>P</i>	Poles of the prototype lowpass filter
<i>K</i>	Gain factor of the prototype lowpass filter
<i>W0</i>	Frequency value to be transformed from the prototype filter
<i>Wt</i>	Desired frequency location in the transformed target filter
<i>Z2</i>	Zeros of the target filter
<i>P2</i>	Poles of the target filter
<i>K2</i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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, allpass1p2bs, iir1p2bs

## 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.

<b>Purpose</b>	Zero-pole-gain lowpass to complex bandstop frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bsc(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bsc(Z,P,K,Wo,Wt)</code> returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.</p> <p>It also returns the numerator, <code>AllpassNum</code>, and the denominator, <code>AllpassDen</code>, of the allpass mapping filter. The prototype lowpass filter is given with zeros, <math>Z</math>, poles, <math>P</math>, and gain factor, <math>K</math>.</p> <p>This transformation effectively places one feature of an original filter, located at frequency <math>-W_o</math>, at the required target frequency location, <math>W_{t1}</math>, and the second feature, originally at <math>+W_o</math>, at the new location, <math>W_{t2}</math>. It is assumed that <math>W_{t2}</math> is greater than <math>W_{t1}</math>. Additionally the transformation swaps passbands with stopbands in the target filter.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>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.</p> <p>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.</p>
<b>Examples</b>	<p>Design a prototype real IIR halfband filter using a standard elliptic approach:</p> <pre>[b, a] = ellip(3,0.1,30,0.409);</pre>

# zpk1p2bsc

```
z = roots(b);  
p = roots(a);  
k = b(1);  
[z2,p2,k2] = zpk1p2bsc(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
<i>Z</i>	Zeros of the prototype lowpass filter
<i>P</i>	Poles of the prototype lowpass filter
<i>K</i>	Gain factor of the prototype lowpass filter
<i>Wo</i>	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.
<i>Wt</i>	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.
<i>Z2</i>	Zeros of the target filter
<i>P2</i>	Poles of the target filter
<i>K2</i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

## See Also

zpkftransf, allpass1p2bsc, iir1p2bsc

<b>Purpose</b>	Zero-pole-gain lowpass to highpass frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2hp(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2hp(Z,P,K,Wo,Wt)</code> returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, 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 <math>W_o</math>, at the required target frequency location, <math>W_t</math>, 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.</p> <p>It also returns the numerator, <code>AllpassNum</code>, and the denominator, <code>AllpassDen</code>, of the allpass mapping filter. The prototype lowpass filter is given with zeros, <math>Z</math>, poles, <math>P</math>, and the gain factor, <math>K</math>.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. After the transformation feature <math>F_2</math> will precede <math>F_1</math> in the target filter. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>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.</p> <p>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.</p>
<b>Examples</b>	<p>Design a prototype real IIR halfband filter using a standard elliptic approach:</p> <pre>[b, a] = ellip(3,0.1,30,0.409); z = roots(b); p = roots(a);</pre>

# zpk1p2hp

```
k = b(1);  
[z2,p2,k2] = zpk1p2hp(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
<i>Z</i>	Zeros of the prototype lowpass filter
<i>P</i>	Poles of the prototype lowpass filter
<i>K</i>	Gain factor of the prototype lowpass filter
<i>W<sub>0</sub></i>	Frequency value to be transformed from the prototype filter
<i>W<sub>t</sub></i>	Desired frequency location in the transformed target filter
<i>Z<sub>2</sub></i>	Zeros of the target filter
<i>P<sub>2</sub></i>	Poles of the target filter
<i>K<sub>2</sub></i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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, allpass1p2hp, iir1p2hp

## 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 lowpass frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2lp(Z,P,K,Wo,Wt)`

**Description** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2lp(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:

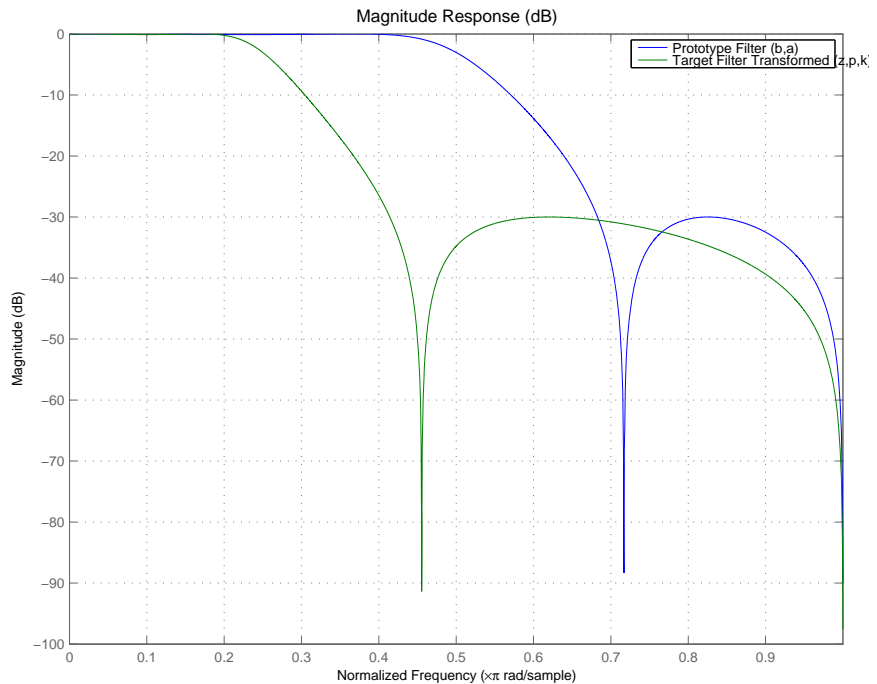
```
[b, a] = ellip(3, 0.1, 30, 0.409);  
z = roots(b);  
p = roots(a);  
k = b(1);  
[z2,p2,k2] = zpk1p2lp(z, p, k, 0.5, 0.25);
```



Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Using `zpk1p2lp` 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
$W_0$	Frequency value to be transformed from the prototype filter
$W_t$	Desired frequency location in the transformed target filter
$Z_2$	Zeros of the target filter
$P_2$	Poles of the target filter
$K_2$	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`, `allpass1p2lp`, `iir1p2lp`

## 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.

<b>Purpose</b>	Zero-pole-gain lowpass to M-band frequency transformation
<b>Syntax</b>	<pre>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2mb(Z,P,K,Wo,Wt) [Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2mb(Z,P,K,Wo,Wt,Pass)</pre>
<b>Description</b>	<p>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2mb(Z,P,K,Wo,Wt) returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, of the target filter transformed from the real lowpass prototype by applying an Mth-order real lowpass to real multibandpass frequency mapping. By default the DC feature is kept at its original location.</p> <p>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2mb(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.</p> <p>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.</p> <p>This transformation effectively places one feature of an original filter, located at frequency <math>W_o</math>, at the required target frequency locations, <math>W_{t1}, \dots, W_{tM}</math>.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>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.</p> <p>This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a</p>

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

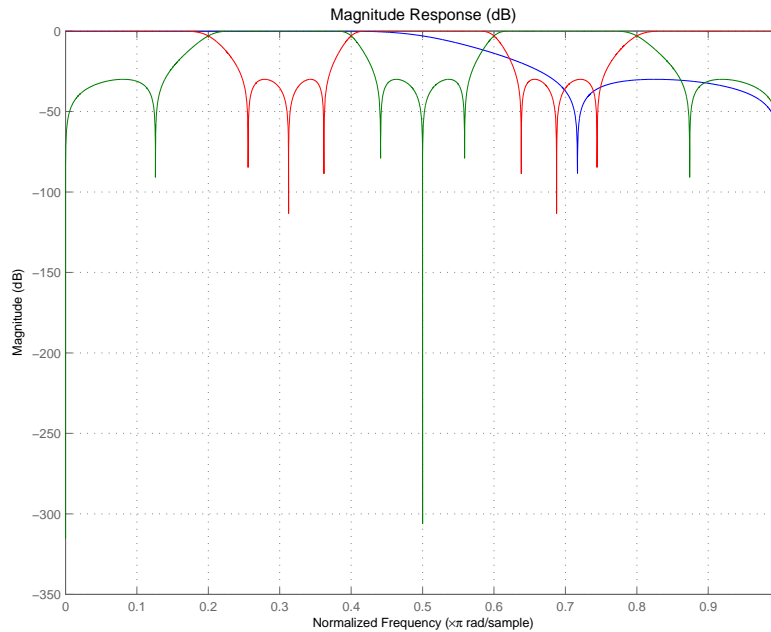
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpk1p2mb(z, p, k, 0.5, [2 4 6 8]/10, 'pass');
[z2,p2,k2] = zpk1p2mb(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
$W_0$	Frequency value to be transformed from the prototype filter
$W_t$	Desired frequency location in the transformed target filter
$Pass$	Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default
$Z_2$	Zeros of the target filter

Variable	Description
<i>P2</i>	Poles of the target filter
<i>K2</i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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, allpass1p2mb, iir1p2mb

## 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.

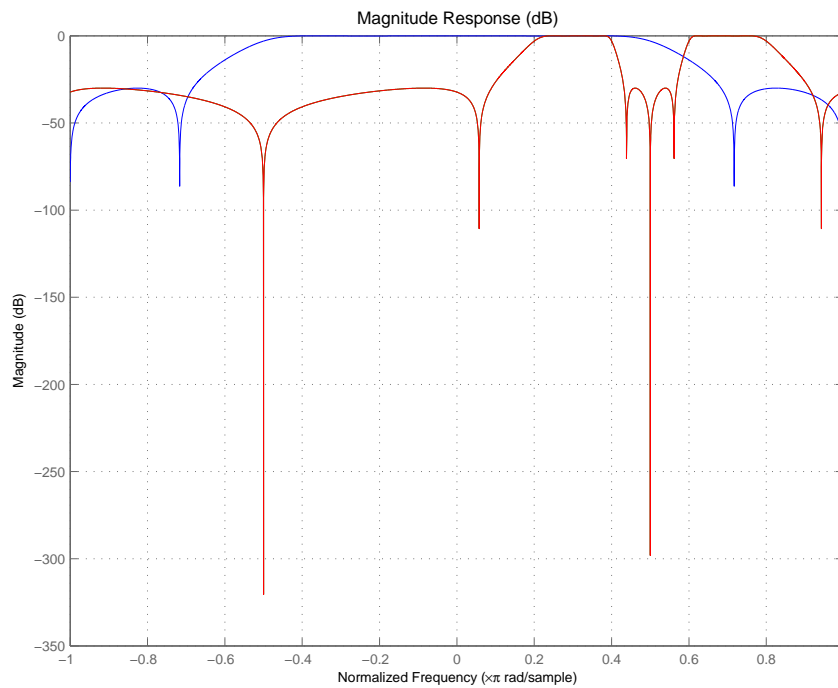
<b>Purpose</b>	Zero-pole-gain lowpass to complex M-band frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1pmbc(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1pmbc(Z,P,K,Wo,Wt)</code> returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, of the target filter transformed from the real lowpass prototype by applying an Mth-order real lowpass to complex multibandpass frequency transformation.</p> <p>It also returns the numerator, <code>AllpassNum</code>, and the denominator, <code>AllpassDen</code>, of the allpass mapping filter. The prototype lowpass filter is given with zeros, <math>Z</math>, poles, <math>P</math>, and gain factor, <math>K</math>.</p> <p>This transformation effectively places one feature of an original filter, located at frequency <math>W_o</math>, at the required target frequency locations, <math>W_{t1}, \dots, W_{tM}</math>.</p> <p>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.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p> <p>This transformation can also be used for transforming other types of filters; e.g., to replicate notch filters and resonators at any required location.</p>
<b>Examples</b>	<p>Design a prototype real IIR halfband filter using a standard elliptic approach:</p> <pre>[b, a] = ellip(3,0.1,30,0.409); z = roots(b); p = roots(a); k = b(1);</pre>

```
[z1,p1,k1] = zpk1p2mbc(z, p, k, 0.5, [2 4 6 8]/10);  
[z2,p2,k2] = zpk1p2mbc(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**

<b>Variable</b>	<b>Description</b>
<i>Z</i>	Zeros of the prototype lowpass filter
<i>P</i>	Poles of the prototype lowpass filter
<i>K</i>	Gain factor of the prototype lowpass filter
<i>Wo</i>	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.
<i>Wt</i>	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.
<i>Z2</i>	Zeros of the target filter
<i>P2</i>	Poles of the target filter
<i>K2</i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

**See Also**

zpkftransf, allpass1p2mbc, iir1p2mbc

**Purpose** Zero-pole-gain lowpass to complex N-point frequency transformation

**Syntax** [Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2xc(Z,P,K,Wo,Wt)

**Description** [Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2xc(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 Nth-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  $W_{o1}, \dots, W_{oN}$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

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

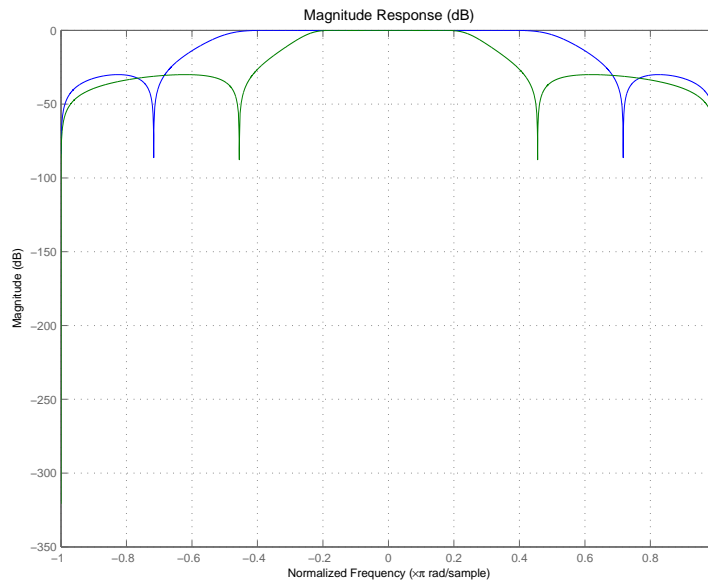
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[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.



# zpk1p2xc

---

## Arguments

Variable	Description
<i>Z</i>	Zeros of the prototype lowpass filter
<i>P</i>	Poles of the prototype lowpass filter
<i>K</i>	Gain factor of the prototype lowpass filter
<i>Wo</i>	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.
<i>Wt</i>	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.
<i>Z2</i>	Zeros of the target filter
<i>P2</i>	Poles of the target filter
<i>K2</i>	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	Denominator of the mapping filter

## See Also

zpkftransf, allpass1p2xc, iir1p2xc

<b>Purpose</b>	Zero-pole-gain lowpass to N-point frequency transformation
<b>Syntax</b>	<pre>[ Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2xn(Z,P,K,Wo,Wt) [ Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2xn(Z,P,K,Wo,Wt,Pass)</pre>
<b>Description</b>	<p>[ Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2xn(Z,P,K,Wo,Wt) returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, of the target filter transformed from the real lowpass prototype by applying an Nth-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.</p> <p>[ Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2xn(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.</p> <p>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.</p> <p>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 <math>W_{o1}, \dots, W_{oN}</math>, at the required target frequency locations, <math>W_{t1}, \dots, W_{tM}</math>.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation. For DC mobility feature <math>F_2</math> will precede <math>F_1</math> after the transformation.</p>

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.

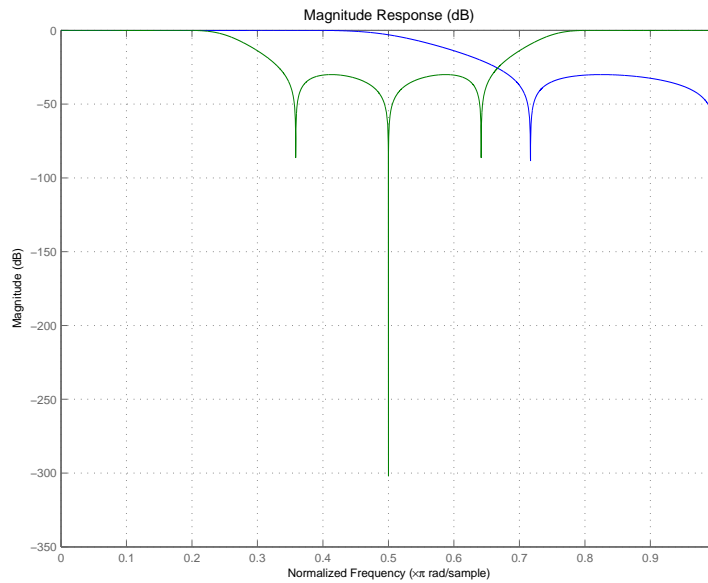
## Examples

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] = zpk1p2xn(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
$\omega_0$	Frequency value to be transformed from the prototype filter
$\omega_t$	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, allpass1p2xn, iir1p2xn

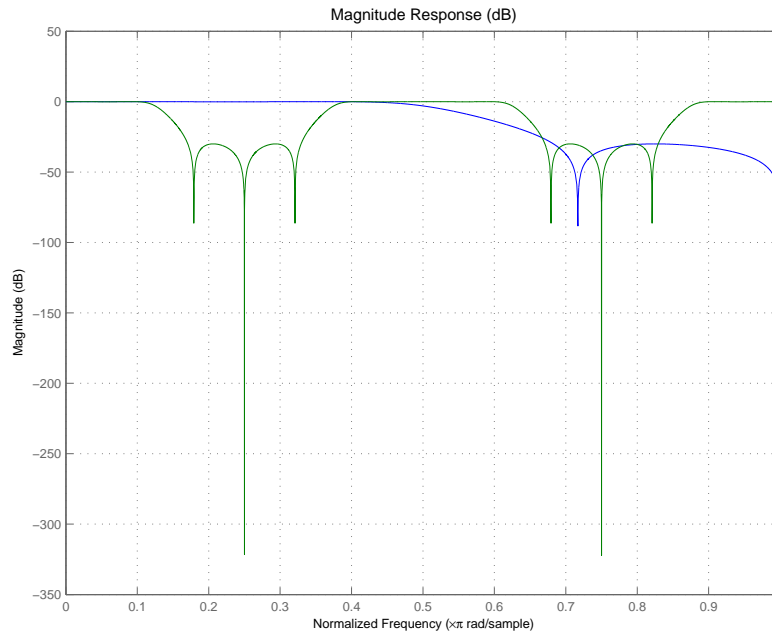
## References

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.



<b>Purpose</b>	Zero-pole-gain complex bandpass frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)</code> returns zeros, <math>Z_2</math>, poles, <math>P_2</math>, and gain factor, <math>K_2</math>, of the target filter being transformed from any prototype by applying an Nth-order rateup frequency transformation, where N is the upsample ratio. Transformation creates N equal replicas of the prototype filter frequency response.</p> <p>It also returns the numerator, <code>AllpassNum</code>, and the denominator, <code>AllpassDen</code>, of the allpass mapping filter. The original lowpass filter is given with zeros, <math>Z</math>, poles, <math>P</math>, and gain factor, <math>K</math>.</p> <p>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, <math>F_1</math> and <math>F_2</math>, with <math>F_1</math> preceding <math>F_2</math>. Feature <math>F_1</math> will still precede <math>F_2</math> after the transformation. However, the distance between <math>F_1</math> and <math>F_2</math> will not be the same before and after the transformation.</p>
<b>Examples</b>	<p>Design a prototype real IIR halfband filter using a standard elliptic approach:</p> <pre>[b, a] = ellip(3,0.1,30,0.409); z = roots(b); p = roots(a); k = b(1);</pre> <p>Upsample the prototype filter four times:</p> <pre>[z2,p2,k2] = zpkrateup(z, p, k, 4);</pre> <p>Verify the result by comparing the prototype filter with the target filter:</p> <pre>fvtool(b, a, k2*poly(z2), poly(p2));</pre> <p>Applying the upsample process creates a bandpass filter, as shown here.</p>



## 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:

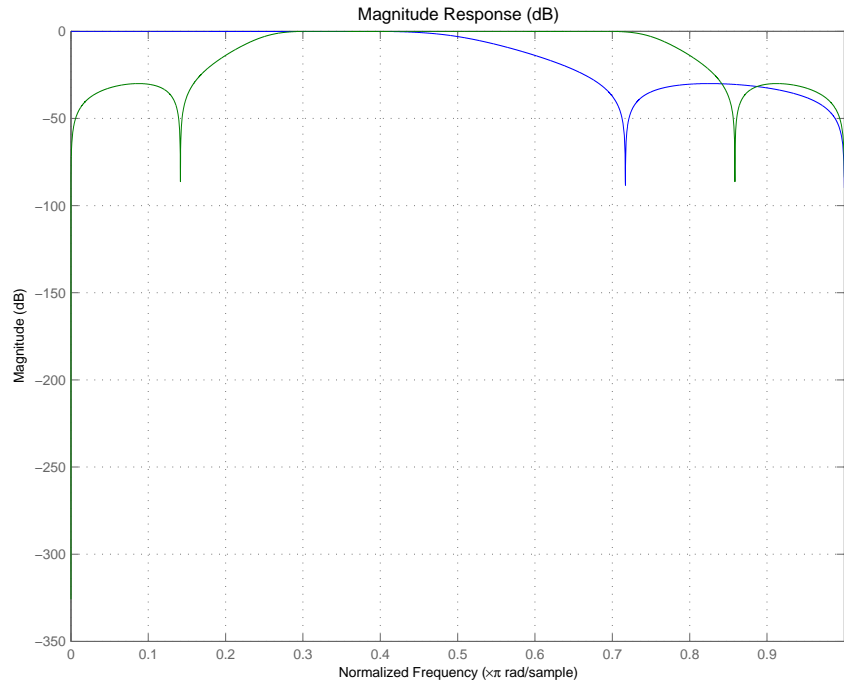
```
[b, a] = ellip(3,0.1,30,0.409);  
z = roots(b);
```

```
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.



# zpkshift

---

## 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
$W_0$	Frequency value to be transformed from the prototype filter
$W_t$	Desired frequency location in the transformed target filter
$Z_2$	Zeros of the target filter
$P_2$	Poles of the target filter
$K_2$	Gain factor of the target filter
<i>AllpassNum</i>	Numerator of the mapping filter
<i>AllpassDen</i>	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] = zpkshifc(Z,P,K,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,0,-0.5)
```

**Description**

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshifc(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] = zpkshifc(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] = zpkshifc(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] = zpkshifc(z, p, k, 0.5, 0.25);
```

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

## Example 2

Hilbert transform:

```
[z2,p2,k2] = zpkshifc(z, p, k, 0, 0.5);  
fvtool(b, a, k2*poly(z2), poly(p2));
```

## Example 3

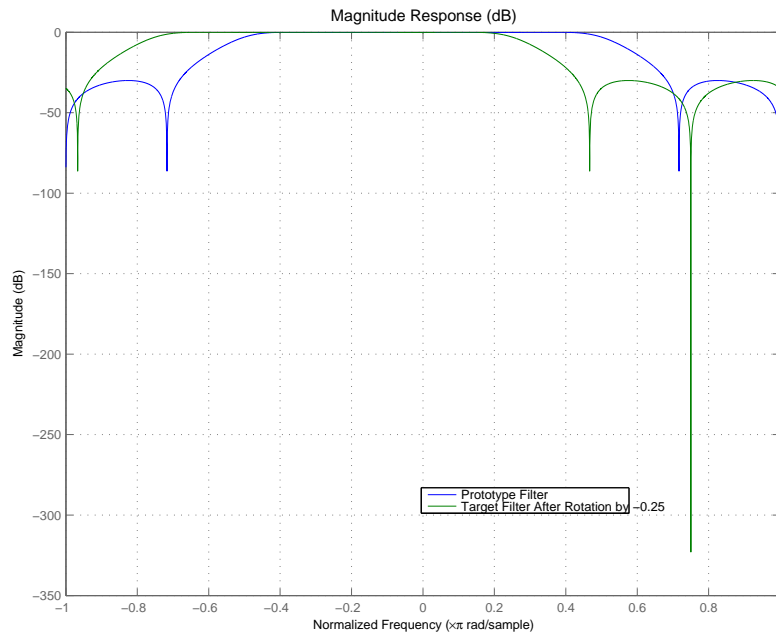
Inverse Hilbert transform:

```
[z2,p2,k2] = zpkshifc(z, p, k, 0, -0.5);  
fvtool(b, a, k2*poly(z2), poly(p2));
```

## Result of Example 1

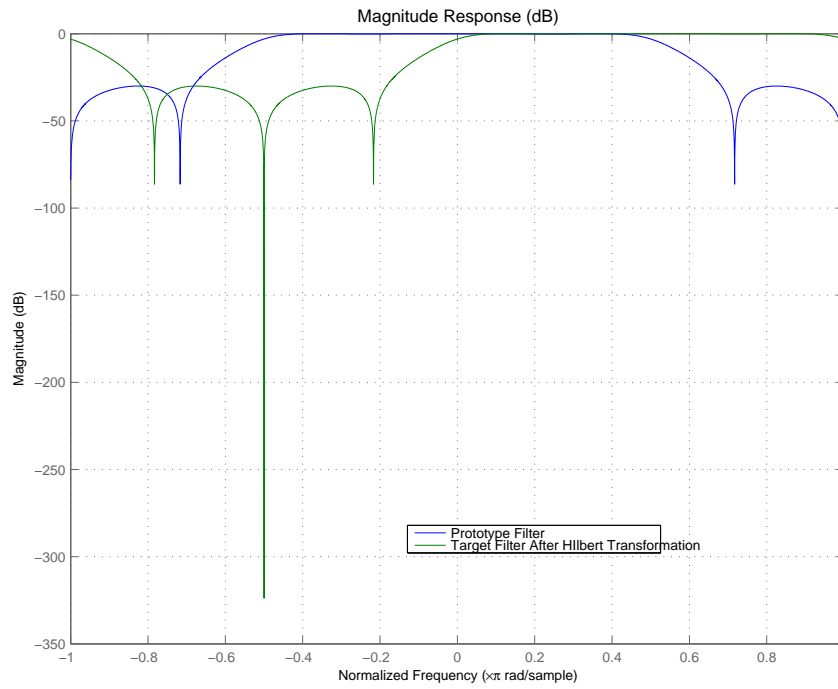
After performing the rotation, the resulting filter shows the features desired.





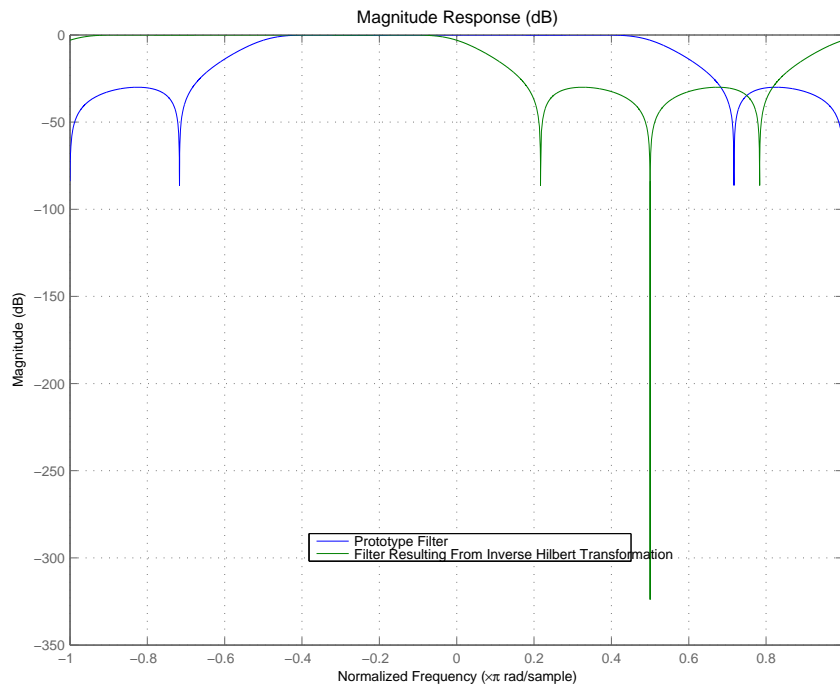
## Result of Example 2

Similar to the first example, performing the Hilbert transformation generates the desired target filter, shown here.



### 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
$W_0$	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`, `allpassshifc`, `iirshifc`

## References

Oppenheim, A.V., R.W. Schaffer 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

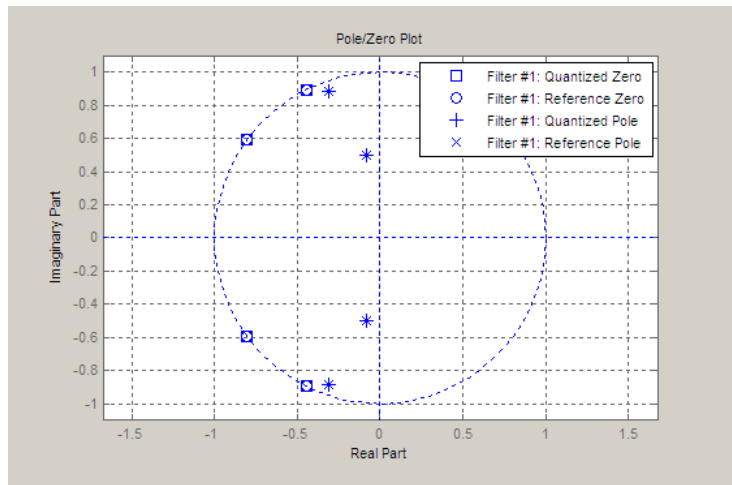
`[zq,pq,kq] = zplane(Hq)` returns the vectors of zeros `zq`, poles `pq`, and gains `kq`. If `Hq` has  $n$  sections, `zq`, `pq`, and `kq` are returned as 1-by- $n$  cell arrays. If there are no zeros (or no poles), `zq` (or `pq`) is set to the empty matrix `[]`.

`[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. Plot the quantized and unquantized poles and zeros associated with this quantized filter.

```
[b,a] = ellip(4,.5,20,.6);  
Hq = dfilt.df2(b, a);  
Hq.arithmetic = 'fixed';  
zplane(Hq);
```

**See Also**

freqz, impz





# Reference for the Properties of Filter Objects

---

- “Fixed-Point Filter Properties” on page 3-2
- “Adaptive Filter Properties” on page 3-102
- “References” on page 3-115
- “Multirate Filter Properties” on page 3-116
- “References” on page 3-132

## Fixed-Point Filter Properties

In this section...
“Overview of Fixed-Point Filters” on page 3-2
“Fixed-Point Objects and Filters” on page 3-2
“Summary — Fixed-Point Filter Properties” on page 3-5
“Property Details for Fixed-Point Filters” on page 3-18

### Overview of Fixed-Point Filters

There is a distinction between fixed-point filters and quantized filters — quantized filters represent a superset that includes fixed-point filters.

When `dfilt` objects have their `Arithmetic` property set to `single` or `fixed`, they are quantized filters. However, after you set the `Arithmetic` property to `fixed`, the resulting filter is both quantized and fixed-point. Fixed-point filters perform arithmetic operations without allowing the binary point to move in response to the calculation — hence the name fixed-point. You can find out more about fixed-point arithmetic in your Fixed-Point Toolbox documentation or from the Help system.

With the `Arithmetic` property set to `single`, meaning the filter uses single-precision floating-point arithmetic, the filter allows the binary point to move during mathematical operations, such as sums or products. Therefore these filters cannot be considered fixed-point filters. But they are quantized filters.

The following sections present the properties for fixed-point filters, which includes all the properties for double-precision and single-precision floating-point filters as well.

### Fixed-Point Objects and Filters

Fixed-point filters depend in part on fixed-point objects from Fixed-Point Toolbox software. You can see this when you display a fixed-point filter at the command prompt.

```
hd=dfilt.df2t
```

```
hd =  
  
    FilterStructure: 'Direct-Form II Transposed'  
    Arithmetic: 'double'  
    Numerator: 1  
    Denominator: 1  
    PersistentMemory: false  
    States: [0x1 double]  
  
set(hd,'arithmetic','fixed')  
hd  
  
hd =  
  
    FilterStructure: 'Direct-Form II Transposed'  
    Arithmetic: 'fixed'  
    Numerator: 1  
    Denominator: 1  
    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'  
  
    AccumWordLength: 40  
    CastBeforeSum: true
```

```
RoundMode: 'convergent'  
OverflowMode: 'wrap'
```

Look at the States property, shown here

```
States: [1x1 embedded.fi]
```

The notation `embedded.fi` indicates that the states are being represented by fixed-point objects, usually called `fi` objects. If you take a closer look at the property `States`, you see how the properties of the `fi` object represent the values for the filter states.

```
hd.states
```

```
ans =
```

```
[]
```

```
DataType: Fixed  
Scaling: BinaryPoint  
Signed: true  
WordLength: 16  
FractionLength: 15
```

```
RoundMode: round  
OverflowMode: saturate  
ProductMode: FullPrecision  
MaxProductWordLength: 128  
SumMode: FullPrecision  
MaxSumWordLength: 128  
CastBeforeSum: true
```

To learn more about `fi` objects (fixed-point objects) in general, refer to your Fixed-Point Toolbox documentation. Commands like the following can help you get the information you are looking for:

```
docsearch(fixed-point object)
```

or

```
docsearch(fi)
```

Either command opens the Help system and searches for information about fixed-point objects in Fixed-Point Toolbox software.

As inputs (data to be filtered), fixed-point filters accept both regular double-precision values and `fi` objects. Which you use depends on your needs. How your filter responds to the input data is determined by the settings of the filter properties, discussed in the next few sections.

## Summary – Fixed-Point Filter Properties

Discrete-time filters in this toolbox use objects that perform the filtering and configuration of the filter. As objects, they include properties and methods that are often referred to as functions — not strictly the same as MATLAB functions but mostly so) to provide filtering capability. In discrete-time filters, or `dfilt` objects, many of the properties are dynamic, meaning they become available depending on the settings of other properties in the `dfilt` object or filter.

### Dynamic Properties

When you use a `dfilt.structure` function to create a filter, MATLAB displays the filter properties in the command window in return (unless you end the command with a semicolon which suppresses the output display). Generally you see six or seven properties, ranging from the property `FilterStructure` to `PersistentMemory`. These first properties are always present in the filter. One of the most important properties is `Arithmetic`. The `Arithmetic` property controls all of the dynamic properties for a filter.

Dynamic properties become available when you change another property in the filter. For example, when you change the `Arithmetic` property value to `fixed`, the display now shows many more properties for the filter, all of them considered dynamic. Here is an example that uses a direct form II filter. First create the default filter:

```
hd=dfilt.df2
```

```
hd =
```

```
FilterStructure: 'Direct-Form II'  
Arithmetic: 'double'
```

```
        Numerator: 1
        Denominator: 1
    PersistentMemory: false
        States: [0x1 double]
```

With the filter `hd` in the workspace, convert the arithmetic to fixed-point. Do this by setting the property `Arithmetic` to `fixed`. Notice the display. Instead of a few properties, the filter now has many more, each one related to a particular part of the filter and its operation. Each of the now-visible properties is dynamic.

```
hd.arithmetic='fixed'
```

```
hd =
```

```
    FilterStructure: 'Direct-Form II'
        Arithmetic: 'fixed'
            Numerator: 1
            Denominator: 1
    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'

    AccumWordLength: 40
        CastBeforeSum: true
```

```
RoundMode: 'convergent'
OverflowMode: 'wrap'
```

Even this list of properties is not yet complete. Changing the value of other properties such as the `ProductMode` or `CoeffAutoScale` properties may reveal even more properties that control how the filter works. Remember this feature about `dfilt` objects and dynamic properties as you review the rest of this section about properties of fixed-point filters.

An important distinction is you cannot change the value of a property unless you see the property listed in the default display for the filter. Entering the filter name at the MATLAB prompt generates the default property display for the named filter. Using `get(filtername)` does not generate the default display — it lists all of the filter properties, both those that you can change and those that are not available yet.

The following table summarizes the properties, static and dynamic, of fixed-point filters and provides a brief description of each. Full descriptions of each property, in alphabetical order, follow the table.

Property Name	Valid Values [Default Value]	Brief Description
<code>AccumFracLength</code>	Any positive or negative integer number of bits [29]	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 — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.
<code>AccumWordLength</code>	Any positive integer number of bits [40]	Sets the word length used to store data in the accumulator/buffer.
<code>Arithmetic</code>	[Double], single, fixed	Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.

<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
CastBeforeSum	[True] or false	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	[True] or 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 <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used.
CoeffFracLength	Any positive or negative integer number of bits [14]	Set the fraction length the filter uses to interpret coefficients. <code>CoeffFracLength</code> is not available until you set <code>CoeffAutoScale</code> to <code>false</code> . Scalar filters include this property.
CoeffWordLength	Any positive integer number of bits [16]	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of addition operations involving denominator coefficients.
DenFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .
Denominator	Any filter coefficient value [1]	Holds the denominator coefficients for IIR filters.



<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
DenProdFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value after you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .
DenStateFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
FracDelay	Any decimal value between 0 and 1 samples	Specifies the fractional delay provided by the filter, in decimal fractions of a sample.
FDAutoScale	[True] or false	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 <code>false</code> enables you to change the <code>FDWordLength</code> and <code>FDFracLength</code> properties to specify the data format applied.
FDFracLength	Any positive or negative integer number of bits [5]	Specifies the fraction length to represent the fractional delay.
FDProdFracLength	Any positive or negative integer number of bits [34]	Specifies the fraction length to represent the result of multiplying the coefficients with the fractional delay.
FDProdWordLength	Any positive or negative integer number of bits [39]	Specifies the word length to represent result of multiplying the coefficients with the fractional delay.
FDWordLength	Any positive or negative integer number of bits [6]	Specifies the word length to represent the fractional delay.

<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
DenStateWordLength	Any positive integer number of bits [16]	Specifies the word length used to represent the states associated with denominator coefficients in the filter.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter sets the output word and fraction lengths, and the accumulator word and fraction lengths automatically to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.
FilterStructure	Not applicable.	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	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret data to be processed by the filter.
InputWordLength	Any positive integer number of bits [16]	Specifies the word length applied to represent input data.
Ladder	Any ladder coefficients in double-precision data type [1]	latticearma filters include this property to store the ladder coefficients.
LadderAccumFracLength	Any positive or negative integer number of bits [29]	latticearma filters use this to define the fraction length applied to values output by the accumulator that stores the results of ladder computations.

<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
LadderFracLength	Any positive or negative integer number of bits [14]	latticearma filters use ladder coefficients in the signal flow. This property determines the fraction length used to interpret the coefficients.
Lattice	Any lattice structure coefficients. No default value.	Stores the lattice coefficients for lattice-based filters.
LatticeAccumFracLength	Any positive or negative integer number of bits [29]	Specifies how the accumulator outputs the results of operations on the lattice coefficients.
LatticeFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length applied to the lattice coefficients.
MultiplicandFracLength	Any positive or negative integer number of bits [15]	Sets the fraction length for values used in product operations in the filter. Direct-form I transposed (df1t) filter structures include this property.
MultiplicandWordLength	Any positive integer number of bits [16]	Sets the word length applied to the values input to a multiply operation (the multiplicands). The filter structure df1t includes this property.
NumAccumFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients.
Numerator	Any double-precision filter coefficients [1]	Holds the numerator coefficient values for the filter.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.

<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
NumProdFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. You can change the property value after you set ProductMode to SpecifyPrecision.
NumStateFracLength	Any positive or negative integer number of bits [15]	For IIR filters, this defines the fraction length applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficients in the filter.
NumStateWordLength	Any positive integer number of bits [16]	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 coefficients in the filter.
OutputFracLength	Any positive or negative integer number of bits — [15] or [12] bits depending on the filter structure	Determines how the filter interprets the filtered data. You can change the value of OutputFracLength after you set OutputMode to SpecifyPrecision.
OutputMode	[AvoidOverflow], BestPrecision, SpecifyPrecision	Sets the mode the filter uses to scale the filtered input data. You have the following choices: <ul style="list-style-type: none"> <li>• AvoidOverflow — directs the filter to set the output data fraction length to avoid causing the data to overflow.</li> <li>• BestPrecision — directs the filter to set the output data fraction length to maximize the precision in the output data.</li> </ul>

Property Name	Valid Values [Default Value]	Brief Description
		<ul style="list-style-type: none"> <li>SpecifyPrecision — lets you set the fraction length used by the filtered data.</li> </ul>
OutputWordLength	Any positive integer number of bits [16]	Determines the word length used for the filtered data.
OverflowMode	Saturate or [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 — hey maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [29]	For the output from a product operation, this sets the fraction length used to interpret the numeric data. This property becomes writable (you can change the value) after you set ProductMode to SpecifyPrecision.

<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
ProductMode	[FullPrecision], KeepLSB, KeepMSB, SpecifyPrecision	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	Any positive number of bits. Default is 16 or 32 depending on the filter structure	Specifies the word length to use for the results of multiplication operations. This property becomes writable (you can change the value) after you set ProductMode to SpecifyPrecision.
PersistentMemory	True or [false]	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. True is the default setting.
RoundMode	[Convergent], ceil, fix, floor, nearest, 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). <ul style="list-style-type: none"> <li>• <b>ceil</b> - Round toward positive infinity.</li> <li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> </ul>

Property Name	Valid Values [Default Value]	Brief Description
		<ul style="list-style-type: none"> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>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.</p>
ScaleValueFracLength	Any positive or negative integer number of bits [29]	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Available only when you disable <code>CoeffAutoScale</code> by setting it to <code>false</code> .
ScaleValues	[2 x 1 double] array with values of 1	Stores the scaling values for sections in SOS filters.
Signed	[True] or false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
sosMatrix	[1 0 0 1 0 0]	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.
SectionInputAuto Scale	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to prevent overflow by data entering a section of an SOS filter. Setting this property to false enables you to change the SectionInputFracLength property to specify the precision used. Available only for SOS filters.
SectionInputFrac Length	Any positive or negative integer number of bits [29]	Section values work with SOS filters. Setting this property controls how your filter interprets the section values between sections of the filter by setting the fraction length. This applies to data entering a section. Compare to SectionOutputFracLength. Available only when you disable SectionInputAutoScale by setting it to false.
SectionInputWord Length	Any positive or negative integer number of bits [29]	Sets the word length used to represent the data moving into a section of an SOS filter.



<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
SectionOutputAutoScale	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to prevent overflow by data leaving a section of an SOS filter. Setting this property to <b>false</b> enables you to change the SectionOutputFracLength property to specify the precision used.
SectionOutputFracLength	Any positive or negative integer number of bits [29]	Section values work with SOS filters. Setting this property controls how your filter interprets the section values between sections of the filter by setting the fraction length. This applies to data leaving a section. Compare to SectionInputFracLength. Available after you disable SectionOutputAutoScale by setting it to <b>false</b> .
SectionOutputWordLength	Any positive or negative integer number of bits [32]	Sets the word length used to represent the data moving out of one section of an SOS filter.
StateFracLength	Any positive or negative integer number of bits [15]	Lets you set the fraction length applied to interpret the filter states.
States	[1x1 embedded fi]	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Notice that the states use <b>fi</b> objects, with the associated properties from those objects. For details, refer to <b>filtstates</b> in your Signal Processing Toolbox documentation or in the Help system.

<b>Property Name</b>	<b>Valid Values [Default Value]</b>	<b>Brief Description</b>
StateWordLength	Any positive integer number of bits [16]	Sets the word length used to represent the filter states.
TapSumFracLength	Any positive or negative integer number of bits [15]	Sets the fraction length used to represent the filter tap values in addition operations. This is available after you set TapSumMode to false. Symmetric and antisymmetric FIR filters include this property.
TapSumMode	FullPrecision, KeepLSB, [KeepMSB], SpecifyPrecision	Determines how the accumulator outputs stored that involve filter tap weights. Choose from full precision (FullPrecision) to prevent overflows, or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when outputting results from the accumulator. To let you set the precision (the fraction length) used by the output from the accumulator, set FilterInternals to SpecifyPrecision.  Symmetric and antisymmetric FIR filters include this property.
TapSumWordLength	Any positive number of bits [17]	Sets the word length used to represent the filter tap weights during addition. Symmetric and antisymmetric FIR filters include this property.

### Property Details for Fixed-Point Filters

When you create a fixed-point filter, you are creating a filter object (a `dfilt` object). In this manual, the terms filter, `dfilt` object, and filter object are used interchangeably. To filter data, you apply the filter object to your data set. The output of the operation is the data filtered by the filter and the filter property values.

Filter objects have properties to which you assign property values. You use these property values to assign various characteristics to the filters you create, including

- The type of arithmetic to use in filtering operations
- The structure of the filter used to implement the filter (not a property you can set or change — you select it by the `dfilt.structure` function you choose)
- The locations of quantizations and cast operations in the filter
- The data formats used in quantizing, casting, and filtering operations

Details of the properties associated with fixed-point filters are described in alphabetical order on the following pages.

### **AccumFracLength**

Except for state-space filters, all `dfilt` objects that use fixed arithmetic have this property that defines the fraction length applied to data in the accumulator. Combined with `AccumWordLength`, `AccumFracLength` helps fully specify how the accumulator outputs data after processing addition operations. As with all fraction length properties, `AccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers.

### **AccumWordLength**

You use `AccumWordLength` to define the data word length used in the accumulator. Set this property to a value that matches your intended hardware. For example, many digital signal processors use 40-bit accumulators, so set `AccumWordLength` to 40 in your fixed-point filter:

```
set(hq,'arithmetic','fixed');  
set(hq,'AccumWordLength',40);
```

Note that `AccumWordLength` only applies to filters whose `Arithmetic` property value is `fixed`.

## Arithmetic

Perhaps the most important property when you are working with `dfilt` objects, `Arithmetic` determines the type of arithmetic the filter uses, and the properties or quantizers that compose the fixed-point or quantized filter. You use strings to set the `Arithmetic` property value.

The next table shows the valid strings for the `Arithmetic` property. Following the table, each property string appears with more detailed information about what happens when you select the string as the value for `Arithmetic` in your `dfilt`.

Arithmetic Property String	Brief Description of Effect on the Filter
<code>double</code>	All filtering operations and coefficients use double-precision floating-point representations and math. When you use <code>dfilt.structure</code> to create a filter object, <code>double</code> is the default value for the <code>Arithmetic</code> property.
<code>single</code>	All filtering operations and coefficients use single-precision floating-point representations and math.
<code>fixed</code>	This string applies selected default values for the properties in the fixed-point filter object, including such properties as coefficient word lengths, fraction lengths, and various operating modes. Generally, the default values match those you use on many digital signal processors. Allows signed fixed data types only. Fixed-point arithmetic filters are available only when you install Fixed-Point Toolbox software with this toolbox.

**double.** When you use one of the `dfilt.structure` methods to create a filter, the `Arithmetic` property value is `double` by default. Your filter is identical to the same filter without the `Arithmetic` property, as you would create if you used Signal Processing Toolbox software.

`Double` means that the filter uses double-precision floating-point arithmetic in all operations while filtering:

- All input to the filter must be double data type. Any other data type returns an error.
- The states and output are doubles as well.
- All internal calculations are done in double math.

When you use `double` data type filter coefficients, the reference and quantized (fixed-point) filter coefficients are identical. The filter stores the reference coefficients as double data type.

**single.** When your filter should use single-precision floating-point arithmetic, set the `Arithmetic` property to `single` so all arithmetic in the filter processing gets restricted to single-precision data type.

- Input data must be single data type. Other data types return errors.
- The filter states and filter output use single data type.

When you choose `single`, you can provide the filter coefficients in either of two ways:

- Double data type coefficients. With `Arithmetic` set to `single`, the filter casts the double data type coefficients to single data type representation.
- Single data type. These remain unchanged by the filter.

Depending on whether you specified single or double data type coefficients, the reference coefficients for the filter are stored in the data type you provided. If you provide coefficients in double data type, the reference coefficients are double as well. Providing single data type coefficients generates single data type reference coefficients. Note that the arithmetic used by the reference filter is always double.

When you use `reffilter` to create a reference filter from the reference coefficients, the resulting filter uses double-precision versions of the reference filter coefficients.

To set the `Arithmetic` property value, create your filter, then use `set` to change the `Arithmetic` setting, as shown in this example using a direct form FIR filter.

```
b=fir1(7,0.45);

hd=dfilt.dffir(b)

hd =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'double'
      Numerator: [1x8 double]
 PersistentMemory: false
      States: [7x1 double]

set(hd,'arithmetic','single')
hd

hd =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'single'
      Numerator: [1x8 double]
 PersistentMemory: false
      States: [7x1 single]
```

**fixed.** Converting your `dfilt` object to use fixed arithmetic results in a filter structure that uses properties and property values to match how the filter would behave on digital signal processing hardware.

---

**Note** The `fixed` option for the property `Arithmetic` is available only when you install Fixed-Point Toolbox software as well as Filter Design Toolbox software.

---

After you set `Arithmetic` to `fixed`, you are free to change any property value from the default value to a value that more closely matches your needs. You cannot, however, mix floating-point and fixed-point arithmetic in your filter when you select `fixed` as the `Arithmetic` property value. Choosing `fixed` restricts you to using either fixed-point or floating point throughout the filter (the data type must be homogenous). Also, all data types must be signed. `fixed` does not support unsigned data types except for unsigned coefficients

when you set the property `Signed` to `false`. Mixing word and fraction lengths within the fixed object is acceptable. In short, using fixed arithmetic assumes

- fixed word length.
- fixed size and dedicated accumulator and product registers.
- the ability to do either saturation or wrap arithmetic.
- that multiple rounding modes are available.

Making these assumptions simplifies your job of creating fixed-point filters by reducing repetition in the filter construction process, such as only requiring you to enter the accumulator word size once, rather than for each step that uses the accumulator.

Default property values are a starting point in tailoring your filter to common hardware, such as choosing 40-bit word length for the accumulator, or 16-bit words for data and coefficients.

In this `dfilt` object example, `get` returns the default values for `dfilt.df1t` structures.

```
[b,a]=butter(6,0.45);
hd=dfilt.df1(b,a)

hd =

    FilterStructure: 'Direct-Form I'
      Arithmetic: 'double'
      Numerator: [1x7 double]
      Denominator: [1x7 double]
 PersistentMemory: false
      States: Numerator: [6x1 double]
            Denominator:[6x1 double]

set(hd,'arithmetic','fixed')
get(hd)
 PersistentMemory: false
    FilterStructure: 'Direct-Form I'
      States: [1x1 filtstates.df1ir]
```

```
        Numerator: [1x7 double]
        Denominator: [1x7 double]
        Arithmetic: 'fixed'
    CoeffWordLength: 16
    CoeffAutoScale: 1
        Signed: 1
        RoundMode: 'convergent'
        OverflowMode: 'wrap'
    InputWordLength: 16
    InputFracLength: 15
        ProductMode: 'FullPrecision'
    OutputWordLength: 16
    OutputFracLength: 15
        NumFracLength: 16
        DenFracLength: 14
    ProductWordLength: 32
    NumProdFracLength: 31
    DenProdFracLength: 29
        AccumWordLength: 40
    NumAccumFracLength: 31
    DenAccumFracLength: 29
        CastBeforeSum: 1
```

Here is the default display for `hd`.

```
hd
```

```
hd =
```

```
    FilterStructure: 'Direct-Form I'
        Arithmetic: 'fixed'
        Numerator: [1x7 double]
        Denominator: [1x7 double]
    PersistentMemory: false
        States: Numerator: [6x1 fi]
              Denominator:[6x1 fi]

    CoeffWordLength: 16
    CoeffAutoScale: true
```



```

Signed: true

InputWordLength: 16
InputFracLength: 15

OutputWordLength: 16
OutputFracLength: 15

ProductMode: 'FullPrecision'

AccumWordLength: 40
CastBeforeSum: true

RoundMode: 'convergent'
OverflowMode: 'wrap'

```

This second example shows the default property values for `dfilt.latticemamax` filter objects, using the coefficients from an `fir1` filter.

```

b=fir1(7,0.45)

hdlat=dfilt.latticemamax(b)

hdlat =

    FilterStructure: [1x45 char]
    Arithmetic: 'double'
    Lattice: [1x8 double]
    PersistentMemory: false
    States: [8x1 double]

hdlat.arithmetic='fixed'

hdlat =

    FilterStructure: [1x45 char]
    Arithmetic: 'fixed'
    Lattice: [1x8 double]
    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'

AccumWordLength: 40
CastBeforeSum: true

RoundMode: 'convergent'
OverflowMode: 'wrap'
```

Unlike the `single` or `double` options for `Arithmetic`, `fixed` uses properties to define the word and fraction lengths for each portion of your filter. By changing the property value of any of the properties, you control your filter performance. Every word length and fraction length property is independent — set the one you need and the others remain unchanged, such as setting the input word length with `InputWordLength`, while leaving the fraction length the same.

```
d=fdesign.lowpass('n,fc',6,0.45)
```

```
d =
```

```
Response: 'Lowpass with cutoff'
Specification: 'N,Fc'
Description: {2x1 cell}
NormalizedFrequency: true
Fs: 'Normalized'
```

```
FilterOrder: 6
Fcutoff: 0.4500
```

```
designmethods(d)
```

```
Design Methods for class fdesign.lowpass:
```

```
butter
```

```
hd=butter(d)
```

```
hd =
```

```
FilterStructure: 'Direct-Form II, Second-Order Sections'
Arithmetic: 'double'
  sosMatrix: [3x6 double]
  ScaleValues: [4x1 double]
PersistentMemory: false
  States: [2x3 double]
```

```
hd.arithmetic='fixed'
```

```
hd =
```

```
FilterStructure: 'Direct-Form II, Second-Order Sections'
Arithmetic: 'fixed'
  sosMatrix: [3x6 double]
  ScaleValues: [4x1 double]
PersistentMemory: false
  States: [1x1 embedded.fi]
```

```
CoeffWordLength: 16
  CoeffAutoScale: true
  Signed: true
```

```
InputWordLength: 16
InputFracLength: 15
```

```
SectionInputWordLength: 16
  SectionInputAutoScale: true

SectionOutputWordLength: 16
Section OutputAutoScale: true

  OutputWordLength: 16
    OutputMode: 'AvoidOverflow'

  StateWordLength: 16
  StateFracLength: 15

    ProductMode: 'FullPrecision'

  AccumWordLength: 40
  CastBeforeSum: true

    RoundMode: 'convergent'
    OverflowMode: 'wrap'

hd.inputWordLength=12

hd =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
  Arithmetic: 'fixed'
  sosMatrix: [3x6 double]
  ScaleValues: [4x1 double]
PersistentMemory: false
  States: [1x1 embedded.fi]

  CoeffWordLength: 16
  CoeffAutoScale: true
  Signed: true

  InputWordLength: 12
  InputFracLength: 15

  SectionInputWordLength: 16
```

```
SectionInputAutoScale: true

SectionOutputWordLength: 16
SectionOutputAutoScale: true

    OutputWordLength: 16
        OutputMode: 'AvoidOverflow'

    StateWordLength: 16
    StateFracLength: 15

        ProductMode: 'FullPrecision'

    AccumWordLength: 40
    CastBeforeSum: true

        RoundMode: 'convergent'
        OverflowMode: 'wrap'
```

Notice that the properties for the lattice filter `hdlat` and direct-form II filter `hd` are different, as befits their differing filter structures. Also, some properties are common to both objects, such as `RoundMode` and `PersistentMemory` and behave the same way in both objects.

**Notes About Fraction Length, Word Length, and Precision.** Word length and fraction length combine to make the format for a fixed-point number, where word length is the number of bits used to represent the value and fraction length specifies, in bits, the location of the binary point in the fixed-point representation. Therein lies a problem — fraction length, which you specify in bits, can be larger than the word length, or a negative number of bits. This section explains how that idea works and how you might use it.

Unfortunately fraction length is somewhat misnamed (although it continues to be used in this User's Guide and elsewhere for historical reasons).

Fraction length defined as the number of fractional bits (bits to the right of the binary point) is true only when the fraction length is positive and less than or equal to the word length. In MATLAB format notation you can use `[word length fraction length]`. For example, for the format `[16 16]`, the second 16 (the

fraction length) is the number of fractional bits or bits to the right of the binary point. In this example, all 16 bits are to the right of the binary point.

But it is also possible to have fixed-point formats of [16 18] or [16 -45]. In these cases the fraction length can no longer be the number of bits to the right of the binary point since the format says the word length is 16 — there cannot be 18 fraction length bits on the right. And how can there be a negative number of bits for the fraction length, such as [16 -45]?

A better way to think about fixed-point format [word length fraction length] and what it means is that the representation of a fixed-point number is a weighted sum of powers of two driven by the fraction length, or the two's complement representation of the fixed-point number.

Consider the format [B L], where the fraction length L can be positive, negative, 0, greater than B (the word length) or less than B. (B and L are always integers and B is always positive.)

Given a binary string  $b(1) b(2) b(3) \dots b(B)$ , to determine the two's-complement value of the string in the format described by [B L], use the value of the individual bits in the binary string in the following formula, where  $b(1)$  is the first binary bit (and most significant bit, MSB),  $b(2)$  is the second, and on up to  $b(B)$ .

The decimal numeric value that those bits represent is given by

$$\text{value} = -b(1) * 2^{(B-L-1)} + b(2) * 2^{(B-L-2)} + b(3) * 2^{(B-L-3)} + \dots + b(B) * 2^{(-L)}$$

L, the fraction length, represents the negative of the weight of the last, or least significant bit (LSB). L is also the step size or the precision provided by a given fraction length.

**Precision.** Here is how precision works.

When all of the bits of a binary string are zero except for the LSB (which is therefore equal to one), the value represented by the bit string is given by  $2^{(-L)}$ . If L is negative, for example  $L=-16$ , the value is  $2^{16}$ . The smallest step between numbers that can be represented in a format where  $L=-16$  is given by  $1 \times 2^{16}$  (the rightmost term in the formula above), which is 65536. Note the precision does not depend on the word length.

Take a look at another example. When the word length set to 8 bits, the decimal value 12 is represented in binary by 00001100. That 12 is the decimal equivalent of 00001100 tells you that you are using [8 0] data format representation — the word length is 8 bits and fraction length 0 bits, and the step size or precision (the smallest difference between two adjacent values in the format [8,0], is  $2^0=1$ .

Suppose you plan to keep only the upper 5 bits and discard the other three. The resulting precision after removing the right-most three bits comes from the weight of the lowest remaining bit, the fifth bit from the left, which is  $2^3=8$ , so the format would be [5,-3].

Note that in this format the step size is 8, I cannot represent numbers that are between multiples of 8.

In MATLAB, with Fixed-Point Toolbox software installed:

```
x=8;
q=quantizer([8,0]); % Word length = 8, fraction length = 0
xq=quantize(q,x);
binxq=num2bin(q,xq);
q1=quantizer([5 -3]); % Word length = 5, fraction length = -3
xq1 = quantize(q1,xq);
binxq1=num2bin(q1,xq1);
binxq

binxq =

00001000

binxq1

binxq1 =

00001
```

But notice that in [5,-3] format, 00001 is the two's complement representation for 8, not for 1;  $q = \text{quantizer}([8 \ 0])$  and  $q1 = \text{quantizer}([5 \ -3])$  are not the same. They cover the about the same range —  $\text{range}(q) > \text{range}(q1)$  — but their quantization step is different —  $\text{eps}(q) = 8$ , and  $\text{eps}(q1) = 1$ .

Look at one more example. When you construct a quantizer `q`

```
q = quantizer([a,b])
```

the first element in `[a,b]` is `a`, the word length used for quantization. The second element in the expression, `b`, is related to the quantization step — the numerical difference between the two closest values that the quantizer can represent. This is also related to the weight given to the LSB. Note that  $2^{-b} = \text{eps}(q)$ .

Now construct two quantizers, `q1` and `q2`. Let `q1` use the format `[32,0]` and let `q2` use the format `[16, -16]`.

```
q1 = quantizer([32,0])  
q2 = quantizer([16,-16])
```

Quantizers `q1` and `q2` cover the same range, but `q2` has less precision. It covers the range in steps of  $2^{16}$ , while `q` covers the range in steps of 1.

This lost precision is due to (or can be used to model) throwing out 16 least-significant bits.

An important point to understand is that in `dfilt` objects and filtering you control which bits are carried from the sum and product operations in the filter to the filter output by setting the format for the output from the sum or product operation.

For instance, if you use `[16 0]` as the output format for a 32-bit result from a sum operation when the original format is `[32 0]`, you take the lower 16 bits from the result. If you use `[16 -16]`, you take the higher 16 bits of the original 32 bits. You could even take 16 bits somewhere in between the 32 bits by choosing something like `[16 -8]`, but you probably do not want to do that.

Filter scaling is directly implicated in the format and precision for a filter. When you know the filter input and output formats, as well as the filter internal formats, you can scale the inputs or outputs to stay within the format ranges. For more information about scaling filters, refer to “Converting from Floating-Point to Fixed-Point”.

Notice that overflows or saturation might occur at the filter input, filter output, or within the filter itself, such as during add or multiply or accumulate



operations. Improper scaling at any point in the filter can result in numerical errors that dramatically change the performance of your fixed-point filter implementation.

### **CastBeforeSum**

Setting the `CastBeforeSum` property determines how the filter handles the input values to sum operations in the filter. After you set your filter `Arithmetic` property value to `fixed`, you have the option of using `CastBeforeSum` to control the data type of some inputs (addends) to summations in your filter. To determine which addends reflect the `CastBeforeSum` property setting, refer to the reference page for the signal flow diagram for the filter structure.

`CastBeforeSum` specifies whether to cast selected addends to summations in the filter to the output format from the addition operation before performing the addition. When you specify `true` for the property value, 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.

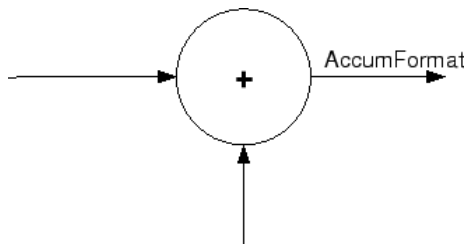
Specifying `CastBeforeSum` to be `false` prevents the addends from being cast to the output format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use.

Notice that the output format for every sum operation reflects the value of the output property specified in the filter structure diagram. Which input property is referenced by `CastBeforeSum` depends on the structure.

Property Value	Description
false	Configures filter summation operations to retain the addends in the format carried from the previous operation.
true	Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually this generates results from the summation that more closely match those found from digital signal processors

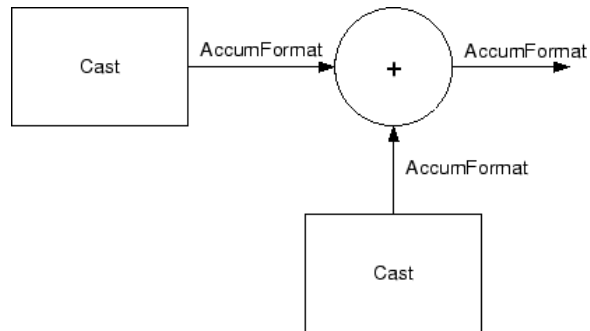
Another point — with `CastBeforeSum` set to `false`, the filter realization process inserts an intermediate data type format to hold temporarily the full precision sum of the inputs. A separate Convert block performs the process of casting the addition result to the accumulator format. This intermediate data format occurs because the Sum block in Simulink always casts input (addends) to the output data type.

**Diagrams of `CastBeforeSum` Settings.** When `CastBeforeSum` is `false`, sum elements in filter signal flow diagrams look like this:



showing that the input data to the sum operations (the addends) retain their format word length and fraction length from previous operations. The addition process uses the existing input formats and then casts the output to the format defined by `AccumFormat`. Thus the output data has the word length and fraction length defined by `AccumWordLength` and `AccumFracLength`.

When `CastBeforeSum` is `true`, sum elements in filter signal flow diagrams look like this:



showing that the input data gets recast to the accumulator format word length and fraction length (`AccumFormat`) before the sum operation occurs. The data output by the addition operation has the word length and fraction length defined by `AccumWordLength` and `AccumFracLength`.

### **CoeffAutoScale**

How the filter represents the filter coefficients depends on the property value of `CoeffAutoScale`. When you create a `dfilt` object, you use coefficients in double-precision format. Converting the `dfilt` object to fixed-point arithmetic forces the coefficients into a fixed-point representation. The representation the filter uses depends on whether the value of `CoeffAutoScale` is `true` or `false`.

- `CoeffAutoScale = true` means the filter chooses the fraction length to maintain the value of the coefficients as close to the double-precision values as possible. When you change the word length applied to the coefficients, the filter object changes the fraction length to try to accommodate the change. `true` is the default setting.
- `CoeffAutoScale = false` removes the automatic scaling of the fraction length for the coefficients and exposes the property that controls the coefficient fraction length so you can change it. For example, if the filter is a direct form FIR filter, setting `CoeffAutoScale = false` exposes the `NumFracLength` property that specifies the fraction length applied to numerator coefficients. If the filter is an IIR filter, setting `CoeffAutoScale = false` exposes both the `NumFracLength` and `DenFracLength` properties.

Here is an example of using `CoeffAutoScale` with a direct form filter.

```
hd2=dfilt.dffir([0.3 0.6 0.3])

hd2 =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'double'
      Numerator: [0.3000 0.6000 0.3000]
 PersistentMemory: false
      States: [2x1 double]

hd2.arithmetic='fixed'

hd2 =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'fixed'
      Numerator: [0.3000 0.6000 0.3000]
 PersistentMemory: false
      States: [1x1 embedded.fi]

    CoeffWordLength: 16
      CoeffAutoScale: true
      Signed: true

    InputWordLength: 16
    InputFracLength: 15

    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'

      ProductMode: 'FullPrecision'

    AccumWordLength: 40
      CastBeforeSum: true

      RoundMode: 'convergent'
      OverflowMode: 'wrap'
```

To this point, the filter coefficients retain the original values from when you created the filter as shown in the Numerator property. Now change the `CoeffAutoScale` property value from `true` to `false`.

```
hd2.coeffautoScale=false
```

```
hd2 =
```

```

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'fixed'
      Numerator: [0.3000 0.6000 0.3000]
    PersistentMemory: false
      States: [1x1 embedded.fi]

    CoeffWordLength: 16
    CoeffAutoScale: false
    NumFracLength: 15
    Signed: true

    InputWordLength: 16
    InputFracLength: 15

    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'

      ProductMode: 'FullPrecision'

    AccumWordLength: 40
    CastBeforeSum: true

      RoundMode: 'convergent'
      OverflowMode: 'wrap'

```

With the `NumFracLength` property now available, change the word length to 5 bits.

Notice the coefficient values. Setting `CoeffAutoScale` to `false` removes the automatic fraction length adjustment and the filter coefficients cannot be represented by the current format of [5 15] — a word length of 5 bits, fraction length of 15 bits.

```
hd2.coeffwordlength=5
```

```
hd2 =
```

```
    FilterStructure: 'Direct-Form FIR'  
      Arithmetic: 'fixed'  
      Numerator: [4.5776e-004 4.5776e-004 4.5776e-004]  
PersistentMemory: false  
      States: [1x1 embedded.fi]  
  
    CoeffWordLength: 5  
      CoeffAutoScale: false  
      NumFracLength: 15  
      Signed: true  
  
    InputWordLength: 16  
    InputFracLength: 15  
  
    OutputWordLength: 16  
      OutputMode: 'AvoidOverflow'  
  
      ProductMode: 'FullPrecision'  
  
    AccumWordLength: 40  
      CastBeforeSum: true  
  
      RoundMode: 'convergent'  
      OverflowMode: 'wrap'
```

Restoring `CoeffAutoScale` to `true` goes some way to fixing the coefficient values. Automatically scaling the coefficient fraction length results in setting the fraction length to 4 bits. You can check this with `get(hd2)` as shown below.

```
hd2.coeffautoScale=true
```

```
hd2 =
```

```
    FilterStructure: 'Direct-Form FIR'  
      Arithmetic: 'fixed'
```

```
        Numerator: [0.3125 0.6250 0.3125]
PersistentMemory: false
        States: [1x1 embedded.fi]

    CoeffWordLength: 5
    CoeffAutoScale: true
    Signed: true

    InputWordLength: 16
    InputFracLength: 15

    OutputWordLength: 16
    OutputMode: 'AvoidOverflow'

    ProductMode: 'FullPrecision'

    AccumWordLength: 40
    CastBeforeSum: true

    RoundMode: 'convergent'
    OverflowMode: 'wrap'

get(hd2)
    PersistentMemory: false
    FilterStructure: 'Direct-Form FIR'
        States: [1x1 embedded.fi]
        Numerator: [0.3125 0.6250 0.3125]
        Arithmetic: 'fixed'
    CoeffWordLength: 5
    CoeffAutoScale: 1
    Signed: 1
    RoundMode: 'convergent'
    OverflowMode: 'wrap'
    InputWordLength: 16
    InputFracLength: 15
    OutputWordLength: 16
    OutputMode: 'AvoidOverflow'
    ProductMode: 'FullPrecision'
    NumFracLength: 4
    OutputFracLength: 12
```

```
ProductWordLength: 21
ProductFracLength: 19
AccumWordLength: 40
AccumFracLength: 19
CastBeforeSum: 1
```

Clearly five bits is not enough to represent the coefficients accurately.

### **CoeffFracLength**

Fixed-point scalar filters that you create using `dfilt.scalar` use this property to define the fraction length applied to the scalar filter coefficients. Like the coefficient-fraction-length-related properties for the FIR, lattice, and IIR filters, `CoeffFracLength` is not displayed for scalar filters until you set `CoeffAutoScale` to `false`. Once you change the automatic scaling you can set the fraction length for the coefficients to any value you require.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 14 bits, with the `CoeffWordLength` of 16 bits.

### **CoeffWordLength**

One primary consideration in developing filters for hardware is the length of a data word. `CoeffWordLength` defines the word length for these data storage and arithmetic locations:

- Numerator and denominator filter coefficients
- Tap sum in `dfilt.dfsymfir` and `dfilt.dfasymfir` filter objects
- Section input, multiplicand, and state values in direct-form SOS filter objects such as `dfilt.df1t` and `dfilt.df2`
- Scale values in second-order filters
- Lattice and ladder coefficients in lattice filter objects, such as `dfilt.latticearma` and `dfilt.latticemamax`
- Gain in `dfilt.scalar`



Setting this property value controls the word length for the data listed. In most cases, the data words in this list have separate fraction length properties to define the associated fraction lengths.

Any positive, integer word length works here, limited by the machine you use to develop your filter and the hardware you use to deploy your filter.

### **DenAccumFracLength**

Filter structures `df1`, `df1t`, `df2`, and `df2t` that use `fixed` arithmetic have this property that defines the fraction length applied to denominator coefficients in the accumulator. In combination with `AccumWordLength`, the properties fully specify how the accumulator outputs data stored there.

As with all fraction length properties, `DenAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. To be able to change the property value for this property, you set `FilterInternals` to `SpecifyPrecision`.

### **DenFracLength**

Property `DenFracLength` contains the value that specifies the fraction length for the denominator coefficients for your filter. `DenFracLength` specifies the fraction length used to interpret the data stored in `C`. Used in combination with `CoeffWordLength`, these two properties define the interpretation of the coefficients stored in the vector that contains the denominator coefficients.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `CoeffWordLength` of 16 bits.

### **Denominator**

The denominator coefficients for your IIR filter, taken from the prototype you start with, are stored in this property. Generally this is a 1-by-N array of data in double format, where N is the length of the filter.

All IIR filter objects include `Denominator`, except the lattice-based filters which store their coefficients in the `Lattice` property, and second-order

section filters, such as `dfilt.df1tsos`, which use the `SosMatrix` property to hold the coefficients for the sections.

### **DenProdFracLength**

A property of all of the direct form IIR `dfilt` objects, except the ones that implement second-order sections, `DenProdFracLength` specifies the fraction length applied to data output from product operations that the filter performs on denominator coefficients.

Looking at the signal flow diagram for the `dfilt.df1t` filter, for example, you see that denominators and numerators are handled separately. When you set `ProductMode` to `SpecifyPrecision`, you can change the `DenProdFracLength` setting manually. Otherwise, for multiplication operations that use the denominator coefficients, the filter sets the fraction length as defined by the `ProductMode` setting.

### **DenStateFracLength**

When you look at the flow diagram for the `dfilt.df1sos` filter object, the states associated with denominator coefficient operations take the fraction length from this property. In combination with the `DenStateWordLength` property, these properties fully specify how the filter interprets the states.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `DenStateWordLength` of 16 bits.

### **DenStateWordLength**

When you look at the flow diagram for the `dfilt.df1sos` filter object, the states associated with the denominator coefficient operations take the data format from this property and the `DenStateFracLength` property. In combination, these properties fully specify how the filter interprets the state it uses.

By default, the value is 16 bits, with the `DenStateFracLength` of 15 bits.

## FilterInternals

Similar to the FilterInternals pane in FDATool, this property controls whether the filter sets the output word and fraction lengths automatically, and the accumulator word and fraction lengths automatically as well, to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. Setting FilterInternals to SpecifyPrecision exposes the output and accumulator related properties so you can set your own word and fraction lengths for them. Note that

## FilterStructure

Every `dfilt` object has a `FilterStructure` property. This is a read-only property containing a string that declares the structure of the filter object you created.

When you construct filter objects, the `FilterStructure` property value is returned containing one of the strings shown in the following table. Property `FilterStructure` indicates the filter architecture and comes from the constructor you use to create the filter.

After you create a filter object, you cannot change the `FilterStructure` property value. To make filters that use different structures, you construct new filters using the appropriate methods, or use `convert` to switch to a new structure.

**Default value.** Since this depends on the constructor you use and the constructor includes the filter structure definition, there is no default value. When you try to create a filter without specifying a structure, MATLAB returns an error.

Filter Constructor Name	FilterStructure Property String and Filter Type
'dfilt.df1'	Direct form I
'dfilt.df1sos'	Direct form I filter implemented using second-order sections
'dfilt.df1t'	Direct form I transposed
'dfilt.df2'	Direct form II

Filter Constructor Name	FilterStructure Property String and Filter Type
'dfilt.df2sos'	Direct form II filter implemented using second order sections
'dfilt.df2t'	Direct form II transposed
'dfilt.dfasymfir'	Antisymmetric finite impulse response (FIR). Even and odd forms.
'dfilt.dffir'	Direct form FIR
'dfilt.dffirt'	Direct form FIR transposed
'dfilt.latticeallpass'	Lattice allpass
'dfilt.latticear'	Lattice autoregressive (AR)
'dfilt.latticemamin'	Lattice moving average (MA) minimum phase
'dfilt.latticemamax'	Lattice moving average (MA) maximum phase
'dfilt.latticearma'	Lattice ARMA
'dfilt.dfsymfir'	Symmetric FIR. Even and odd forms
'dfilt.scalar'	Scalar

**Filter Structures with Quantizations Shown in Place.** To help you understand how and where the quantizations occur in filter structures in this toolbox, the figure below shows the structure for a Direct Form II filter, including the quantizations (fixed-point formats) that compose part of the fixed-point filter. You see that one or more quantization processes, specified by the *\*format* label, accompany each filter element, such as a delay, product, or summation element. The input to or output from each element reflects the result of applying the associated quantization as defined by the word length and fraction length format. Wherever a particular filter element appears in a filter structure, recall the quantization process that accompanies the element as it appears in this figure. Each filter reference page, such as the `dfilt.df2` reference page, includes the signal flow diagram showing the formatting elements that define the quantizations that occur throughout the filter flow.

For example, a product quantization, either numerator or denominator, follows every product (gain) element and a sum quantization, also either



To specify the filter structure, you select the appropriate `dfilt.structure` method to construct your filter. Refer to the function reference information for `dfilt` and `set` for details on setting property values for quantized filters.

The figures in the following subsections of this section serve as aids to help you determine how to enter your filter coefficients for each filter structure. Each subsection contains an example for constructing a filter of the given structure.

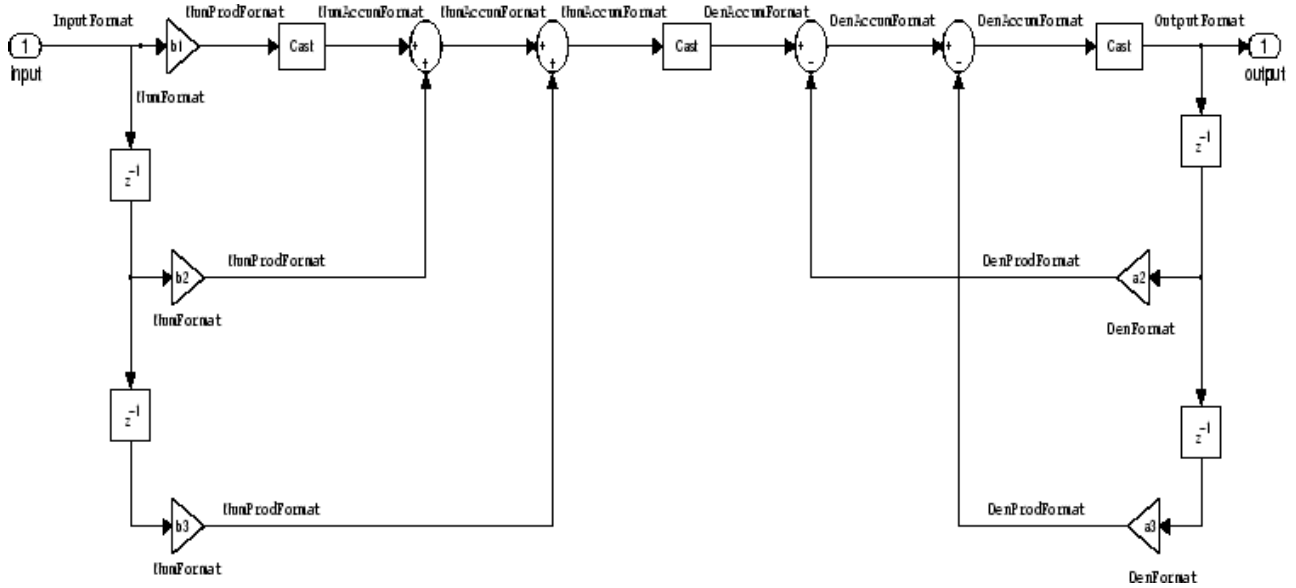
Scale factors for the input and output for the filters do not appear in the block diagrams. The default filter structures do not include, nor assume, the scale factors. For filter scaling information, refer to `scale` in the Help system.

**About the Filter Structure Diagrams.** In the diagrams that accompany the following filter structure descriptions, you see the active operators that define the filter, such as sums and gains, and the formatting features that control the processing in the filter. Notice also that the coefficients are labeled in the figure. This tells you the order in which the filter processes the coefficients.

While the meaning of the block elements is straightforward, the labels for the formats that form part of the filter are less clear. Each figure includes text in the form *labelFormat* that represents the existence of a formatting feature at that point in the structure. The *Format* stands for formatting object and the *label* specifies the data that the formatting object affects.

For example, in the `dfilt.df2` filter shown above, the entries `InputFormat` and `OutputFormat` are the formats applied, that is the word length and fraction length, to the filter input and output data. For example, filter properties like `OutputWordLength` and `InputWordLength` specify values that control filter operations at the input and output points in the structure and are represented by the formatting objects `InputFormat` and `OutputFormat` shown in the filter structure diagrams.

**Direct Form I Filter Structure.** The following figure depicts the *direct form I* filter structure that directly realizes a transfer function with a second-order numerator and denominator. The numerator coefficients are numbered  $b(i)$ ,  $i = 1, 2, 3$ ; the denominator coefficients are numbered  $a(i)$ ,  $i = 1, 2, 3$ ; and the states (used for initial and final state values in filtering) are labeled  $z(i)$ . In the figure, the Arithmetic property is set to fixed.



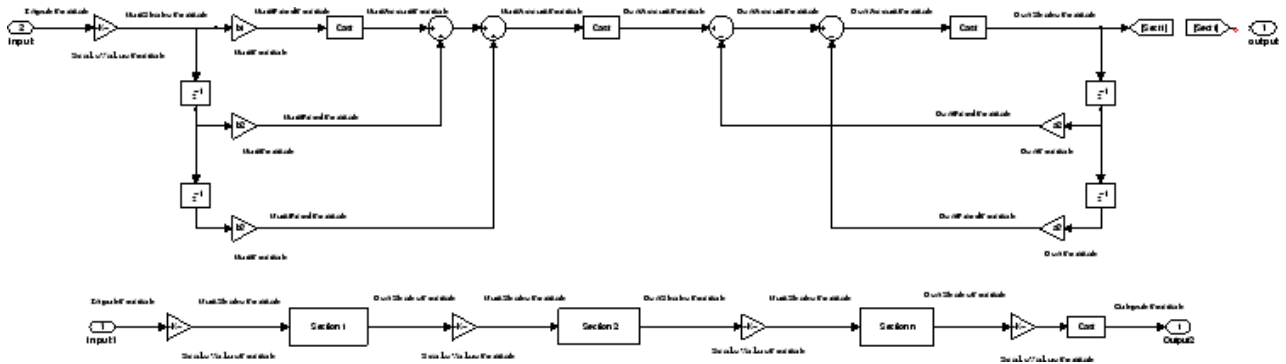
**Example – Specifying a Direct Form I Filter.** You can specify a second-order direct form I structure for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1(b,a);
```

To create the fixed-point filter, set the Arithmetic property to `fixed` as shown here.

```
set(hq,'arithmetic','fixed');
```

**Direct Form I Filter Structure With Second-Order Sections.** The following figure depicts a *direct form I* filter structure that directly realizes a transfer function with a second-order numerator and denominator and second-order sections. The numerator coefficients are numbered  $b(i)$ ,  $i = 1, 2, 3$ ; the denominator coefficients are numbered  $a(i)$ ,  $i = 1, 2, 3$ ; and the states (used for initial and final state values in filtering) are labeled  $z(i)$ . In the figure, the Arithmetic property is set to `fixed` to place the filter in fixed-point mode.



**Example — Specifying a Direct Form I Filter with Second-Order Sections.** You can specify an eighth-order direct form I structure for a quantized filter `hq` with the following code.

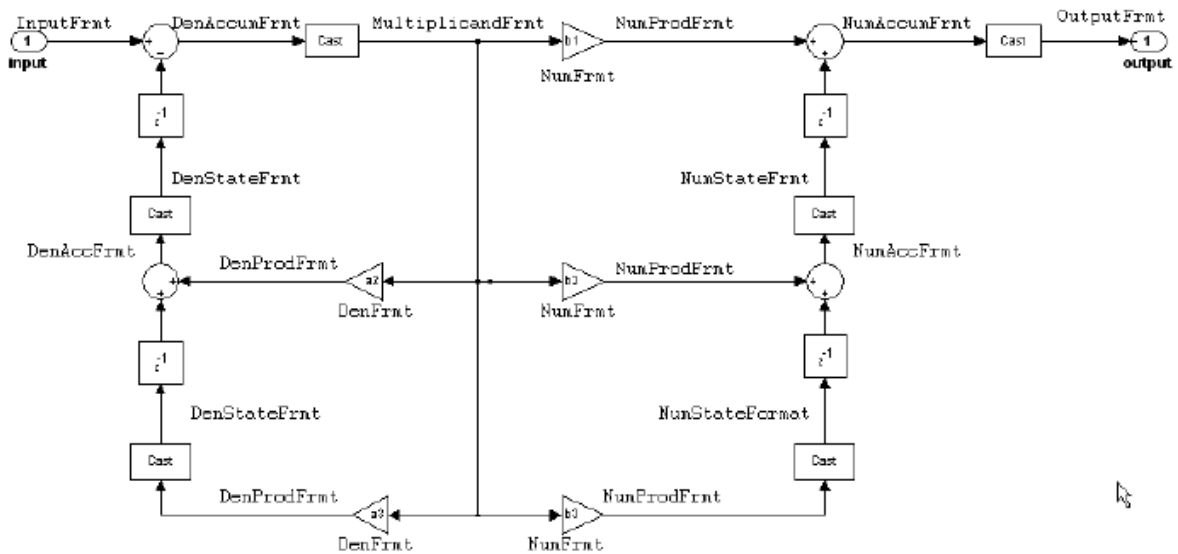
```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1sos(b,a);
```

To create the fixed-point filter, set the `Arithmetic` property to `fixed`, as shown here.

```
set(hq,'arithmetic','fixed');
```

**Direct Form I Transposed Filter Structure.** The next signal flow diagram depicts a *direct form I transposed* filter structure that directly realizes a transfer function with a second-order numerator and denominator. The numerator coefficients are  $b(i)$ ,  $i = 1, 2, 3$ ; the denominator coefficients are  $a(i)$ ,  $i = 1, 2, 3$ ; and the states (used for initial and final state values in filtering) are labeled  $z(i)$ . With the `Arithmetic` property value set to `fixed`, the figure shows the filter with the properties indicated.

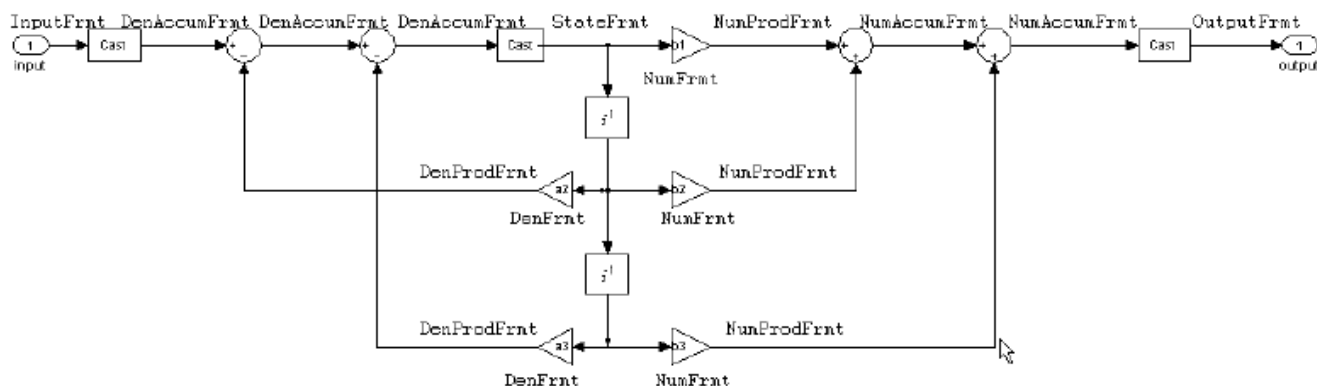




**Example – Specifying a Direct Form I Transposed Filter.** You can specify a second-order direct form I transposed filter structure for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1t(b,a);
set(hq,'arithmetic','fixed');
```

**Direct Form II Filter Structure.** The following graphic depicts a *direct form II* filter structure that directly realizes a transfer function with a second-order numerator and denominator. In the figure, the Arithmetic property value is fixed. Numerator coefficients are named  $b(i)$ ; denominator coefficients are named  $a(i)$ ,  $i = 1, 2, 3$ ; and the states (used for initial and final state values in filtering) are named  $z(i)$ .



Use the method `dfilt.df2` to construct a quantized filter whose `FilterStructure` property is `Direct-Form II`.

**Example – Specifying a Direct Form II Filter.** You can specify a second-order direct form II filter structure for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df2(b,a);
hq.arithmetic = 'fixed'
```

To convert your initial double-precision filter `hq` to a quantized or fixed-point filter, set the `Arithmetic` property to `fixed`, as shown.

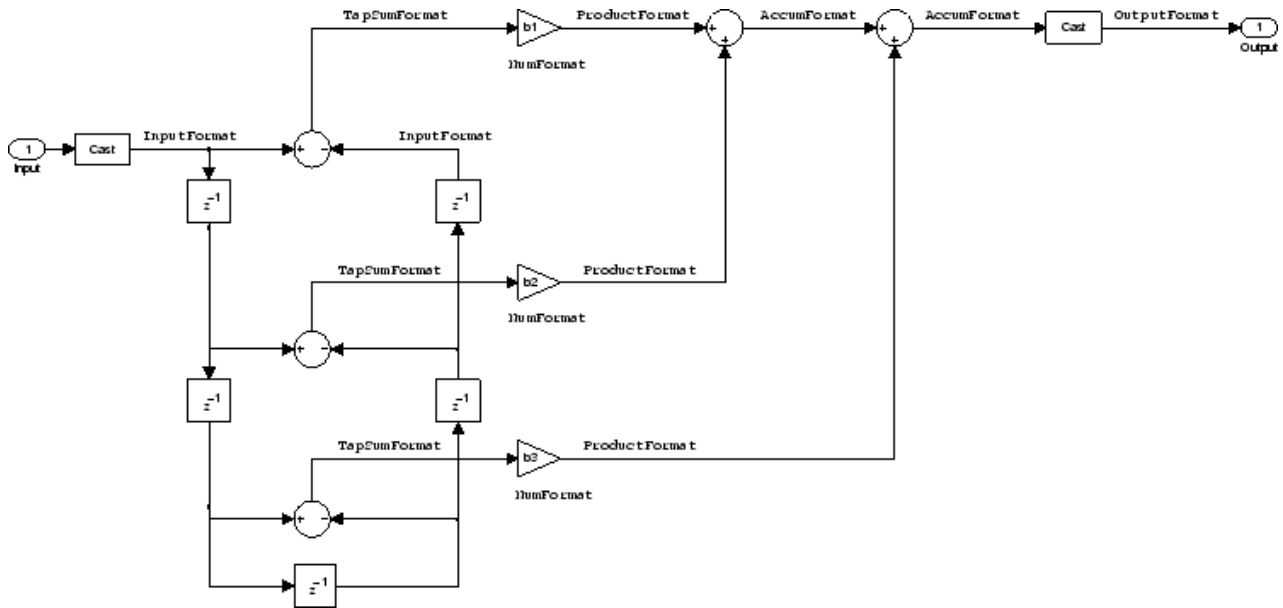
#### Direct Form II Filter Structure With Second-Order Sections

The following figure depicts *direct form II* filter structure using second-order sections that directly realizes a transfer function with a second-order numerator and denominator sections. In the figure, the `Arithmetic` property value is `fixed`. Numerator coefficients are labeled  $b(i)$ ; denominator coefficients are labeled  $a(i)$ ,  $i = 1, 2, 3$ ; and the states (used for initial and final state values in filtering) are labeled  $z(i)$ .





**Direct Form Antisymmetric FIR Filter Structure (Any Order).** The following figure depicts a *direct form antisymmetric FIR* filter structure that directly realizes a second-order antisymmetric FIR filter. The filter coefficients are labeled  $b(i)$ , and the initial and final state values in filtering are labeled  $z(i)$ . This structure reflects the Arithmetic property set to fixed.



Use the method `dfilt.dfasmfir` to construct the filter, and then set the Arithmetic property to fixed to convert to a fixed-point filter with this structure.

**Example – Specifying an Odd-Order Direct Form Antisymmetric FIR Filter.** Specify a fifth-order direct form antisymmetric FIR filter structure for a fixed-point filter `hq` with the following code.

```
b = [-0.008 0.06 -0.44 0.44 -0.06 0.008];
hq = dfilt.dfasmfir(b);
set(hq,'arithmetic','fixed')
```

```
hq
```

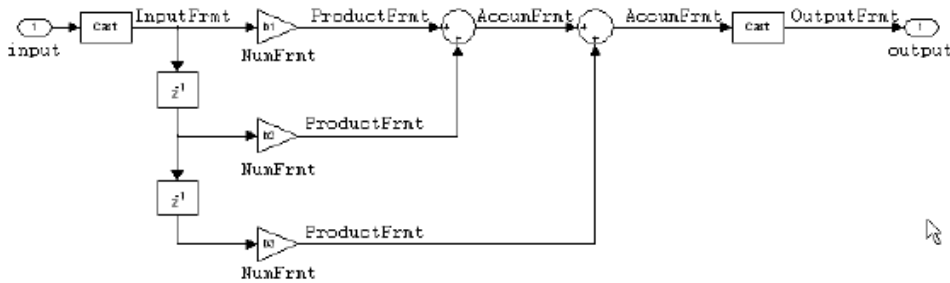
```
hq =  
  
    FilterStructure: 'Direct-Form Antisymmetric FIR'  
        Arithmetic: 'fixed'  
            Numerator: [-0.0080 0.0600 -0.4400 0.4400 -0.0600 0.0080]  
PersistentMemory: false  
        States: [1x1 fi object]  
  
    CoeffWordLength: 16  
        CoeffAutoScale: true  
            Signed: true  
  
    InputWordLength: 16  
    InputFracLength: 15  
  
    OutputWordLength: 16  
        OutputMode: 'AvoidOverflow'  
  
        TapSumMode: 'KeepMSB'  
    TapSumWordLength: 17  
  
        ProductMode: 'FullPrecision'  
  
    AccumWordLength: 40  
  
    CastBeforeSum: true  
        RoundMode: 'convergent'  
        OverflowMode: 'wrap'  
  
    InheritSettings: false
```

**Example — Specifying an Even-Order Direct Form Antisymmetric FIR Filter.** You can specify a fourth-order direct form antisymmetric FIR filter structure for a fixed-point filter `hq` with the following code.

```
b = [-0.01 0.1 0.0 -0.1 0.01];  
hq = dfilt.dfasymfir(b);  
hq.arithmetic='fixed'  
  
hq =
```

```
FilterStructure: 'Direct-Form Antisymmetric FIR'  
  Arithmetic: 'fixed'  
    Numerator: [-0.0100 0.1000 0 -0.1000 0.0100]  
PersistentMemory: false  
  States: [1x1 fi object]  
  
  CoeffWordLength: 16  
    CoeffAutoScale: true  
      Signed: true  
  
  InputWordLength: 16  
    InputFracLength: 15  
  
  OutputWordLength: 16  
    OutputMode: 'AvoidOverflow'  
  
    TapSumMode: 'KeepMSB'  
  TapSumWordLength: 17  
  
    ProductMode: 'FullPrecision'  
  
  AccumWordLength: 40  
  
    CastBeforeSum: true  
      RoundMode: 'convergent'  
        OverflowMode: 'wrap'  
  
  InheritSettings: false
```

**Direct Form Finite Impulse Response (FIR) Filter Structure.** In the next figure, you see the signal flow graph for a *direct form finite impulse response (FIR)* filter structure that directly realizes a second-order FIR filter. The filter coefficients are  $b(i)$ ,  $i = 1, 2, 3$ , and the states (used for initial and final state values in filtering) are  $z(i)$ . To generate the figure, set the `Arithmetic` property to `fixed` after you create your prototype filter in double-precision arithmetic.



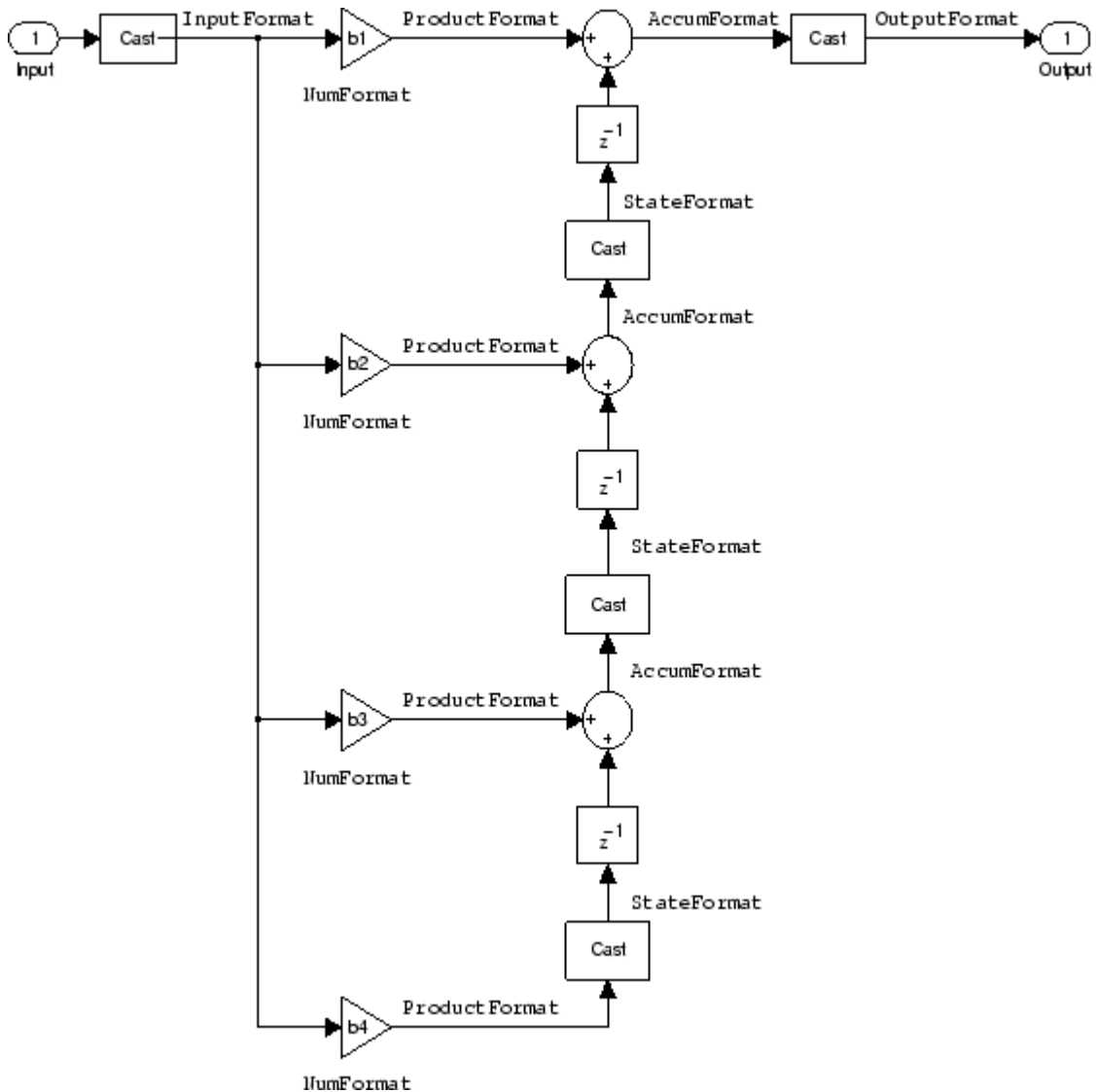
Use the `dfilt.dffir` method to generate a filter that uses this structure.

**Example — Specifying a Direct Form FIR Filter.** You can specify a second-order direct form FIR filter structure for a fixed-point filter `hq` with the following code.

```
b = [0.05 0.9 0.05];
hd = dfilt.dffir(b);
hq = set(hd,'arithmetic','fixed');
```

**Direct Form FIR Transposed Filter Structure.** This figure uses the filter coefficients labeled  $b(i)$ ,  $i = 1, 2, 3$ , and states (used for initial and final state values in filtering) are labeled  $z(i)$ . These depict a *direct form finite impulse response (FIR) transposed* filter structure that directly realizes a second-order FIR filter.



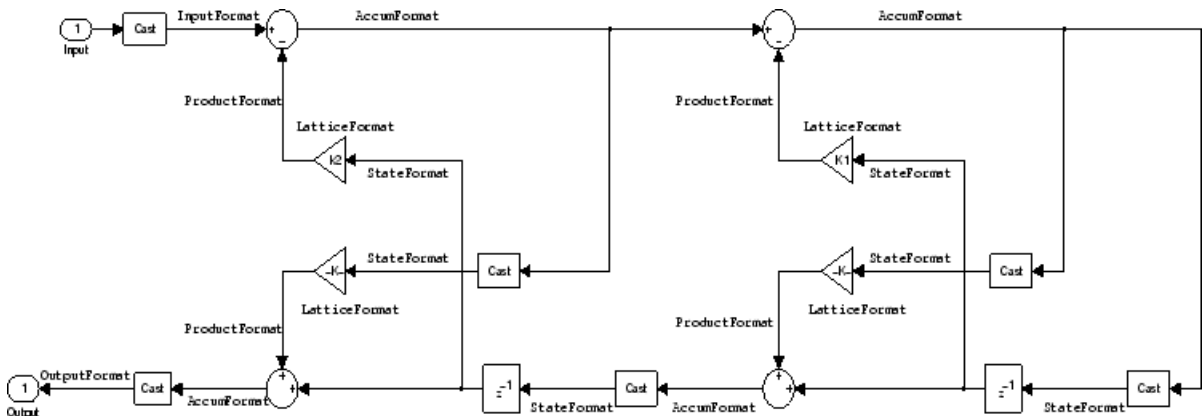


With the Arithmetic property set to fixed, your filter matches the figure. Using the method `dfilt.dffirt` returns a double-precision filter that you convert to a fixed-point filter.

**Example – Specifying a Direct Form FIR Transposed Filter.** You can specify a second-order direct form FIR transposed filter structure for a fixed-point filter `hq` with the following code.

```
b = [0.05 0.9 0.05];
hd=dfilt.dffirt(b);
hq = copy(hd);
hq.arithmetic = 'fixed';
```

**Lattice Allpass Filter Structure.** The following figure depicts the *lattice allpass* filter structure. The pictured structure directly realizes third-order lattice allpass filters using fixed-point arithmetic. The filter reflection coefficients are labeled  $k1(i)$ ,  $i = 1, 2, 3$ . The states (used for initial and final state values in filtering) are labeled  $z(i)$ .



To create a quantized filter that uses the lattice allpass structure shown in the figure, use the `dfilt.latticeallpass` method and set the `Arithmetic` property to `fixed`.

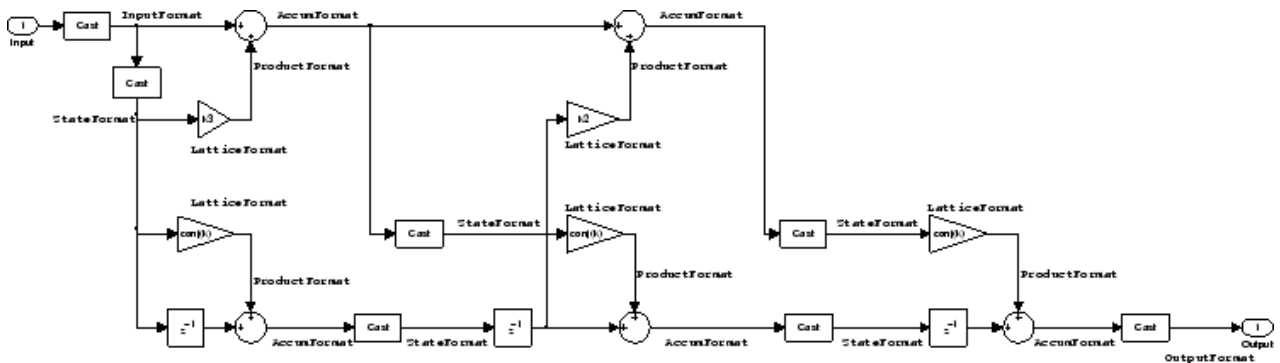
**Example – Specifying a Lattice Allpass Filter.** You can create a third-order lattice allpass filter structure for a quantized filter `hq` with the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticeallpass(k);
set(hq,'arithmetic','fixed');
```

**Lattice Moving Average Maximum Phase Filter Structure.** In the next figure you see a *lattice moving average maximum phase* filter structure. This signal flow diagram directly realizes a third-order lattice moving average (MA) filter with the following phase form depending on the initial transfer function:

- When you start with a minimum phase transfer function, the upper branch of the resulting lattice structure returns a minimum phase filter. The lower branch returns a maximum phase filter.
- When your transfer function is neither minimum phase nor maximum phase, the lattice moving average maximum phase structure will not be maximum phase.
- When you start with a maximum phase filter, the resulting lattice filter is maximum phase also.

The filter reflection coefficients are labeled  $k(i)$ ,  $i = 1, 2, 3$ . The states (used for initial and final state values in filtering) are labeled  $z(i)$ . In the figure, we set the Arithmetic property to fixed to reveal the fixed-point arithmetic format features that control such options as word length and fraction length.



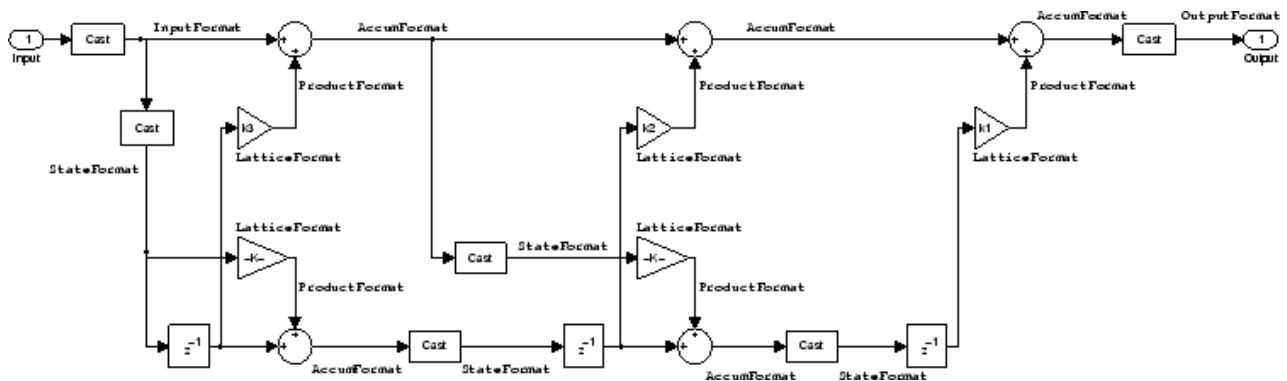
**Example — Constructing a Lattice Moving Average Maximum Phase Filter.** Constructing a fourth-order lattice MA maximum phase filter structure for a quantized filter `hq` begins with the following code.

```
k = [.66 .7 .44 .33];
hd=dfilt.latticemamax(k);
```



- When you start with a minimum phase transfer function, the upper branch of the resulting lattice structure returns a minimum phase filter. The lower branch returns a minimum phase filter.
- When your transfer function is neither minimum phase nor maximum phase, the lattice moving average minimum phase structure will not be minimum phase.
- When you start with a minimum phase filter, the resulting lattice filter is minimum phase also.

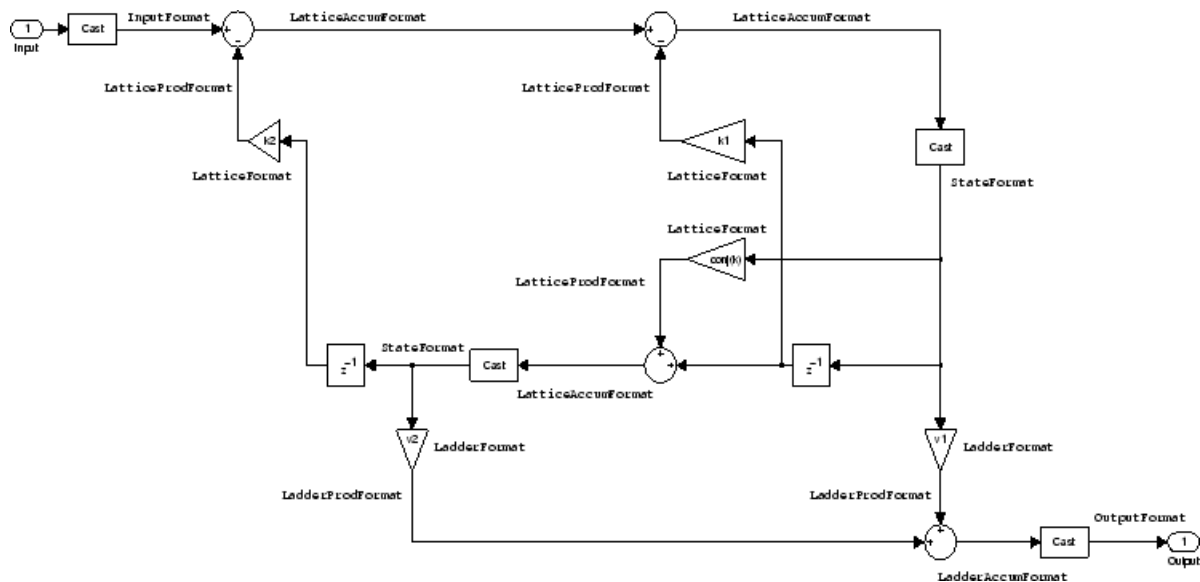
The filter reflection coefficients are labeled  $k(i)$ ,  $i = 1, 2, 3$ . The states (used for initial and final state values in filtering) are labeled  $z(i)$ . This figure shows the filter structure when the Arithmetic property is set to fixed to reveal the fixed-point arithmetic format features that control such options as word length and fraction length.



**Example — Specifying a Minimum Phase Lattice MA Filter.** You can specify a third-order lattice MA filter structure for minimum phase applications using variations of the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticemamin(k);
set(hq,'arithmetic','fixed');
```

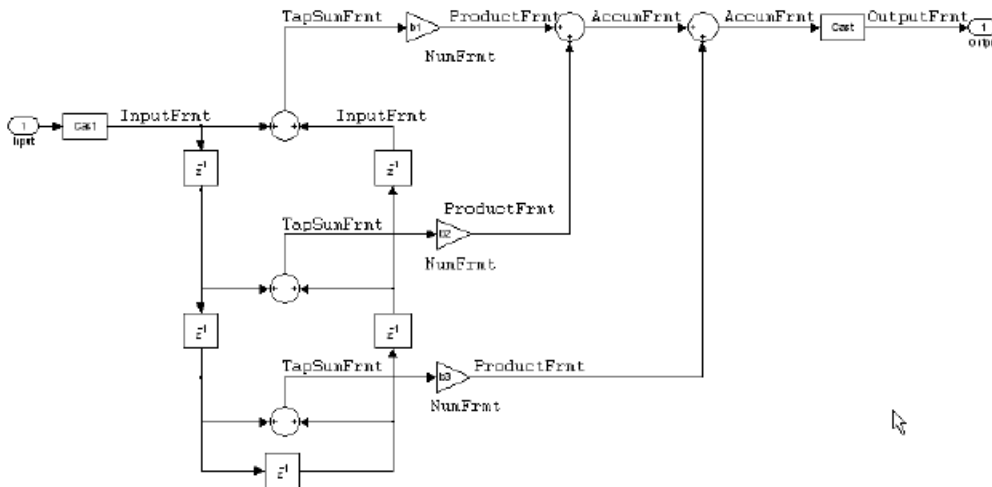
**Lattice Autoregressive Moving Average (ARMA) Filter Structure.** The figure below depicts a *lattice autoregressive moving average (ARMA)* filter structure that directly realizes a fourth-order lattice ARMA filter. The filter reflection coefficients are labeled  $k(i)$ ,  $(i) = 1, \dots, 4$ ; the ladder coefficients are labeled  $v(i)$ ,  $(i) = 1, 2, 3$ ; and the states (used for initial and final state values in filtering) are labeled  $z(i)$ .



**Example – Specifying an Lattice ARMA Filter.** The following code specifies a fourth-order lattice ARMA filter structure for a quantized filter `hq`, starting from `hd`, a floating-point version of the filter.

```
k = [.66 .7 .44 .66];
v = [1 0 0];
hd=dfilt.latticearma(k,v);
hq.arithmetic = 'fixed';
```

**Direct Form Symmetric FIR Filter Structure (Any Order).** Shown in the next figure, you see signal flow that depicts a *direct form symmetric FIR* filter structure that directly realizes a fifth-order direct form symmetric FIR filter. Filter coefficients are labeled  $b(i)$ ,  $i = 1, \dots, n$ , and states (used for initial and final state values in filtering) are labeled  $z(i)$ . Showing the filter structure used when you select fixed for the Arithmetic property value, the first figure details the properties in the filter object.



**Example — Specifying an Odd-Order Direct Form Symmetric FIR Filter.** By using the following code in MATLAB, you can specify a fifth-order direct form symmetric FIR filter for a fixed-point filter `hq`:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
hd=dfilt.dfsymfir(b);
set(hq,'arithmetic','fixed');
```

**Assigning Filter Coefficients.** The syntax you use to assign filter coefficients for your floating-point or fixed-point filter depends on the structure you select for your filter.

**Converting Filters Between Representations.** Filter conversion functions in this toolbox and in Signal Processing Toolbox software let you convert filter transfer functions to other filter forms, and from other filter forms to transfer function form. Relevant conversion functions include the following functions.

<b>Conversion Function</b>	<b>Description</b>
ca2tf	Converts from a coupled allpass filter to a transfer function.
cl2tf	Converts from a lattice coupled allpass filter to a transfer function.
convert	Convert a discrete-time filter from one filter structure to another.
sos	Converts quantized filters to create second-order sections. We recommend this method for converting quantized filters to second-order sections.
tf2ca	Converts from a transfer function to a coupled allpass filter.
tf2cl	Converts from a transfer function to a lattice coupled allpass filter.
tf2latc	Converts from a transfer function to a lattice filter.
tf2sos	Converts from a transfer function to a second-order section form.
tf2ss	Converts from a transfer function to state-space form.
tf2zp	Converts from a rational transfer function to its factored (single section) form (zero-pole-gain form).
zp2sos	Converts a zero-pole-gain form to a second-order section form.
zp2ss	Conversion of zero-pole-gain form to a state-space form.
zp2tf	Conversion of zero-pole-gain form to transfer functions of multiple order sections.



Note that these conversion routines do not apply to `dfilt` objects.

The function `convert` is a special case — when you use `convert` to change the filter structure of a fixed-point filter, you lose all of the filter states and settings. Your new filter has default values for all properties, and it is not fixed-point.

To demonstrate the changes that occur, convert a fixed-point direct form I transposed filter to direct form II structure.

```
hd=dfilt.df1t

hd =

    FilterStructure: 'Direct-Form I Transposed'
      Arithmetic: 'double'
      Numerator: 1
      Denominator: 1
 PersistentMemory: false
      States: Numerator: [0x0 double]
            Denominator:[0x0 double]

hd.arithmetic='fixed'
hd =

    FilterStructure: 'Direct-Form I Transposed'
      Arithmetic: 'fixed'
      Numerator: 1
      Denominator: 1
 PersistentMemory: false
      States: Numerator: [0x0 fi]
            Denominator:[0x0 fi]

convert(hd,'df2')

Warning: Using reference filter for structure conversion.
Fixed-point attributes will not be converted.

ans =
```

```
FilterStructure: 'Direct-Form II'  
Arithmetic: 'double'  
Numerator: 1  
Denominator: 1  
PersistentMemory: false  
States: [0x1 double]
```

You can specify a filter with  $L$  sections of arbitrary order by

- 1 Factoring your entire transfer function with `tf2zp`. This converts your transfer function to zero-pole-gain form.
- 2 Using `zp2tf` to compose the transfer function for each section from the selected first-order factors obtained in step 1.

---

**Note** You are not required to normalize the leading coefficients of each section's denominator polynomial when you specify second-order sections, though `tf2sos` does.

---

## Gain

`dfilt.scalar` filters have a gain value stored in the `gain` property. By default the gain value is one — the filter acts as a wire.

## InputFracLength

`InputFracLength` defines the fraction length assigned to the input data for your filter. Used in tandem with `InputWordLength`, the pair defines the data format for input data you provide for filtering.

As with all fraction length properties in `dfilt` objects, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, in this case `InputWordLength`, as well.

## InputWordLength

Specifies the number of bits your filter uses to represent your input data. Your word length option is limited by the arithmetic you choose — up to 32 bits for

double, float, and fixed. Setting Arithmetic to single (single-precision floating-point) limits word length to 16 bits. The default value is 16 bits.

## Ladder

Included as a property in `dfilt.latticearma` filter objects, Ladder contains the denominator coefficients that form an IIR lattice filter object. For instance, the following code creates a high pass filter object that uses the lattice ARMA structure.

```
[b,a]=cheby1(5,.5,.5,'high')

b =

    0.0282   -0.1409    0.2817   -0.2817    0.1409   -0.0282

a =

    1.0000    0.9437    1.4400    0.9629    0.5301    0.1620

hd=dfilt.latticearma(b,a)

hd =

    FilterStructure: [1x44 char]
      Arithmetic: 'double'
      Lattice: [1x6 double]
      Ladder: [1 0.9437 1.4400 0.9629 0.5301 0.1620]
 PersistentMemory: false
      States: [6x1 double]

hd.arithmetic='fixed'

hd =

    FilterStructure: [1x44 char]
      Arithmetic: 'fixed'
      Lattice: [1x6 double]
      Ladder: [1 0.9437 1.4400 0.9629 0.5301 0.1620]
```

```
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'

    AccumWordLength: 40
    CastBeforeSum: true

    RoundMode: 'convergent'
    OverflowMode: 'wrap'
```

#### **LadderAccumFracLength**

Autoregressive, moving average lattice filter objects (`latticearma`) use ladder coefficients to define the filter. In combination with `LadderFracLength` and `CoeffWordLength`, these three properties specify or reflect how the accumulator outputs data stored there. As with all fraction length properties, `LadderAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. The default value is 29 bits.

#### **LadderFracLength**

To let you control the way your `latticearma` filter interprets the denominator coefficients, `LadderFracLength` sets the fraction length applied to the ladder coefficients for your filter. The default value is 14 bits.

As with all fraction length properties, `LadderFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers.

## Lattice

When you create a lattice-based IIR filter, your numerator coefficients (from your IIR prototype filter or the default `dfilt` lattice filter function) get stored in the `Lattice` property of the `dfilt` object. The properties `CoeffWordLength` and `LatticeFracLength` define the data format the object uses to represent the lattice coefficients. By default, lattice coefficients are in double-precision format.

## LatticeAccumFracLength

Lattice filter objects (`latticeallpass`, `latticearma`, `latticeamax`, and `latticeamin`) use lattice coefficients to define the filter. In combination with `LatticeFracLength` and `CoeffWordLength`, these three properties specify how the accumulator outputs lattice coefficient-related data stored there. As with all fraction length properties, `LatticeAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. By default, the property is set to 31 bits.

## LatticeFracLength

To let you control the way your filter interprets the denominator coefficients, `LatticeFracLength` sets the fraction length applied to the lattice coefficients for your lattice filter. When you create the default lattice filter, `LatticeFracLength` is 16 bits.

As with all fraction length properties, `LatticeFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers.

## MultiplicandFracLength

Each input data element for a multiply operation has both word length and fraction length to define its representation. `MultiplicandFracLength` sets the fraction length to use when the filter object performs any multiply operation during filtering. For default filters, this is set to 15 bits.

As with all word and fraction length properties, `MultiplicandFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers.

### **MultiplicandWordLength**

Each input data element for a multiply operation has both word length and fraction length to define its representation. `MultiplicandWordLength` sets the word length to use when the filter performs any multiply operation during filtering. For default filters, this is set to 16 bits. Only the `df1t` and `df1tsos` filter objects include the `MultiplicandFracLength` property.

Only the `df1t` and `df1tsos` filter objects include the `MultiplicandWordLength` property.

### **NumAccumFracLength**

Filter structures `df1`, `df1t`, `df2`, and `df2t` that use fixed arithmetic have this property that defines the fraction length applied to numerator coefficients in output from the accumulator. In combination with `AccumWordLength`, the `NumAccumFracLength` property fully specifies how the accumulator outputs numerator-related data stored there.

As with all fraction length properties, `NumAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. 30 bits is the default value when you create the filter object. To be able to change the value for this property, set `FilterInternals` for the filter to `SpecifyPrecision`.

### **Numerator**

The numerator coefficients for your filter, taken from the prototype you start with or from the default filter, are stored in this property. Generally this is a 1-by-N array of data in double format, where N is the length of the filter.

All of the filter objects include `Numerator`, except the lattice-based and second-order section filters, such as `dfilt.latticema` and `dfilt.df1tsos`.

## NumFracLength

Property `NumFracLength` contains the value that specifies the fraction length for the numerator coefficients for your filter. `NumFracLength` specifies the fraction length used to interpret the numerator coefficients. Used in combination with `CoeffWordLength`, these two properties define the interpretation of the coefficients stored in the vector that contains the numerator coefficients.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `CoeffWordLength` of 16 bits.

## NumProdFracLength

A property of all of the direct form IIR `dfilt` objects, except the ones that implement second-order sections, `NumProdFracLength` specifies the fraction length applied to data output from product operations the filter performs on numerator coefficients.

Looking at the signal flow diagram for the `dfilt.df1t` filter, for example, you see that denominators and numerators are handled separately. When you set `ProductMode` to `SpecifyPrecision`, you can change the `NumProdFracLength` setting manually. Otherwise, for multiplication operations that use the numerator coefficients, the filter sets the word length as defined by the `ProductMode` setting.

## NumStateFracLength

All the variants of the direct form I structure include the property `NumStateFracLength` to store the fraction length applied to the numerator states for your filter object. By default, this property has the value 15 bits, with the `CoeffWordLength` of 16 bits, which you can change after you create the filter object.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well.

#### **NumStateWordLength**

When you look at the flow diagram for the `df1sos` filter object, the states associated with the numerator coefficient operations take the data format from this property and the `NumStateFracLength` property. In combination, these properties fully specify how the filter interprets the state it uses.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 16 bits, with the `NumStateFracLength` of 11 bits.

#### **OutputFracLength**

To define the output from your filter object, you need both the word and fraction lengths. `OutputFracLength` determines the fraction length applied to interpret the output data. Combining this with `OutputWordLength` fully specifies the format of the output.

Your fraction length can be any negative or positive integer, or zero. In addition, the fraction length you specify can be larger than the associated word length. Generally, the default value is 11 bits.

#### **OutputMode**

Sets the mode the filter uses to scale the filtered (output) data. You have the following choices:

- `AvoidOverflow` — directs the filter to set the property that controls the output data fraction length to avoid causing the data to overflow. In a `df2` filter, this would be the `OutputFracLength` property.
- `BestPrecision` — directs the filter to set the property that controls the output data fraction length to maximize the precision in the output data. For `df1t` filters, this is the `OutputFracLength` property. When you change the word length (`OutputWordLength`), the filter adjusts the fraction length to maintain the best precision for the new word size.
- `SpecifyPrecision` — lets you set the fraction length used by the filtered data. When you select this choice, you can set the output fraction length using the `OutputFracLength` property to define the output precision.



All filters include this property except the direct form I filter which takes the output format from the filter states.

Here is an example that changes the mode setting to `bestprecision`, and then adjusts the word length for the output.

```
hd=dfilt.df2

hd =

    FilterStructure: 'Direct-Form II'
      Arithmetic: 'double'
      Numerator: 1
      Denominator: 1
    PersistentMemory: false
      States: [0x1 double]

hd.arithmetic='fixed'

hd =

    FilterStructure: 'Direct-Form II'
      Arithmetic: 'fixed'
      Numerator: 1
      Denominator: 1
    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'

        AccumWordLength: 40
        CastBeforeSum: true

        RoundMode: 'convergent'
        OverflowMode: 'wrap'

get(hd)
    PersistentMemory: false
FilterStructure: 'Direct-Form II'
    States: [1x1 embedded.fi]
        Numerator: 1
        Denominator: 1
        Arithmetic: 'fixed'
    CoeffWordLength: 16
    CoeffAutoScale: 1
        Signed: 1
        RoundMode: 'convergent'
        OverflowMode: 'wrap'
    InputWordLength: 16
    InputFracLength: 15
    OutputWordLength: 16
        OutputMode: 'AvoidOverflow'
        ProductMode: 'FullPrecision'
    StateWordLength: 16
    StateFracLength: 15
        NumFracLength: 14
        DenFracLength: 14
    OutputFracLength: 13
    ProductWordLength: 32
    NumProdFracLength: 29
    DenProdFracLength: 29
        AccumWordLength: 40
        NumAccumFracLength: 29
        DenAccumFracLength: 29
        CastBeforeSum: 1

hd.outputMode='bestprecision'
```

```
hd =  
  
    FilterStructure: 'Direct-Form II'  
        Arithmetic: 'fixed'  
        Numerator: 1  
        Denominator: 1  
    PersistentMemory: false  
        States: [1x1 embedded.fi]  
  
    CoeffWordLength: 16  
        CoeffAutoScale: true  
        Signed: true  
  
    InputWordLength: 16  
    InputFracLength: 15  
  
    OutputWordLength: 16  
        OutputMode: 'BestPrecision'  
  
    StateWordLength: 16  
    StateFracLength: 15  
  
        ProductMode: 'FullPrecision'  
  
    AccumWordLength: 40  
        CastBeforeSum: true  
  
        RoundMode: 'convergent'  
        OverflowMode: 'wrap'  
  
hd.outputWordLength=8;  
  
get(hd)  
    PersistentMemory: false  
    FilterStructure: 'Direct-Form II'  
        States: [1x1 embedded.fi]  
        Numerator: 1  
        Denominator: 1  
        Arithmetic: 'fixed'
```

```
CoeffWordLength: 16
  CoeffAutoScale: 1
    Signed: 1
      RoundMode: 'convergent'
        OverflowMode: 'wrap'
          InputWordLength: 16
            InputFracLength: 15
              OutputWordLength: 8
                OutputMode: 'BestPrecision'
                  ProductMode: 'FullPrecision'
                    StateWordLength: 16
                      StateFracLength: 15
                        NumFracLength: 14
                          DenFracLength: 14
                            OutputFracLength: 5
                              ProductWordLength: 32
                                NumProdFracLength: 29
                                  DenProdFracLength: 29
                                    AccumWordLength: 40
                                      NumAccumFracLength: 29
                                        DenAccumFracLength: 29
                                          CastBeforeSum: 1
```

Changing the `OutputWordLength` to 8 bits caused the filter to change the `OutputFracLength` to 5 bits to keep the best precision for the output data.

### OutputWordLength

Use the property `OutputWordLength` to set the word length used by the output from your filter. Set this property to a value that matches your intended hardware. For example, some digital signal processors use 32-bit output so you would set `OutputWordLength` to 32.

```
[b,a] = butter(6,.5);
hd=dfilt.df1t(b,a);

set(hd,'arithmetic','fixed')

hd

hd =
```

```

    FilterStructure: 'Direct-Form I Transposed'
      Arithmetic: 'fixed'
      Numerator: [1x7 double]
      Denominator: [1 0 0.7777 0 0.1142 0 0.0018]
    PersistentMemory: false
      States: Numerator: [6x1 fi]
             Denominator:[6x1 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'

    AccumWordLength: 40
    CastBeforeSum: true

    RoundMode: 'convergent'
    OverflowMode: 'wrap'

    hd.outputwordLength=32

    hd =

    FilterStructure: 'Direct-Form I Transposed'
      Arithmetic: 'fixed'

```

```
    Numerator: [1x7 double]
    Denominator: [1 0 0.7777 0 0.1142 0 0.0018]
PersistentMemory: false
    States: Numerator: [6x1 fi]
           Denominator:[6x1 fi]

    CoeffWordLength: 16
    CoeffAutoScale: true
    Signed: true

    InputWordLength: 16
    InputFracLength: 15

    OutputWordLength: 32
    OutputMode: 'AvoidOverflow'

MultiplicandWordLength: 16
MultiplicandFracLength: 15

    StateWordLength: 16
    StateAutoScale: true

    ProductMode: 'FullPrecision'

    AccumWordLength: 40
    CastBeforeSum: true

    RoundMode: 'convergent'
    OverflowMode: 'wrap'
```

When you create a filter object, this property starts with the value 16.

### **OverflowMode**

The `OverflowMode` property is specified as one of the following two strings indicating how to respond to overflows in fixed-point arithmetic:

- `'saturate'` — saturate overflows.

When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the applicable word length and fraction length properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest. `saturate` is the default value for `OverflowMode`.

- 'wrap' — wrap all overflows to the range of representable values.

When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number. You can learn more about modular arithmetic in Fixed-Point Toolbox documentation.

These rules apply to the `OverflowMode` property.

- Applies to the accumulator and output data only.
- Does not apply to coefficients or input data. These always saturate the results.
- Does not apply to products. Products maintain full precision at all times. Your filters do not lose precision in the products.

---

**Note** Numbers in floating-point filters that extend beyond the dynamic range overflow to  $\pm\text{inf}$ .

---

## ProductFracLength

After you set `ProductMode` for a fixed-point filter to `SpecifyPrecision`, this property becomes available for you to change. `ProductFracLength` sets the fraction length the filter uses for the results of multiplication operations. Only the FIR filters such as asymmetric FIRs or lattice autoregressive filters include this dynamic property.

Your fraction length can be any negative or positive integer, or zero. In addition, the fraction length you specify can be larger than the associated word length. Generally, the default value is 11 bits.

## **ProductMode**

This property, available when your filter is in fixed-point arithmetic mode, specifies how the filter outputs the results of multiplication operations. All `dfilt` objects include this property when they use fixed-point arithmetic.

When available, you select from one of the following values for `ProductMode`:

- `FullPrecision` — means the filter automatically chooses the word length and fraction length it uses to represent the results of multiplication operations. The setting allow the product to retain the precision provided by the inputs (multiplicands) to the operation.
- `KeepMSB` — means you specify the word length for representing product operation results. The filter sets the fraction length to discard the LSBs, keep the higher order bits in the data, and maintain the precision.
- `KeepLSB` — means you specify the word length for representing the product operation results. The filter sets the fraction length to discard the MSBs, keep the lower order bits, and maintain the precision. Compare to the `KeepMSB` option.
- `SpecifyPrecision` — means you specify the word length and the fraction length to apply to data output from product operations.

When you switch to fixed-point filtering from floating-point, you are most likely going to throw away some data bits after product operations in your filter, perhaps because you have limited resources. When you have to discard some bits, you might choose to discard the least significant bits (LSB) from a result since the resulting quantization error would be small as the LSBs carry less weight. Or you might choose to keep the LSBs because the results have MSBs that are mostly zero, such as when your values are small relative to the range of the format in which they are represented. So the options for `ProductMode` let you choose how to maintain the information you need from the accumulator.

For more information about data formats, word length, and fraction length in fixed-point arithmetic, refer to “Notes About Fraction Length, Word Length, and Precision” on page 3-29.



## ProductWordLength

You use `ProductWordLength` to define the data word length used by the output from multiplication operations. Set this property to a value that matches your intended application. For example, the default value is 32 bits, but you can set any word length.

```
set(hq,'arithmetic','fixed');
set(hq,'ProductWordLength',64);
```

Note that `ProductWordLength` applies only to filters whose `Arithmetic` property value is `fixed`.

## PersistentMemory

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 object. `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` — the filter does not retain memory about filtering operations from one to the next. Maintaining memory (setting `PersistentMemory` to `true`) lets you filter large data sets as collections of smaller subsets and get the same result.

In this example, filter `hd` first filters data `xtot` in one pass. Then you can use `hd` to filter `x` as two separate data sets. The results `ytot` and `ysec` are the same in both cases.

```
xtot=[x,x];
ytot=filter(hd,xtot)
ytot =

           0  -0.0003   0.0005  -0.0014   0.0028  -0.0054   0.0092

reset(hm1); % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hd,x) filter(hd,x)]

ysec =

           0  -0.0003   0.0005  -0.0014   0.0028  -0.0054   0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

### **RoundMode**

The `RoundMode` property value specifies the rounding method used for quantizing numerical values. Specify the `RoundMode` property values as one of the following five strings.

<b>RoundMode String</b>	<b>Description of Rounding Algorithm</b>
Ceiling	Round toward positive infinity.
Floor	Round toward negative infinity.
Nearest	Round toward nearest. Ties round toward positive infinity.
Nearest (Convergent)	Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.
Round	Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
Zero	Round toward zero.

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**

Filter structures `df1sos`, `df1tsos`, `df2sos`, and `df2tsos` that use fixed arithmetic have this property that defines the fraction length applied to the scale values the filter uses between sections. In combination with `CoeffWordLength`, these two properties fully specify how the filter interprets and uses the scale values stored in the property `ScaleValues`. As with fraction length properties, `ScaleValueFracLength` can be any integer,

including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter.

### ScaleValues

The `ScaleValues` property values are specified as a scalar (or vector) that introduces scaling for inputs (and the outputs from cascaded sections in the vector case) during filtering:

- When you only have a single section in your filter:
  - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
  - Specify the `ScaleValues` property as a vector of length 2 if you want to specify scaling to the input (scaled with the first entry in the vector) and the output (scaled with the last entry in the vector).
- When you have  $L$  cascaded sections in your filter:
  - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
  - Specify the value for the `ScaleValues` property as a vector of length  $L+1$  if you want to scale the inputs to every section in your filter, along with the output:

The first entry of your vector specifies the input scaling

Each successive entry specifies the scaling at the output of the next section

The final entry specifies the scaling for the filter output.

The default value for `ScaleValues` is 0.

The interpretation of this property is described as follows with diagrams in “Interpreting the `ScaleValues` Property” on page 3-84.

---

**Note** The value of the `ScaleValues` property is not quantized. Data affected by the presence of a scaling factor in the filter is quantized according to the appropriate data format.

---

When you apply `normalize` to a fixed-point filter, the value for the `ScaleValues` property is changed accordingly.

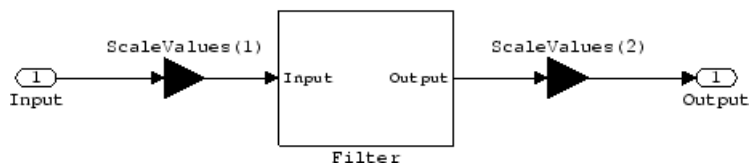
It is good practice to choose values for this property that are either positive or negative powers of two.

**Interpreting the `ScaleValues` Property.** When you specify the values of the `ScaleValues` property of a quantized filter, the values are entered as a vector, the length of which is determined by the number of cascaded sections in your filter:

- When you have only one section, the value of the `ScaleValues` property can be a scalar or a two-element vector.
- When you have  $L$  cascaded sections in your filter, the value of the `ScaleValues` property can be a scalar or an  $L+1$ -element vector.

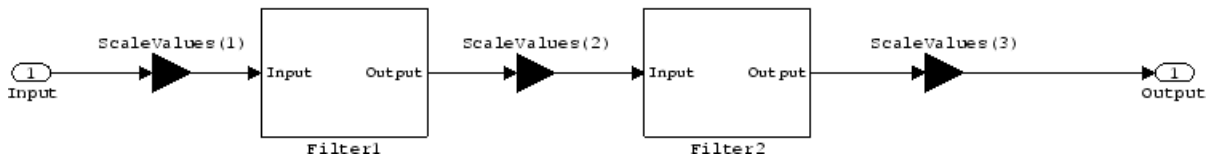
The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with only one section.

#### Application of `ScaleValues` to a Single Section



The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with two sections.

### Application of ScaleValues to Multiple Sections



### Signed

When you create a `dfilt` object for fixed-point filtering (you set the property `Arithmetic` to `fixed`, the property `Signed` specifies whether the filter interprets coefficients as signed or unsigned. This setting applies only to the coefficients. While the default setting is `true`, meaning that all coefficients are assumed to be signed, you can change the setting to `false` after you create the fixed-point filter.

For example, create a fixed-point direct-form II transposed filter with both negative and positive coefficients, and then change the property value for `Signed` from `true` to `false` to see what happens to the negative coefficient values.

```
hd=dfilt.df2t(-5:5)

hd =

    FilterStructure: 'Direct-Form II Transposed'
    Arithmetic: 'double'
    Numerator: [-5 -4 -3 -2 -1 0 1 2 3 4 5]
    Denominator: 1
    PersistentMemory: false
    States: [10x1 double]

set(hd,'arithmetic','fixed')
hd.numerator

ans =

    -5    -4    -3    -2    -1     0
```

```

          1      2      3      4      5

set(hd,'signed',false)
hd.numerator

ans =

      0      0      0      0      0      0
      1      2      3      4      5

```

Using unsigned coefficients limits you to using only positive coefficients in your filter. `Signed` is a dynamic property — you cannot set or change it until you switch the setting for the `Arithmetic` property to `fixed`.

### SosMatrix

When you convert a `dfilt` object to second-order section form, or create a second-order section filter, `sosMatrix` holds the filter coefficients as property values. Using the `double` data type by default, the matrix is in [sections coefficients per section] form, displayed as [15-x-6] for filters with 6 coefficients per section and 15 sections, [15 6].

To demonstrate, the following code creates an order 30 filter using second-order sections in the direct-form II transposed configuration. Notice the `sosMatrix` property contains the coefficients for all the sections.

```

d = fdesign.lowpass('n,fc',30,0.5);
hd = butter(d);

hd =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
    Arithmetic: 'double'
      sosMatrix: [15x6 double]
    ScaleValues: [16x1 double]
 PersistentMemory: false
       States: [2x15 double]

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

```

```

SectionInputWordLength: 16
  SectionInputAutoScale: true

```

```

SectionOutputWordLength: 16
  SectionOutputAutoScale: true

```

```

OutputWordLength: 16
  OutputMode: 'AvoidOverflow'

```

```

StateWordLength: 16
StateFracLength: 15

```

```

ProductMode: 'FullPrecision'

```

```

AccumWordLength: 40
  CastBeforeSum: true

```

```

RoundMode: 'convergent'
OverflowMode: 'wrap'

```

```
hd.sosMatrix
```

```
ans =
```

```

1.0000    2.0000    1.0000    1.0000         0    0.9005

```

1.0000	2.0000	1.0000	1.0000	0	0.7294
1.0000	2.0000	1.0000	1.0000	0	0.5888
1.0000	2.0000	1.0000	1.0000	0	0.4724
1.0000	2.0000	1.0000	1.0000	0	0.3755
1.0000	2.0000	1.0000	1.0000	0	0.2948
1.0000	2.0000	1.0000	1.0000	0	0.2275
1.0000	2.0000	1.0000	1.0000	0	0.1716
1.0000	2.0000	1.0000	1.0000	0	0.1254
1.0000	2.0000	1.0000	1.0000	0	0.0878
1.0000	2.0000	1.0000	1.0000	0	0.0576
1.0000	2.0000	1.0000	1.0000	0	0.0344
1.0000	2.0000	1.0000	1.0000	0	0.0173
1.0000	2.0000	1.0000	1.0000	0	0.0062
1.0000	2.0000	1.0000	1.0000	0	0.0007

The SOS matrix is an M-by-6 matrix, where M is the number of sections in the second-order section filter. Filter `hd` has M equal to 15 as shown above (15 rows). Each row of the SOS matrix contains the numerator and denominator coefficients (b's and a's) and the scale factors of the corresponding section in the filter.

### SectionInputAutoScale

Second-order section filters include this property that determines who the filter handles data in the transitions from one section to the next in the filter.

How the filter represents the data passing from one section to the next depends on the property value of `SectionInputAutoScale`. The representation the filter uses between the filter sections depends on whether the value of `SectionInputAutoScale` is `true` or `false`.

- `SectionInputAutoScale = true` means the filter chooses the fraction length to maintain the value of the data between sections as close to the output values from the previous section as possible. `true` is the default setting.
- `SectionInputAutoScale = false` removes the automatic scaling of the fraction length for the intersection data and exposes the property that controls the coefficient fraction length (`SectionInputFracLength`) so you can change it. For example, if the filter is a second-order, direct form FIR filter, setting `SectionInputAutoScale` to `false` exposes the



`SectionInputFracLength` property that specifies the fraction length applied to data between the sections.

### **SectionInputFracLength**

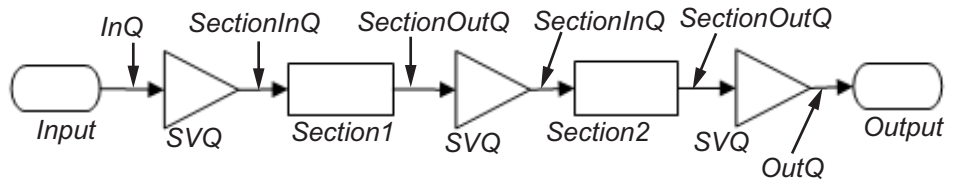
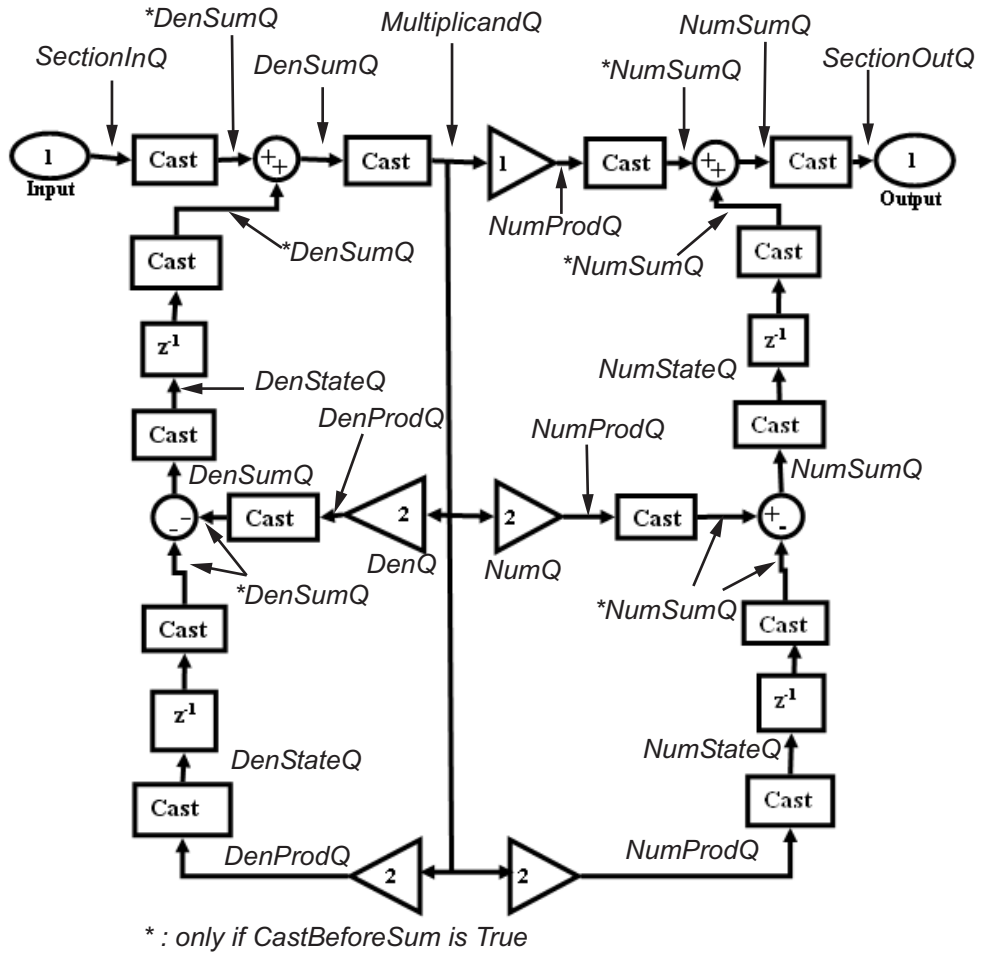
Second-order section filters use quantizers at the input to each section of the filter. The quantizers apply to the input data entering each filter section. Note that the quantizers for each section are the same. To set the fraction length for interpreting the input values, use the property value in `SectionInputFracLength`.

In combination with `CoeffWordLength`, `SectionInputFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`. As with all word and fraction length properties, `SectionInputFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

### **SectionInputWordLength**

SOS filters are composed of sections, each one a second-order filter. Filtering data input to the filter involves passing the data through each filter section. `SectionInputWordLength` specifies the word length applied to data as it enters one filter section from the previous section. Only second-order implementations of direct-form I transposed and direct-form II transposed filters include this property.

The following diagram shows an SOS filter composed of sections (the bottom part of the diagram) and a possible internal structure of each Section (the top portion of the diagram), in this case — a direct form I transposed second order sections filter structure. Note that the output of each section is fed through a multiplier. If the gain of the multiplier =1, then the last Cast block of the Section is ignored, and the format of the output is NumSumQ.



`SectionInputWordLength` defaults to 16 bits.

### **SectionOutputAutoScale**

Second-order section filters include this property that determines who the filter handles data in the transitions from one section to the next in the filter.

How the filter represents the data passing from one section to the next depends on the property value of `SectionOutputAutoScale`. The representation the filter uses between the filter sections depends on whether the value of `SectionOutputAutoScale` is `true` or `false`.

- `SectionOutputAutoScale = true` means the filter chooses the fraction length to maintain the value of the data between sections as close to the output values from the previous section as possible. `true` is the default setting.
- `SectionOutputAutoScale = false` removes the automatic scaling of the fraction length for the intersection data and exposes the property that controls the coefficient fraction length (`SectionOutputFracLength`) so you can change it. For example, if the filter is a second-order, direct form FIR filter, setting `SectionOutputAutoScale = false` exposes the `SectionOutputFracLength` property that specifies the fraction length applied to data between the sections.

### **SectionOutputFracLength**

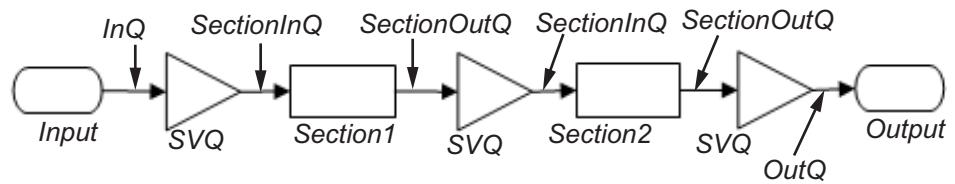
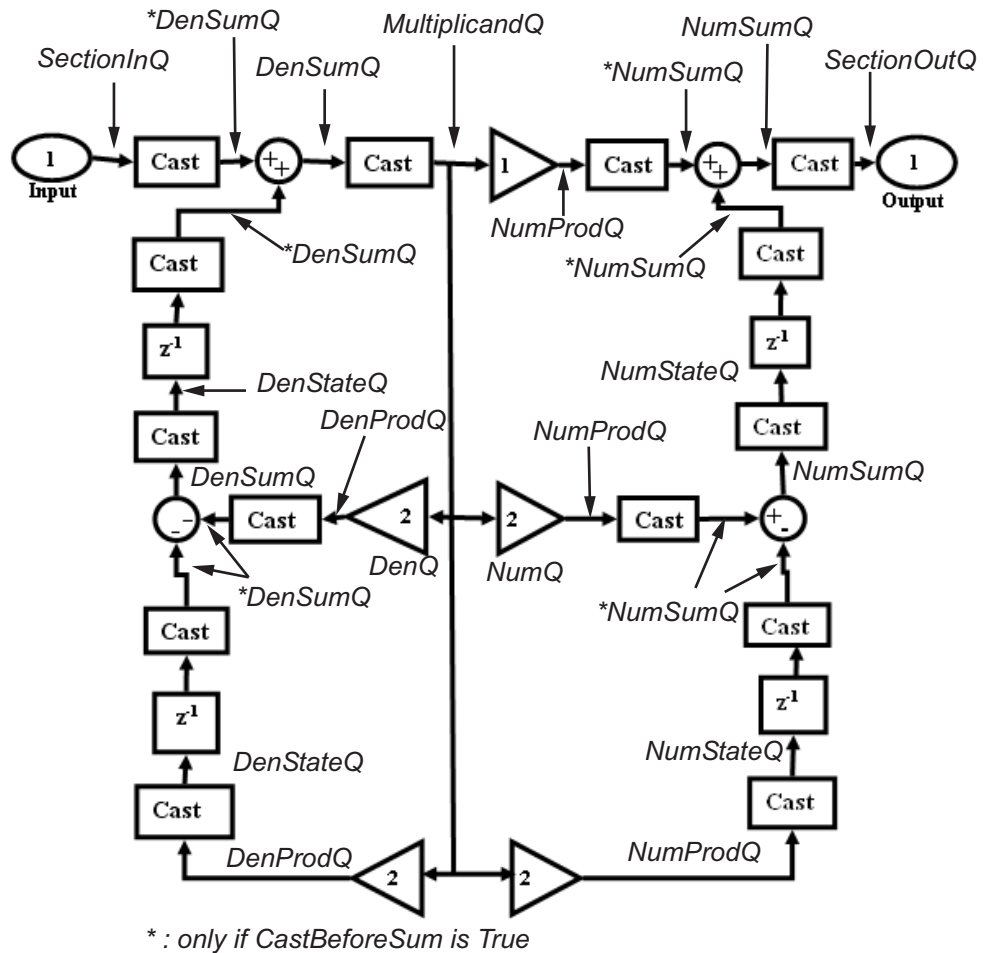
Second-order section filters use quantizers at the output from each section of the filter. The quantizers apply to the output data leaving each filter section. Note that the quantizers for each section are the same. To set the fraction length for interpreting the output values, use the property value in `SectionOutputFracLength`.

In combination with `CoeffWordLength`, `SectionOutputFracLength` determines how the filter interprets and uses the state values stored in the property `States`. As with all fraction length properties, `SectionOutputFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

### **SectionOutputWordLength**

SOS filters are composed of sections, each one a second-order filter. Filtering data input to the filter involves passing the data through each filter section. `SectionOutputWordLength` specifies the word length applied to data as it leaves one filter section to go to the next. Only second-order implementations direct-form I transposed and direct-form II transposed filters include this property.

The following diagram shows an SOS filter composed of sections (the bottom part of the diagram) and a possible internal structure of each Section (the top portion of the diagram), in this case — a direct form I transposed second order sections filter structure. Note that the output of each section is fed through a multiplier. If the gain of the multiplier =1, then the last Cast block of the Section is ignored, and the format of the output is NumSumQ.



SectionOutputWordLength defaults to 16 bits.

### StateAutoScale

Although all filters use states, some do not allow you to choose whether the filter automatically scales the state values to prevent overruns or bad arithmetic errors. You select either of the following settings:

- `StateAutoScale = true` means the filter chooses the fraction length to maintain the value of the states as close to the double-precision values as possible. When you change the word length applied to the states (where allowed by the filter structure), the filter object changes the fraction length to try to accommodate the change. `true` is the default setting.
- `StateAutoScale = false` removes the automatic scaling of the fraction length for the states and exposes the property that controls the coefficient fraction length so you can change it. For example, in a direct form I transposed SOS FIR filter, setting `StateAutoScale = false` exposes the `NumStateFracLength` and `DenStateFracLength` properties that specify the fraction length applied to states.

Each of the following filter structures provides the `StateAutoScale` property:

- `df1t`
- `df1tsos`
- `df2t`
- `df2tsos`
- `dffirt`

Other filter structures do not include this property.

### StateFracLength

Filter states stored in the property `States` have both word length and fraction length. To set the fraction length for interpreting the stored filter object state values, use the property value in `StateFracLength`.

In combination with `CoeffWordLength`, `StateFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`.

As with all fraction length properties, `StateFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

## States

Digital filters are dynamic systems. The behavior of dynamic systems (their response) depends on the input (stimulus) to the system and the current or previous *state* of the system. You can say the system has memory or inertia. All fixed- or floating-point digital filters (as well as analog filters) have states.

Filters use the states to compute the filter output for each input sample, as well using them while filtering in loops to maintain the filter state between loop iterations. This toolbox assumes zero-valued initial conditions (the dynamic system is at rest) by default when you filter the first input sample. Assuming the states are zero initially does not mean the states are not used; they are, but arithmetically they do not have any effect.

Filter objects store the state values in the property `States`. The number of stored states depends on the filter implementation, since the states represent the delays in the filter implementation.

When you review the display for a filter object with fixed arithmetic, notice that the states return an embedded `fi` object, as you see here.

```
b = ellip(6,3,50,300/500);
hd=dfilt.dffir(b)

hd =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'double'
      Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858 0.2938 0.0773]
 PersistentMemory: false
           States: [6x1 double]
```

```
hd.arithmetic='fixed'

hd =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'fixed'
      Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858 0.2938 0.0773]
    PersistentMemory: false
      States: [1x1 embedded.fi]

    CoeffWordLength: 16
      CoeffAutoScale: 'on'
      Signed: 'on'

    InputWordLength: 16
    InputFracLength: 15

    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'

      ProductMode: 'FullPrecision'

    AccumWordLength: 40
      CastBeforeSum: 'on'

      RoundMode: 'convergent'
      OverflowMode: 'wrap'

    InheritSettings: 'off'
```

`fi` objects provide fixed-point support for the filters. To learn more about the details about `fi` objects, refer to your Fixed-Point Toolbox documentation.

The property `States` lets you use a `fi` object to define how the filter interprets the filter states. For example, you can create a `fi` object in MATLAB, then assign the object to `States`, as follows:

```
statefi=fi([],16,12)
```



```

statefi =

[]
  DataTypeMode = Fixed-point: binary point scaling
  Signed = true
  Wordlength = 16
  Fractionlength = 12

```

This `fi` object does not have a value associated (notice the `[]` input argument to `fi` for the value), and it has word length of 16 bits and fraction length of 12 bit. Now you can apply `statefi` to the `States` property of the filter `hd`.

```

set(hd,'States',statefi);
Warning: The 'States' property will be reset to the value
specified at construction before filtering.
Set the 'PersistentMemory' flag to 'True'
to avoid changing this property value.
hd

```

```

hd =

  FilterStructure: 'Direct-Form FIR'
  Arithmetic: 'fixed'
  Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858
             0.2938 0.0773]
  PersistentMemory: false
  States: [1x1 embedded.fi]

  CoeffWordLength: 16
  CoeffAutoScale: 'on'
  Signed: 'on'

  InputWordLength: 16
  InputFracLength: 15

  OutputWordLength: 16
  OutputMode: 'AvoidOverflow'

  ProductMode: 'FullPrecision'
  AccumWordLength: 40

```

```
CastBeforeSum: 'on'  
  
    RoundMode: 'convergent'  
    OverflowMode: 'wrap'
```

## StateWordLength

While all filters use states, some do not allow you to directly change the state representation — the word length and fraction lengths — independently. For the others, `StateWordLength` specifies the word length, in bits, the filter uses to represent the states. Filters that do not provide direct state word length control include:

- `df1`
- `dfasymfir`
- `dffir`
- `dfsymfir`

For these structures, the filter derives the state format from the input format you choose for the filter — except for the `df1` IIR filter. In this case, the numerator state format comes from the input format and the denominator state format comes from the output format. All other filter structures provide control of the state format directly.

## TapSumFracLength

Direct-form FIR filter objects, both symmetric and antisymmetric, use this property. To set the fraction length for output from the sum operations that involve the filter tap weights, use the property value in `TapSumFracLength`. To enable this property, set the `TapSumMode` to `SpecifyPrecision` in your filter.

As you can see in this code example that creates a fixed-point asymmetric FIR filter, the `TapSumFracLength` property becomes available after you change the `TapSumMode` property value.

```
hd=dfilt.dfasymfir  
  
hd =
```

```
        FilterStructure: 'Direct-Form Antisymmetric FIR'  
          Arithmetic: 'double'  
            Numerator: 1  
PersistentMemory: false  
          States: [0x1 double]  
  
set(hd,'arithmetic','fixed');  
hd  
  
hd =  
  
        FilterStructure: 'Direct-Form Antisymmetric FIR'  
          Arithmetic: 'fixed'  
            Numerator: 1  
PersistentMemory: false  
          States: [1x1 embedded.fi]  
  
        CoeffWordLength: 16  
          CoeffAutoScale: true  
            Signed: true  
  
        InputWordLength: 16  
          InputFracLength: 15  
  
        OutputWordLength: 16  
          OutputMode: 'AvoidOverflow'  
  
          TapSumMode: 'KeepMSB'  
        TapSumWordLength: 17  
  
          ProductMode: 'FullPrecision'  
  
        AccumWordLength: 40  
  
          CastBeforeSum: true  
            RoundMode: 'convergent'  
          OverflowMode: 'wrap'
```

With the filter now in fixed-point mode, you can change the `TapSumMode` property value to `SpecifyPrecision`, which gives you access to the `TapSumFracLength` property.

```
set(hd,'TapSumMode','SpecifyPrecision');
hd

hd =

    FilterStructure: 'Direct-Form Antisymmetric FIR'
      Arithmetic: 'fixed'
      Numerator: 1
 PersistentMemory: false
      States: [1x1 embedded.fi]

    CoeffWordLength: 16
      CoeffAutoScale: true
      Signed: true

    InputWordLength: 16
    InputFracLength: 15

    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'

      TapSumMode: 'SpecifyPrecision'
    TapSumWordLength: 17
    TapSumFracLength: 15

      ProductMode: 'FullPrecision'

    AccumWordLength: 40

    CastBeforeSum: true
      RoundMode: 'convergent'
      OverflowMode: 'wrap'
```

In combination with `TapSumWordLength`, `TapSumFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`.

As with all fraction length properties, `TapSumFracLength` can be any integer, including integers larger than `TapSumWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

### **TapSumMode**

This property, available only after your filter is in fixed-point mode, specifies how the filter outputs the results of summation operations that involve the filter tap weights. Only symmetric (`dfilt.dfsymfir`) and antisymmetric (`dfilt.dfasymfir`) FIR filters use this property.

When available, you select from one of the following values:

- **FullPrecision** — means the filter automatically chooses the word length and fraction length to represent the results of the sum operation so they retain all of the precision provided by the inputs (addends).
- **KeepMSB** — means you specify the word length for representing tap sum summation results to keep the higher order bits in the data. The filter sets the fraction length to discard the LSBs from the sum operation. This is the default property value.
- **KeepLSB** — means you specify the word length for representing tap sum summation results to keep the lower order bits in the data. The filter sets the fraction length to discard the MSBs from the sum operation. Compare to the **KeepMSB** option.
- **SpecifyPrecision** — means you specify the word and fraction lengths to apply to data output from the tap sum operations.

### **TapSumWordLength**

Specifies the word length the filter uses to represent the output from tap sum operations. The default value is 17 bits. Only `dfasymfir` and `dfsymfir` filters include this property.

## Adaptive Filter Properties

<b>In this section...</b>
“Property Summaries” on page 3-102
“Property Details for Adaptive Filter Properties” on page 3-107

### Property Summaries

The following table summarizes the adaptive filter properties and provides a brief description of each. Full descriptions of each property, in alphabetical order, are given in the subsequent section.

Property	Description
Algorithm	Reports the algorithm the object uses for adaptation. When you construct your adaptive filter object, this property is set automatically by the constructor, such as <code>adaptfilt.nlms</code> creating an adaptive filter that uses the normalized LMS algorithm. You cannot change the value — it is read only.
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 $(1 - \text{step}) \cdot \text{lambda}$ is the input argument that represents AvgFactor
BkwdPredErrorPower	Returns the minimum mean-squared prediction error. Refer to [3] in the bibliography for details about linear prediction.
BkwdPrediction	Returns the predicted samples generated during adaptation. Refer to [3] in the bibliography for details about linear prediction.

Property	Description
Blocklength	Block length for the coefficient updates. This must be a positive integer such that $(1/\text{blocklength})$ is also an integer. For faster execution, <code>blocklength</code> should be a power of two. <code>blocklength</code> defaults to two.
Coefficients	Vector containing the initial filter coefficients. It must be a length <code>l</code> vector where <code>l</code> is the number of filter coefficients. <code>coeffs</code> defaults to length <code>l</code> vector of zeros when you do not provide the argument for input.
ConversionFactor	Conversion factor defaults to the matrix $[1 \ -1]$ that specifies soft-constrained initialization. This is the <code>gamma</code> input argument for some of the fast transversal algorithms.
Delay	Update delay given in time samples. This scalar should be a positive integer—negative delays do not work. <code>delay</code> defaults to 1 for most algorithms.
DesiredSignalStates	Desired signal states of the adaptive filter. <code>dstates</code> defaults to a zero vector with length equal to $(\text{blocklen} - 1)$ or $(\text{swblocklen} - 1)$ depending on the algorithm.
EpsilonStates	Vector of the epsilon values of the adaptive filter. <code>EpsilonStates</code> defaults to a vector of zeros with $(\text{projectord} - 1)$ elements.
ErrorStates	Vector of the adaptive filter error states. <code>ErrorStates</code> defaults to a zero vector with length equal to $(\text{projectord} - 1)$ .
FFTCoefficients	Stores the discrete Fourier transform of the filter coefficients in <code>coeffs</code> .
FFTStates	Stores the states of the FFT of the filter coefficients during adaptation.
FilteredInputStates	Vector of filtered input states with length equal to $l - 1$ .

<b>Property</b>	<b>Description</b>
FilterLength	Contains the length of the filter. Note that this is not the filter order. Filter length is 1 greater than filter order. Thus a filter with length equal to 10 has filter order equal to 9.
ForgettingFactor	Determines how the RLS adaptive filter uses past data in each iteration. You use the forgetting factor to specify whether old data carries the same weight in the algorithm as more recent data.
FwdPredErrorPower	Returns the minimum mean-squared prediction error in the forward direction. Refer to [3] in the bibliography for details about linear prediction.
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. Called <code>delta</code> as an input argument, this defaults to one.
InvCov	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix. Dimensions are 1-by-1, where 1 is the filter length.
KalmanGain	Empty when you construct the object, this gets populated after you run the filter.
KalmanGainStates	Contains the states of the Kalman gain updates during adaptation.



Property	Description
Leakage	Contains the setting for leakage in the adaptive filter algorithm. Using a leakage factor that is not 1 forces the weights to adapt even when they have found the minimum error solution. Forcing the adaptation can improve the numerical performance of the LMS algorithm.
OffsetCov	Contains the offset covariance matrix.
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.
Power	A vector of 2*1 elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
ProjectionOrder	Projection order of the affine projection algorithm. <code>projectord</code> defines the size of the input signal covariance matrix and defaults to two.
ReflectionCoeffs	Coefficients determined for the reflection portion of the filter during adaptation.
ReflectionCoeffsStep	Size of the steps used to determine the reflection coefficients.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states and coefficients from previous filtering runs.
SecondaryPathCoeffs	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.
SecondaryPathEstimate	An estimate of the secondary path filter model.

Property	Description
SecondaryPathStates	The states of the secondary path filter, the unknown system.
SqrtCov	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
SqrtInvCov	Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
States	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros whose length depends on the chosen algorithm. Usually the length is a function of the filter length <code>l</code> and another input argument to the filter object, such as <code>projectord</code> .
StepSize	Reports the size of the step taken between iterations of the adaptive filter process. Each <code>adaptfilt</code> object has a default value that best meets the needs of the algorithm.
SwBlockLength	Block length of the sliding window. This integer must be at least as large as the filter length. <code>swblocklen</code> defaults to 16.

Like `dfilt` objects, `adaptfilt` objects have properties that govern their behavior and store some of the results of filtering operations. The following pages list, in alphabetical order, the name of every property associated with `adaptfilt` objects. Note that not all `adaptfilt` objects have all of these properties. To view the properties of a particular adaptive filter, such as an `adaptfilt.bap` filter, use `get` with the object handle, like this:

```

ha = adaptfilt.bap(32,0.5,4,1.0);
get(ha)
    PersistentMemory: false
        Algorithm: 'Block Affine Projection FIR Adaptive Filter'
        FilterLength: 32
        Coefficients: [1x32 double]

```

```
States: [35x1 double]
StepSize: 0.5000
ProjectionOrder: 4
OffsetCov: [4x4 double]
```

`get` shows you the properties for `ha` and the values for the properties. Entering the object handle returns the same values and properties without the formatting of the list and the more familiar property names.

## Property Details for Adaptive Filter Properties

### Algorithm

Reports the algorithm the object uses for adaptation. When you construct you adaptive filter object, this property is set automatically. You cannot change the value—it is read only.

### 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  $(1 - \text{step})$ . `lambda` is the input argument that represent `AvgFactor`

### BkwdPredErrorPower

Returns the minimum mean-squared prediction error in the backward direction. Refer to [3] in the bibliography for details about linear prediction.

### BkwdPrediction

When you use an adaptive filter that does backward prediction, such as `adaptfilt.ftf`, one property of the filter contains the backward prediction coefficients for the adapted filter. With these coefficient, the forward coefficients, and the system under test, you have the full set of knowledge of how the adaptation occurred. Two values stored in properties compose the `BkwdPrediction` property:

- `Coefficients`, which contains the coefficients of the system under test, as determined using backward predictions process.

- Error, which is the difference between the filter coefficients determined by backward prediction and the actual coefficients of the sample filter. In this example, `adaptfilt.ftf` identifies the coefficients of an unknown FIR system.

```

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);

ha

ha =

        Algorithm: 'Fast Transversal Least-Squares Adaptive Filter'
        FilterLength: 32
        Coefficients: [1x32 double]
           States: [31x1 double]
ForgettingFactor: 0.9900
        InitFactor: 0.1000
        FwdPrediction: [1x1 struct]
        BkwdPrediction: [1x1 struct]
        KalmanGain: [32x1 double]
ConversionFactor: 0.7338
        KalmanGainStates: [32x1 double]
PersistentMemory: false

ha.coefficients

ans =

Columns 1 through 8

    -0.0055    0.0048    0.0045    0.0146   -0.0009    0.0002   -0.0019    0.0008

```

Columns 9 through 16

-0.0142 -0.0226 0.0234 0.0421 -0.0571 -0.0807 0.1434 0.4620

Columns 17 through 24

0.4564 0.1532 -0.0879 -0.0501 0.0331 0.0361 -0.0266 -0.0220

Columns 25 through 32

0.0231 0.0026 -0.0063 -0.0079 0.0032 0.0082 0.0033 0.0065

ha.bkwdprediction

ans =

Coeffs: [1x32 double]

Error: 82.3394

>> ha.bkwdprediction.coeffs

ans =

Columns 1 through 8

0.0067 0.0186 0.1114 -0.0150 -0.0239 -0.0610 -0.1120 -0.1026

Columns 9 through 16

0.0093 -0.0399 -0.0045 0.0622 0.0997 0.0778 0.0646 -0.0564

Columns 17 through 24

0.0775 0.0814 0.0057 0.0078 0.1271 -0.0576 0.0037 -0.0200

Columns 25 through 32

-0.0246 0.0180 -0.0033 0.1222 0.0302 -0.0197 -0.1162 0.0285

### **Blocklength**

Block length for the coefficient updates. This must be a positive integer such that  $(1/\text{blocklen})$  is also an integer. For faster execution, `blocklen` should be a power of two. `blocklen` defaults to two.

### **Coefficients**

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 defaults to the matrix  $[1 \ -1]$  that specifies soft-constrained initialization. This is the `gamma` input argument for some of the fast transversal algorithms.

### **Delay**

Update delay given in time samples. This scalar should be a positive integer — negative delays do not work. `delay` defaults to 1 for most algorithms.

### **DesiredSignalStates**

Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to  $(\text{blocklen} - 1)$  or  $(\text{swblocklen} - 1)$  depending on the algorithm.

### **EpsilonStates**

Vector of the epsilon values of the adaptive filter. `EpsilonStates` defaults to a vector of zeros with  $(\text{projectord} - 1)$  elements.

### **ErrorStates**

Vector of the adaptive filter error states. `ErrorStates` defaults to a zero vector with length equal to  $(\text{projectord} - 1)$ .

### **FFTCoefficients**

Stores the discrete Fourier transform of the filter coefficients in `coeffs`.

**FFTStates**

Stores the states of the FFT of the filter coefficients during adaptation.

**FilteredInputStates**

Vector of filtered input states with length equal to  $1 - 1$ .

**FilterLength**

Contains the length of the filter. Note that this is not the filter order. Filter length is 1 greater than filter order. Thus a filter with length equal to 10 has filter order equal to 9.

**ForgettingFactor**

Determines how the RLS adaptive filter uses past data in each iteration. You use the forgetting factor to specify whether old data carries the same weight in the algorithm as more recent data.

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.

**FwdPredErrorPower**

Returns the minimum mean-squared prediction error in the forward direction. Refer to [3] in the bibliography for details about linear prediction.

**FwdPrediction**

Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.

**InitFactor**

Returns the 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.

### **InvCov**

Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix. Dimensions are 1-by-1, where 1 is the filter length.

### **KalmanGain**

Empty when you construct the object, this gets populated after you run the filter.

### **KalmanGainStates**

Contains the states of the Kalman gain updates during adaptation.

### **Leakage**

Contains the setting for leakage in the adaptive filter algorithm. Using a leakage factor that is not 1 forces the weights to adapt even when they have found the minimum error solution. Forcing the adaptation can improve the numerical performance of the LMS algorithm.

### **OffsetCov**

Contains the offset covariance matrix.

### **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 input signal amplitude becomes very small, or dividing by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to one.

### **Power**

A vector of 2\*1 elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.



**ProjectionOrder**

Projection order of the affine projection algorithm. `projectord` defines the size of the input signal covariance matrix and defaults to two.

**ReflectionCoeffs**

For adaptive filters that use reflection coefficients, this property stores them.

**ReflectionCoeffsStep**

As the adaptive filter changes coefficient values during adaptation, the step size used between runs is stored here.

**PersistentMemory**

Determines 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`.

**SecondaryPathCoeffs**

A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.

**SecondaryPathEstimate**

An estimate of the secondary path filter model.

**SecondaryPathStates**

The states of the secondary path filter, the unknown system.

**SqrtCov**

Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.

### **SqrtInvCov**

Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.

### **States**

Vector of the adaptive filter states. `states` defaults to a vector of zeros whose length depends on the chosen algorithm. Usually the length is a function of the filter length `l` and another input argument to the filter object, such as `projectord`.

### **StepSize**

Reports the size of the step taken between iterations of the adaptive filter process. Each `adaptfilt` object has a default value that best meets the needs of the algorithm.

### **SwBlockLength**

Block length of the sliding window. This integer must be at least as large as the filter length. `swblocklength` defaults to 16.

## References

- [1] 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.
- [2] Hayes, M.H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, 1996.
- [3] Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

## Multirate Filter Properties

<b>In this section...</b>
“Property Summaries” on page 3-116
“Property Details for Multirate Filter Properties” on page 3-121

### Property Summaries

The following table summarizes the multirate filter properties and provides a brief description of each. Full descriptions of each property are given in the subsequent section.

<b>Name</b>	<b>Values</b>	<b>Default</b>	<b>Description</b>
BlockLength	Positive integers	100	Length of each block of data input to the FFT used in the filtering. <code>fftfirinterp</code> multirate filters include this property.
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.
FilterInternals	FullPrecision, MinWordlengths, SpecifyWordLengths, SpecifyPrecision	FullPrecision	Controls whether the filter sets the output word and fraction lengths, and the accumulator word and fraction lengths automatically to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code>

Name	Values	Default	Description
			exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.
FilterStructure	mfilt structure string	None	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. You cannot set this property — it is always read only and results from your choice of mfilt object.
InputOffset	Integers	0	Contains the number of input data samples processed without generating an output sample. $\text{InputOffset} = \text{mod}(\text{length}(nx), m)$ where $nx$ is the number of input samples that have been processed so far and $m$ is the decimation factor.
InterpolationFactor	Positive integers	2	Interpolation factor for the filter. 1 specifies the amount to increase the input sampling rate.
NumberOfSections	Any positive integer	2	Number of sections used in the decimator, or in the comb and integrator portions of CIC filters.

Name	Values	Default	Description
Numerator	Array of double values	No default values	Vector containing the coefficients of the FIR lowpass filter used for interpolation.
OverflowMode	saturate, [wrap]	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	Values depend on filter type. Either double, single, or fixed-point object	0	Stores the value remaining in the accumulator after the filter processes the last input sample. The stored value for PolyphaseAccum affects the next output when PersistentMemory is true and InputOffset is not equal to 0. Always provides full precision values. Compare the AccumWordLength and AccumFracLength.

Name	Values	Default	Description
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.
RateChangeFactors	[1,m]	[2,3] or [3,2]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal. The default changes depending on whether the filter decimates or interpolates.
States	Any m+1-by-n matrix of double values	2-by-2 matrix, int32	Stored conditions for the filter, including values for the integrator and comb sections. n is the number of filter sections and m is the differential delay. Stored in a filtstates object.

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 proper word lengths for each section, according to the information in “Constraints and Word Length Considerations” on page 2-1000. 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.
SpecifyWordLengths	Vector of integers	[ 16 16 16 16] bits	
WordLengthPerSection	Any integer or a vector of length 2*n	16	Defines the word length used in each section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter WordLengthPerSection as a scalar or vector of length 2*n, where n is the number of sections. When



Name	Values	Default	Description
			WordLengthPerSection is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.

The following sections provide details about the properties that govern the way multirate filter work. Creating any multirate filter object puts in place a number of these properties. The following pages list the `mfilt` object properties in alphabetical order.

## Property Details for Multirate Filter Properties

### BitsPerSection

Any integer or a vector of length  $2*n$ .

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 `bps` as a scalar or vector of length  $2*n$ , where  $n$  is the number of sections. When `bps` is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.

### BlockLength

Length of each block of input data used in the filtering.

`mfilt.fftfirinterp` objects process data in blocks whose length is determined by the value you set for the `BlockLength` property. By default the property value is 100. When you set the `BlockLength` value, try choosing a value so that  $[BlockLength + \text{length}(\text{filter order})]$  is a power of two.

Larger block lengths generally reduce the computation time.

**DecimationFactor**

Decimation factor for the filter.  $m$  specifies the amount to reduce the sampling rate of the input signal. It must be an integer. You can enter any integer value. The default value is 2.

**DifferentialDelay**

Sets the differential delay for the filter. Usually a value of one or two is appropriate. While you can set any value, the default is one and the maximum is usually two.

**FilterInternals**

Similar to the FilterInternals pane in FDATool, this property controls whether the filter sets the output word and fraction lengths automatically, and the accumulator word and fraction lengths automatically as well, to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. Setting `FilterInternals` to `SpecifyPrecision` exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.

**About FilterInternals Mode.** There are four usage modes for this that you set using the `FilterInternals` property in multirate filters.

- `FullPrecision` — All word and fraction lengths set to  $B_{max} + 1$ , called  $B_{accum}$  by Fred Harris in [2]. Full precision is the default setting.
- `MinWordLengths` — Minimum Word Lengths
- `SpecifyWordLengths` — Specify Word Lengths
- `SpecifyPrecision` — Specify Precision

**Full Precision**

In full precision mode, the word lengths of all sections and the output are set to  $B_{accum}$  as defined by

$$B_{accum} = \text{ceil}(N_{secs}(\text{Log}_2(D \times M)) + \text{InputWordLength})$$

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 Word Lengths

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 [3] in the following References section) 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 word length 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  $2 \times (\text{NumberOfSections})$ , 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  $B_{accum}$ , the CIC algorithm 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 calar such as  $B_{accum}$ , the design 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
```

### **FilterStructure**

Reports the type of filter object, such as a decimator or fractional integrator. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Because of the length of the names of multirate filters, `FilterStructure` often returns a vector specification for the string. For example, when you use `mfilt.firfracinterp` to design a filter, `FilterStructure` returns as [1x49 char].

```
hm=mfilt.firfracinterp

hm =

    FilterStructure: [1x49 char]
      Numerator: [1x72 double]
RateChangeFactors: [3 2]
  PersistentMemory: false
          States: [24x1 double]
```

### **InputOffset**

When you decimate signals whose length is not a multiple of the decimation factor  $M$ , the last samples —  $(nM + 1)$  to  $[(n+1)(M) - 1]$ , where  $n$  is an integer — are processed and used to track where the filter stopped processing input data and when to expect the next output sample. If you think of the filtering process

as generating an output for a block of input data, `InputOffset` contains a count of the number of samples in the last incomplete block of input data.

---

**Note** `InputOffset` applies only when you set `PersistentMemory` to `true`. Otherwise, `InputOffset` is not available for you to use.

---

Two different cases can arise when you decimate a signal:

- 1** The input signal is a multiple of the filter decimation factor. In this case, the filter processes the input samples and generates output samples for all inputs as determined by the decimation factor. For example, processing 99 input samples with a filter that decimates by three returns 33 output samples.
- 2** The input signal is not a multiple of the decimation factor. When this occurs, the filter processes all of the input samples, generates output samples as determined by the decimation factor, and has one or more input samples that were processed but did not generate an output sample.

For example, when you filter 100 input samples with a filter which has decimation factor of 3, you get 33 output samples, and 1 sample that did not generate an output. In this case, `InputOffset` stores the value 1 after the filter run.

`InputOffset` equal to 1 indicates that, if you divide your input signal into blocks of data with length equal to your filter decimation factor, the filter processed one sample from a new (incomplete) block of data. Subsequent inputs to the filter are concatenated with this single sample to form the next block of length `m`.

One way to define the value stored in `InputOffset` is

$$\text{InputOffset} = \text{mod}(\text{length}(nx), m)$$

where `nx` is the number of input samples in the data set and `m` is the decimation factor.

Storing `InputOffset` in the filter allows you to stop filtering a signal at any point and start over from there, provided that the `PersistentMemory`

property is set to `true`. Being able to resume filtering after stopping a signal lets you break large data sets in to smaller pieces for filtering. With `PersistentMemory` set to `true` and the `InputOffset` property in the filter, breaking a signal into sections of arbitrary length and filtering the sections is equivalent to filtering the entire signal at once.

```
xtot=[x,x];
ytot=filter(hm1,xtot)
ytot =

    0   -0.0003   0.0005  -0.0014   0.0028  -0.0054   0.0092

reset(hm1); % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hm1,x) filter(hm1,x)]

ysec =

    0   -0.0003   0.0005  -0.0014   0.0028  -0.0054   0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

### InterpolationFactor

Amount to increase the sampling rate. Interpolation factor for the filter. It specifies the amount to increase the input sampling rate. It must be an integer. Two is the default value. You may use any positive value.

### NumberOfSections

Number of sections used in the multirate filter. By default multirate filters use two sections, but any positive integer works.

### OverflowMode

The `OverflowMode` property is specified as one of the following two strings indicating how to respond to overflows in fixed-point arithmetic:

- `'saturate'` — saturate overflows.

When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the applicable



word length and fraction length properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.

- 'wrap' — wrap all overflows to the range of representable values.

When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number. You can learn more about modular arithmetic in Fixed-Point Toolbox documentation.

These rules apply to the `OverflowMode` property.

- Applies to the accumulator and output data only.
- Does not apply to coefficients or input data. These always saturate the results.
- Does not apply to products. Products maintain full precision at all times. Your filters do not lose precision in the products.

**Default value:** 'saturate'

---

**Note** Numbers in floating-point filters that extend beyond the dynamic range overflow to  $\pm\text{inf}$ .

---

## PolyphaseAccum

The idea behind `PolyphaseAccum` and `AccumWordLength/AccumFracLength` is to distinguish between the adders that always work in full precision (`PolyphaseAccum`) from the others [the adders that are controlled by the user (through `AccumWordLength` and `AccumFracLength`) and that may introduce quantization effects when you set property `FilterInternals` to `SpecifyPrecision`].

Given a product format determined by the input word and fraction lengths, and the coefficients word and fraction lengths, doing full precision accumulation means allowing enough guard bits to avoid overflows and underflows.

Property `PolyphaseAccum` stores the value that was in the accumulator the last time your filter ran out of input samples to process. The default value for `PolyphaseAccum` affects the next output only if `PersistentMemory` is true and `InputOffset` is not equal to 0.

`PolyphaseAccum` stores data in the format for the filter arithmetic. Double-precision filters store doubles in `PolyphaseAccum`. Single-precision filter store singles in `PolyphaseAccum`. Fixed-point filters store `fi` objects in `PolyphaseAccum`.

### **PersistentMemory**

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.

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 object. `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 — the filter retains memory about filtering operations from one to the next. Maintaining memory lets you filter large data sets as collections of smaller subsets and get the same result.

```
xtot=[x,x];
ytot=filter(hm1,xtot)
ytot =

    0   -0.0003   0.0005   -0.0014   0.0028   -0.0054   0.0092
reset(hm1); % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hm1,x) filter(hm1,x)]

ysec =

    0   -0.0003   0.0005   -0.0014   0.0028   -0.0054   0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

## RateChangeFactors

Reports the decimation ( $m$ ) and interpolation ( $l$ ) factors for the filter object when you create fractional integrators and decimators, although  $m$  and  $l$  are used as arguments to both decimators and integrators, applying the same meaning. Combining these factors as input arguments to the fractional decimator or integrator results in the final rate change for the signal.

For decimating filters, the default is `[2,3]`. For integrators, `[3,2]`.

## States

Stored conditions for the filter, including values for the integrator and comb sections.  $m$  is the differential delay and  $n$  is the number of sections in the filter.

**About the States of Multirate Filters.** In the `states` property you find the states for both the integrator and comb portions of the filter, stored in a `filtstates` object. `states` is a matrix of dimensions  $m+1$ -by- $n$ , with the states in CIC filters 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.

In the state matrix, state values are specified and stored in `double` format.

`States` stores conditions for the delays between each interpolator phase, the filter states, and the states at the output of each phase in the filter, including values for the interpolator and comb states.

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.

`States` defaults to a vector of zeros that has length equal to `nstates(hm)`

## References

- [1] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.
- [2] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.
- [3] Hogenauer, E. B., “An Economical Class of Digital Filters for Decimation and Interpolation,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 2, April 1981, pp. 155-162.
- [4] Lyons, Richard G., *Understanding Digital Signal Processing*, Prentice Hall PTR, 2004
- [5] Mitra, S.K., *Digital Signal Processing*, McGraw-Hill, 1998.
- [6] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Inc., 1996.

## A

- AccumFracLength 3-19
- AccumWordLength 3-19
- adaptfilt
  - about 2-2
  - copying 2-9
- adaptfilt object properties
  - Algorithm 3-107
  - AvgFactor 3-107
  - BkwdPredErrorPower 3-107
  - BkwdPrediction 3-107
  - Blocklength 3-110
  - Coefficients 3-110
  - ConversionFactor 3-110
  - Delay 3-110
  - DesiredSignalStates 3-110
  - EpsilonStates 3-110
  - ErrorStates 3-110
  - FFTCoefficients 3-110
  - FFTStates 3-111
  - FilteredInputStates 3-111
  - FilterLength 3-111
  - ForgettingFactor 3-111
  - FwdPredErrorPower 3-111
  - FwdPrediction 3-111
  - InitFactor 3-111
  - InvCov 3-112
  - KalmanGain 3-112
  - KalmanGainStates 3-112
  - Offset 3-112
  - OffsetCov 3-112
  - Power 3-112
  - ProjectionOrder 3-113
  - ReflectionCoeffsStep 3-113
  - ResetBeforeFiltering 3-113
  - SecondaryPathCoeffs 3-113
  - SecondaryPathEstimate 3-113
  - SecondaryPathStates 3-113
  - SqrtInvCov 3-114
  - States 3-114

- StepSize 3-114
- SwBlockLength 3-114
- adaptive filter properties
  - SqrtCov 3-113
- addstages method 2-257
- Algorithm 3-107
- antisymmetricfir 3-53
- arithmetic
  - about fixed-point 3-20
- arithmetic property
  - double 3-20
  - fixed 3-22
  - single 3-21
- AvgFactor 3-107

## B

- BkwdPredErrorPower 3-107
- BkwdPrediction 3-107
- block method 2-257
- Blocklength 3-110

## C

- cascade method 2-257
- CoeffAutoScale 3-35
- CoeffFracLength 3-40
- Coefficients 3-110
- coefficients method 2-257
- CoeffWordLength 3-40
- ConversionFactor 3-110
- convert filters 3-65
- convert method 2-257

## D

- delay 2-283
- Delay 3-110
- DenAccumFracLength 3-41
- DenFracLength 3-41
- Denominator 3-41

- DenProdFracLength 3-42
  - DenStateFracLength 3-42
  - DenStateWordLength 3-42
  - DesiredSignalStates 3-110
  - df1 3-46
  - df1t 3-48
  - df2 3-49
  - df2t 3-51
  - dfilt 2-6
    - cascade 2-273
    - df1 2-285
    - df1sos 2-294
    - df1t 2-304
    - df1tsos 2-315
    - df2 2-328
    - df2sos 2-338
    - df2t 2-349
    - df2tsos 2-360
    - direct-form antisymmetric FIR 2-373
    - direct-form FIR transposed 2-388
    - direct-form II transposed (df2t) 2-349
    - direct-form IIR 2-381
    - direct-form symmetric FIR 2-395
    - lattice allpass 2-411
    - lattice autoregressive 2-420
    - lattice moving-average maximum 2-438
    - lattice moving-average minimum 2-447
    - parallel 2-457
    - scalar 2-458
    - See also* Signal Processing Toolbox
      - documentation
  - dfilt function 2-252
    - convert structures 2-264
    - copying 2-263
    - delay 2-283
    - FFT FIR 2-408
    - methods 2-256
    - structures 2-252
  - dfilt properties
    - arithmetic 3-20
    - dfilt.cascade 2-273
    - dfilt.delay function 2-283
    - dfilt.df1 2-285
    - dfilt.df1sos 2-294
    - dfilt.df1t 2-304
    - dfilt.df1tsos 2-315
    - dfilt.df2 2-328
    - dfilt.df2sos 2-338
    - dfilt.df2t 2-349
    - dfilt.df2tsos 2-360
    - dfilt.dffir 2-381
    - dfilt.dffirt 2-388
    - dfilt.dfsymfir 2-395
    - dfilt.fftir function 2-408
    - dfilt.latticeallpass 2-411
    - dfilt.latticear 2-420
    - dfilt.latticemamax 2-438
    - dfilt.latticemamin 2-447
    - dfilt.parallel 2-457
    - dfilt.scalar 2-458
    - differentiators
      - least square linear-phase FIR 2-831
    - direct-form I 3-47
      - transposed 3-48
    - direct-form II 3-49
      - transposed 3-51
    - double
      - property value 3-20
    - dynamic properties 3-5
- E**
- EpsilonStates 3-110
  - ErrorStates 3-110
- F**
- farrow filter 2-483
  - fcfwrite method 2-258
  - fdesign

- reference 2-503
- FFTCoefficients 3-110
- fftcoeffs method 2-258
- FFTStates 3-111
- filter
  - initial conditions 2-9
  - states 2-9
- filter conversions 3-66
- filter design
  - multirate 1-10
- filter method 2-258
- filter sections
  - specifying 3-66
- filter structures
  - about 3-43
  - all-pass lattice 3-58
  - direct-form antisymmetric FIR 3-53
  - direct-form FIR 3-55
  - direct-form I 3-46
  - direct-form I SOS IIR 3-47
  - direct-form I transposed 3-48
  - direct-form I transposed IIR 3-48
  - direct-form II 3-49
  - direct-form II IIR 3-49
  - direct-form II SOS IIR 3-50
  - direct-form II transposed 3-51
  - direct-form II transposed IIR 3-51
  - direct-form symmetric FIR 3-63
  - direct-form transposed FIR 3-56
  - FIR transposed 3-56
  - fixed-point 3-45
  - lattice allpass 3-58
  - lattice AR 3-60
  - lattice ARMA 3-62
  - lattice autoregressive moving average 3-62
  - lattice moving average maximum phase 3-59
  - lattice moving average minimum phase 3-60
- FilteredInputStates 3-111
- filterinternals
  - fixed-point filter 3-43
  - multirate filter 3-122
- FilterLength 3-111
- filters
  - converting 3-65
  - FIR 3-43
  - impulse response 2-949
  - initial conditions using dfilter 2-264
  - lattice 3-43
  - objects 2-252
  - overlap-add using dfilter.fftfir 2-408
  - state-space 3-43
  - states 2-264
- FilterStructure property 3-43
- finite impulse response
  - antisymmetric 3-53
  - symmetric 3-63
- fir 3-55
- FIR filters 3-43
  - least square linear phase 2-829
- firls function 2-829
- firt 3-56
- firtype method 2-258
- fixed
  - arithmetic property value 3-22
- fixed-point filter properties
  - AccumFracLength 3-19
  - AccumWordLength 3-19
  - Arithmetic 3-20
  - CastBeforeSum 3-33
  - CoeffAutoScale 3-35
  - CoeffFracLength 3-40
  - CoeffWordLength 3-40
  - DenAccumFracLength 3-41
  - DenFracLength 3-41
  - Denominator 3-41
  - DenProdFracLength 3-42
  - DenStateFracLength 3-42
  - DenStateWordLength 3-42
  - FilterStructure 3-43
- fixed-point filter states 3-95

- fixed-point filter structures 3-45
- fixed-point filters
  - dynamic properties 3-5
- ForgettingFactor 3-111
- fraction length
  - about 3-29
  - negative number of bits 3-29
- frequency response 2-854
- freqz 2-854
- freqz method 2-258
- FwdPredErrorPower 3-111
- FwdPrediction 3-111

## G

- grpdelay method 2-258

## H

- hilbert transform function
  - using firls 2-830

## I

- impz method 2-258
- impzlength method 2-258
- info method
  - dfilt function 2-258
- InitFactor 3-111
- initial conditions 2-9
  - using dfilt states 2-264
- InvCov 3-112
- isallpass method 2-259
- iscascade method 2-259
- isfir method 2-259
- islinphase method 2-259
- ismaxphase method 2-259
- isminphase method 2-259
- isparallel method 2-259
- isreal method 2-259

- isscalar method 2-259
- issos method 2-259
- isstable method 2-259

## K

- KalmanGain 3-112
- KalmanGainStates 3-112

## L

- latcallpass 3-58
- lattice filters
  - allpass 3-58
  - AR 3-60
  - ARMA 3-62
  - autoregressive 3-60
  - MA 3-60
  - moving average maximum phase 3-59
  - moving average minimum phase 3-60
- latticear 3-60
- latticearma 3-62
- latticeca 3-59
- laticemamax 3-59
- laticemamin 3-60
- least squares method FIR 2-829
- linear phase filters
  - least squares FIR 2-829

## M

- mfilt object 2-992
- mfilt objects 1-10
- multiple sections
  - specifying 3-66
- multirate filter functions 1-10
- multirate filter states 3-131
- multirate object 2-992
  - See also* mfilt



**N**

negative fraction length  
     interpret 3-29  
 nsections method 2-260  
 nstages method 2-260  
 nstate method 2-260

**O**

object  
     adaptfilt 2-2  
     changing properties 2-9 2-263  
     filter 2-252  
     mfilt 2-992  
     viewing parameters 2-8  
     viewing properties 2-263  
 object properties  
     AccumWordLength 3-19  
 Offset 3-112  
 OffsetCov 3-112  
 order method 2-260  
 overlap-add filter 2-408

**P**

parallel method 2-260  
 PersistentMemory 3-113  
 phasez method 2-260  
 plots  
     zero-pole, command for 2-1251  
 pole-zero plots 2-1251  
 polyphase filters 1-10  
     *See also* multirate filter functions  
 Power 3-112  
 precision 3-30  
 ProjectionOrder 3-113  
 properties  
     dynamic 3-5  
     FilterStructure 3-43  
     ScaleValues 3-83

**Q**

quantized filters  
     architecture 3-43  
     direct-form FIR 3-55  
     direct-form FIR transposed 3-56  
     direct-form symmetric FIR 3-63  
     filtering data 2-663 2-665  
     finite impulse response 3-56  
     frequency response 2-854  
     lattice allpass 3-58  
     lattice AR 3-60  
     lattice ARMA 3-62  
     lattice coupled-allpass 3-58  
     lattice MA maximum phase 3-59  
     lattice MA minimum phase 3-60  
     reference filter 3-63  
     scaling 3-83  
     specifying 3-63  
     specifying coefficients for multiple  
         sections 3-66  
     structures 3-43  
     symmetric FIR 3-53  
     zero-pole plots 2-1251  
 quantized filters properties  
     ScaleValues 3-83

**R**

realizemdl method 2-261  
 reference coefficients  
     specifying 3-63  
 ReflectionCoeffs 3-113  
 ReflectionCoeffsStep 3-113  
 removestage method 2-261  
 represent numeric data 3-29  
 ResetBeforeFiltering  
     (PersistentMemory) 3-113

**S**

ScaleValues property 3-83  
    interpreting 3-84  
scaling  
    implementing for quantized filters 3-84  
    quantized filters 3-83  
second-order sections  
    normalizing 3-66  
SecondaryPathCoeffs 3-113  
SecondaryPathEstimate 3-113  
SecondaryPathStates 3-113  
setstage method 2-261  
single  
    property value 3-21  
sos method 2-262  
SqrtCov 3-113  
SqrtInvCov 3-114  
ss method 2-262  
States 3-114  
states, fixed-point filter 3-95  
states, multirate filter 3-131  
StepSize 3-114  
stepz method 2-262  
SwBlockLength 3-114

symmetricfir 3-63

**T**

tf method 2-262

**V**

vectors  
    weighting 2-830

**W**

word length  
    about 3-29

**Z**

zero-pole plots 2-1251  
zerophase method 2-262  
zpk method 2-263  
zplane 2-1251  
    plotting options 2-1251  
zplane method 2-263