

# Filter Design Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

## 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 User's Guide*

© COPYRIGHT 2000–2006 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc. Other product or brand names are 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 2000	Online only	New for Version 1.0
September 2000	First printing	Revised for Vesion 2.0 (Release 12)
June 2001	Online only	Revised for Version 2.1 (Release 12.1)
July 2002	Online only	Revised for Version 2.2 (Release 13)
November 2002	Online only	Revised for Version 2.5
June 2004	Online only	Revised for Version 3.0 (Release 14)
October 2004	Online only	Revised for Version 3.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.2 (Release 14SP2)
September 2005	Online only	Revised for Version 3.3 (Release 14SP3)
March 2006	Online only	Revised for Version 3.4 (Release 2006a)



## What Is Filter Design Toolbox?

1

<b>Introducing the Filter Design Toolbox</b> .....	1-2
Key Features .....	1-2
<b>Getting Started with the Toolbox</b> .....	1-4
Using Specification Objects to Design Filters .....	1-4
Getting General Filter Specification Object Help .....	1-5
Creating a Filter Specification Object .....	1-5
Changing Specifications for Specification Objects .....	1-6
Setting Design Parameters .....	1-7
Normalizing Frequency Specifications .....	1-10
Designing Filters From Filter Specification Objects .....	1-11
Using Design Time Options .....	1-12
Comparing Designs .....	1-13
Example—Creating a Fixed-Point IIR Filter .....	1-14
<b>Selected Bibliography</b> .....	1-24

## Designing Fixed-Point Filters

2

<b>Designing Fixed-Point Filters</b> .....	2-3
The Filter Design Process .....	2-3
Designing a Filter With Floating-Point Coefficients .....	2-6
Converting the Filter to Fixed-Point .....	2-7
Quantizing Filter Coefficients with Automatic Scaling .....	2-9
Scaling Filter Coefficients Manually .....	2-10
Specifying Arithmetic Rules .....	2-12
<b>Working with Fixed-Point Direct-Form FIR Filters</b> .....	2-14
Obtaining the Filter .....	2-14
Creating the Direct-Form FIR Fixed-Point Filter .....	2-15

Comparing Quantized Coefficients to Nonquantized Coefficients	2-15
Determining the Number of Bits being Used	2-16
Determining the Proper Coefficient Word Length	2-17
Fixed-Point Filtering	2-18
Generating a Baseline Output for Comparison	2-20
Computing the Fixed-Point Filter Output	2-21
Reducing Filter Output Quantization	2-21
The Advantages of Guard Bits	2-22
Avoiding Overflow Without Guard Bits	2-26
<b>Constructing Fixed-Point Filters</b>	<b>2-28</b>
Defining Quantized and Fixed-Point Filters	2-28
Constructors for Fixed-Point Filters	2-29
Constructing a Quantized Filter from a Filter Specification Object	2-30
Copying Filters to Inherit Properties	2-31
Fixed-Point Arithmetic Filter Structures	2-32
<b>Data Type Handling in Discrete-Time Filters</b>	<b>2-36</b>
Filter Input Signals, Coefficients, and States	2-36
The CastBeforeSum Filter Property	2-42
<b>Introduction to Fixed-Point Arithmetic</b>	<b>2-45</b>
Binary Point Interpretation	2-46
Precision and Dynamic Range	2-49
Overflows and Scaling	2-50

## Designing Multirate Filters

# 3

<b>Introducing Multirate Filters</b>	<b>3-2</b>
<b>Getting Started—Designing Multirate Filters</b>	<b>3-4</b>
Creating Multirate Filters	3-4
Getting and Setting Filter Coefficients	3-6
Analyzing Multirate and Multistage Filters	3-8
Filtering with Multirate Filters	3-9

Specifying Initial Conditions to the Filter .....	3-11
Streaming Data to the Filter .....	3-12
Filtering Multichannel Signals .....	3-13
Generating Simulink Blocks .....	3-15
Getting Help About Multirate Filters .....	3-15
<b>FIR Decimation—Filtering with FIR Decimators .....</b>	<b>3-18</b>
Creating FIR Decimators .....	3-18
Understanding Input Sample Processing and the InputOffset Property .....	3-19
Filtering with FIR Decimators .....	3-21
<b>CIC Filter Example—Using CIC Decimation Filters .....</b>	<b>3-24</b>
Creating CIC Decimator filters .....	3-24
Analyzing CIC Decimation Filters .....	3-26
About the MSB at the Filter Output .....	3-27
Working with Section Word Lengths .....	3-28
CIC Filter States .....	3-31
Filter Implementation—Signal Flow Graph .....	3-33
Reference .....	3-35
<b>Analyzing Multirate and Multistage Filters .....</b>	<b>3-36</b>
Analyzing Single-Stage Multirate Filters .....	3-37
Comparing Interpolators .....	3-38
Performing Multistage Filter Analysis .....	3-40
Analyzing Multistage Interpolators .....	3-42
Analyzing a Multistage Sample-Rate Converter .....	3-43
Analyzing Other Multistage Configurations .....	3-45
<b>Audio Example—Audio Sample Rate Conversion .....</b>	<b>3-47</b>
Creating the Multirate Filters .....	3-47
Decreasing the Sample Rate by a Fractional Factor .....	3-48
Constructing the Fractional Decimator .....	3-48
Filtering to Change the Sample Rate .....	3-49
Comparing the Resampled Signals .....	3-49
Increasing the Sample Rate by a Fractional Factor .....	3-51
Plotting the Original Signal and the Reconverted Signal .....	3-52
Converting from 48 kHz to 44.1 kHz .....	3-53
Plotting the 48 kHz Signal and the 44.1 kHz Signal .....	3-54

<b>Introducing Adaptive Filtering</b> .....	<b>4-2</b>
<b>Getting Started with Adaptive Filters</b> .....	<b>4-4</b>
Tutorial Contents .....	4-4
Create the Signals for Adaptation .....	4-4
Construct Two Adaptive Filters .....	4-5
Choose the Step Size .....	4-6
Set the Adapting Filter Step Size .....	4-7
Filter with the Adaptive Filters .....	4-7
Compute the Optimal Solution .....	4-8
Plot the Results .....	4-8
Compare the Final Coefficients .....	4-9
Reset the Filter Before Filtering .....	4-10
Investigate Convergence Through Learning Curves .....	4-10
<b>Overview of Adaptive Filters and Applications</b> .....	<b>4-14</b>
Choosing an Adaptive Filter .....	4-16
System Identification .....	4-17
Inverse System Identification .....	4-18
Noise Cancellation (or Interference Cancellation) .....	4-18
Prediction .....	4-19
<b>Adaptive Filters in the Filter Design Toolbox</b> .....	<b>4-21</b>
Algorithms .....	4-21
Using Adaptive Filter Objects .....	4-25
<b>Examples of Adaptive Filters That Use LMS Algorithms</b> .	<b>4-26</b>
adaptfilt.lms Example—System Identification .....	4-27
adaptfilt.nlms Example—System Identification .....	4-31
adaptfilt.sd Example—Noise Cancellation .....	4-34
adaptfilt.se Example—Noise Cancellation .....	4-38
adaptfilt.ss Example—Noise Cancellation .....	4-42

<b>Example of Adaptive Filter That Uses RLS Algorithm . . .</b>	<b>4-47</b>
adapfilt.rls Example—Inverse System Identification . . . . .	4-48
<b>Selected Bibliography . . . . .</b>	<b>4-52</b>

## Digital Frequency Transformations

# 5

<b>Introduction . . . . .</b>	<b>5-2</b>
<b>Definition of the Problem . . . . .</b>	<b>5-3</b>
Selecting Features Subject to Transformation . . . . .	5-6
Mapping from Prototype Filter to Target Filter . . . . .	5-8
Summary of Frequency Transformations . . . . .	5-9
<b>Frequency Transformations for Real Filters . . . . .</b>	<b>5-11</b>
Real Frequency Shift . . . . .	5-12
Real Lowpass to Real Lowpass . . . . .	5-13
Real Lowpass to Real Highpass . . . . .	5-15
Real Lowpass to Real Bandpass . . . . .	5-17
Real Lowpass to Real Bandstop . . . . .	5-19
Real Lowpass to Real Multiband . . . . .	5-21
Real Lowpass to Real Multipoint . . . . .	5-23
<b>Frequency Transformations for Complex Filters . . . . .</b>	<b>5-26</b>
Complex Frequency Shift . . . . .	5-26
Real Lowpass to Complex Bandpass . . . . .	5-28
Real Lowpass to Complex Bandstop . . . . .	5-29
Real Lowpass to Complex Multiband . . . . .	5-31
Real Lowpass to Complex Multipoint . . . . .	5-33
Complex Bandpass to Complex Bandpass . . . . .	5-35

## Using FDATool with the Filter Design Toolbox

# 6

<b>Designing Advanced Filters in FDATool</b> .....	<b>6-5</b>
<b>Switching FDATool to Quantization Mode</b> .....	<b>6-8</b>
<b>Quantizing Filters in the Filter Design and Analysis Tool</b>	<b>6-12</b>
Coefficients Options .....	<b>6-13</b>
Input/Output Options .....	<b>6-14</b>
Filter Internals Options .....	<b>6-17</b>
Filter Internals Options for CIC Filters .....	<b>6-21</b>
<b>Analyzing Filters with a Noise-Based Method</b> .....	<b>6-23</b>
Using the Magnitude Response Estimate Method .....	<b>6-23</b>
Comparing the Estimated and Theoretical Magnitude Responses .....	<b>6-28</b>
Choosing Quantized Filter Structures .....	<b>6-28</b>
Converting the Structure of a Quantized Filter .....	<b>6-28</b>
Converting Filters to Second-Order Sections Form .....	<b>6-29</b>
<b>Scaling Second-Order Section Filters</b> .....	<b>6-30</b>
<b>Reordering the Sections of Second-Order Section Filters</b>	<b>6-38</b>
Switching FDATool to Reorder Filters .....	<b>6-38</b>
<b>Viewing SOS Filter Sections</b> .....	<b>6-46</b>
<b>Importing and Exporting Quantized Filters</b> .....	<b>6-53</b>
To Export Quantized Filters .....	<b>6-55</b>
<b>Importing XILINX Coefficient (.COE) Files</b> .....	<b>6-58</b>
<b>Transforming Filters</b> .....	<b>6-59</b>
Original Filter Type .....	<b>6-60</b>
Frequency Point to Transform .....	<b>6-63</b>
Transformed Filter Type .....	<b>6-64</b>
Specify Desired Frequency Location .....	<b>6-64</b>



<b>Designing Multirate Filters in FDATool</b> .....	<b>6-70</b>
Switching FDATool to Multirate Filter Design Mode .....	<b>6-70</b>
Controls on the Multirate Design Panel .....	<b>6-71</b>
Quantizing Multirate Filters .....	<b>6-81</b>
<b>Realizing Filters as Simulink Subsystem Blocks</b> .....	<b>6-84</b>
About the Realize Model Panel in FDATool .....	<b>6-84</b>
<b>Getting Help for FDATool</b> .....	<b>6-89</b>
The What's This? Option .....	<b>6-89</b>
Additional Help for FDATool .....	<b>6-89</b>

## Reference for the Properties of Filter Objects

# 7

<b>Overview</b> .....	<b>7-2</b>
<b>Fixed-Point Filter Properties</b> .....	<b>7-3</b>
<b>Adaptive Filter Properties</b> .....	<b>7-103</b>
<b>Multirate Filter Properties</b> .....	<b>7-117</b>

## Function Reference

# 8

<b>Functions — By Category</b> .....	<b>8-2</b>
Adaptive Filter Constructors .....	<b>8-3</b>
Discrete-Time Filter Constructors .....	<b>8-6</b>
Filter Specification Objects — Response Types .....	<b>8-8</b>
Filter Specification Objects — Design Methods .....	<b>8-9</b>
Multirate Filter Constructors .....	<b>8-10</b>
Filter Analysis Methods .....	<b>8-12</b>
Fixed-Point Filter Construction and Property Functions ....	<b>8-17</b>

Quantized Filter Analysis Functions .....	8-17
SOS Conversion Functions .....	8-19
Filter Design Functions .....	8-19
Filter Conversion Functions .....	8-20
<b>Functions — Alphabetical List .....</b>	<b>8-21</b>

## Bibliography

### A

Advanced Filters .....	A-2
Adaptive Filters .....	A-2
Multirate Filters .....	A-3
Frequency Transformations .....	A-3

## Index

# What Is Filter Design Toolbox?

---

Introducing the Filter Design Toolbox  
(p. 1-2)

Describes the toolbox briefly

Getting Started with the Toolbox  
(p. 1-4)

Introduces filter design in the toolbox by presenting two demonstrations

Selected Bibliography (p. 1-24)

Lists some books that offer details about digital filtering and digital signal processing

## Introducing the Filter Design Toolbox

The Filter Design Toolbox is a collection of tools that provides advanced techniques for designing, simulating, and analyzing digital filters. It extends the capabilities of the Signal Processing Toolbox with filter architectures and design methods for complex real-time DSP applications, including adaptive filtering and multirate filtering, as well as filter transformations.

Used with the Fixed-Point Toolbox, the Filter Design Toolbox provides functions that simplify the design of fixed-point filters and the analysis of quantization effects. When used with the Filter Design HDL Coder, the Filter Design Toolbox lets you generate VHDL and Verilog code for fixed-point filters.

### Key Features

- FIR filter design, including minimum-order, minimum-phase, constrained-ripple, halfband, Nyquist, interpolated FIR, and nonlinear phase
- IIR filter design, including arbitrary magnitude and phase, group-delay equalizers, constrained-pole radius, peaking, notching, and comb filters
- Multirate filter design, analysis, and implementation, including cascaded integrator-comb (CIC) fixed-point multirate filters and compensators
- Farrow filter design
- Multirate, multistage filter design
- Wave digital filter design
- IIR filters implemented in second-order sections, including design, scaling, and section reordering
- Analysis and implementation of digital filters in single-precision floating-point and fixed-point arithmetic
- Perfect reconstruction and two-channel FIR filter bank design
- Round-off noise analysis for filters implemented in single-precision floating point or fixed point
- FIR and IIR filter transformations, including lowpass to lowpass, lowpass to highpass, and lowpass to multiband

- Adaptive filter design, analysis, and implementation, including LMS-based, RLS-based, lattice-based, frequency-domain, fast transversal, and affine projection adaptive filters
- C code header file generation from filter designs in FDATool. The header file includes the filter coefficients and information about the filter design
- VHDL and Verilog code generation for fixed-point filters with the Filter Design HDL Coder

## Getting Started with the Toolbox

This section provides an example to get you started using Filter Design Toolbox. You can run the code in this example from the Help browser (select the code, right-click the selection, and choose **Evaluate Selection** from the context menu) or you can enter the code on the command line. This exercise also introduces Filter Design and Analysis Tool (FDATool). You use it to design and analyze filters, and to quantize filters.

As you follow the example, you are introduced to some of the basic tasks of designing a filter and using FDATool. You will engage some of the quantization capabilities of the toolbox, and a few of the filter analyses provided as well.

Before you begin this example, start MATLAB<sup>®</sup> and verify that you have installed Signal Processing and Filter Design Toolboxes (type `ver` at the command prompt). You should see Filter Design Toolbox, Signal Processing Toolbox, and Fixed-Point Toolbox (to do fixed-point filter design and analysis) among others, in the list of installed products.

## Using Specification Objects to Design Filters

The filter specification (`fdesign`) objects let you design many single rate, multirate, and multistage filters, such as lowpass, highpass, bandpass, and bandstop IIR and FIR using a range of design algorithms. and many other types of filters with a variety of constraints. The design process computes the filter coefficients using the various algorithms available in the Signal Processing and Filter Design Toolboxes and associates a particular filter structure to those coefficients.

This tutorial review of filter design contains the following sections:

- “Getting General Filter Specification Object Help” on page 1-5
- “Creating a Filter Specification Object” on page 1-5
- “Changing Specifications for Specification Objects” on page 1-6
- “Setting Design Parameters” on page 1-7
- “Normalizing Frequency Specifications” on page 1-10
- “Using Design Time Options” on page 1-12
- “Comparing Designs” on page 1-13

## Getting General Filter Specification Object Help

Entering `help fdesign` in the command window opens the help for filter specification objects. Various hyperlinks in the help enable you to navigate to all of the help for the filter specification objects.

You can also enter

```
help responses
help fdesign/responses
```

at the command prompt for information about the response types you can specify for filter specification objects. Both forms return the same information.

## Creating a Filter Specification Object

To create a filter specification object, you need to select the response to be used. For example, to create a lowpass filter you would type:

```
d = fdesign.lowpass
d =

Response: 'Lowpass'
Specification: 'Fp,Fst,Ap,Ast'
Description: {'Passband Frequency';'Stopband...
Frequency';'Passband Ripple (dB)';'Stopband Attenuation (dB)'}
NormalizedFrequency: true
Fpass: 0.45
Fstop: 0.55
Apass: 1
Astop: 60
```

Notice that each specification is listed as an abbreviation. `Fp` is the abbreviation for `Fpass` (the passband frequency edge) and `Fst` is the abbreviation for `Fstop` (the stopband frequency edge).

The `Description` property provides a full description of the properties that are added by the Specification.

```
get(d, 'description')
ans =

    'Passband Frequency'
    'Stopband Frequency'
```

```
'Passband Ripple (dB)'  
'Stopband Attenuation (dB)'
```

## Changing Specifications for Specification Objects

The Specification property allows you to select different design parameters. This is a string which lists the specifications that will be used for the design. To see all valid specifications type:

```
set(d, 'Specification')  
ans =
```

```
'Fp,Fst,Ap,Ast'  
'N,F3dB'  
'N,F3dB,Ap'  
'N,F3dB,Ap,Ast'  
'N,F3dB,Ast'  
'N,F3dB,Fst'  
'N,Fc'  
'N,Fc,Ap,Ast'  
'N,Fp,Ap'  
'N,Fp,Ap,Ast'  
'N,Fp,F3dB'  
'N,Fp,Fst'  
'N,Fp,Fst,Ap'  
'N,Fp,Fst,Ast'  
'N,Fst,Ap,Ast'  
'N,Fst,Ast'  
'Nb,Na,Fp,Fst'
```

Changing the Specification changes which the properties for the the object:

```
set(d, 'Specification', 'N,Fc');  
d  
d =
```

```
Response: 'Lowpass'  
Specification: 'N,Fc'  
Description: {'Filter Order';'Cutoff Frequency'}  
NormalizedFrequency: true  
FilterOrder: 10
```



```
Fcutoff: 0.5
```

## Setting Design Parameters

You can set design parameters after creating your specification object, or you can pass the specifications when you construct your object.

For example:

```
specs = 'N,Fp,Fst';
d = fdesign.lowpass(specs)
d =

    Response: 'Lowpass'
    Specification: 'N,Fp,Fst'
    Description: {'Filter Order'; 'Passband Frequency'; 'Stopband...
    Frequency'}
    NormalizedFrequency: true
    FilterOrder: 10
    Fpass: 0.45
    Fstop: 0.55
```

After specifying the specification to use, then specify the values for those specifications.

```
N = 40; % Filter Order.
Fpass = .33; % Passband Frequency Edge.
Fstop = .4; % Stopband Frequency Edge.
d = fdesign.lowpass(specs, N, Fpass, Fstop)
d =

    Response: 'Lowpass'
    Specification: 'N,Fp,Fst'
    Description: {'Filter Order'; 'Passband Frequency'; 'Stopband
    Frequency'}
    NormalizedFrequency: true
    FilterOrder: 40
    Fpass: 0.33
    Fstop: 0.4
```

You can also specify a sampling frequency after all of the specifications have been entered.

```
Fpass = 1.3;
```

```
Fstop = 1.6;
Fs = 4.5; % Sampling Frequency
d = fdesign.lowpass(specs, N, Fpass, Fstop, Fs)
d =
```

```
Response: 'Lowpass'
Specification: 'N,Fp,Fst'
Description: {'Filter Order';'Passband Frequency';'Stopband
Frequency'}
NormalizedFrequency: false
Fs: 4.5
FilterOrder: 40
Fpass: 1.3
Fstop: 1.6
```

Amplitude specifications can be given in linear or squared units by providing a flag to the `fdesign` method. However, the specifications are always stored in dB.

```
Apass = .0575;
specs = 'N,Fp,Ap';
d = fdesign.lowpass(specs, N, Fpass, Apass, Fs, 'linear')
d =
```

```
Response: 'Lowpass'
Specification: 'N,Fp,Ap'
Description: {'Filter Order';'Passband Frequency';'Passband
Ripple (dB)'}
NormalizedFrequency: false
Fs: 4.5
FilterOrder: 40
Fpass: 1.3
Apass: 0.999980343384991
Apass = .95;
```

```
d = fdesign.lowpass(specs, N, Fpass, Apass, Fs, 'squared')
d =
```

```
Response: 'Lowpass'
Specification: 'N,Fp,Ap'
Description: {'Filter Order';'Passband Frequency';'Passband
Ripple (dB)'}
NormalizedFrequency: false
Fs: 4.5
FilterOrder: 40
Fpass: 1.3
Apass: 0.999980343384991
Apass = .95;
```

```

NormalizedFrequency: false
Fs: 4.5
FilterOrder: 40
Fpass: 1.3
Apass: 0.222763947111522

```

An alternative way of changing specifications is by using `setspecs`. Work with `setspecs` the same way as the design function.

```

specs = 'N,F3dB,Ap';
F3dB = .9;
Apass = 1;
Fs = 2.5;
setspecs(d, specs, N, F3dB, Apass, Fs);
d
d =

```

```

Response: 'Lowpass'
Specification: 'N,F3dB,Ap'
Description: {'Filter Order'; '3dB Frequency'; ...
'Passband Ripple (dB)'}
NormalizedFrequency: false
Fs: 2.5
FilterOrder: 40
F3dB: 0.9
Apass: 1

```

If your object is already set to the correct Specification you can omit that input argument from your `setspecs` command.

```

F3dB = 1.1;
Apass = .5;
Fs = 3;
setspecs(d, N, F3dB, Apass, Fs);
d
d =

```

```

Response: 'Lowpass'
Specification: 'N,F3dB,Ap'
Description: {'Filter Order'; '3dB Frequency'; ...
'Passband Ripple (dB)'}

```

```
NormalizedFrequency: false
Fs: 3
FilterOrder: 40
F3dB: 1.1
Apass: 0.5
```

## Normalizing Frequency Specifications

To normalize your frequency specifications, use `normalizefreq` with the filter specification object.

```
normalizefreq(d);
d
d =

    Response: 'Lowpass'
    Specification: 'N,F3dB,Ap'
    Description: {'Filter Order';'3dB Frequency';...
'Passband Ripple (dB)'}
    NormalizedFrequency: true
    FilterOrder: 40
    F3dB: 0.7333333333333333
    Apass: 0.5
```

`normalizefreq` also unnormalizes the frequency specifications.

```
newFs = 3.1;
normalizefreq(d, false, newFs);
d
d =

    Response: 'Lowpass'
    Specification: 'N,F3dB,Ap'
    Description: {'Filter Order';'3dB Frequency';...
'Passband Ripple (dB)'}
    NormalizedFrequency: false
    Fs: 3.1
    FilterOrder: 40
    F3dB: 1.136666666666667
    Apass: 0.5
```

## Designing Filters From Filter Specification Objects

To design filters you use `design`.

```
d = fdesign.lowpass;
Hd = design(d)
Hd =

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

With no additional (or optional) inputs this syntax for `design` uses the default filter design method to design the default filter. To determine which method was used, use the `designmethods` method with the 'default' flag.

```
designmethods(d, 'default')
```

```
Default Design Method for class fdesign.lowpass (Fp,Fst,Ap,Ast):
equiripple
```

Specifying the command without outputs launches FVTool.

```
design(d)
```

For a complete list of design methods that apply to `d`, use `designmethods` without additional input arguments.

```
designmethods(d)

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

To get a better description of each design method use the **full** input argument to return the full names for the design methods.

```
designmethods(d, 'full')
Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):
Butterworth
Chebyshev Type I
Chebyshev Type II
Elliptic
Equiripple
Interpolated FIR
Kaiser Window
Multistage Equiripple
```

`designmethods` also accepts **fir** or **iir** flags to return only FIR algorithms or IIR algorithms.

To design a filter with a specific algorithm, specify the design algorithm at design time.

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

## Using Design Time Options

Many methods have options that are method-specific. For help on these design options, use `help` and pass the desired algorithm name as an input argument.

```
help(d, 'ellip')
DESIGN Design a Elliptic iir filter.
HD = DESIGN(D, 'ellip') designs a Elliptic 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 an Elliptic filter and matches the frequency and magnitude specification for the band

MATCH exactly. The other band will exceed the specification. MATCH can be 'stopband', 'passband' or 'both', and is 'both' by default.

```
% Example #1 - Compare passband and stopband MatchExactly.
d      = fdesign.lowpass('Fp,Fst,Ap,Ast', .1, .3, 1, 60);
Hd     = design(d, 'ellip', 'MatchExactly', 'passband');
Hd(2) = design(d, 'ellip', 'MatchExactly', 'stopband');

% Compare the passband edges in FVTool.
fvtool(Hd);
axis([.09 .11 -2 0]);
```

You specify the design options as parameter name/parameter value pairs when you design the filter.

```
design(d, 'ellip', 'MatchExactly', 'passband')
```

If you wish, you can provide these parameters in a structure. The `designopts` method returns a valid structure for your object and specified algorithm with the default values. Here is an example that uses `designopts` and a structure `do`. The example starts by getting the default design-time options.

```
do = designopts(d, 'ellip');
```

Now use the `MatchExactly` option for the stopband.

```
do.MatchExactly = 'stopband';
design(d, 'ellip', do);
```

## Comparing Designs

`design` can also help you investigate various designs simultaneously, by adding an optional input argument that specifies the kinds of filter to design.

Adding the input argument `allfir` directs `design` to return all of the FIR filters the available design methods can create. Begin by designing all of the FIR filters.

```
design(d, 'allfir');
```

The following code returns all of the IIR filters available.

```
design(d, 'alliir');
```

## Example—Creating a Fixed-Point IIR Filter

**Example Background.** To introduce you to designing fixed-point filters in the toolbox, this example uses Filter Design and Analysis Tool (FDATool) to design an IIR filter. In this case, use the Chebyshev I filter design method to begin the design process.

During the example, you have the chance to export filters to your MATLAB workspace, filter some data with the filter, and use the scaling features in FDATool to improve the filter performance.

One of the salient points in this example is, while second-order section (SOS) implementations are generally good starting points for fixed-point filter design, you might find that without scaling your SOS filter, the SOS implementation may not meet your needs, as this example shows.

### To Create a Fixed-Point Filter in FDATool

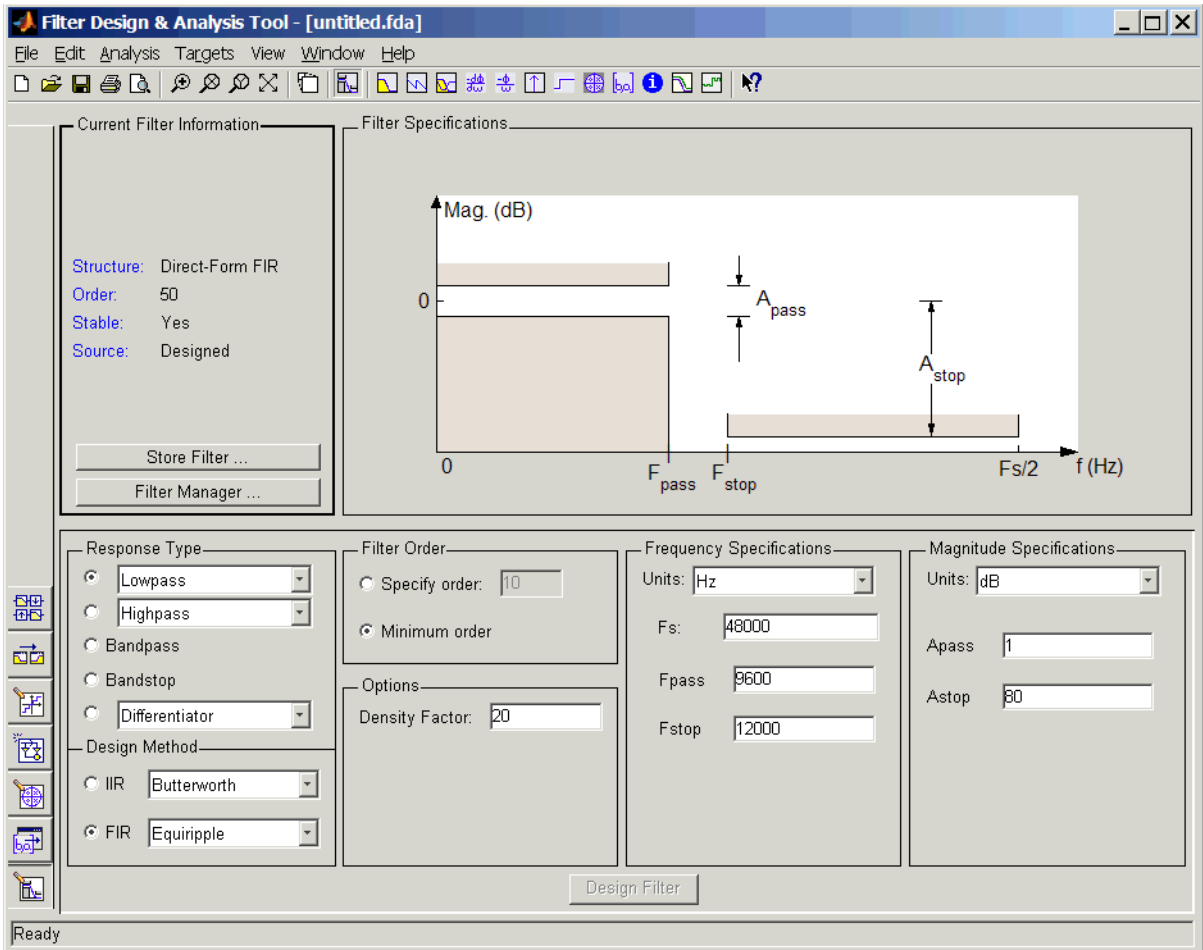
Filter Design and Analysis Tool (FDATool) is one tool this toolbox provides to help you design and analyze filters. From the various design panels in the tool, such as the filter design panel or the multirate filter design panel, you can design FIR and IIR filters, import or export filters, analyze filters, and more.

As an introduction to using the toolbox, this tutorial takes you through designing, quantizing, and scaling a filter in FDATool.

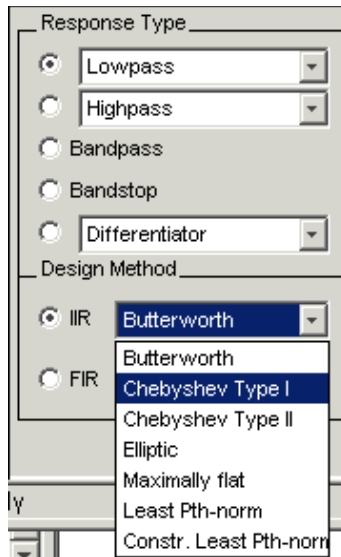
- 1 Open Filter Design and Analysis Tool by entering  
`fdatool`

at the MATLAB command prompt. FDATool opens to show you the following dialog.



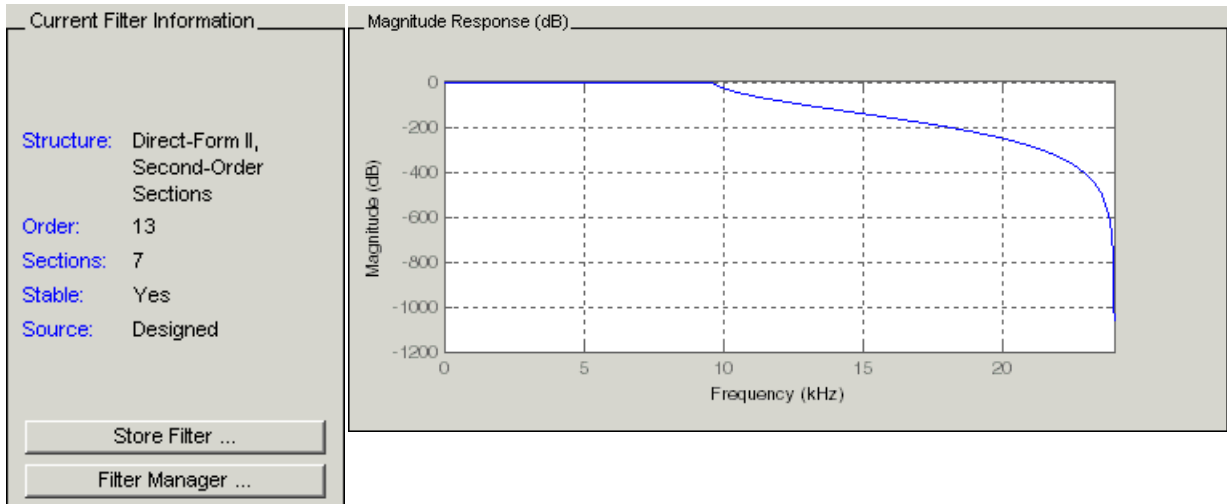


- Under **Design Method** in the bottom pane, select Chebyshev Type I from the **IIR** list and click **Design Filter**.



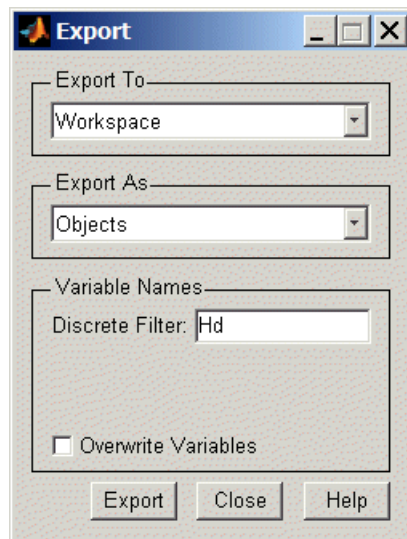
FDATool designs a double-precision lowpass filter using the Chebyshev I design method and displays the filter magnitude response in the FDATool analysis area. Your new uses seven second-order sections. In the **Current Filter Information** area in FDATool, reproduced in the next figure, you see your filter described by various filter parameters including the filter order (13) and the structure (direct-form II using second-order sections).

In the figure, next to the current filter information, the curve presents the filter magnitude response. As intended, it shows a lowpass filter with the end of the passband at about 9600 Hz.




Now export this filter to your workspace so you can use it to filter some data.

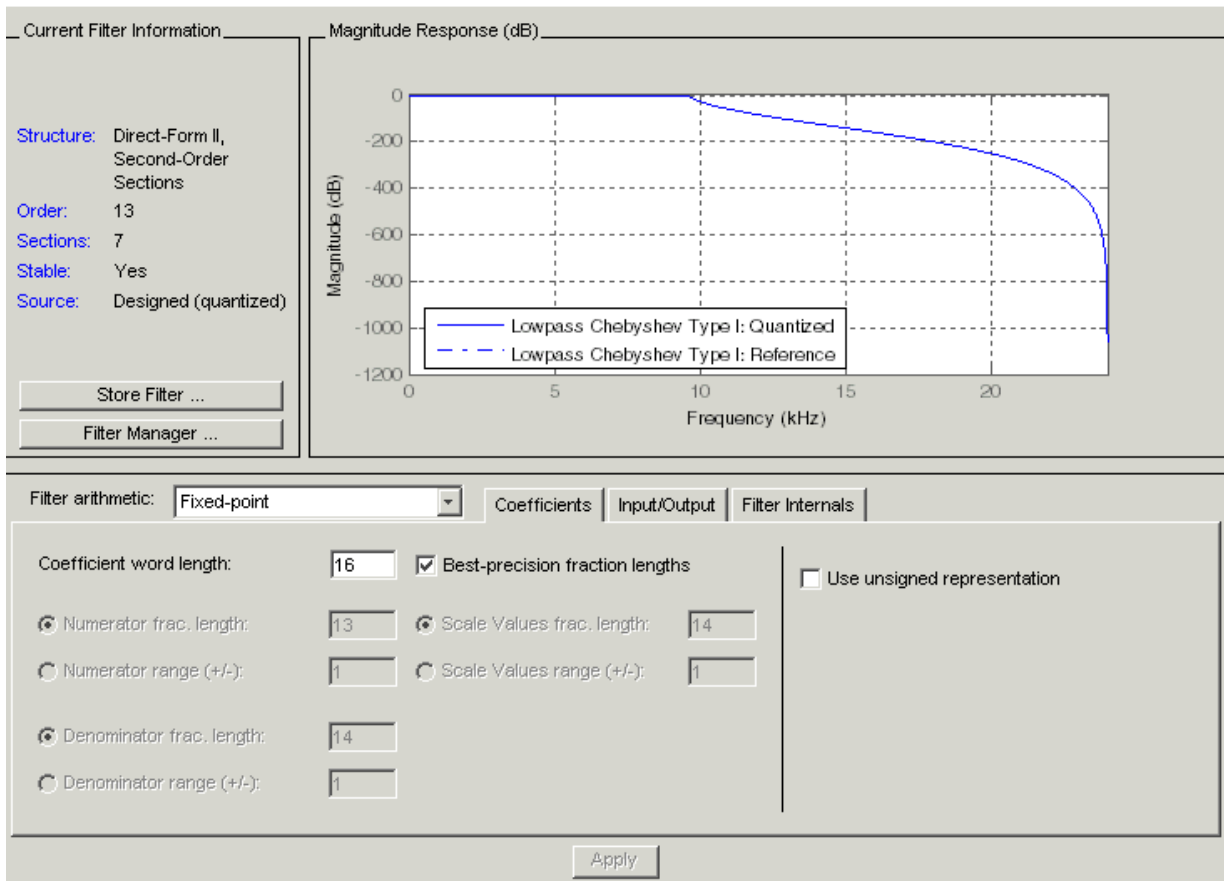
- 3 On the FDATool menu bar, select **File**—>**Export** to open the **Export** dialog.
- 4 To export the filter to your workspace as a filter object, select **Workspace** from **Export To** and select **Objects** for **Export As**. The export dialog looks like this after you make your selections.



- 5 Click **Export** to export the filter with the variable name shown in the **Export** dialog. When you return to your workspace in MATLAB, you see the new object. In this case, the new object is named Hd.
- 6 In MATLAB, create a vector of random data (with values between 0 and 1) and filter the data with Hd.  

```
x=rand(1000,1);  
y = filter(Hd,x);
```

Now y contains the data filtered by running x through the filter Hd.
- 7 Back in FDATool, click  on the side bar to switch FDATool to quantization mode.
- 8 With the quantization pane displayed in FDATool, switch **Filter arithmetic** to fixed-point. Now you see the quantization pane in FDATool, as shown in this figure.



In the analysis area, Fdatool shows the magnitude responses for two filters—your fixed-point (quantized) filter and the reference filter that accompanies the fixed-point version. Turn on the filter legend (select **View—Legend** from the menu bar) to help you identify which response belongs to each filter.

Zooming in on the curves shows that the two filter responses are very similar. Note that your fixed-point filter used the default settings in the

quantization pane—16-bit coefficients and fraction lengths selected to ensure the best precision.

**9** Now export the quantized filter to your workspace as an object. Since you are going to use the same variable name `Hd` for the quantized filter in your workspace, select **Overwrite variables** in the **Export** dialog.

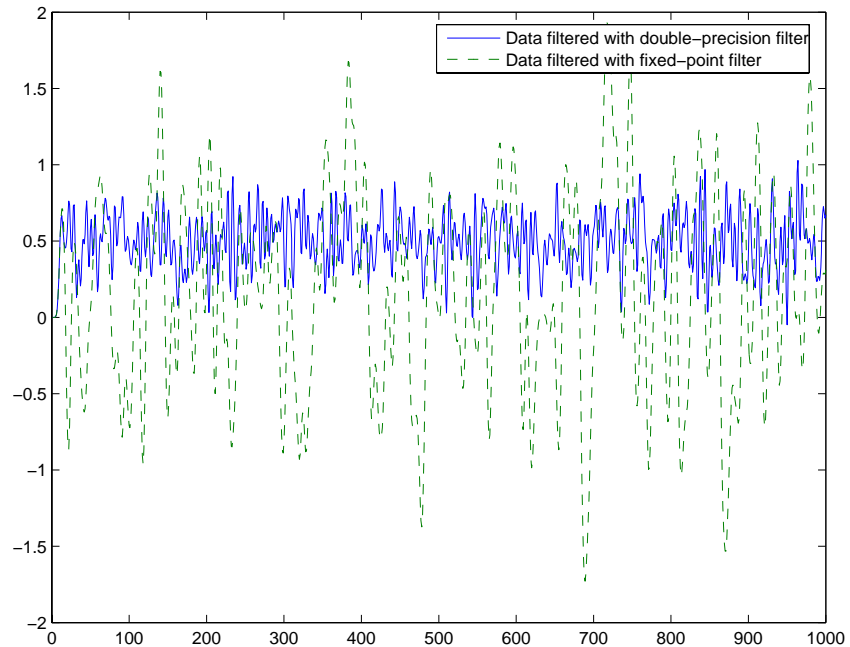
**10** Back in MATLAB, perform the filter process again, using the quantized filter `Hd` and the signal `x`.

```
yq = filter(Hd,x);
```

**11** This is the important step. Plot `y` and `yq` to see how the filtering process results differed between the double-precision filter `Hd` and the fixed-point filter `Hd`.

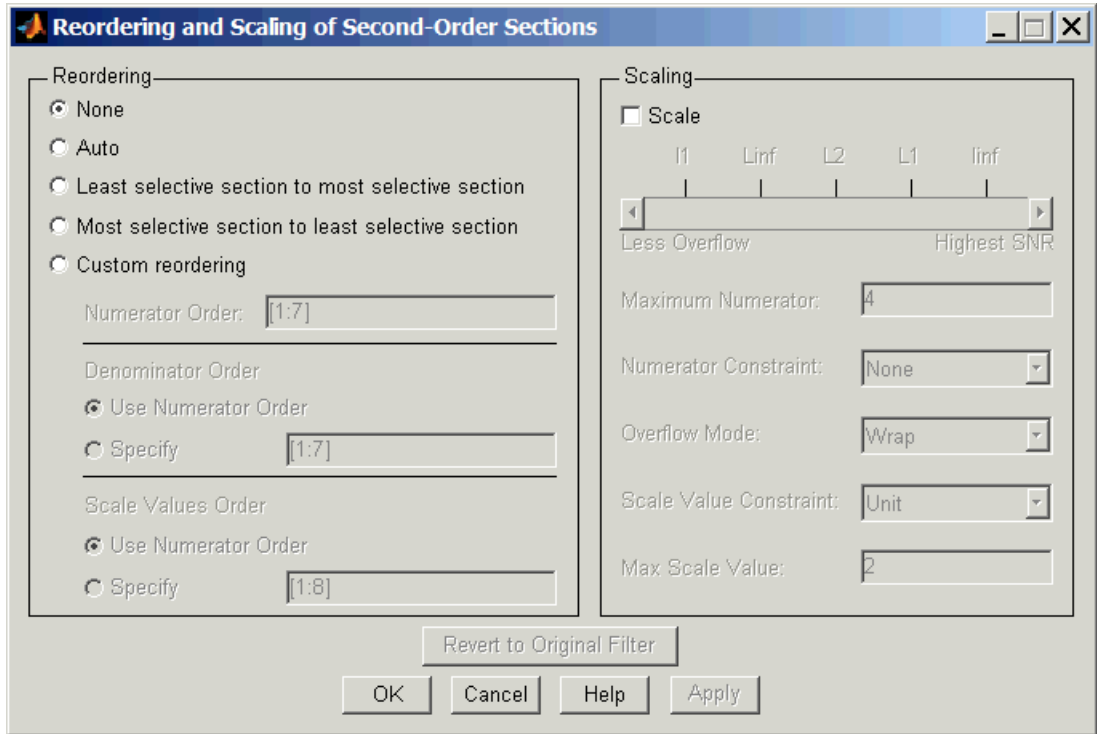
```
plot([y,yq]) % The results are not close to matching.
```

A look at the plot reveals that the results of filtering the same data (`x`) with each filter were very different. Recall that the magnitude responses seemed to be the same. So quantizing the filter affected the filtering performance in a way that the magnitude response curve does not show. The answer is that the arithmetic performed by the filter after quantization is very different from the double-precision filter before quantization.



Again, return to FDATool, which should still be open on your desktop. You are going to fix the discrepancy between  $y$  and  $y_q$  by reordering the sections of the fixed-point filter and scaling the filter to improve the performance after quantization.

- 1 To access the scaling and SOS filter reordering capability in FDATool, select **Edit**→**Reorder and Scale Second-Order Sections** from the menu bar. The **Reordering and Scaling of Second-Order Sections** dialog opens, shown below. Note the default settings:
  - No reordering option is selected.
  - Scaling is not selected.

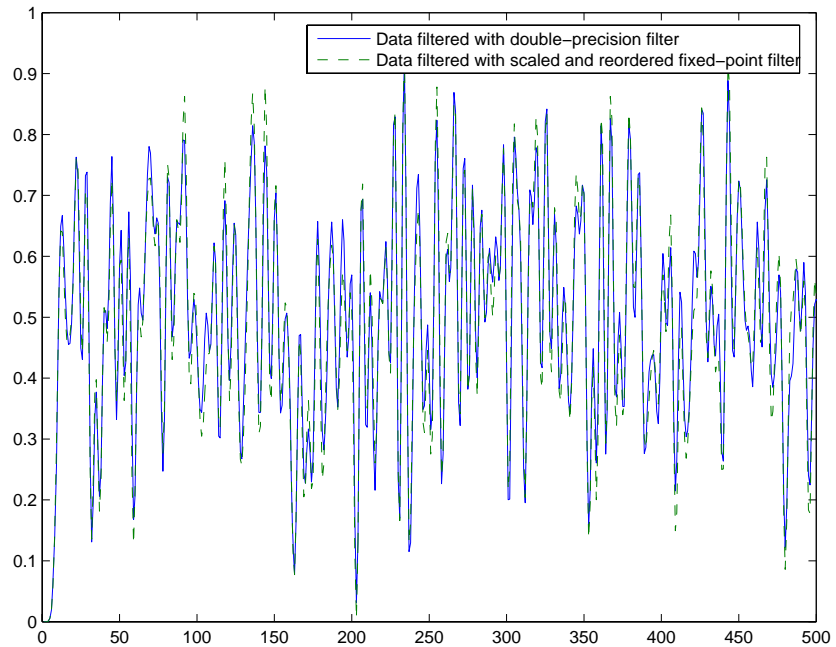


- 2 Review the settings. Set **Reordering** to Auto.
- 3 Select **Scale** in the **Scaling** area. Review the default settings to be sure **Maximum Numerator** is 4 and **Overflow Mode** is Wrap.
- 4 Click **OK** to close the dialog and scale and reorder the filter.
- 5 One more time, export the now-scaled quantized filter to your workspace as Hd.
- 6 Filter the data x again, using the latest Hd filter—now reordered and scaled.  
`yqs = filter(Hd,x);`
- 7 Finally, plot y and yqs to see if the filtering performance matches now.



```
plot([y,yqs]) % y and yqs are identical.
```

Here is the plot showing the results. Scaling and reordering the fixed-point filter restores the filtering performance to match the double-precision filter performance. The results demonstrate the power of scaling and reordering SOS filters.



## Selected Bibliography

For further information about the algorithms and computer models used to design filters and apply quantization in the toolbox, refer to one or more of the following references.

### Digital Filters

[1] Antoniou, Andreas, *Digital Filters*, Second Edition, McGraw-Hill, Inc., 1993.

[2] Mitra, Sanjit K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, Inc., 1998.

[3] Oppenheim, Alan V., R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, Inc., 1989.

### Quantization and Signal Processing

[4] Lapsley, Phil, J. Bier, A. Shoham, and E.A. Lee, *DSP Processor Fundamentals*, IEEE Press, 1997.

[5] McClellan, James H., C.S. Burrus, A.V. Oppenheim, T.W. Parks, R.W. Schaffer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB 5*, Prentice-Hall, Inc., 1998.

[6] Roberts, Richard A. and C.T. Mullis, *Digital Signal Processing*, Addison-Wesley Publishing Company, 1987.

[7] Van Loan, Charles, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

# Designing Fixed-Point Filters

---

Designing Fixed-Point Filters (p. 2-3)	Introduces fixed-point filters to get you started using your own filters
Working with Fixed-Point Direct-Form FIR Filters (p. 2-14)	Uses the direct-form FIR filter to introduce some analytical approaches from the toolbox and fixed-point filters
Constructing Fixed-Point Filters (p. 2-28)	Describes how you construct quantized filters in the toolbox
Data Type Handling in Discrete-Time Filters (p. 2-36)	Provides details about how discrete-time filters handle different data types as input, coefficients, and states
Introduction to Fixed-Point Arithmetic (p. 2-45)	Introduces the concepts underlying fixed-point arithmetic that relate to fixed-point filters

In the Filter Design Toolbox you can implement and analyze single-input single-output filters either as fixed-point filters, or as single-precision or double-precision floating-point filters. Both the single-precision floating-point and fixed-point filters are referred to as *quantized filters*.

You can create a quantized filter from a reference filter, that is, a filter whose coefficients and arithmetic operations you want to quantize in some fashion.

When you apply a quantized filter to data, not only are the filter coefficients quantized to your specification, but so are

- The data that you filter, both input and output
- The results of any arithmetic operations that occur during filtering

Refer to “Bibliography” for a list of relevant references on quantized filtering.

This chapter covers what you need to know to construct and use quantized filters:

- Getting Started with fixed-point filters
- Constructing quantized and fixed-point filters
- Fixed-point filter properties
- Filtering data with fixed-point filters
- Transformation functions for fixed-point filter coefficients
- Working with fixed-point direct-form FIR filters

Most of the filters you create in this toolbox are objects with properties. You can find much of the basic information you need to know about setting and retrieving property values in your MATLAB documentation by reading about the set and get functions.

## Designing Fixed-Point Filters

As filter designers begin to use digital filters in applications where power limitations and size constraints drive the filter design, they move from double-precision, floating-point filters to fixed-point filters. This tutorial shows you how to analyze the quantization effects introduced by such a conversion using discrete-time filter objects (`dfilt` objects).

This exercise covers the following filter development and analysis processes:

- “Designing a Filter With Floating-Point Coefficients” on page 2-6
- “Converting the Filter to Fixed-Point” on page 2-7
- “Quantizing Filter Coefficients with Automatic Scaling” on page 2-9
- “Scaling Filter Coefficients Manually” on page 2-10
- “Specifying Arithmetic Rules” on page 2-12

Each section builds on the contents and filters from preceding sections, so progressing through the tutorial from the start is most effective. Otherwise, code examples that depend on earlier tutorial sections might not work properly.

### The Filter Design Process

The toolbox uses a three step process to design filters.

- 1** Use `fdesign.response` to create a filter specifications object. For example, use `fdesign.bandpass` or `fdesign.decimator`.
- 2** Use `designmethods` to find out which design methods apply to your filter specification object.
- 3** Use one of the design methods from step 2 to design your filter from your specification object. Two of the design methods might be `ellip` or `cheby2`.

Now you have your filter and you can analyze it, test it, filter with it, or create other filters from your specification object to compare to the first filter.

Here is one example that design two highpass filters using different design methods, followed by a plot that shows both filter magnitude responses.

Notice that the example specifies multiple filter response features in the specification string argument.

- fp1—cutoff of the first passband
- fst1—first edge of the stopband
- fst2—second edge of the stopband
- fp2—edge of the second passband
- ap1—attenuation in the first passband
- ast—attenuation in the stopband
- ap2—attenuation in the second passband

```
d=fdesign.bandstop('fp1,fst1,fst2,fp2,ap1,ast,ap2',0.35,0.40,0.55,...  
0.60,1,50,1)
```

```
d =
```

```
           Response: 'Bandstop'  
Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'  
Description: {7x1 cell}  
NormalizedFrequency: true  
           Fpass1: 0.35  
           Fstop1: 0.4  
           Fstop2: 0.55  
           Fpass2: 0.6  
           Apass1: 1  
           Astop: 50  
           Apass2: 1
```

```
designmethods(d)
```

```
Design Methods for class fdesign.bandstop  
(Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2):
```

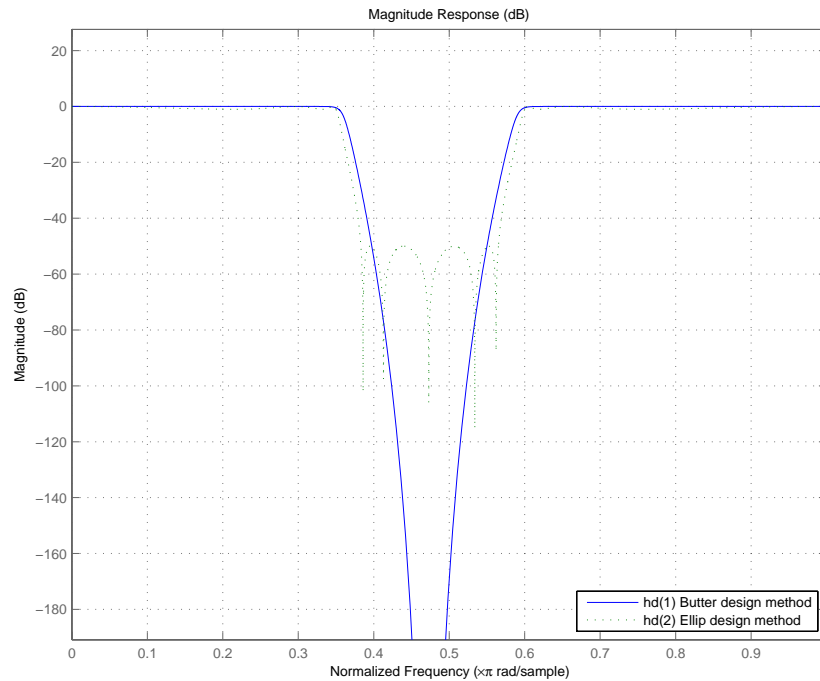
```
butter  
cheby1  
cheby2  
ellip  
equiripple  
kaiserwin
```

```
hd(1)=design(d,'butter','filterstructure','df1sos');  
hd(2)=design(d,'ellip','filterstructure','df1sos');
```

```
hd(1)
ans =
    FilterStructure: 'Direct-Form I, Second-Order Sections'
      Arithmetic: 'double'
      sosMatrix: [13x6 double]
      ScaleValues: [14x1 double]
 PersistentMemory: false

hd(2)
ans =
    FilterStructure: 'Direct-Form I, Second-Order Sections'
      Arithmetic: 'double'
      sosMatrix: [5x6 double]
      ScaleValues: [6x1 double]
 PersistentMemory: false

fvtool(hd)
```



### Designing a Filter With Floating-Point Coefficients

Begin this tutorial by designing a lowpass filter specifications object `d`, specifying the filter values `Fp`, `Fc`, `Ap`, and `Ast`. Then use the `kaiserwin` method to design a direct-form FIR filter from `d`.

```
d=fdesign.lowpass(0.40,0.54,0.05,50)
```

```
d =
```

```
Response: 'Lowpass'  
Specification: 'Fp,Fst,Ap,Ast'  
Description: {4x1 cell}  
NormalizedFrequency: true  
Fpass: 0.4
```



```
Fstop: 0.54
Apass: 0.05
Astop: 50
```

`d` contains the specifications for a lowpass filter.

Design the filter from `d` by applying the `kaiserwin` design method and specify the direct-form FIR filter structure.

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

```
hd =
```

```
    FilterStructure: 'Direct-Form FIR'
           Arithmetic: 'double'
           Numerator: [1x44 double]
 PersistentMemory: false
```

## Converting the Filter to Fixed-Point

`dfilt` objects in the Filter Design Toolbox include a property `Arithmetic` that provides the capability to analyze the filter in double-precision floating-point arithmetic, single-precision floating-point arithmetic, and fixed-point arithmetic.

With the Fixed-Point Toolbox installed, you can set the `Arithmetic` property of the `dfilt` object `hd` to `fixed` to turn quantization on and implement filters that perform fixed-point arithmetic.

The examples in this section discuss fixed-point filters and assume that you have installed the Fixed Point Toolbox.

### Fixed-Point Filter Properties

Setting the `Arithmetic` property to `fixed` adds filter properties to the `dfilt` object. The default display of the filter object properties enhances the readability of the properties by grouping them together in a logical manner.

```
hd.Arithmetic='fixed'
```

```
hd =
```

```
    FilterStructure: 'Direct-Form FIR'
```

```
        Arithmetic: 'fixed'
        Numerator: [1x44 double]
PersistentMemory: false

    CoeffWordLength: 16
    CoeffAutoScale: true
        Signed: true

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'FullPrecision'
```

Notice that only writable properties show in the Command Window listing.

Some filter properties, such as `CoeffAutoScale`, control the display of other properties. `CoeffAutoScale` controls the display of `NumFracLength` and whether you can write (change) the property value for `NumFracLength`.

In contrast to the property display that the filter handle `hd` generates, the `get` function returns the complete collection of properties and property values for the filter, whether you can change the property value or not.

```
get(hd)
    PersistentMemory: 0
    FilterStructure: 'Direct-Form FIR'
        States: [43x1 embedded.fi]
        Numerator: [1x44 double]
        Arithmetic: 'fixed'
    CoeffWordLength: 16
    CoeffAutoScale: 1
        Signed: 1
        RoundMode: 'convergent'
    OverflowMode: 'wrap'
    InputWordLength: 16
    InputFracLength: 15
    NumFracLength: 16
    FilterInternals: 'FullPrecision'
    OutputWordLength: 33
    OutputFracLength: 31
    ProductWordLength: 31
    ProductFracLength: 31
```

```
AccumWordLength: 33
AccumFracLength: 31
```

## Quantizing Filter Coefficients with Automatic Scaling

To determine the number of bits the filter is using to represent the filter coefficients, you look at the value of the `CoeffWordLength` property.

```
hd.CoeffWordLength
ans =

    16
```

To determine how the coefficients are being scaled, look at the `NumFracLength` property.

```
hd.NumFracLength
ans =

    16
```

This tells you that the filter coefficients are 16 bits long (the word length), and the least significant bit (LSB) is weighted by  $2^{-16}$  (the fraction length). The section “Notes About Fraction Length, Word Length, and Precision” on page 2-46 provides more information about interpreting the fraction length in the data format.

16 bits is the default value the filters use for coefficient word lengths. To understand the scaling, look at the `CoeffAutoScale` setting.

```
hd.CoeffAutoScale % Returns a logical true = 1.

ans =

    1
```

When the `CoeffAutoScale` property is true (=1), the filter adjusts the coefficient fraction length to avoid overflow each time you change the coefficient word length. Verify this automatic scaling by changing the number of bits used to quantize the coefficients from 16 bits to 24 bits.

```
hd.CoeffWordLength = 24;
```

```
hd.NumFracLength
ans =
```

```
24
```

The  $2^{-24}$  weight has been computed automatically to represent the coefficients with the best precision possible while using the round-to-nearest value round for the filter property `RoundMode`. “RoundMode” on page 7-85 provides further information about `RoundMode`.

### Scaling Filter Coefficients Manually

Setting the `CoeffAutoScale` property to `false` turns the `NumFracLength` property writable and visible in the display.

```
h1 = copy(hd); % Keep a copy of the original object for...
      % latter comparison
h1.CoeffAutoScale = false
h1 =
```

```
FilterStructure: 'Direct-Form FIR'
Arithmetic: 'fixed'
Numerator: [1x102 double]
PersistentMemory: false
States: [1x1 embedded.fi]
```

```
CoeffWordLength: 24
CoeffAutoScale: false
NumFracLength: 24
Signed: true
```

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

```
OutputWordLength: 16
OutputMode: 'AvoidOverflow'
```

```
ProductMode: 'FullPrecision'
```

```
AccumWordLength: 40
CastBeforeSum: true
```

```
RoundMode: 'convergent'  
OverflowMode: 'wrap'
```

The quantized coefficients are always rounded to the nearest value and saturated when overflow occurs.

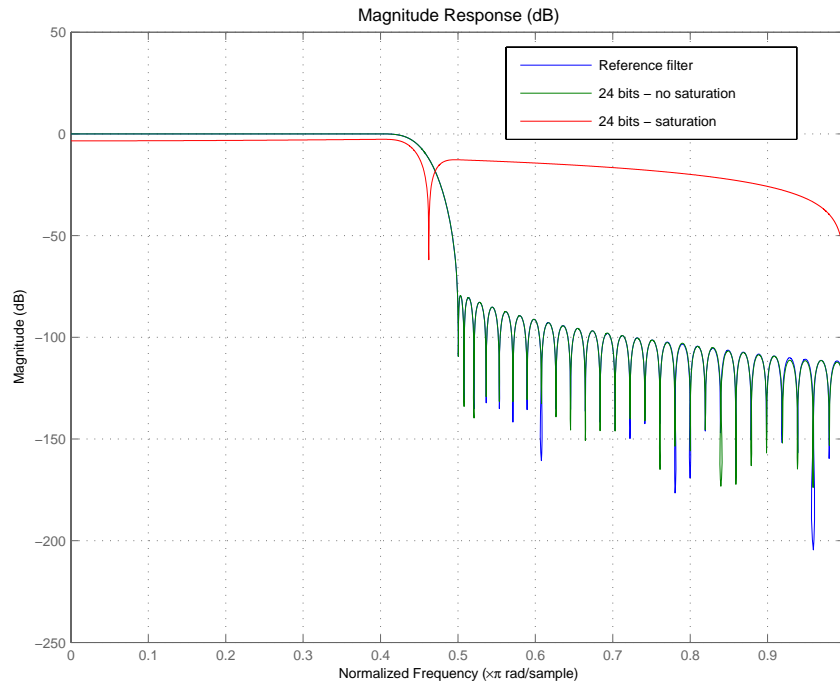
Because the scaling process chose the fraction length to avoid overflow, increasing the fraction length saturates the quantized coefficients, introducing severe distortion in the magnitude response of the filter. Try increasing the numerator fraction length to 25 bits.

```
h1.NumFracLength = 25;
```

This is more clear when you plot the magnitude response to show the effect of saturating the coefficients. Here is the code to display the response.

```
href = reffilter(hd); % Get the reference double-precision...  
                    % floating-point filter.  
hfvt = fvtool(href,hd,h1);  
set(hfvt,'ShowReference','off'); % Reference already displayed.  
legend(hfvt, 'Reference filter', '24 bits - no saturation',...  
       '24 bits - saturation')
```

Saturating the coefficients compromises the filter cutoff performance considerably, shown in the figure.



### Specifying Arithmetic Rules

After you quantize the coefficients, you need to pay attention to the filter internal settings that govern how arithmetic is done inside the filter. For the remainder of this tutorial, you use a classic 16-bit word length filter.

```
hd.CoeffWordLength = 16;
```

One property—`ProductMode`—helps you simulate different filter arithmetic scenarios in the multipliers and adders of the filter.

Setting these properties to specify full precision (set the property values to `FullPrecision`) allows you to determine the minimum resources required to avoid losing precision during filtering.

```

hd.ProductMode = 'FullPrecision'; % (default)
[hd.ProductWordLength hd.ProductFracLength]
ans =

    32    31

[hd.AccumWordLength hd.AccumFracLength]
ans =

    39    31

```

Given an input format of [16 15] and coefficients format of [16 16]—the current settings for `hd`—these responses indicate that you need

- a product register twice the size of the coefficients (or twice the size of the input).
- an accumulator register with seven guard bits to allow for bit growth during the accumulation process.

They also tell you the position of the binary point in those registers— the `AccumFracLength` and `ProductFracLength` property values.

Starting from this scenario that allows your filter to perform most accurately, you can introduce constraints on the product or the accumulator register or both. The `KeepMSB` option for the fraction length properties sets the fraction lengths automatically to avoid overflows while the `KeepLSB` option sets the fraction lengths automatically to avoid underflows.

Finally, the `SpecifyPrecision` option give you full control of the settings. You need to run your filter to see the effect of these settings on the output.

For further discussion about product and accumulator settings, refer to the tutorial “Working with Fixed-Point Direct-Form FIR Filters” on page 2-14.

# Working with Fixed-Point Direct-Form FIR Filters

This chapter ends with a tutorial that illustrates various aspects of working with direct-form FIR filters using fixed-point arithmetic.

As you follow this example, you learn about these topics:

- “Obtaining the Filter” on page 2-14
- “Creating the Direct-Form FIR Fixed-Point Filter” on page 2-15
- “Comparing Quantized Coefficients to Nonquantized Coefficients” on page 2-15
- “Determining the Number of Bits being Used” on page 2-16
- “Determining the Proper Coefficient Word Length” on page 2-17
- “Fixed-Point Filtering” on page 2-18
- “Generating a Baseline Output for Comparison” on page 2-20
- “Computing the Fixed-Point Filter Output” on page 2-21
- “Reducing Filter Output Quantization” on page 2-21
- “The Advantages of Guard Bits” on page 2-22
- “Avoiding Overflow Without Guard Bits” on page 2-26

Each section builds on the contents and filters from preceding sections. Progressing through the tutorial from the start is most effective. Otherwise, code examples that depend on earlier tutorial sections may not work properly.

## Obtaining the Filter

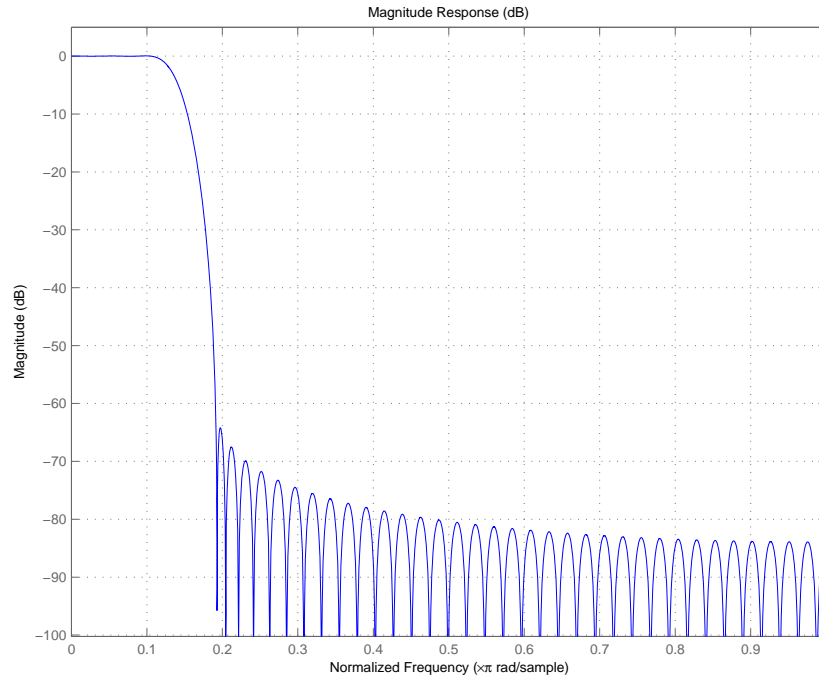
For this tutorial, the FIR filter is not critical. Given the importance of direct-form FIR filters, use the direct-form structure here—it does not even need to have linear phase. This demonstration uses a `firls` design method to obtain the filter.

To display the filter, pass the filter object to the Filter Visualization Tool (FVTool).

```
d = fdesign.lowpass('N,Fp,Fst',80,.11,.19); % Order, and cutoff
                                     % cutoff freqs.
hd = design(d,'firls','Wpass',1,'Wstop',100);
hfvt = fvtool(hd);
```



Here is the magnitude response for `hd` as shown by FVTool.



## Creating the Direct-Form FIR Fixed-Point Filter

To create the fixed-point direct-form FIR filter, change the `Arithmetic` property setting for `hd` to fixed-point arithmetic.

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

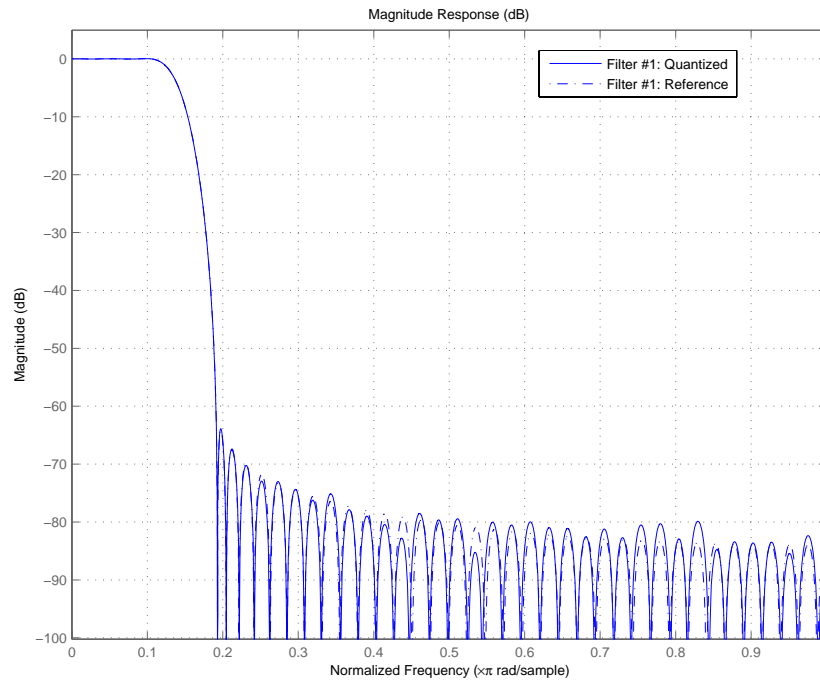
## Comparing Quantized Coefficients to Nonquantized Coefficients

There are several parameters for a fixed-point direct-form FIR filter. To start with, concentrate on the coefficient word length and fraction length (scaling).

Use the Filter Visualization Tool to compare the quantized coefficients filter to the nonquantized (reference) coefficient filter.

```
hfvt=fvtool(hd,'Legend','on');
```

FVTool returns the plot of the magnitude responses for both filters—the quantized filter and the corresponding reference filter.



### Determining the Number of Bits being Used

To determine the number of bits being used in the fixed-point filter `hd`, look at the `CoeffWordLength` property value. Check the `CoeffAutoScale` setting to determine how the filter is scaling the coefficients.

```
get(hd,'CoeffWordLength')  
get(hd,'NumFracLength')
```

```
ans =

    16

ans =

    17
```

These values tell us that `hd` uses 16 bits to represent the coefficients, and the least significant bit (LSB) is weighted by  $2^{-17}$ . 16 bits is the default coefficient word length the filter uses for coefficients, but the  $2^{-17}$  weight has been computed automatically to represent the coefficients with the best possible precision, given the `CoeffWordLength` value.

You control this scaling through the `CoeffAutoScale` property. Set `CoeffAutoScale` to `false` to give yourself manual control of the coefficient scaling. The next command verifies that autoscaling is enabled in filter `hd`.

```
get(hd,'CoeffAutoScale') % Returns a logical true.

ans =

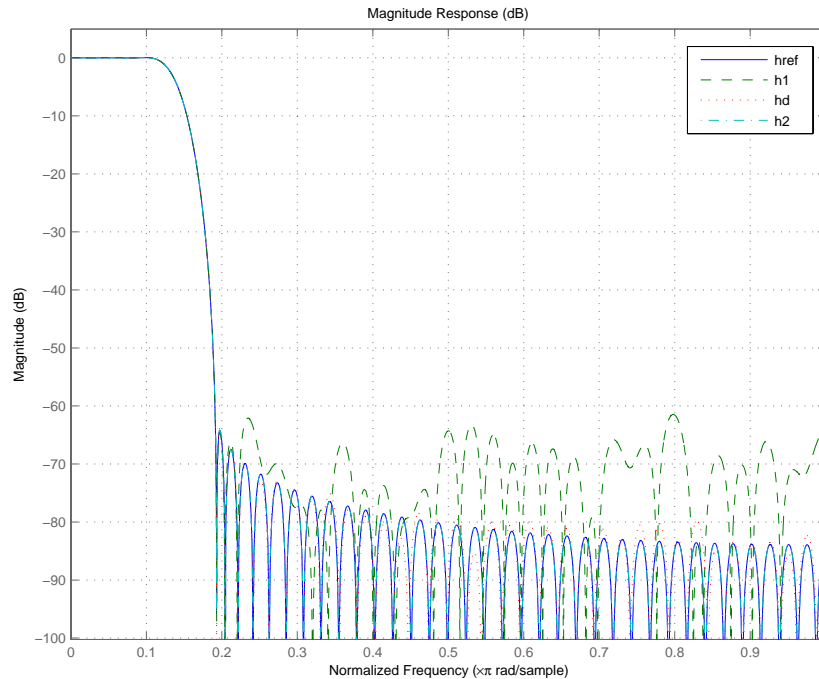
    1
```

## Determining the Proper Coefficient Word Length

Make several copies of the filter to try different word lengths. Allow the coefficient autoscaling process to determine the best precision in each case.

In the figure that follows the code presented here, you see the magnitude responses for the various versions of `hd` (`h1`, `h2`, and the reference filter) so you can compare the effects of changing the coefficient word length.

```
h1 = copy(hd);
set(h1,'CoeffWordLength',12); % Use 12 bits.
h2 = copy(hd);
set(h2,'CoeffWordLength',24); % Use 24 bits.
href = reffilter(hd);
set(hfv, 'Filters', [href, h1, h2], 'ShowReference', 'off');
legend(hfv, 'Reference filter', '12 bits', '16 bits (original...
CoeffWordLength', '24 bits');
```



12 bits (filter h1) is not enough to represent this filter accurately. 16 bits is enough for many applications.

The remaining sections of this tutorial use 16 bits to represent the filter coefficients.

As a rule of thumb, expect an attainable attenuation in the stop band of about 5 dB per bit of coefficient length—16-bit coefficients provide about 80 dB attenuation.

### Fixed-Point Filtering

The main purpose of this tutorial is to evaluate the accuracy of the fixed-point filter when compared to a double-precision floating-point version of the same filter.

Through the sections to come you see that representing the filter coefficients so the magnitude response of the fixed-point filter is close to the double-precision filter does not ensure the performance of the fixed-point filter during filtering.

### Generating Random Test Input Data

To evaluate the accuracy of the fixed-point filter, filter some random data with both filters. Create 1000 data points with range of [-1,1) to generate random, uniformly distributed white-noise data using 16 bits of word length.

```
rand('state',0); % Make results reproducible by initializing the
                % random generator.
x = (rand(1000,1)*2-1); % 1000 Data points in the range [-1,1).
xin = fi(x,true,16,15);
```

Now `xin` is an array of integers with 1000 members, represented as a fixed-point object (a `fi` object).

```
get(xin)

DataTypeMode: 'Fixed-point: binary point scaling'
              DataType: 'Fixed'
              Scaling: 'BinaryPoint'
              Signed: 1
              WordLength: 16
              FractionLength: 15
              FixedExponent: -15
              Slope: 3.0518e-005
SlopeAdjustmentFactor: 1
              Bias: 0
              RoundMode: 'round'
              OverflowMode: 'saturate'
              ProductMode: 'FullPrecision'
              ProductWordLength: 32
              MaxProductWordLength: 128
              ProductFractionLength: 30
              SumMode: 'FullPrecision'
              SumWordLength: 32
              MaxSumWordLength: 128
              SumFractionLength: 30
              CastBeforeSum: 1
```

Your Fixed-Point Toolbox documentation provides more information about `fi` objects.

### Generating a Baseline Output for Comparison

When you evaluate the accuracy of fixed-point filtering, consider three quantities for comparing between the quantized filter and the reference filter:

- 1 The ideal filtered output—this is the goal. Compute it using the reference coefficients and double-precision floating-point arithmetic.
- 2 The best-you-can-hope-for filtered output—this is the best you can hope to achieve. Compute this using the quantized coefficients and double-precision floating-point arithmetic.
- 3 The filtered output you can actually attain with the quantized filter—this is the output you compute using the quantized coefficients and fixed-point arithmetic (compare this to number 2).

Compare what you can actually attain (number 3) to the best you can hope for (number 2). To compute the best-you-can-hope-for, cast the fixed-point filter to double-precision and filter with double-precision floating-point arithmetic, provided by filter `hdouble`.

```
xdouble = double(xin); % Cast the input data to doubles.  
hdouble = double(hd); % Convert hd to double-precision.  
ydouble = filter(hdouble,xdouble);
```

Notice that you had to cast the input data `xin` to double format to use it with the double-precision filter `hdouble`. Double-precision filters require double-precision input values.

For completeness, this is how you compute the ideal output (number 1 in the preceding list). Then you can see how much quantizing just the filter coefficients affects the filter output.

```
yideal = filter(href,xdouble); % Reference filter, double data.  
norm(yideal-ydouble) % Total error.
```

```
ans =
```

```
3.4887e-004
```

```
norm(yideal-ydouble,inf) % Maximum deviation.

ans =

3.7218e-005
```

## Computing the Fixed-Point Filter Output

Now perform the actual fixed-point filtering. Again, the best you can hope to achieve is to have an output identical to ydouble.

```
y = filter(hd,xin);
norm(double(y)-ydouble) % Total error.

ans =

0.0

norm(double(y)-ydouble,inf) % Maximum deviation.

ans =

0.0
```

The error between the filtered results is exactly zero. The accumulator is not introducing any quantization error. The results of products are represented with full precision, the default setting.

From that fact we know that no quantization errors are occurring there either. Finally, the output and accumulator share the same specification for word and fraction length which eliminates errors induced by quantization at the output.

## Reducing Filter Output Quantization

To isolate any other quantization errors that are being introduced in the filter, you can eliminate quantization error at the output completely by setting the output format to have the same specifications as the accumulator. Think of this as being able to look inside the accumulator.

```
set(hd,'FilterInternals','SpecifyPrecision');
set(hd,'AccumWordLength',get(hd,'ProductWordLength'));
```

```
set(hd, 'OutputWordLength', get(hd, 'AccumWordLength'));
y = filter(hd, xin);
norm(double(y)-ydouble) % Total error.
ans =

    8.0623

norm(double(y)-ydouble, inf) % Maximum deviation.
ans =

    0.5000
```

The errors are exactly zero, indicating that the accumulator is not adding further quantization to the output. The arithmetic products (multiplies) are set by default to use full precision, so you know that no errors are occurring in multiplication operations.

Usually it is not possible to have a full 40-bit output of the filter, so you must expect some difference between `y` and `ydouble`. Nevertheless, you have verified that in this filtering case, the difference between the ideal filter and the quantized filter is due to output quantization. This is not always the case—in some cases bits get lost in the accumulator. In fact overflow can occur in the accumulator.

### The Advantages of Guard Bits

If you compare the product word and fraction lengths with the accumulator word and fraction lengths, by looking at the filter properties `ProductWordLength`, `ProductFracLength`, `AccumWordLength`, and `AccumFracLength`, as shown here

```
get(hd, 'ProductWordLength')

ans =

    31

get(hd, 'ProductFracLength')

ans =
```



```
33
```

```
get(hd, 'AccumWordLength')
```

```
ans =
```

```
35
```

```
get(hd, 'AccumFracLength')
```

```
ans =
```

```
33
```

You see that the accumulator has 4 extra bits available (AccumWordLength is 35 bits). Having extra accumulator bits is typical of many fixed-point DSP processors. The extra bits are usually referred to as guard bits. They provide a safety valve for overflows that occur during filtering calculations.

Using `info` provides the same information in one display.

```
info(hd)
Discrete-Time FIR Filter (real)
-----
Filter Structure : Direct-Form FIR
Filter Length   : 81
Stable          : Yes
Linear Phase    : Yes (Type 1)
Arithmetic      : fixed
Numerator       : s16,18 -> [-1.250000e-001 1.250000e-001)
Input           : s16,15 -> [-1 1)
Filter Internals : Full Precision
  Output        : s35,33 -> [-2 2) (auto determined)
  Product       : s31,33 -> [-1.250000e-001 1.250000e-001) (auto determined)
  Accumulator   : s35,33 -> [-2 2) (auto determined)
  Round Mode    : No rounding
  Overflow Mode : No overflow

Measurements
Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.064538
3-dB Point         : 0.10001
6-dB Point         : 0.11
Stopband Edge      : 0.15183
Passband Ripple    : 0.01 dB
Stopband Atten.    : 60 dB
Transition Width    : 0.087288
```

The easiest way of appreciating the value of guard bits is to remove them and see what happens (adjust the output settings accordingly).

```
set(hd,'FilterInternals','SpecifyPrecision');
set(hd,'AccumWordLength',get(hd,'ProductWordLength'));
set(hd,'OutputWordLength',get(hd,'AccumWordLength'));
```

```
hd
```

```
hd =
```

```
    FilterStructure: 'Direct-Form FIR'
           Arithmetic: 'fixed'
           Numerator: [1x81 double]
 PersistentMemory: false

    CoeffWordLength: 16
           CoeffAutoScale: true
           Signed: true

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'SpecifyPrecision'

    OutputWordLength: 31
    OutputFracLength: 32

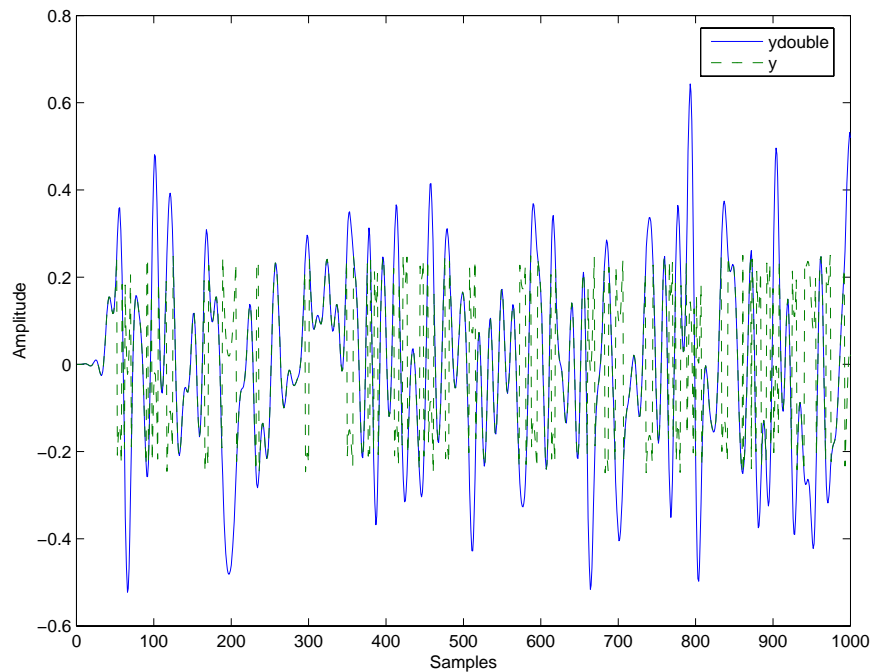
    ProductWordLength: 31
    ProductFracLength: 32

    AccumWordLength: 31
    AccumFracLength: 32

           RoundMode: 'convergent'
           OverflowMode: 'wrap'
```

Now the accumulator word length matches the product word length of 31 bits, and the output word length matches the accumulator word length, 32 bits. Now use `hd` to filter some data, and plot the results.

```
y = filter(hd,xin);  
norm(double(y)-ydouble)    % Total error.  
  
ans =  
  
    3.4641  
  
norm(double(y)-ydouble,inf) % Maximum deviation.  
ans =  
  
    1  
  
plot([ydouble,double(y)])  
xlabel('Samples'); ylabel('Amplitude')  
legend('ydouble','y')
```



The total error is large because overflow occurred during filtering. The representable range for the accumulator and output formats [32 32] is -0.5 to 0.5.

In the plot, one of the values around sample 800 is larger than 0.5, indicating an overflow. Recall that you set the output settings equal to the accumulator settings. You removed the guard bits by setting the accumulator word length to 32 bits. So the overflow is occurring in the accumulator itself.

### Avoiding Overflow Without Guard Bits

It is possible not to have overflow even when guard bits are not available in the accumulator.

```
set(hd, 'OutputFracLength', get(hd, 'AccumFracLength'));
y = filter(hd, xin);
norm(double(y)-ydouble) % Total error.
norm(double(y)-ydouble, inf) % Maximum deviation.
```

```
ans =
```

```
2.4442e-006
```

```
ans =
```

```
2.5332e-007
```

If the filter uses 16 bits for the output word length and sets the output mode to maintain the best precision for this word length, the resulting error is much larger—almost two orders of magnitude.

```
set(hd, 'OutputWordLength', 16);
set(hd, 'OutputMode', 'BestPrecision');
y = filter(hd, xin);
norm(double(y)-ydouble) % Total error.
```

```
ans =
```

```
2.7627e-004
```

```
norm(double(y)-ydouble, inf) % Maximum deviation.
```

```
ans =

    1.5400e-005
```

From the earlier plots of  $y$  and  $y_{\text{double}}$ , you might have realized that one extra bit was all that would have been required to avoid overflow in those examples.

You can improve the results slightly with this one bit change, but remember that this is specific to the filter coefficients and input signal in this tutorial.

Reducing the accumulator fraction length from 32 bits to 31 bits provides one more bit in the integer part of the accumulator word and reduces the filtering error.

```
set(hd, 'AccumFracLength', 31);
y = filter(hd, xin);
norm(double(y)-ydouble)      % Total error.
norm(double(y)-ydouble, inf) % Maximum deviation.
ans =

    2.7623e-004
```

```
ans =

    1.5251e-005
```

The errors are the same as when the filter used 39 bits for the accumulator and  $2^{-32}$  to scale the least-significant bit. This indicates that the errors in filtering are due to quantization effects between the accumulator and the output.

### Constructing Fixed-Point Filters

You construct filters by

- Using an `fdesign.response` object combined with a filter design method such as `butter`
- Using the appropriate filter constructor function `dfilt.structure`, where `structure` is the filter topology to implement
- Using FDATool design features
- Copying an existing filter

All filter characteristics are stored as properties that you can set or retrieve. These filter characteristics include

- Filter structure
- Reference filter coefficients
- Filter topology (single section or cascaded second-order sections)
- Fixed-point filter data format parameters such as
  - Quantization parameters (word lengths, fraction lengths, and precisions).
  - Data type (signed or unsigned fixed-point, double-precision or single-precision floating-point, and signed or unsigned integers)
  - Rounding method used in quantization
  - Overflow method used in quantization
- Scaling factors for each section of a second-order section filter

You can specify quantized filter properties by creating a quantized filter with default property values and then changing some or all of these property values later.

### Defining Quantized and Fixed-Point Filters

With the `dfilt` objects in this toolbox you can create quantized and fixed-point filter objects that you use to filter signals or data.

In this user's guide, we distinguish between *fixed-point* and *quantized* filters only very rarely—mostly we use the terms interchangeably. There is a difference between them that is worth recalling when you work with the filter objects in this toolbox.

Quantized means using limited precision arithmetic, either fixed-point or floating-point. Underlying all the filters in this toolbox, including the floating-point filters, is quantized arithmetic.

Roughly explained, quantizing is the act of reducing the precision with which you represent numeric quantities.

With this in mind, we approximate ideal arithmetic (arithmetic with infinite precision) using double-precision, floating-point arithmetic and we refer to floating-point filters as nonquantized, or reference, filters.

Fixed-point arithmetic is a subset of quantized arithmetic, and fixed-point filters are thus a subset of quantized filters. In fixed-point arithmetic, the word length and fraction length you use limit the precision of your results. Arithmetic operations occur without moving the binary, or radix, point. Hence the name fixed-point or fixed binary-point arithmetic.

In summary, quantized filters use limited precision arithmetic and data representations. Fixed-point filters use limited precision representations and fixed-point arithmetic where the binary point location does not change.

## Constructors for Fixed-Point Filters

The most direct way to create a fixed-point arithmetic filter (a `fixedfilt` object) is to create one with the default properties. Fixed-point filter object construction requires these steps:

- Create a default double-precision lowpass filter `hd` by entering something like this command pair. First create a filter specifications object, and then design the filter.

```
d = fdesign.lowpass;  
hd = design(d,'equiripple');
```

- Change the Arithmetic property setting for filter object `hd` to `fixed`.

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

MATLAB displays a listing of all of the properties of the filter `hd` you created, along with the associated property values. All properties are set to defaults when you construct a fixed-point filter this way.

## Constructing a Quantized Filter from a Filter Specification Object

You construct quantized filters by constructing default filters or filters with specified filter coefficients. Begin with a set of nonquantized filter coefficients to implement in a quantized filter.

For this example, start with a filter specification object that defines the response of the filter to design. This code specifies the filter order, cutoff frequency, and attenuations for the filter design.

```
d = fdesign.lowpass('n,fp,fst,ap',3,0.5,0.6,3);
```

To implement `d` as a quantized filter, use one of the design methods in the toolbox to design the filter and then change the value of the Arithmetic property to `fixed`:

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

```
hd =
```

```
    FilterStructure: 'Direct-Form II, Second-Order Sections'  
      Arithmetic: 'double'  
      sosMatrix: [2x6 double]  
      ScaleValues: [0.187365400536859;0.860421232522936;1]  
 PersistentMemory: false
```

```
set(hd,'arithmetic','fixed'); % Convert to quantized filter.
```

Because filters designed with second-order section topologies are more robust against quantization errors than those composed of higher-order transfer functions, `ellip` constructs the `dfilt` object as an SOS filter.

## Constructing a Fixed-Point Filter in Second-Order Sections

By default, many of the filter design functions in the toolbox return filters that use second-order sections. In most cases, this is a desirable feature when you are using fixed-point arithmetic because SOS filters tend to resist errors caused by quantization.



```

hs = fdesign.bandpass(.3, .4, .6, .7, 80, .5, 60); % Specify the
        passband edges and attenuations.
designmethods(hs) % Find an appropriate design method.

Design Methods for class fdesign.bandpass:

butter
cheby1
cheby2
ellip
equiripple
kaiserwin

hd=design(hs,'butter') % Design the filter.

hd =

        FilterStructure: 'Direct-Form II, Second-Order Sections'
        Arithmetic: 'double'
        sosMatrix: [13x6 double]
        ScaleValues: [14x1 double]
        PersistentMemory: false

```

## Copying Filters to Inherit Properties

When you have a quantized filter `hd` with the property values set the way you want them, you can create a new quantized filter `hd2` with the same property values as `hd` by entering

```
hd2 = copy(hd)
```

This function is convenient to use when you are changing a small number of properties on a set of filters.

For example, create a 16-bit precision filter `hd2` from an FIR reference filter with

```

hd = design((fdesign.lowpass('N,fc,ap,ast',80,0.5,.05,50)),...
'equiripple') % Reference filter with double-precision coeffs.;
hd2 = hd;

```

hd2 inherits the property values for hd, but is an independent object that you can change without affecting hd.

### Fixed-Point Arithmetic Filter Structures

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 object constructors. In some instances, function `convert` allows you to change the structure of an existing filter object.

You specify the filter structure by selecting the appropriate `dfilt.structure` method to construct your filter. For information about setting properties for fixed-point filter objects, refer to the reference information for `dfilt` in your Signal Processing Toolbox documentation and in this user's guide, and `get` and `set` in your MATLAB documentation.

The figures included in the reference page for each filter structure, such as `dfilt.dfasymfir`, act as aids to help you determine how to enter your filter coefficients for each filter structure and how the filter performs quantizations in the filter signal flow. Each reference page also contains an example for constructing a filter of the given structure.

Filter Constructor Name	FilterStructure Property String and Filter Type
<code>dfilt.dfasymfir</code>	Antisymmetric finite impulse response (FIR)
<code>dfilt.df1</code>	Direct form I
<code>dfilt.df1sos</code>	Direct form I filter implemented using second-order sections
<code>dfilt.df1t</code>	Direct form I transposed
<code>dfilt.df2</code>	Direct form II
<code>dfilt.df2sos</code>	Direct form II filter implemented using second order sections

<b>Filter Constructor Name</b>	<b>FilterStructure Property String and Filter Type</b>
<code>dfilt.df2t</code>	Direct form II transposed.
<code>dfilt.dffir</code>	Direct form FIR
<code>dfilt.dffirt</code>	Direct form FIR transposed
<code>dfilt.latticear</code>	Lattice autoregressive (AR)
<code>dfilt.latticemamin</code>	Lattice moving average (MA) minimum phase
<code>dfilt.latticemamax</code>	Lattice moving average (MA) maximum phase
<code>dfilt.latticearma</code>	Lattice ARMA
<code>dfilt.dfsymfir</code>	Symmetric FIR. Even and odd forms
<code>dfilt.scalar</code>	Scalar

### Fixed-Point Arithmetic Filter Structure Diagrams

To help you understand where quantizations occur in filter structures like those provided in the toolbox, the next figure presents the structure for a direct-form 2 filter, including the quantizations that the quantized filter incorporates. You see that one or more quantizations accompany each filter element, such as a delay, coefficient, or summation element. The input to or output from each element reflects the result of applying the associated quantization.

Wherever a particular filter element appears in a filter structure, recall the quantization that accompanies the element. For example, a product quantization, either numerator or denominator, follows every coefficient element. A sum quantization, also either numerator or denominator, follows each sum element.

In this figure, you see the structure for a direct-form 2 IIR filter, with the arithmetic property value set to 'fixed'.



data at the labeled location in the filter. `InputFormat` refers to the `InputWordLength` and `InputFracLength` filter properties and `OutputFormat` refers to the `OutputWordLength` and `OutputFracLength` filter properties.

Property names like `CoeffWordLength` and `DenFracLength` define the properties that control filter operations with coefficients or denominator coefficients at that point in the structure and are properties of the filter.

# Data Type Handling in Discrete-Time Filters

In this section you learn how discrete-time filters (`dfilt` objects) handle different data types in significant filtering areas:

- Different data types as input data to your filter
- Different data types to represent your filter coefficients
- Different data types representing the states of your filter
- Reference filter coefficients

How these varied filter areas respond is driven primarily by the value you set for the `Arithmetic` property of the filter object. The next sections cover each of the areas noted above, discussing how each responds when you set the value for the `Arithmetic` property.

Property `Arithmetic` accepts one of three valid entries:

- `Double`
- `Single`
- `Fixed`

Each option affects how the filter handles the states, coefficients, input and output data, and filter arithmetic. And what you use as input to the filter object.

## Filter Input Signals, Coefficients, and States

Filter object properties and their values directly affect how and in what form your filter works with input data, the filter coefficients, and the states of the filter.

In many cases, fixed-point filters use fixed-point objects to handle fixed-point values such as coefficients, input, or filter states. The Fixed-Point Toolbox documentation provides details about the fixed-point, or `fi`, object that `dfilt` objects use.

## Input Data and the Arithmetic Property Setting

The `Arithmetic` property setting controls the handling and quantization of input to the filter. All arithmetic property settings—`double`, `single`, `fixed`—support the same input data types:

- Double-precision floating-point
- Single-precision floating-point
- `int*`
- `uint*`
- `fi` objects

Each Arithmetic property value refines how the filter accepts input data. When you specify one of the following values for `Arithmetic`, this is what happens in the filter.

- `double`

The filter casts the input data to double-precision format. The filter states and output are double data type as well. This is the default value for the filter `Arithmetic` property. The resulting filter is considered double-precision and floating-point.

- `single`

The filter casts the input data to single-precision format. Both the filter states and the output from the filter are in single data type. This is a quantized filter that uses single-precision floating-point data format.

- `fixed`

The filter casts the input data to fixed-point (`fi`) objects to use fixed-point formats defined by the filter properties [`InputWordLength` `InputFracLength`], adds properties to the filter object for configuring the filter, and switches the filter to using fixed-point arithmetic. The added properties let you determine the data formats (the word length and fraction length) the filter uses for all filter operations and data.

### Filter Coefficients and the Arithmetic Property Setting

Changing the arithmetic mode controls the format the filter uses to represent coefficients. Discrete-time filters accept coefficients in any of the following formats:

- double-precision floating-point
- single-precision floating-point
- `int*`

- `uint*`
- `fi` objects

Depending on the setting for `Arithmetic`, whether `double`, `single`, or `fixed`, the filter handles the coefficients in the following manner:

- `double`  
The filter casts the coefficients to `double` data type. Reference coefficients for the filter are stored in the data type in which you provide them. In this case, the quantized and reference coefficients for the filter are identical.
- `single`  
The filter casts the coefficients to `singles`. `single` data type coefficients are unchanged. Reference coefficients for the filter are stored in the data type that you use to provide them.
- `fixed`  
The filter casts the coefficients to `fixed-point (fi)` objects, using the `[InputWordLength InputFracLength]` filter properties to format the coefficients. The resulting `fixed-point` filter stores the reference filter coefficients in the data type that you supply. When you use `reffilter`, you get back a reference filter whose coefficients are `double-precision` approximations to the actual reference coefficients.

### Arithmetic Property Setting and Filter States

How the filter stores and operates on filter states depends on the setting of the `Arithmetic` property. You can provide the states in any of the following formats:

- `double-precision floating-point`
- `single-precision floating-point`
- `int*`
- `uint*`
- `fixed-point (fi)` objects

When you set the `Arithmetic` property value you change how the filter responds to the state values.

- `double`  
The filter casts the states to `double-precision` data type.



- `single`  
The filter casts the filter states to single-precision data type.
- `fixed`  
The filter casts the states to fixed-point objects, using the `[InputWordLength InputFracLength]` filter properties to format the states as
  - Fixed-point objects
  - Double
 Other data types return an error in MATLAB.

When you set `PersistentMemory` to `true`, the word length and fraction length settings for the filter states must be the same as the filter input word length and fraction length. If these settings do not match, you receive an error.

Note that the filter does not store reference values for the states.

Disabling the autoscaling filter properties such as `CoeffAutoScale`, `InputAutoScale`, and `OutputAutoScale` results in all the additional fraction length properties becoming available in the filter. To make disabling the automatic scaling for a filter easier, use `specifyall`. When you use

```
specifyall(hd)
```

all of the automatic control properties of `hd` are set to `SpecifyPrecision`:

- `ProductMode`
- `OutputMode`

`specifyall` also disables the automatic scaling provided by

- `CoeffAutoScale`
- All other `*AutoScale` properties for the filter, since this varies from structure to structure

With autoscaling disabled you have access to the fraction length properties for coefficients, the accumulator, products, and output values, which lets you set the precision yourself.

`specifyall` also helps you return your filter to the default automatic modes. Use the syntax

```
specifyall(hd, false)
```

to reset filter `hd` to the default automatic mode settings.

You may want more information about filter states after you read this review. Refer to `filtstates` in your Signal Processing Toolbox documentation for detail about filter states and the `filtstates` object the filters use.

### Reference Filter Coefficients for Fixed-Point Filters

Quantized or fixed-point filters in the toolbox have both quantized coefficients (or fixed-point coefficients) that result from changing the `Arithmetic` property to `fixed` or `single`, and reference coefficients. You can access both sets from the command line.

How the toolbox stores the reference coefficients for a filter depends on the data type you use to specify the coefficients—reference filter coefficients are stored in the data type you specified when you constructed the filter. Retaining the specified data type prevents the memory for storing the coefficients from growing unnecessarily.

When you view the fixed-point filter coefficients, you see double-precision approximations to the actual fixed-point or quantized coefficients used for filtering. In many cases, the approximation is exact, including when your filter uses single or double arithmetic.

When the `Arithmetic` property value is `fixed`, the approximation is exact if the software can store the fixed-point values exactly as a double data type value. Otherwise, you see the double data type approximation of the value.

Returning the double-precision approximations enables the software to represent the leading denominator coefficient of an IIR filter exactly as a 1, even if you are working in a fractional mode, such as Q15.

You use the function `reffilter` to return a filter that has the reference coefficients that accompany any fixed-point filter. For example, create a fixed-point direct form filter `hd` with

```
d=fdesign.lowpass('n,fc,ap,ast',5,0.45,0.1,50); % Order, cutoff,  
                                     % and filter attenuations in dB.  
hd = design(d);  
hd.arithmetic='fixed';
```

which has fixed point coefficients

```
hd.numerator  
  
ans =
```

```
-0.0122    0.1192    0.3959    0.3959    0.1192    -0.0122
```

Now change the word length the filter uses to represent the numerator coefficients.

```
hd.coeffautoScale=false

hd =

    FilterStructure: 'Direct-Form FIR'
           Arithmetic: 'fixed'
           Numerator: [1x6 double]
 PersistentMemory: false

    CoeffWordLength: 16
    CoeffAutoScale: false
    NumFracLength: 16
    Signed: true

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'FullPrecision'
set(hd, 'coeffWordLength',14');
hd.numerator

ans =
```

```
-0.0122    0.1192    0.1250    0.1250    0.1192    -0.0122
```

Using `reffilter` returns a filter object with reference coefficients, as follows:

```
hdref=reffilter(hd)

hdref =

    FilterStructure: 'Direct-Form FIR'
           Arithmetic: 'double'
           Numerator: [1x6 double]
 PersistentMemory: false

hdref.Numerator

ans =
```

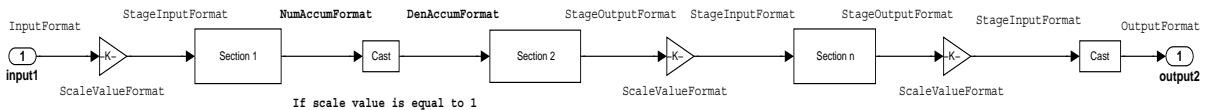
-0.0122    0.1192    0.3959    0.3959    0.1192    -0.0122

`hdref` has the original filter coefficients and is a double-precision filter. The reference filter coefficients match the original set of fixed-point coefficients for `hd`, but not the coefficients as represented by 14 bits.

### Fixed-Point Filters and Second-Order Sections

Listed within the `dfilt` methods for creating quantized filters you find methods that return second-order section (SOS) versions of the direct-form IIR filters—`df1sos`, `df1tsos`, `df2sos`, and `df2tsos`.

The following figure shows how the second-order sections combine to form a filter, in this case a direct-form II SOS filter. This diagram (or a similar one) appears with each SOS filter structure as well.



Combining this figure with the structures and signal flows for each SOS filter helps you work out the details about quantization in the SOS filter.

Using second-order sections is not the same as cascading the filters, as the `dfilt.cascade` or `dfilt.parallel` methods in the Signal Processing Toolbox allow you to do with any `dfilt` objects.

### The `CastBeforeSum` Filter Property

Setting the `CastBeforeSum` property determines how the filter handles the input values to sum operations in the filter.

After you set the 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 specific filter structure.

`CastBeforeSum` specifies whether to cast selected inputs to summations in the filter to the summation output format before performing the addition.

### Setting CastBeforeSum to True

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 errors to your filter results.

### Setting CastBeforeSum to False

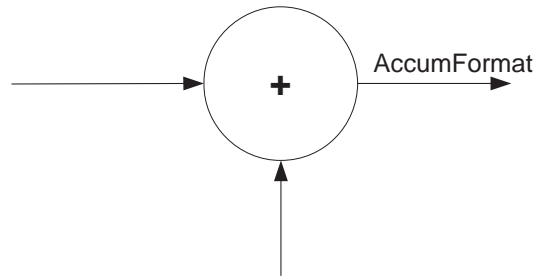
Specifying `CastBeforeSum` to be `false` prevents the addends from being cast to the output format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use.

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 `CastBeforeSum` refers to depends on the structure.

Property Value	Description
<code>false</code>	Configures filter summation operations to retain the addends in the format carried from the previous operation.
<code>true</code>	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

### Diagrams of CastBeforeSum Settings

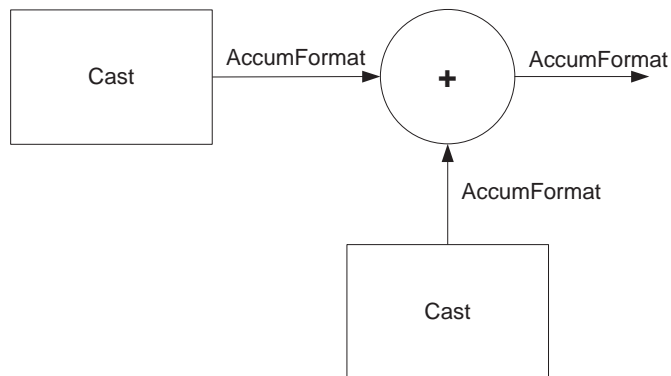
When `CastBeforeSum` is `false`, sum elements in filter signal flow diagrams look like this:



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 cast 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`.

## Introduction to Fixed-Point Arithmetic

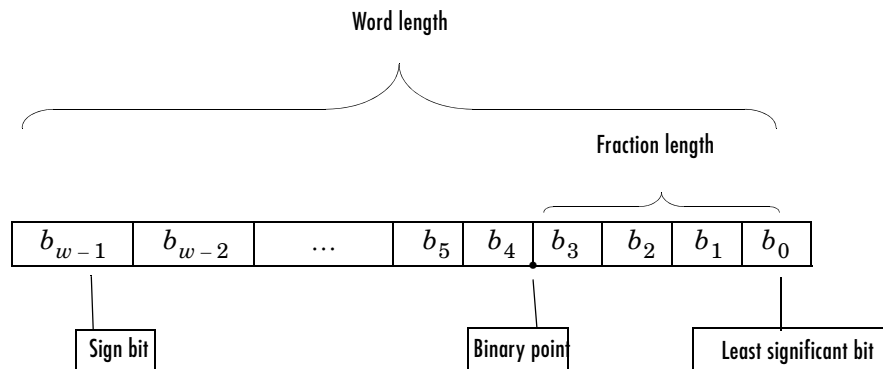
You specify how numbers are quantized using fixed-point arithmetic in this toolbox with two quantities:

- Word length in bits
- Fraction length in bits

This toolbox does bit-true fixed-point arithmetic for all word lengths. It properly handles overflows and the results are bit-true when the numbers are scaled properly. For example, (small numbers + small numbers) works correctly and (large numbers + large numbers) are right as well.

Fraction lengths used to represent numeric values can be any positive or negative integer, including integers larger than the associated word length for the value.

A general representation for a signed two's-complement binary fixed-point number is



where

- $b_i$  are the binary digits (bits, 0s or 1s).
- The word length in bits is given by  $w$ .
- The most significant bit (MSB) is the leftmost bit. It is represented by the location  $b_{w-1}$ . In Filter Design Toolbox, this value represents the sign bit; a 1 indicates the number is negative, and a 0 indicates it is nonnegative.

- The least significant bit (LSB) is the rightmost bit, represented by the location  $b_0$ .
- The binary point is shown four places to the left of the LSB for this example.
- The fraction length  $f$  is the distance from the LSB to the binary point.

### Binary Point Interpretation

Where you place the binary point determines how fixed-point numbers are interpreted in two's complement arithmetic. For example, the 5-bit binary number

- 10110 represents the integer  $-2^4+2^2+2 = -10$ .
- 10.110 represents  $-2+2^{-1}+2^{-2} = -1.25$ .
- 1.0110 represents  $-2^{-0}+2^{-2}+2^{-3} = -0.625$ .

### 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, the format notation is [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.

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.

For related information about scaling filters, refer to “Quantizing Filter Coefficients with Automatic Scaling” on page 2-9, which provides a discussion of how the toolbox scales filters automatically and how you can scale them yourself.

## Precision

Here is how precision works.

When all of the bits of a binary string are 0 except for the LSB (which is therefore equal to 1), 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 that the precision does not depend on the word length.

Look at another example. When the word length is set to 8 bits, the decimal value of 12 is represented in binary by 00001100. That the decimal value 12 is equivalent to binary 00001100 indicates that the data format [8 0] is being used—the word length is 8 bits and fraction length 0 bits, and the 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 3. The resulting precision after removing the right-most 3 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].

In this format the precision is 8. The [5 -3] format cannot represent numbers that are between multiples of 8.

In MATLAB, with the Fixed-Point Toolbox 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 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. b, second element in the expression, 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 (they have the same word length), 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 is that by setting the format for the output from the sum or product operation in `dfilt` objects and filtering, you control which bits are carried from the filter sum and product operations to the filter output.

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 are taking the lower 16 bits from the result. If you use [16 -16], you are taking 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].

## Precision and Dynamic Range

A fixed-point quantization scheme determines the dynamic range of the numbers that can be applied to it. Numbers outside this range are always mapped to fixed-point numbers within the range when you quantize them.

Precision is the distance between successive numbers occurring within the dynamic range in a fixed-point representation. The dynamic range and precision depend on the word length and the fraction length.

For a signed fixed-point number with word length  $w$  and fraction length  $f$ , the range is from  $-2^{w-f-1}$  to  $2^{w-f-1}-2^{-f}$ .

For an unsigned fixed-point number with word length  $w$  and fraction length  $f$ , the range is from 0 to  $2^{w-f}-2^{-f}$ .

In both cases the precision is  $2^{-f}$ .

### Overflows and Scaling

When you quantize a number outside of the dynamic range for your specified format, *overflows* occur. Overflows occur more frequently with fixed-point quantization than with floating-point, because the dynamic range of fixed-point numbers is much less than that of floating-point numbers with equivalent word lengths.

Overflows can occur when you create a fixed-point quantized filter from an arbitrary floating-point design. You can normalize your fixed-point filter coefficients and introduce a corresponding scaling factor for filtering to avoid overflows in the coefficients.

In this toolbox you can specify how you want overflows to be handled:

- Saturate on the overflow
- Wrap on the overflow

For more about scaling and filters with fraction lengths that exceed the word length, refer to “Quantizing Filter Coefficients with Automatic Scaling” on page 2-9, which provides a discussion of how the toolbox scales filters automatically and how you can scale them yourself.

# Designing Multirate Filters

---

Introducing Multirate Filters (p. 3-2)	Introduces multirate filters and discusses uses, specifications, and definitions
Getting Started—Designing Multirate Filters (p. 3-4)	Provides a tutorial to show you how to design multirate filters
FIR Decimation—Filtering with FIR Decimators (p. 3-18)	Designs an FIR decimator and uses it to filter a signal
CIC Filter Example—Using CIC Decimation Filters (p. 3-24)	Develops, explains, and uses cascaded integrator-comb decimators
Analyzing Multirate and Multistage Filters (p. 3-36)	Provides information about using the toolbox analytical capabilities to analyze multirate filters
Audio Example—Audio Sample Rate Conversion (p. 3-47)	Demonstrates sample rate decimations of a 48 kHz signal to 32 kHz (broadcast audio rate) and 44.1 kHz (CD audio rate)

# Introducing Multirate Filters

Over the last few years, developments in multirate filter design and implementation have brought rapid growth in applying multirate filtering to signals in digital signal processing. Improved processors and development tools allow system designers to use multirate filters in a broad range of application areas, such as:

- POTS audio encryption—encrypts voice sent over plain old telephone systems (POTS).
- Digital audio—sound handled in digital rather than analog form. Encompasses various signal compression schemes, analog-to-digital conversion techniques, and the opposite conversions, signal reproduction, and audio improvements.
- Subband speech and image coding—uses the techniques of separating a signal or image into subbands that each containing only a portion of the original signal. Then processing the subbands through filters before reconstructing the original signal from the processed subbands.

Polyphase filters—filters that separate an input signal into constituent bands that are easier to process, and can then be either recombined or used after processing—represent one way to accomplish signal separation. Filter performance depends on the phase differences between the input signals.

- Transmultiplexer design—uses filters to convert time division multiplexing (TDM) signals to frequency division multiplexing (FDM) format, and the reverse. FDM combines numerous signals for transmission on a single communications line or channel. Each signal is assigned a different frequency (subchannel) within the main channel. TDM puts multiple data streams in a single signal by separating the signal into many segments, each having a very short duration. Based on the timing of the signals, each individual data stream is reassembled at the receiving end.

These represent a few of the growing number of areas in which systems designers use multirate filters.

Listed below are the examples in this chapter that introduce multirate filters. Each example includes a tutorial that uses toolbox features to demonstrate how you work with multirate filters:

- “Getting Started—Designing Multirate Filters” on page 3-4

- “Audio Example—Audio Sample Rate Conversion” on page 3-47
- “CIC Filter Example—Using CIC Decimation Filters” on page 3-24
- “Audio Example—Audio Sample Rate Conversion” on page 3-47

## Getting Started—Designing Multirate Filters

This section demonstrates how to use the multirate filter (`mfilt`) objects available in the toolbox. By following these procedures you get introduced to multirate filter development. This tutorial covers the following tasks:

- “Creating Multirate Filters” on page 3-4
- “Getting and Setting Filter Coefficients” on page 3-6
- “Analyzing Multirate and Multistage Filters” on page 3-8
- “Specifying Initial Conditions to the Filter” on page 3-11
- “Streaming Data to the Filter” on page 3-12
- “Filtering Multichannel Signals” on page 3-13
- “Generating Simulink Blocks” on page 3-15
- “Getting Help About Multirate Filters” on page 3-15

### Creating Multirate Filters

To develop a multirate filter (`mfilt`) object, you select the filter structure to be used by selecting the constructor function, such as `mfilt.firdecim` or `mfilt.firinterp`.

Entering `helpwin mfilt` at the prompt gives you a list of all supported structures and constructor functions.

Most multirate filter constructors take the coefficients of the filter as an optional final right-hand input argument. If you do not specify the coefficients, the toolbox functions design a default filter according to the interpolation or decimation factor(s) you provide as input for `L` or `M` in the calling syntax, or both in the case of fractional rate changer filters.

Here is an example that creates an interpolating filter with order of three interpolation and a decimating filter that decimates by two.

```
l = 3; % Interpolation factor
m = 2; % Decimation factor
hm1 = mfilt.firinterp(l);
hm2 = mfilt.firdecim(m);
```

Both filter constructors return direct-form FIR polyphase Nyquist filters by default. Nyquist filters tend to be well-suited for decimation and interpolation work, because the form is computationally efficient due to the zero-valued



coefficients inherent in the design. Used as interpolators, Nyquist filters preserve the nonzero samples of the upsampled output of the interpolator.

```

hm1

hm1 =

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

hm2

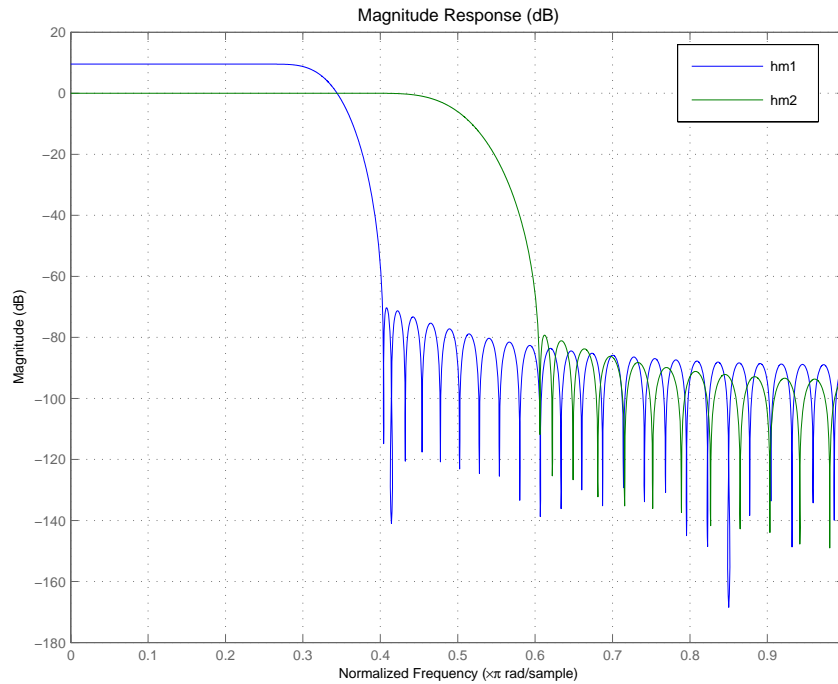
hm2 =

    FilterStructure: 'Direct-Form Transposed FIR Polyphase
Decimator'
      Arithmetic: 'double'
      Numerator: [1x48 double]
DecimationFactor: 2
  PersistentMemory: false

```

Filter hm1 is a direct-form FIR polyphase interpolator filter with the cutoff frequency of  $\pi/1$  and gain of 1. hm2 is a direct-form transposed FIR polyphase decimator with a cutoff frequency of  $\pi/m$  and a gain of 1.

For confirmation, here is the frequency response displayed by the Filter Visualization Tool (FVTool).



hm1 and hm2 are filters and `mfilt` objects. As objects, they work with a range of functions (methods) such as `filter`, `freqz`, and `tf`, or `display`.

### Getting and Setting Filter Coefficients

To access and manipulate the coefficients of a filter as a regular MATLAB vector, you use the common object functions `set` and `get` or dot notation. You can always get the coefficients from the `mfilt` object (`filter`). To modify the coefficients of an existing `mfilt` object, you set new ones. Direct-form FIR structures like those of hm1 and hm2 have numerator coefficients only—also known as the filter weights.

Here are the filter coefficients for hm2.

```
b = get(hm2,'numerator') % Could use command hm2.numerator as well. Assign the
                        % coefficients to vector b.
b =
Columns 1 through 8
    0    -0.0001         0    0.0004         0   -0.0010         0    0.0022
Columns 9 through 16
    0   -0.0043         0    0.0077         0   -0.0128         0    0.0207
Columns 17 through 24
    0   -0.0331         0    0.0542         0   -0.1002         0    0.3163
Columns 25 through 32
    0.5000    0.3163         0   -0.1002         0    0.0542         0   -0.0331
Columns 33 through 40
    0    0.0207         0   -0.0128         0    0.0077         0   -0.0043
Columns 41 through 48
```

After you get the coefficients, create a new Nyquist FIR filter bmod and set the coefficients of hm2 to the coefficients from bmod.

```
bmod = firnyquist(8,m,kaiser(9,0.1102*(80-8.71)));
set(hm2,'Numerator',bmod); % Set the modified coefficients.
hm2.numerator
```

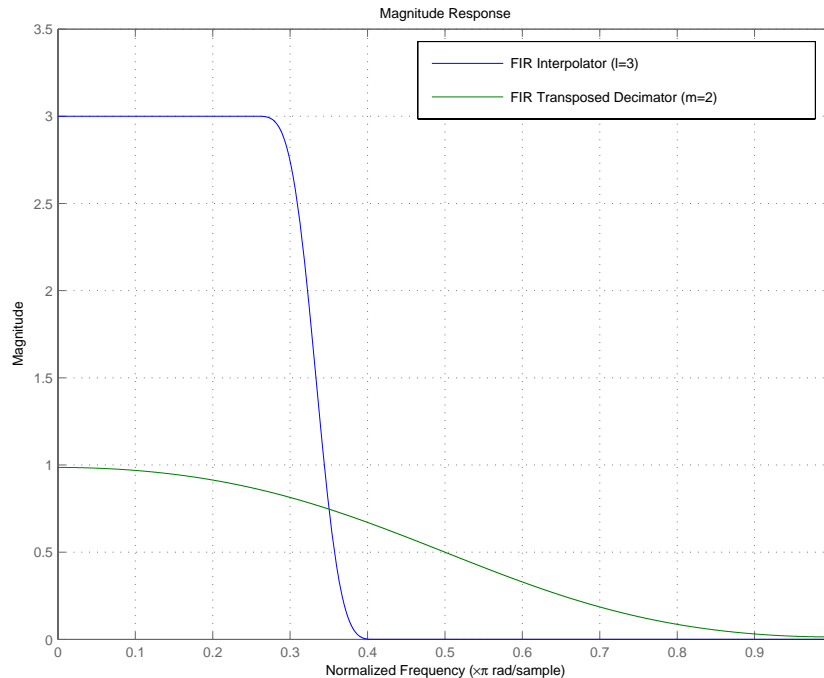
```
ans =
Columns 1 through 6
    0   -0.0092         0    0.2522    0.5000    0.2522
Columns 7 through 9
    0   -0.0092         0
```

You do not have to use a Nyquist filter to get new filter coefficients; other FIR filter design techniques in the toolbox work as well.

## Analyzing Multirate and Multistage Filters

Analyzing multirate or multistage filter objects is similar to analyzing discrete-time filter (`dfilt`) objects. Many if not all of the analysis functions for `dfilt` objects apply to `mfilt` objects equally. In particular, the Filter Visualization Tool (FVTool) provides most of the filter analysis tools you need.

```
h = fvtool(hm1,hm2);
```

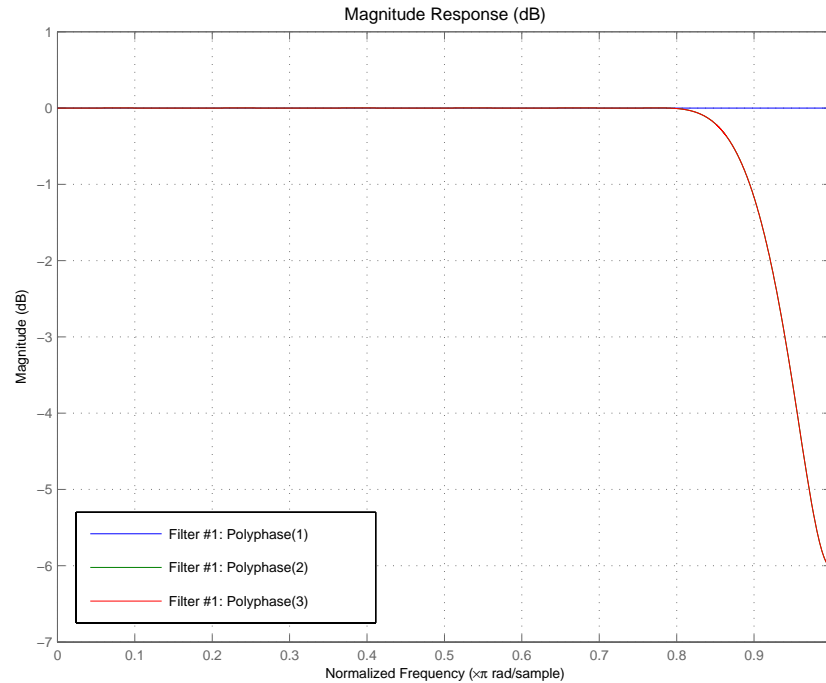


But one difference is very important. In analyzing multirate and multistage filters, the filter sample rates become important. The toolbox and tools let you specify sample rates for all of your analyses.

Additionally, `polyphase` for `mfilt` objects provides a tool for analyzing the polyphase components of `mfilt` objects. Calling the `polyphase` method without

output arguments (as shown here using filter hm1) starts an FVTool session with the polyphase subfilters ready for you to analyze.

```
polyphase(hm1)
```



`polyphase(hm)` lets you analyze your filter in more detail, such as checking that the group delay of each filter phase is flat, the desirable state.

## Filtering with Multirate Filters

By default, multirate filters begin with zero-valued filter states. Furthermore the `PersistentMemory` property is set to `false`, meaning that the filter object properties, such as the filter states, are reset before each filter run. This built-in reset process allows you to filter the same sequence input data sequence twice and produce the same output. For example:

```
x = 1:6;  
y1 = filter(hm2,x) % First run
```

```
y1 =
```

```
0    -0.0184    0.9676
```

At this point, you can verify that the filter hm2 holds nonzero final conditions in the filter states.

```
zf1 = hm2.States
```

```
zf1 =
```

```
3.0133  
3.4904  
-0.0369  
0
```

Run the filter again using the same input data x.

```
y2 = filter(hm2,x) % Second run
```

```
y2 =
```

```
0    -0.0184    0.9676
```

```
zf2 = hm2.States
```

```
zf2 =
```

```
3.0133  
3.4904  
-0.0369
```

After the second run, the states of the filter are the same as they were after the first run. With `PersistentMemory` property set to `false`, the filter states were reinitialized to zeros before the second run.

## Specifying Initial Conditions to the Filter

You make it possible to specify the initial conditions for your filter by setting both of the following:

- The `PersistentMemory` property to `true`
- The `States` property to your initial conditions (ICs)

Setting the `PersistentMemory` property to `true` is essential in the process of specifying initial conditions. If you set your filter ICs to specific values but you do not enable the filter memory, when you use the filter with input data the ICs get reset to zeros before the filter runs. As a result you lose your desired ICs and the results of filtering are not correct, or not what you might anticipate.

When you set the ICs, if you provide a scalar, that value is expanded to the correct number of states. If you specify a vector of values, its length must be equal to the number of states for the filter.

For example, using `hm2` as the filter, experiment with setting the filter states before filtering an input data set.

```

hm2.persistentmemory='true'

hm2.States=zf1

hm2 =

    FilterStructure: 'Direct-Form Transposed FIR Polyphase Decimator'
    Arithmetic: 'double'
    Numerator: [1x9 double]
    DecimationFactor: 2
    PersistentMemory: false

y3=filter(hm2,x)

y3 =

    2.9580    4.9853    2.4440

zf3=hm2.states

zf3 =

    2.9580

```

```
3.4904
-0.0369
```

As you might have anticipated, the filter output and the filter states are different now than they were after the first run.

## Streaming Data to the Filter

Setting the filter property `PersistentMemory` to `true` is a valuable feature when you are filtering streaming data. Since breaking a signal into sections and filtering the sections in a loop is equivalent to filtering the entire signal at once, this example simulates filtering streaming data by using the filter `hm2` in a loop.

```
reset(hm2); % Clear history of the filter by resetting all states.
xsec = reshape(x(:),2,3); % Break the input signal into
                           % three sections.
yloop = zeros(1,3); % Preallocate memory for storing
                   % intermediate results.
for i=1:3,
    yloop(i)=filter(hm2,xsec(:,i));
end
yloop

yloop =

    0   -0.0184    0.9676

y1

y1 =

    0   -0.0184    0.9676
```

You have verified that `yloop` (the signal filtered by three sections) is equal to `y1` (entire signal filtered at once). Without changing the property value for `PersistentMemory`, this test does not work.

Note that sample mode is a special case where you feed your input data to your filter one sample at a time. In this operating mode, debugging and cosimulation might be easier to do.



## Filtering Multichannel Signals

Up to this point you have only done single channel filtering, entering a vector of data  $x$  for the filter. When the input signal  $x$  is a matrix, the filter interprets each column of  $x$  as an independent input channel. Thus an 11-by-4 matrix provides 4 channels of input data where each channel contains 11 samples.

As was true for the streaming data case, sample-by-sample filtering is a special case. In sample mode operation, you have to provide a third input argument to filter that defines the input matrix dimension, in this case one dimension:

```
y = filter(hm,2,1)
```

Before you can continue this tutorial and experiment with multichannel filtering, you must either reset your filter to the initial states, or set the `PersistentMemory` property to `false`. The toolbox does not let you switch between single channel and multichannel filtering unless `PersistentMemory` is `false` or you reset the filter manually. If you forget to do this step, MATLAB returns an error message to tell you to reset your filter.

This example begins by resetting `hm2` and defining some data to filter.

```
reset(hm2);
x = randn(10,3); % Three channel signal; each channel providing
                % ten samples.
y = filter(hm2,x)
```

```
y =
```

```

           0           0           0
-0.0094    0.0095   -0.0022
 0.0794    0.3678    0.5956
 0.0440   -0.2253    1.1980
 0.6913    0.3884    0.3812
```

```
zf = hm2.States
```

```
zf =
```

```

 0.9268   -0.0027    0.4663
-0.5359   -0.6960    0.3092
 0.0066    0.0123   -0.0029
```

Notice that the filter object stores the final conditions for each channel separately. Each column of the States property corresponds to one input channel or column in the input matrix  $x$ .

### Filtering Multichannel Data in Loops

When  $x$  is a matrix, the filter treats each matrix column as an independent channel. 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. 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 (each individual row containing the states for one channel of input data) and `size(x,2)` columns (one column containing the filter states for each channel).

Here is an example that uses the `dim` input argument to filter the multichannel input data matrix  $x$ .

```

Fs = 44.1e3;           % Original sampling frequency 44.1kHz
n = [0:10239].';     % 10240 samples, 0.232s signal.
x = sin(2*pi*1e3/Fs*n); % Original signal, sinusoid at 1kHz.
M = 2;               % Decimation factor.
Hm = mfilter.firdecim(M); % We use the default filter.

% No initial conditions
y1 = filter(Hm,x);   % PersistentMemory is false.
zf = Hm.States;      % Final conditions.

% Non-zero initial conditions.
Hm.PersistentMemory = true;
Hm.States = 1;       % Uses scalar expansion.
y2 = filter(Hm,x);
stem([y1(1:60) y2(1:60)]) % Different sequences at the
                        % beginning.

% Streaming data
reset(Hm);           % Clear filter history.
y3 = filter(Hm,x);   % Filter the entire signal in one
                        % block.

```

```

reset(Hm);                % Clear filter history.
yloop = [];
xblock = reshape(x,[2048 5]);
% Filtering the signal section by section is equivalent to
% filtering the entire signal at once.
for i=1:5,
    yloop = [yloop; filter(Hm,xblock(:,i))];
end

```

## Generating Simulink Blocks

When the Signal Processing Blockset is installed, you can generate a Simulink® block of the `mfilt` object if the Signal Processing Blockset supports the filter structure. For example `hm1`, the direct-form FIR polyphase interpolator that you have been using throughout these examples, can be rendered as a Simulink block.

```
block(hm1,'destination','new','blockname','FIR Interp');
```

This figure shows the block as generated by the toolbox from the filter `hm1`.



## Getting Help About Multirate Filters

Entering `helpwin mfilt` in the MATLAB Command Window returns a list of multirate structures that the toolbox supports, as well as functions that operate on `mfilt` objects. For further information about a particular structure or function, enter `helpwin mfilt.functionname`, which returns the help information about `functionname` in a formatted HTML view, or enter `help mfilt.functionname` that returns the help information as plain text. For example:

```
help mfilt.firinterp % Help on the FIRINTERP structure
```

returns the following text in the Command Window.

```
FIRINTERP Direct-Form FIR Polyphase Interpolator.
```

`Hm = mfilt.FIRINTERP(L,NUM)` returns a direct-form FIR polyphase interpolator `Hm`.

`L` is the interpolation factor. It must be an integer. If not specified, it defaults to 2.

`NUM` is a vector containing the coefficients of the FIR lowpass filter used for interpolation. If omitted, a low-pass Nyquist filter of gain `L`

EXAMPLE: Interpolation by a factor of 2 (used to convert from 22.05kHz to 44.1kHz)

```
L = 2; % Interpolation factor.
Hm = mfilt.firinterp(L); % We use the default filter.
Fs = 22.05e3; % Original sampling frequency: 22.05kHz.
n = 0:5119; % 5120 samples, 0.232 second long signal.
x = sin(2*pi*1e3/Fs*n); % Original signal, sinusoid at 1kHz.
y = filter(Hm,x); % 10240 samples, still 0.232 seconds.
stem(n(1:22)/Fs,x(1:22),'filled') % Plot original sampled at
% 22.05kHz.

hold on % Plot interpolated signal (44.1kHz) in
% red.

stem(n(1:44)/(Fs*L),y(25:68),'r')
xlabel('Time (sec)');
ylabel('Signal value')
```

See also `mfilt/HOLDINTERP`, `mfilt/LINEARINTERP`, `mfilt/FFTFIRINTERP`, `mfilt/FIRFRACINTERP`, `mfilt/CICINTERP`, `mfilt/CICINTERPZEROLAT`, `FDESIGN/INTERP`, `FDESIGN/SRC`.

You can also enter

```
help mfilt.polyphase
```

at the MATLAB prompt to return this information about polyphase.

**POLYPHASE** Polyphase decomposition of multirate filters.

`P=POLYPHASE(Hm)` returns the polyphase matrix of the multirate filter `Hm`. The *i*th row of the matrix `P` represents the *i*th subfilter.

POLYPHASE(Hm) called with no outputs launches the Filter Visualization Tool (FVTool) with all the polyphase subfilters to allow analysis of each component individually.

To use the online help system, use the doc function instead of help.

```
doc mfilt
```

opens the Help browser and displays the general help text for multirate filter objects.

To obtain information about CIC decimation filter objects, enter one of the following commands:

```
help mfilt.cicdecim  
doc mfilt.cicdecim
```

at the command prompt, depending on which structure you need to know about.

For a complete list of the multirate filters that are available in the toolbox, enter `help mfilt`.

## FIR Decimation—Filtering with FIR Decimators

This section demonstrates how you can decrease the sampling rate of a signal using FIR decimators from the toolbox. To show you how this works, this section takes you through the following tasks:

- “Creating FIR Decimators” on page 3-18
- “Understanding Input Sample Processing and the InputOffset Property” on page 3-19
- “Filtering with FIR Decimators” on page 3-21

### Creating FIR Decimators

The Filter Design Toolbox supports different structures to perform decimation including different FIR-based structures and cascaded integrator-comb (CIC) structures. Entering `helpwin mfilt` at the prompt gives you a list of all supported structures.

Start by defining the filter decimation factor for your FIR decimator.

```
m = 3; % Specify the decimation factor as m.
```

Because the toolbox uses objects to implement multirate filters, you use the same methods to create most decimators. First you specify the decimation factor and then the FIR filter coefficients. If you do not include filter coefficients when you construct the filter, the toolbox filter constructor returns a lowpass filter with a cutoff frequency of  $(\pi/\text{decimation factor})$  and a gain of 1. This example uses `mfilt.firdecim` to create a direct-form polyphase FIR decimator. After constructing the filter, you can change the filter coefficients that are stored in the Numerator property.

Begin by designing an FIR decimator with the decimation factor set to 3.

```
hm1 = mfilt.firdecim(m); % Default decimator filter
```

`mfilt.firdecim` produces filters that decimate signals by integer factors. To change the sampling rate of a signal by a fractional factor, you might use a direct-form FIR polyphase sample rate converter. One way to create such a rate-changing filter is `mfilt.firsrc`. This structure uses  $L$  polyphase subfilters where  $L$  is the interpolation factor. Sample rate converters use both a decimation factor and interpolation factor to perform fractional rate changing.

```
l = 2; % Set the interpolation factor.  
hm2 = mfilt.firsrc(l,m); % Create the rate changing filter.
```

Here is the configuration information about hm2.

```
hm2 =  
  
    FilterStructure: 'Direct-Form FIR Polyphase Sample-Rate Converter'  
      Numerator: [1x72 double]  
RateChangeFactors: [2 3]  
PersistentMemory: false  
          States: [35x1 double]
```

## Understanding Input Sample Processing and the InputOffset Property

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, where each block has  $M$  elements, every complete input data block yields one output sample. Incomplete blocks of data (one or more input samples up to one less than the decimation factor) increment the `InputOffset` property by one for each sample in the incomplete block.

---

**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 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

```
InputOffset = mod(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).

All of the preceding discussion applies to interpolation filters as well, with appropriate changes from decimation to interpolation.

## Filtering with FIR Decimators

After creating your decimator, you are ready to filter data. Rather than use random data, as you did earlier, this example uses a more realistic data set.

For this example, define the input signal `x` as a 1 kHz sinusoid sampled at 44.1 kHz. Here is one way to create `x[n]`.

```
N = 159;
fs = 44.1e3;
n = 0:N-1;
x = sin(2*pi*n*1e3/fs); % Signal as required. 159 data points.
```

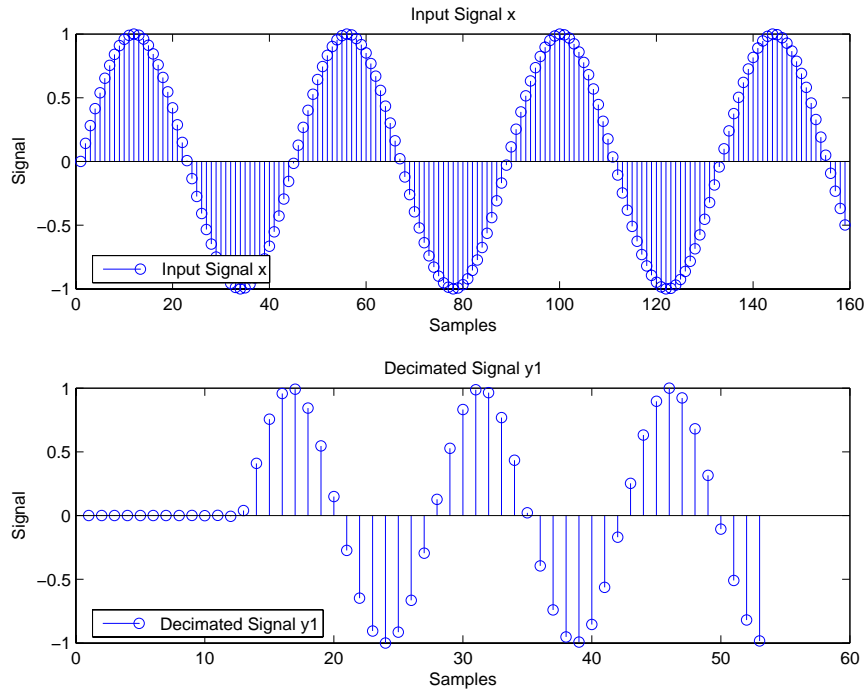
Now you can use filter `hm1` you designed earlier to try decimating `x`.

### Filtering with the Direct-Form FIR Polyphase Decimator `hm1`

You have data and a decimator in your workspace. Applying the filter to the data takes two steps—reset the filter and use `filter` to apply the decimator to `x`.

```
reset(hm1) % Reset the filter history and states to zeros.
y1 = filter(hm1,x);
```

Two stem plots give a sense of the decimation.



$y_1$  contains 53 samples—one-third of the number in  $x$ . Filter `hm1` decimated  $x$  by two-thirds. Since multirate filters support sample-by-sample processing, all input samples passed through the filter.

For further information about filtering options in general and specifying initial conditions for filters in particular, refer to “Getting Started—Designing Multirate Filters” on page 3-4.

The previous stem plot shows a feature of the filter—a delay of a number of samples before the filter starts to output the decimated input signal. Called the transient response, the length of the transient response of the decimator is equal to half the order of a polyphase subfilter. For `hm1`, the subfilter order is 24, so the transient response should be 12 samples. This is also the group delay of the filter.

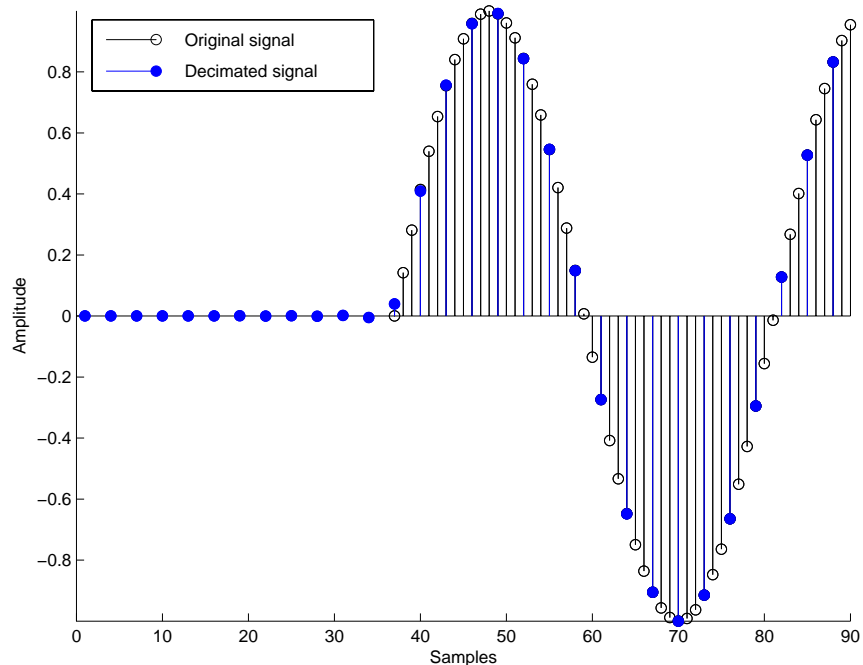
From the plot, it appears that the transient response is about 12 samples long. The next plot makes this more clear by plotting the decimated signal with a delayed version of the input  $x$ .

```
delay = mean(grpdelay(hm1)); % Constant group delay equal to its
                             % mean.
tx = delay+[1:length(x)];
ty = 1:m:m*length(y1);
```

Plot the output of the direct-form FIR polyphase decimator  $hm1$  and overlay a shifted version of the original signal using  $tx$  and  $ty$ .

```
stem(tx,x,'k');hold on;stem(ty,y1,'filled');
```

Using the delayed signals makes the transient response clear.



## CIC Filter Example—Using CIC Decimation Filters

This demonstration shows how to use multirate cascaded integrator-comb (CIC) decimation filters in the Filter Design Toolbox. CIC filters are efficient, multiplierless structures that are often used in high-decimation ratio or high-interpolation ratio systems.

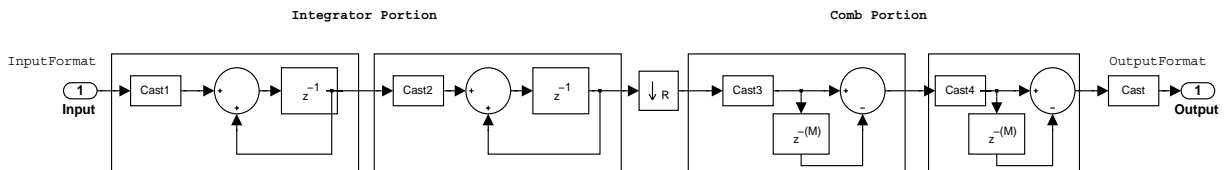
Digital down converters and digital up converters commonly use CIC filters. Refer to the demonstration program “Design of a Digital Down-Converter for GSM (Group Speciale Mobile)” in the Filter Design Toolbox demos for an example that uses a CIC decimator for digital down-conversion processing of a signal.

To help you understand what CIC filters do and how, this example includes the following sections:

- “Creating CIC Decimator filters” on page 3-24
- “Analyzing CIC Decimation Filters” on page 3-26
- “Working with Section Word Lengths” on page 3-28
- “CIC Filter States” on page 3-31
- “Filter Implementation—Signal Flow Graph” on page 3-33
- “Reference” on page 3-35

### Creating CIC Decimator filters

The Filter Design Toolbox provides a CIC decimating filter structure—the Cascaded Integrator-Comb Decimator. As you see in the figure below, the structure is optimized for pipelined implementations such as might be used on field-programmable gate arrays (FPGAs). The following Simulink model provides a signal-flow graph of the structure.



With the Fixed-Point Toolbox installed (required for you to use CIC filters), you create a default cascaded integrator-comb decimator object with this command

```
hm = mfilt.cicdecim
```

at the prompt. MATLAB returns the CIC filter with the specifications shown here.

```
hm =

    FilterStructure: 'Cascaded Integrator-Comb Decimator'
      Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 2
PersistentMemory: false

    InputWordLength: 16
    InputFracLength: 15

SectionWordLengthMode: 'MinWordLengths'

    OutputWordLength: 16
```

The CIC decimation filter comprises three portions—an integrator portion, a rate change factor, and a comb portion. Similarly, you can completely specify a CIC decimation filter with three parameters—a decimation factor  $r$ , the number of individual integrator or comb sections  $n$ , and the differential delay of the comb section  $m$ .

The display of the multirate filter object (`mfilt`) in the Command Window groups the filter properties together in a logical manner, making the filter specification more clear.

Only the writable properties appear in the display by default. Changing a filter property, such as resetting `PersistentMemory` from `false` to `true` reveals more properties as they become writable—in this case the `States` property appears when `PersistentMemory` is `true`.

Unlike other multirate filters and discrete-time objects, CIC filter objects allow only fixed-point arithmetic (the `Arithmetic` property is always set to `fixed`) since these filters are inherently fixed-point filters. Check the value of the `Arithmetic` property

```
set(hm,'arithmetic')  
  
ans =  
  
    'fixed'
```

to see that `fixed` is the only option. As with all filter objects, and all objects in general, the `get` function returns the complete set of properties (read-only and writable) for the filter and object.

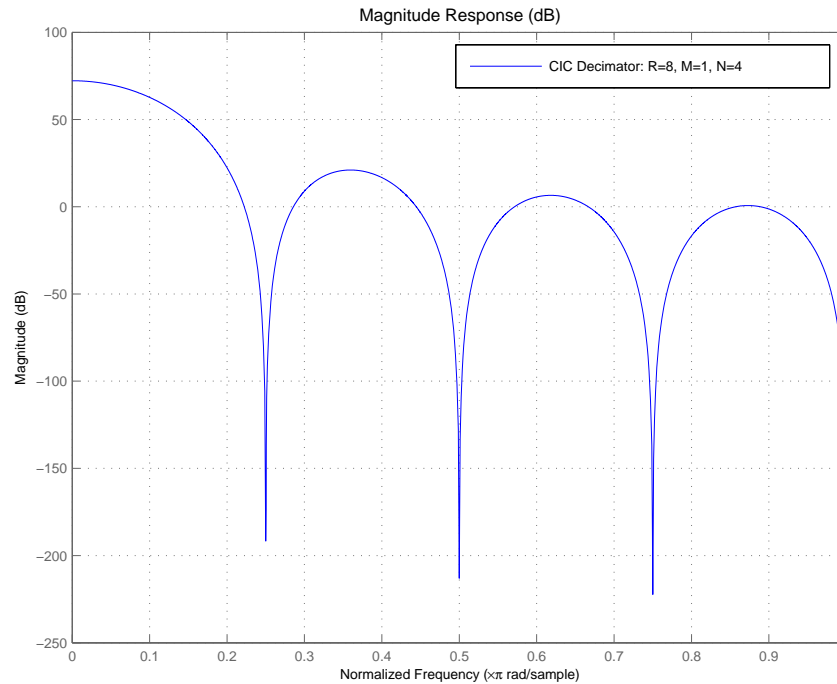
```
get(hm)
```

### Analyzing CIC Decimation Filters

Analyzing CIC filters is the same as analyzing any multirate filter object in the Filter Design Toolbox. The Filter Visualization Tool (FVTool) provides graphical access to all analyses.

```
hm = mfilt.cicdecim(8,1,4);  
hfvt=fvtool(hm);  
hfvt.showreference='off';
```

FVTool returns the magnitude response for `hm`, shown here. As `hm` is a fixed-point filter, we suppress the reference filter in the display by setting the `ShowReference` property in FVTool to `off`.



After you have the filter displayed in FVTool, you can use any of the filter analysis capabilities provided to learn more about  $h_m$ . To perform an analysis, select one of the analytical options, such as **Impulse Response** or **Round-off Noise Power Spectrum** from **Analysis** on the FVTool menu bar.

## About the MSB at the Filter Output

A significant consideration in CIC filters is the size (number of bits) of data that can pass through the filter without loss. The most significant bit (MSB) of the filter represents the maximum number of bits that can be propagated through the filter while maintaining the integrity of the data.

Parameters  $R$ ,  $M$ ,  $N$  and the `InputWordLength` specify the MSB of the filter output. Since the output of the integrator sections of the filter can grow without bounds, the MSB at the filter output is also the MSB for all filter sections.

Called  $B_{\max}$  in the reference, the maximum word length in the filter, or most significant bit (MSB), is both the maximum word length for all of the filter sections as well as the MSB at the filter output.

Hogenauer defines  $B_{\max}$ , the MSB at the filter output, as follows:

$$B_{max} = \lceil N \log_2 RM + B_{in} - 1 \rceil$$

with

- N is the number of filter sections
- M is the comb portion differential delay
- R is the decimation factor
- $B_{in}$  is the input word length in bits

## Working with Section Word Lengths

CIC filters include a property that defines how you specify the section word and fraction lengths for the filter. Called `SectionWordLengthMode`, this property specifies the specific data format (word length and fraction length) the filter uses when accumulating data in the integrator sections or subtracting data in the comb sections. `SectionWordLengthMode` can take one of two values:

- `MinWordLengths`—the filter calculates the optimal section word lengths given the filter parameters R (the rate change factor), M (the differential delay), N (the number of filter sections), and the input and output word lengths.
- `SpecifyWordLengths`—you specify the word lengths for the sections by entering a scalar or a vector of length  $2*N$ . When you provide a scalar, the filter method expands the scalar into a vector with  $2*N$  elements, applying the same word length to all sections. If you specify a vector, it must meet these requirements:
  - It must contain  $2*N$  elements.
  - The values of the vector elements must be monotonically decreasing.

When you construct a new CIC decimating filter, `SectionWordLengthMode` is set to `MinWordLength` by default.

Using `hm` as an example, here is the `SectionWordLengthMode`.

```
set(hm, 'SectionWordLengthMode')
```



```
ans =

    'MinWordLengths'
    'SpecifyWordLengths'
```

In the reference provided later in this section ([1] on page 3-35), Hogenauer shows that during filtering you can discard least significant bits (LSBs) from each section (refer to Equation 21 of the reference) of the filter so long as the error introduced by removing the LSBs is acceptable at the filter output. In this case, the section word lengths reported by the filter are computed by subtracting the LSBs from the maximum word lengths in the filter (refer to Equation 11 in the reference for details).

To help connect the CIC filter designs in the toolbox to the analysis by Hogenauer, the next example designs a CIC decimator that matches the design on page 159 of the Hogenauer paper.

```
m=1; % Set the differential delay to one.
n=4; % Specify the number of sections.
r=25; % Set the rate change factor.
inwl=16; % Set the word length at the filter input.
outwl=16; % Set the filter output word length.

% With the specifications prepared, design the CIC decimator.
hm=mfilt.cicdecim(r,m,n,inwl,outwl);
```

hm reproduces the referenced filter exactly. To see the correspondence, check that the word lengths applied to each filter section match those developed in the reference example, where the MSB is 34 bits.

<b>Filter Section</b>	<b>Number of LSBs Discarded</b>	<b>Word Length Calculated in [1] on page 3-35 (MSB-Discarded LSBs)</b>
1	1	33 (34-1)
2	6	28 (34-6)
3	9	25 (34-9)
4	13	21 (34-13)

Filter Section	Number of LSBs Discarded	Word Length Calculated in [1] on page 3-35 (MSB-Discarded LSBs)
5	14	20 (34-14)
6	15	19 (34-15)
7	16	18 (34-16)
8	17	17 (34-17)

In the referenced paper by Hogenauer, [1] on page 3-35, the MSB is also called  $B_{\max}$ . Use `get` to verify the match.

```
get(hm, 'sectionwordlengths')

ans =

    33    28    25    21    20    19    18    17
```

For cases where you enter the word lengths explicitly when you construct the filter, rather than letting the `mfilt` constructor determine them, by setting `SectionWordLengthMode` to `SpecifyWordLengths`, you enter the word lengths to use as either a scalar or a vector of length  $2*n$ . Recall from earlier that the input vector containing the section word lengths must meet two criteria—the number of elements must be twice the number of filter sections  $n$ , and the element values must be monotonically decreasing.

As you see in this example, when you enter the word length as a scalar, the filter constructor expands the scalar to apply it as the section word length for all of the filter sections.

```
set(hm, 'sectionWordLengthMode', 'SpecifyWordLengths');
hm.sectionWordLengths=32;
get(hm, 'sectionWordLengths')

ans =

    32    32    32    32    32    32    32    32
```

## CIC Filter States

The States property of CIC decimation filters contains an object—`filtstates.cic`. This object represents or stores the initial conditions of the filter before filtering and the final conditions after filtering. `filtstates.cic` has two properties, `Integrator` and `Comb`, that correspond to their respective portions of the filter. When you construct a CIC filter, the states contain zeros. After you filter data with the filter, the states contain the values stored in the filter delay elements. To demonstrate the filter states, the following example creates a decimator, and then applies the filter to a set of fixed-point input data.

```
% Construct the input data set for filter filter some fixed-point
% ones.
x = fi(ones(1,10),true,16,0);
% Construct a decimator to use to filter x.
hm = mfilt.cicdecim(2,1,2,16,16,16);
```

Take a look at `x` and `hm` to see what you have.

```
x
x =
    1     1     1     1     1     1     1     1     1     1

    DataTypeMode: Fixed-point: binary point scaling
        Signed: true
        WordLength: 16
    FractionLength: 0

        RoundMode: round
    OverflowMode: saturate
        ProductMode: FullPrecision
    MaxProductWordLength: 128
        SumMode: FullPrecision
    MaxSumWordLength: 128
    CastBeforeSum: true

hm
hm =
```

```
        FilterStructure: 'Cascaded Integrator-Comb Decimator'  
            Arithmetic: 'fixed'  
DifferentialDelay: 1  
NumberOfSections: 2  
DecimationFactor: 2  
PersistentMemory: false  
  
        InputWordLength: 16  
        InputFracLength: 15  
  
SectionWordLengthMode: 'SpecifyWordLengths'  
  
        SectionWordLengths: [16 16 16 16]  
  
        OutputWordLength: 16  
get(hm,'states')  
  
ans =  
  
        Integrator: [2x1 States]  
        Comb: [2x1 States]
```

At this point, the states for the filter are zeros. That changes after you filter a set of data.

```
hm.inputfraclength = 0; % Set the input to use integer data.  
y = filter(hm,x);
```

You can extract the final states by using the `int` function and assigning the output to a variable.

```
sts = int(Hm.states)  
sts =
```

```
    10    45  
    28    13
```

As you see, the states now contain nonzero values related to the filtering operation.

This states matrix has dimensions  $M+1$ -by- $N$ , where  $M$  is the differential delay of the comb section and  $N$  is the number of sections. Filter `hm` stores the integrator sections states (`hm.states.integrator`) in the first row of the states matrix and stores the states for the comb portion in the remaining rows in the matrix.

You might have noticed that the `States` property is not displayed by the default filter display. When `PersistentMemory` is set to `false`, you do not see the `states` property in the default listing in MATLAB.

```

hm % Generate the default filter display.

hm =

    FilterStructure: 'Cascaded Integrator-Comb Decimator'
    Arithmetic: 'fixed'
    DifferentialDelay: 1
    NumberOfSections: 2
    DecimationFactor: 2
    PersistentMemory: false

    InputWordLength: 16
    InputFracLength: 15

    SectionWordLengthMode: 'MinWordLengths'

    OutputWordLength: 16

```

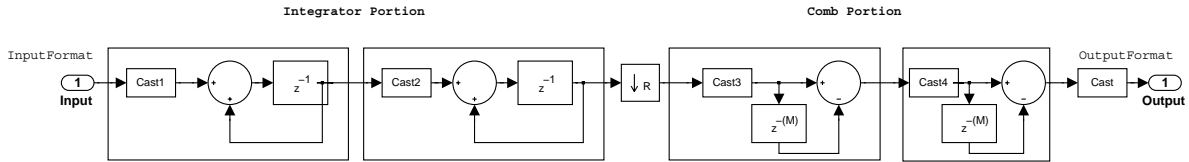
Setting `PersistentMemory` to `true` reveals the `States` property in the filter display. However, when you use `get` to review the properties, you see the `States` property listed in all instances.

For more information about the `fi` object used in `x` above, refer to the Fixed-Point Toolbox documentation in the online Help system.

## Filter Implementation—Signal Flow Graph

The toolbox implements a structure that differs slightly from the one in the referenced paper by [1] on page 3-35. The difference lies in the location of the delays in the integrator portion of the filter. We made this change to optimize the filter for pipelining on hardware such as field-programmable gate arrays (FPGAs). The following figure shows the flow graph as implemented by

`mfilt.cicdecim`. After the table following the figure, is a short example that should help interpret the entries in the figure.



The word length and fraction length at each stage of the decimator are shown in the following table. Either you specify the word length for each filter stage in the `SectionWordLengths` property as a vector of integers, or you let the filter constructor set the word lengths by making `MinWordLengths` the value for `SectionWordLengthMode`. The calculation for each fraction length is shown below:

#### Decimator Word Lengths and Fraction Lengths

Position in the Signal Flow	Word Length	Fraction Length
Filter Input	InputWL	InputFL
1 <sup>st</sup> Section Output	SectionOneWL	InputFL
2 <sup>nd</sup> Section Output	SectionTwoWL	InputFL (SectionTwoWL - SectionOneWL)
3 <sup>rd</sup> Section Output	SectionThreeWL	SectionTwoFL + (SectionThreeWL - SectionTwoWL)
4 <sup>th</sup> Section Output	SectionFourWL	SectionThreeFL + (SectionFourWL - SectionThreeWL)
N <sup>th</sup> Section Output	Section(N)WL	Section(N-1)FL + (Section(N)WL - Section(N-1)WL)
Filter Output	OutputWL	FinalSectionFL + (OutputWL - FinalSectionWL)

## Reference

The following paper formed the basis for developing the CIC filters in the Filter Design Toolbox. Many more details of the CIC multirate filters are discussed in this reference.

[1] 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.

## Analyzing Multirate and Multistage Filters

Multirate filter analysis presents some differences from analyzing single-rate discrete-time filters. While most of the same analytical tools apply, the primary difference is the filter sample rate—what the sample rate is, how it is defined, and where. Filter sample rate, called  $F_s$  in the toolbox, changes depending on the type of multirate filter you are using. Or more precisely, how the sample rate is defined changes according to the multirate filter under discussion.

Generally, filter sample rate refers to the rate at which the filter is running:

- For decimators, the filter sample rate equals the sample rate at the filter input, prior to decimating the input.
- For interpolators, the filter sample rate is equal to the sample rate at the output of the filter, after interpolation.
- For sample rate change filters,  $F_s$  is the input rate multiplied by the interpolation factor. The decimation factor does not apply to define the sample rate.

When you provide a sampling frequency for the analysis, the analytical tool, such as FVTool, assume that the rate specified is the sampling frequency at which the filter is operating.

Another feature of analyzing multirate filters that have more than one stage is that the analysis process applies to a filter that is the overall equivalent of the multistage filter under consideration. Recognizing that the analytical tool you choose first computes an equivalent filter makes understanding the analytical process somewhat easier.

For example, a multistage filter that included

- Multiple interpolators
- Multiple decimators

might be reduced to an equivalent filter with

- One equivalent interpolation filter
- One equivalent decimation stage

For more about how the tools develop the equivalent filter they use to analyze your filter, refer to “Performing Multistage Filter Analysis” on page 3-40.



A pair of definitions will help as you read this section:

- Multirate filters consist of *sections*.
- Multistage filters are the result of using `cascade` or `parallel` (refer to `dfilt` in the Signal Processing Toolbox documentation for more information about parallel and cascade filter design) to create filters by combining other filters. Each filter that composes the multistage filter is called a *stage*.

This tutorial demonstrates how to perform analysis on single-stage and multistage multirate filters by presenting the following topics:

- “Analyzing Single-Stage Multirate Filters” on page 3-37
- “Comparing Interpolators” on page 3-38
- “Performing Multistage Filter Analysis” on page 3-40
- “Analyzing Multistage Interpolators” on page 3-42
- “Analyzing a Multistage Sample-Rate Converter” on page 3-43
- “Analyzing Other Multistage Configurations” on page 3-45

## Analyzing Single-Stage Multirate Filters

You analyze single-stage multirate filters at the rate the filter is operating. As mentioned in the introduction to this tutorial section, the sample rate you use depends on the filter you are analyzing.

The following plot overlays the magnitude response of a sample-rate converter, an interpolator, and a decimator. For the first filter, the input sampling frequency is  $1000/5$  and the output sampling frequency is  $1000/3$ . For the interpolator, the input  $f_s$  is  $1000/4$  and the output  $f_s$  is  $1000$ . Finally, for the decimator, the input  $f_s$  is  $1000$  and the output  $f_s$  is  $1000/3$ .

Here are the commands to create the three filters to analyze.

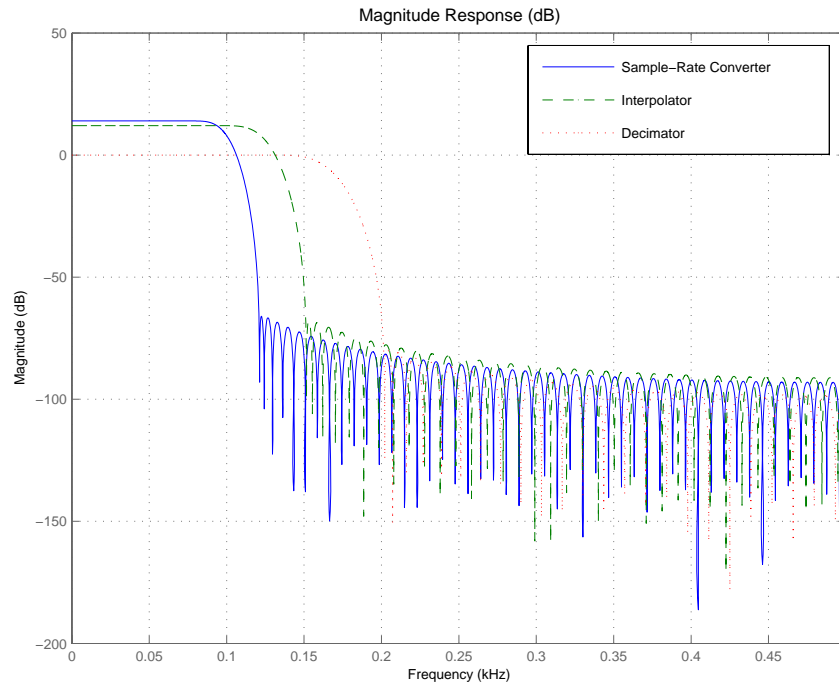
```
h1 = mfilt.firsrc(5,3); % Use a default filter.  
h2 = mfilt.firinterp(4); % Use a default filter.  
h3 = mfilt.firdecim(3); % Use a default filter.
```

Now you need to specify the sampling rate and the number of points in the FFT used.

```
fs = 1000; nfft = 8192;
```

With the filters in your workspace and the sampling frequency set, use FVTool to visualize the filters using a common sampling rate.

```
fvtool(h1,h2,h3,'fs',fs);
```



## Comparing Interpolators

Interpolators and decimators exhibit a lowpass magnitude response. Simple interpolators, like the CIC interpolator and the hold or linear interpolators, have a poor lowpass response. However, they are easy to implement and, except for the linear interpolator, they do not require the filter to perform multiplications in real-time while filtering data. The following plot compares the lowpass response of four different interpolators:

- An FIR interpolator (`mfilt.firinterp`)

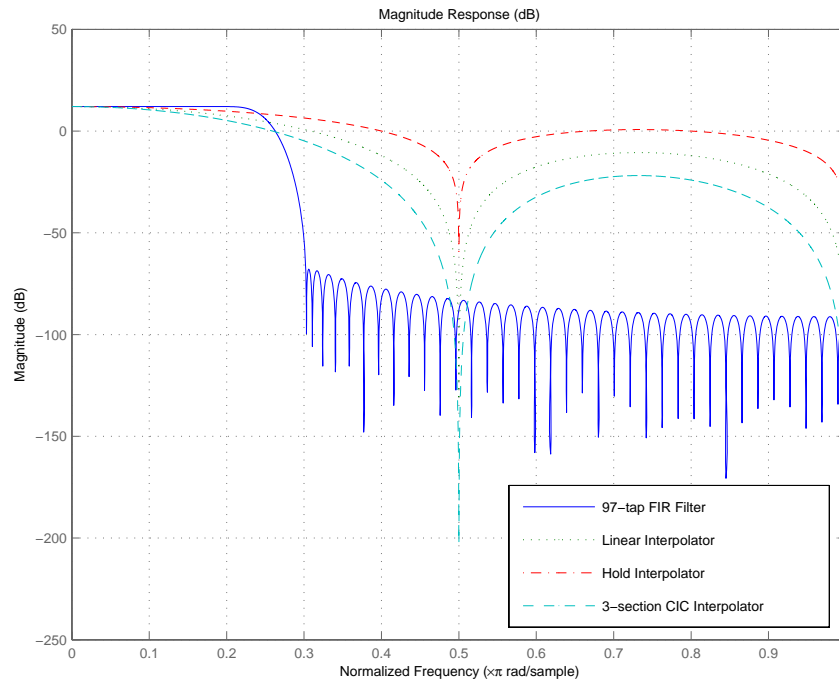
- A linear interpolator (`mfilt.linearinterp`)
- A hold interpolator (`mfilt.holdinterp`)
- A CIC interpolator (`mfilt.cicinterp`)

They each have an interpolation factor of 4. You can see that the quality of the lowpass filter, such as the sharpness of the lowpass cutoff, depends on which type of interpolator you use. By design, the CIC interpolator has more gain than the other interpolators. For the purposes of this analysis, we include a scalar in cascade with the CIC filter to normalize its gain. Normalizing the gain makes comparing the different filters easier.

```
h(1) = mfilt.firinterp(4); % Use the default filter.
h(2) = mfilt.linearinterp(4);
h(3) = mfilt.holdinterp(4);
hcic = mfilt.cicinterp(4,1,3); % 3-section CIC with
                               % differential delay = 1.
hscalar = dfilt.scalar(1/gain(hcic));
h(4) = cascade(hscalar, hcic); % Add a gain correction...
                               % filter in cascade.
```

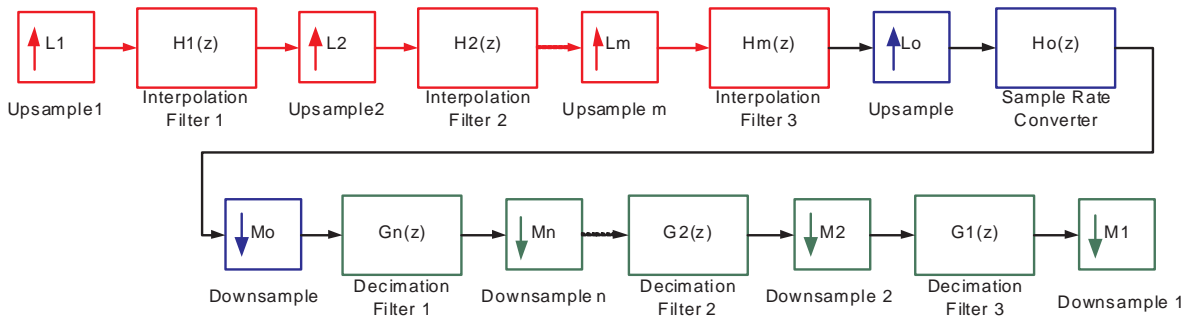
Use FVTool to see the results of the four filters. An interesting trick you might notice—naming the filters as indexes of the variable `h` lets you plot all four interpolators by passing `h` to FVTool.

```
fvtool(h);
```



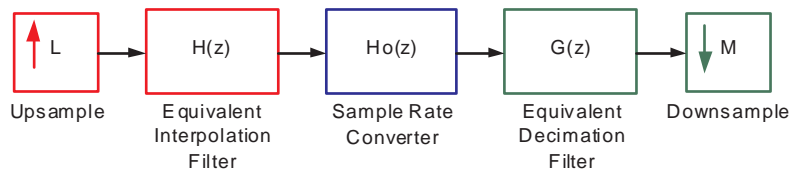
## Performing Multistage Filter Analysis

Using the tools provided in the toolbox, either from the command line or in FVtool, you can analyze multirate filters of the following form.



In a multirate filter, any of the blue, red, or green sections is optional. Since that is true, you can perform analysis on multistage interpolators, multistage decimators, or multistage sample-rate converters.

When you choose to perform the analysis, the analysis tool computes an equivalent overall filter for the interpolation section and/or the decimation section as shown in the next figure, and performs the requested analysis on the equivalent filter.



In the equivalent filter shown in the figure, the following conversions apply.

- Upsample block  $L = L_0 * L_1 * L_2 * \dots * L_m$ ; (convolved interpolators)
- Downsample block  $M = M_0 * M_1 * M_2 * \dots * M_n$ ; (convolved decimators)
- Interpolator transfer function  

$$H(z) = H_1(z^{(L_0 * L_1 * \dots * L_m)}) * H_2(z^{(L_0 * L_2 * \dots * L_m)}) \dots H_m(z^{(L_0)});$$
- Decimator transfer function  

$$G(z) = G_1(z^{(M_0 * M_1 * \dots * M_n)}) * G_2(z^{(M_0 * M_2 * \dots * M_n)}) \dots G_n(z^{(M_0)})$$

Finally, filters  $H(z)$ ,  $G(z)$ , and  $H_0(z)$  are all operating at the same rate and can be combined into a single filter on which to perform the analysis. If you specify

a sampling frequency as an input to the analytical tool, the analysis assumes that the single overall filter (equivalent to the subfilters that have been combined) is operating at the rate you specified.

## Analyzing Multistage Interpolators

Here is an example of how you might analyze a multistage interpolator. Refer to the demo “Design of a Digital Down-Converter for GSM” in the Filter Design Toolbox demos for an example in which the Global System for Mobile Communications (GSM) uses a multistage decimator.

This section cascades four interpolators to form a four stage filter. The fourth interpolator is a CIC filter. In this case, the sampling frequency specified for the filter corresponds to the output of the four stage interpolator because this is the rate at which the equivalent filter operates.

```
h(1) = mfilt.firinterp(4);  
h(2) = mfilt.firinterp(2);  
h(3) = mfilt.firinterp(2);  
h(4) = mfilt.cicinterp(16);  
hc = cascade(h);
```

```
hc
```

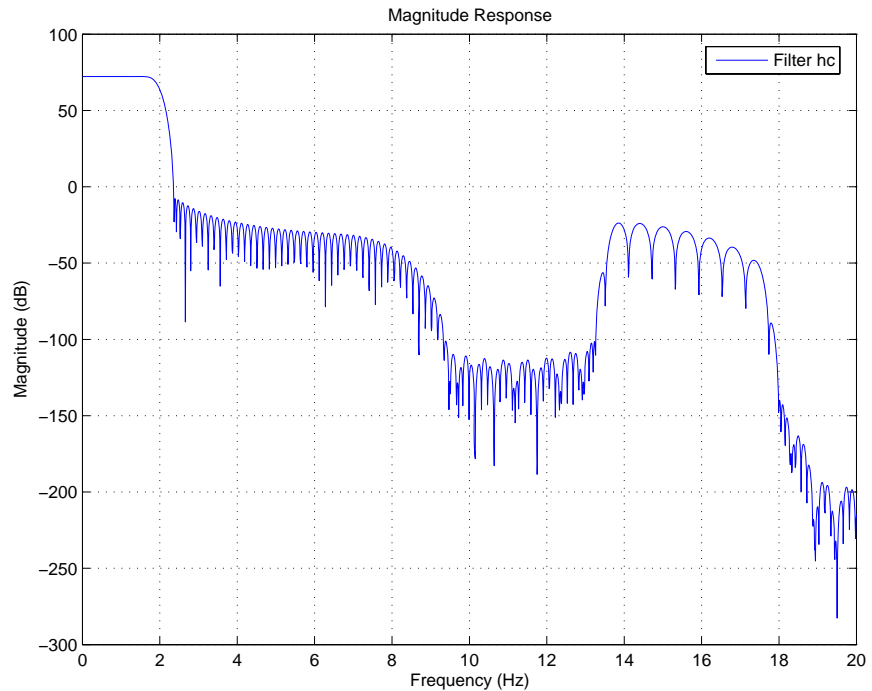
```
hc =
```

```
FilterStructure: Cascade  
Stage(1): Direct-Form FIR Polyphase Interpolator  
Stage(2): Direct-Form FIR Polyphase Interpolator  
Stage(3): Direct-Form FIR Polyphase Interpolator  
Stage(4): Cascaded Integrator-Comb Interpolator  
PersistentMemory: false
```

To perform the analysis on `hc`, compute the frequency response between 0 and 200 Hz. set the sampling frequency  $F_s$  to 1000 Hz.

```
[hf, f] = freqz(hc, 0:1e-2:20, 1000);  
plot(f, 20*log10(abs(hf)))
```

`freqz` returns the transfer function for the cascaded filter at the sampling frequency you entered as an input argument.



## Analyzing a Multistage Sample-Rate Converter

To demonstrate working with multistage sample rate converters, add some decimation stages to filter `hc` to form a multistage sample-rate converter. Again, the sampling frequency `fs` you specify as input to `freqz` once again represents and is assumed to be the rate of the equivalent filter. And this is the rate at which the frequency response of `hc2` is analyzed. This `fs` is the fastest rate in the entire system in this case.

```
h(5) = mfilt.firsrc(2,3);
h(6) = mfilt.cicdecim(13);
h(7) = mfilt.firdecim(5);
hc2 = cascade(h)
```

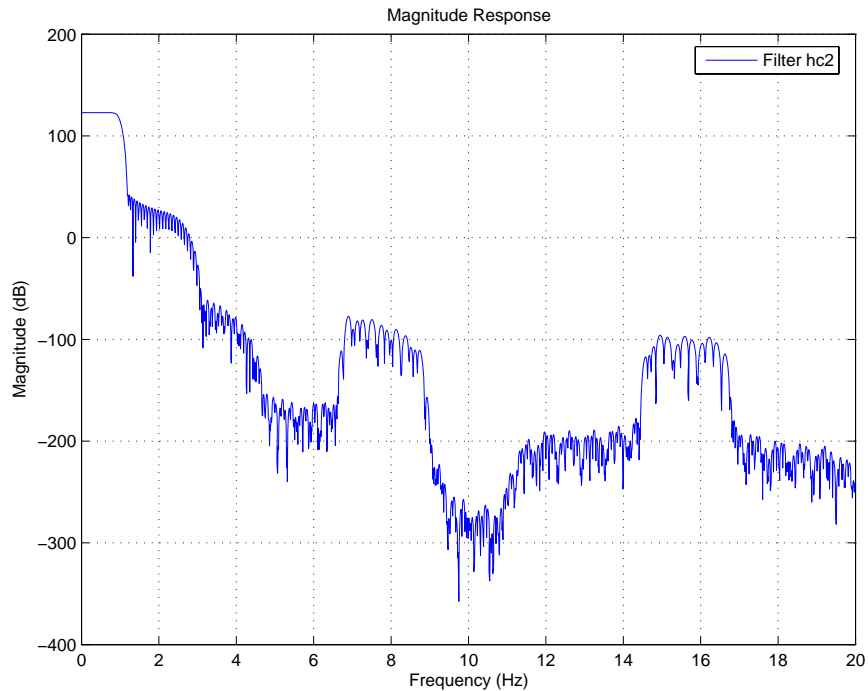
```
hc2 =  
  
    FilterStructure: Cascade  
        Stage(1): Direct-Form FIR Polyphase Interpolator  
        Stage(2): Direct-Form FIR Polyphase Interpolator  
        Stage(3): Direct-Form FIR Polyphase Interpolator  
        Stage(4): Cascaded Integrator-Comb Interpolator  
        Stage(5): Direct-Form FIR Polyphase Sample-Rate Converter  
        Stage(6): Cascaded Integrator-Comb Decimator  
        Stage(7): Direct-Form FIR Polyphase Decimator  
    PersistentMemory: false
```

As you did in the preceding section, compute the frequency response between 0 and 200 Hz using  $F_s$  equal to 1000 Hz.

```
[hf, f] = freqz(hc2, 0:1e-2:20, 1000);  
plot(f, 20*log10(abs(hf)))
```

The figure show the frequency response of `hc2`, the result of adding decimators and a rate changing filter to `hc`.

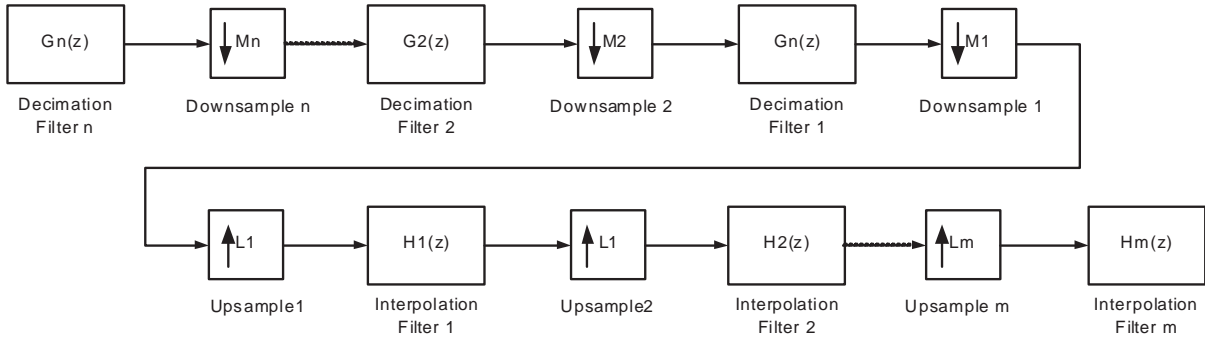




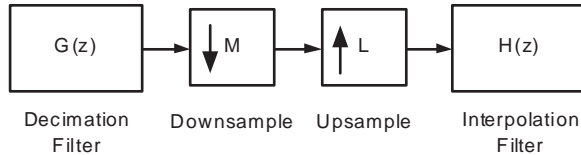
## Analyzing Other Multistage Configurations

In addition to the multistage filters `hc` and `hc2` shown, the toolbox lets you analyze multistage filters where decimation occurs prior to interpolation, provided the overall filter interpolation and decimation factors are the same. Notice that this does not necessarily mean that there is an equal number of decimation and interpolation stages.

One multistage structure that you could analyze in the toolbox is this one:



In this case, the analysis tools develop two equivalent filters as shown in the next figure, where  $M = M1 * M2 * \dots * Mn = L1 * L2 * \dots * Lm = L$ .



Because the overall interpolation factor  $L$  is equal to the overall decimation factor  $M$ , the equivalent filters are operating at the same rate.

As before, when you provide a sampling frequency for the analysis, the tools assume that the supplied rate is the rate at which both filters are operating. For this case, this would also be equal to the input and output rate for these filters.

To see a demonstration about this type of analysis, where you are analyzing multistage, multirate filters, refer to “Multirate Multistage FIR Filter Design” in the Filter Design Toolbox demos.

## Audio Example—Audio Sample Rate Conversion

For a more concrete application of multirate filters, this section illustrates multirate filters that you might use to perform sample rate conversion on different audio formats. During this section, you create each of the following:

- FIR sample rate conversion filter
- FIR fractional interpolator
- FIR fractional decimator

To do these tasks, this section contains the following topics:

- “Creating the Multirate Filters” on page 3-47
- “Decreasing the Sample Rate by a Fractional Factor” on page 3-48
- “Constructing the Fractional Decimator” on page 3-48
- “Filtering to Change the Sample Rate” on page 3-49
- “Comparing the Resampled Signals” on page 3-49
- “Increasing the Sample Rate by a Fractional Factor” on page 3-51
- “Plotting the Original Signal and the Reconverted Signal” on page 3-52
- “Converting from 48 kHz to 44.1 kHz” on page 3-53
- “Plotting the 48 kHz Signal and the 44.1 kHz Signal” on page 3-54

### Creating the Multirate Filters

All fractional sample rate conversion filters are created in the same way. You specify the interpolation factor  $L$ , and the decimation factor  $M$ , and the FIR filter coefficients.  $L$  and  $M$  must be relatively prime.

Two integers  $a$  and  $b$  are relatively prime when they do not share any common factors. For example, 21 and 54 are not relatively prime—3 is a factor common to both. 14 and 25 are relatively prime.

When  $L$  and  $M$  are not relatively prime, they are converted to relatively prime factors and you get a warning in MATLAB.

If you do not provide filter coefficients when you construct your filter, the filter design process returns a lowpass filter with a cutoff frequency of  $\pi/\max(L,M)$  and a gain of  $L$  in the passband.

Begin by designing a default rate change filter `hm1`.

```
hm1 = mfilt.firsrc(4,3); % Default sample rate change filter.  
hm2 = mfilt.firfracinterp(8,6);
```

Warning: L and M are not relatively prime. Converting ratio 8/6 to 4/3.

The cutoff frequency of the filter should be approximately  $\pi/4$ .

MATLAB notifies you that the factors 8 and 6 do not meet the relatively prime specification and reduces each by the common factor 2. Then MATLAB designs the filter.

## Decreasing the Sample Rate by a Fractional Factor

Suppose you are converting an audio signal recorded at 48kHz to 32kHz for broadcasting. Consider the following audio sample recorded at 48kHz (Copyright 2002 FingerBomb) by loading the sample into MATLAB and then playing the file.

```
load audio48;
```

To listen to the original 48 kHz signal, you can use an audio player object in MATLAB.

```
p48 = audioplayer(signal48kHz,Fs48); % Create audio player  
                                     % object.  
play(p48); % Play the track. Use stop(p48) to stop play.
```

In all, the track lasts about 9 seconds.

## Constructing the Fractional Decimator

Reducing the 48kHz sample rate for the signal to 32 kHz requires decimating the signal by two-thirds (discard one sample out of every three). Decimation by two-thirds is an example of fractional decimation.

The interpolation factor for this case is 2 and the decimation factor is 3. You can use a fractional decimator to achieve this sample rate modification. To avoid making this example more complicated, use the default filter that `mfilt.firfracdecim` designs for now.

```
hfd = mfilt.firfracdecim(2,3); % Use default decimator filter.
```

```

hfd
hfd =
    FilterStructure: 'Direct-Form FIR Polyphase Fractional Decimator'
    Numerator: [1x72 double]
    RateChangeFactors: [2 3]
    PersistentMemory: false
    States: [36x1 double]

```

You could also use your own lowpass filter by specifying the coefficients as a third input argument

```
hfd = mfilt.firfracdecim(1,m,coeffs)
```

where `coeffs` contains the FIR filter coefficients to use.

## Filtering to Change the Sample Rate

To use the fractional decimator `hfd` to convert the sample rate of the signal, you invoke the filter method with the signal `signal48kHz` and `hfd`.

```
s32 = filter(hfd,signal48kHz);
```

Once again, you can use an `audioplayer` object to listen to the down-converted signal.

```
p32 = audioplayer(s32,32e3); % Create a new audio player.
play(p32);
```

## Comparing the Resampled Signals

You now have about 9 seconds of audio. Of course, you can find the exact length in seconds from

```
length(signal48kHz)/Fs48 % Or length(s32)/32e3.
```

```
ans =
```

```
8.9634
```

For clarity, you should overlay the two signals on a plot to compare them. Because the audio track contains some 430,000 samples, you show only a small signal segment. You also have to account for the delay the filter introduces in the 32 kHz signal (the transient response mentioned earlier). Filter `hfd` has a

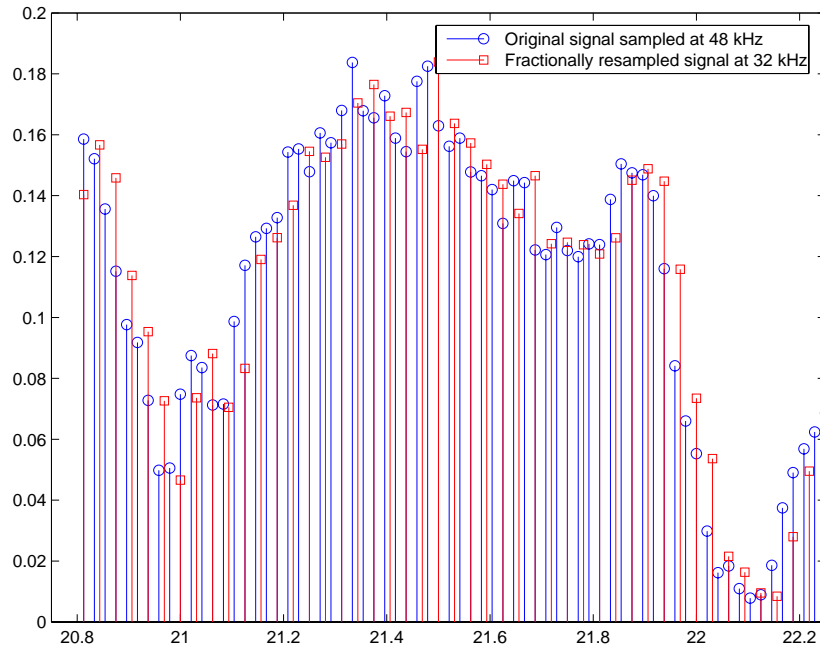
group-delay of 36 samples. Since it is running three times faster than the 32 kHz signal, the delay is equivalent to 12 low speed samples.

Note that there are three samples of the 48 kHz signal for every two samples of the 32 kHz signal. Now to pick some audio data samples to display.

To make the overlay work, you need the same starting point for each signal. The following code finds common points for the 48 kHz and 32 kHz signals and displays them in a stem plot.

```
xindx = 999:1500; % 0.0105 seconds of audio at 48 kHz.
figure
stem(xindx/Fs48*1e3,signal48kHz(xindx));
hold on;
xindx2 = xindx(1)*32e3/48e3:xindx(end)*32e3/48e3; % Find the same
                                                    % start and
                                                    % stop times.

stem(xindx2/32,s32(xindx2+12),'r'); % Add 12 samples to account
                                     % for filter transient delay.
```



## Increasing the Sample Rate by a Fractional Factor

You can convert the broadcast quality signal at 32 kHz back to 48 kHz with a fractional interpolator, perhaps to store it on a digital audio tape (DAT). Moving from 32 to 48 requires upsampling by 50 percent, achieved using an interpolation factor of 3 and decimation by 2. Again, you use the fractional FIR interpolator.

```
hfi = mfilter.firfracinterp(3,2);  
s48 = filter(hfi,s32);
```

Listening to the up-converted audio might be interesting. Use an audio player again.

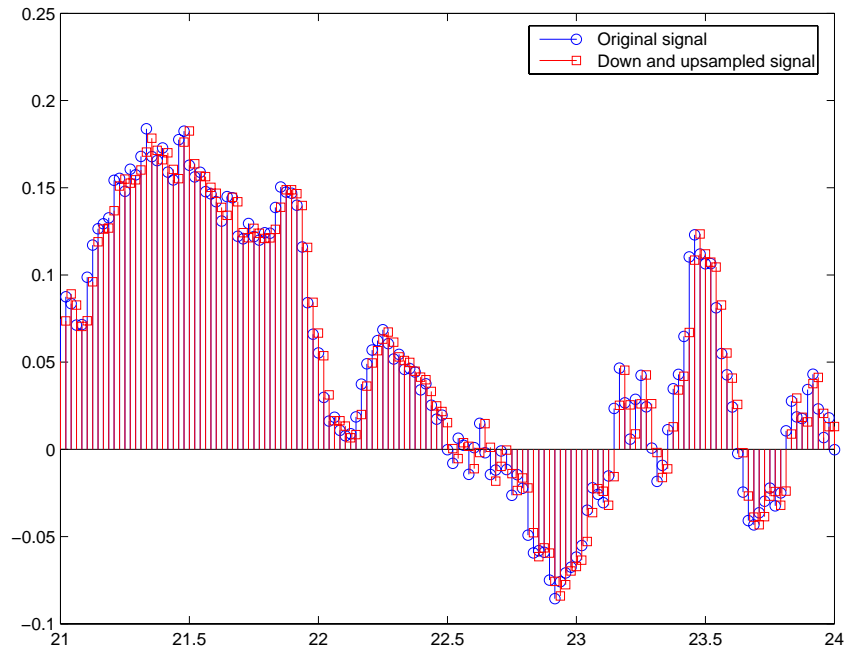
```
ps48 = audioplayer(s48,Fs48);  
play(ps48);
```

## Plotting the Original Signal and the Reconverted Signal

To compare both 48 kHz signals—the original and the twice-converted signal, you must account for the delay introduced by both the fractional decimation and the fractional interpolation processes when you converted the signal down to 32 kHz and back to 48 kHz. In the stem plot shown here, notice that most of the reconverted signal samples have moved slightly from where they were originally. This is distortion introduced by converting down to 32 kHz by decimation and then converting back up to 48 kHz by interpolation.

```
figure;  
xindx = 1000:1500;  
stem(xindx/Fs48*1e3,signal48kHz(xindx));  
hold on;  
stem(xindx/Fs48*1e3,s48(1037:1537),'r'); % Account for the  
                                           % process-induced  
                                           % delays.
```





Different filters achieve different results. You used the default filters which do not optimize the output.

## Converting from 48 kHz to 44.1 kHz

To convert from studio quality audio at 48 kHz to CD quality audio, 44.1 kHz, you would use a multirate filter better suited for this ratio change (interpolation factor of 147, decimation factor of 160; decimation by 1.088). To avoid the startup delay (latency) introduced by the filter, preload half of the filter states with the beginning of the signal. Doing this step compensates for the delay caused by filtering and decimation. For this rate change, you use the FIR sample rate change multirate filter—`firsrc`.

```
hsrc = mfilt.firsrc(147,160) % Use default filter coefficients.
hsrc =
```

```
FilterStructure: 'Direct-Form FIR Polyphase Sample-Rate Converter'  
  Numerator: [1x3840 double]  
RateChangeFactors: [147 160]  
PersistentMemory: true  
  States: [26x1 double]
```

```
hsrc.persistentmemory = true;    % Allows you to set the states  
                                % to eliminate delay.  
hsrc.States(13:-1:1) = signal48kHz(1:13); % Preload the states.  
s441 = filter(hsrc,signal48kHz(14:end)); % This takes a few  
                                         % seconds.
```

Again, you can play the down-converted signal at 44.1 kHz with a MATLAB audio player.

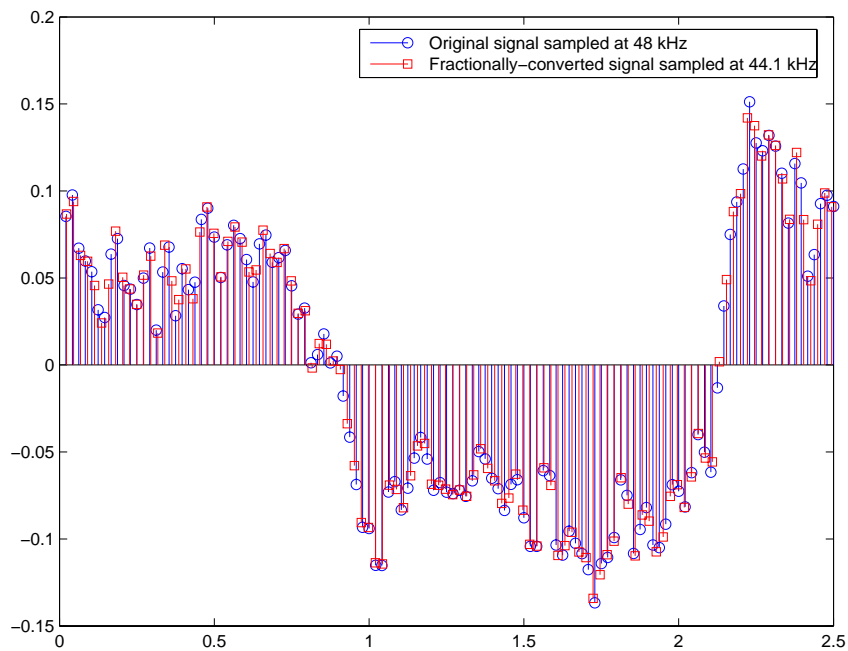
```
p441 = audioplayer(s441,44.1e3);  
play(p441);
```

When you are doing sample-rate conversion with large values of  $L$  or  $M$ , as you are in this case where  $L=147$  and  $M=160$ , 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.

## Plotting the 48 kHz Signal and the 44.1 kHz Signal

Now compare segments of the two signals graphically. In this case you can verify visually in the stem plot shown that the filter does not introduce delay since you compensated for its group delay by preloading the states.

```
figure  
xindx = 1:160;  
stem(xindx/Fs48*1e3,signal48kHz(xindx));  
hold on  
xindx2 = 1:147;  
stem(xindx2/44.1,s441(xindx2),'r');
```





# Designing Adaptive Filters

---

Introducing Adaptive Filtering (p. 4-2)	Provides a little background on the development of adaptive filters and the contents of this section
Getting Started with Adaptive Filters (p. 4-4)	Uses a signal enhancement application to introduce adaptive filters
Overview of Adaptive Filters and Applications (p. 4-14)	Provides a short discussion about adaptive filters and their uses
System Identification (p. 4-17)	Learn about the questions to ask when you need an adaptive filter
Adaptive Filters in the Filter Design Toolbox (p. 4-21)	Learn about the adaptive filter objects provided in the toolbox
Examples of Adaptive Filters That Use LMS Algorithms (p. 4-26)	Presents examples of adaptive filters that use LMS algorithms to determine filter coefficients
Example of Adaptive Filter That Uses RLS Algorithm (p. 4-47)	Presents examples of adaptive filters that use RLS algorithms to determine filter coefficients
Selected Bibliography (p. 4-52)	Lists a few books that cover adaptive filters in both detail and with broad scope

# Introducing Adaptive Filtering

Over the past three decades, digital signal processors have made great advances in increasing speed and complexity, and reducing power consumption. As a direct result, real-time adaptive filtering is quickly becoming essential for the future of communications, both wired and wireless.

In the following sections, this guide presents an overview of adaptive filtering; discussions of some of the common applications for adaptive filters; and details about the adaptive filters available in the toolbox.

Listed below are the sections that cover adaptive filters in this guide. Within each section, examples and a short discussion of the theory of the filters introduce the adaptive filter concepts.

- “Getting Started with Adaptive Filters” on page 4-4 introduces adaptive filtering through a worked example.
- “Overview of Adaptive Filters and Applications” on page 4-14 presents a general discussion of adaptive filters and their applications.
  - “System Identification” on page 4-17—Using adaptive filters to identify the response of an unknown system such as a communications channel or a telephone line.
  - “Inverse System Identification” on page 4-18—Using adaptive filters to develop a filter that has a response that is the inverse of an unknown system.
  - “Noise Cancellation (or Interference Cancellation)” on page 4-18—Performing active noise cancellation where the filter adapts in real-time to remove noise by keeping the error small.
  - “Prediction” on page 4-19—describes using adaptive filters to predict a signal’s future values.
- “System Identification” on page 4-17 describes the important considerations for selecting an adaptive filter for an application.
- “Adaptive Filters in the Filter Design Toolbox” on page 4-21 lists the adaptive filters included in the toolbox.
- “Examples of Adaptive Filters That Use LMS Algorithms” on page 4-26 presents a discussion of using LMS techniques to perform the filter adaptation process.

- “Example of Adaptive Filter That Uses RLS Algorithm” on page 4-47 discusses adaptive filters based on the RMS techniques for minimizing the total error between the known and unknown systems.

For more detailed information about adaptive filters and adaptive filter theory, refer to the books listed in “Selected Bibliography” on page 4-52.

# Getting Started with Adaptive Filters

This demonstration illustrates one way to use a few of the adaptive filter algorithms provided in the toolbox.

This example uses a signal enhancement application as an illustration. While there are about 30 different adaptive filtering algorithms included with the toolbox, this example demonstrates two algorithms—least means square (LMS), `adaptfilt.lms`, and normalized LMS, `adaptfilt.nlms`, for adaptation.

## Tutorial Contents

As you follow this tutorial, you encounter these subjects.

- “Create the Signals for Adaptation” on page 4-4
- “Construct Two Adaptive Filters” on page 4-5
- “Choose the Step Size” on page 4-6
- “Set the Adapting Filter Step Size” on page 4-7
- “Filter with the Adaptive Filters” on page 4-7
- “Compute the Optimal Solution” on page 4-8
- “Plot the Results” on page 4-8
- “Compare the Final Coefficients” on page 4-9
- “Reset the Filter Before Filtering” on page 4-10
- “Compute the Learning Curves” on page 4-11
- “Compute the Theoretical Learning Curves” on page 4-12

## Create the Signals for Adaptation

The goal is to use an adaptive filter to extract a desired signal from a noise-corrupted signal by filtering out the noise. The desired signal (the output from the process) is a sinusoid with 1000 samples.

```
n = (1:1000)';  
s = sin(0.075*pi*n);
```

To perform adaptation requires two signals:



- a reference signal
- a noisy signal that contains both the desired signal and an added noise component.

### Generate the Noise Signal

To create a noise signal, assume that the noise  $v_1$  is autoregressive, meaning that the value of the noise at time  $t$  depends only on its previous values and on a random disturbance.

```
v = 0.8*randn(1000,1); % Random noise part.
ar = [1, 1/2];        % Autoregression coefficients.
v1 = filter(1,ar,v);  % Noise signal. Applies a 1-D digital
                    % filter.
```

### Corrupt the Desired Signal to Create a Noisy Signal

To generate the noisy signal that contains both the desired signal and the noise, add the noise signal  $v_1$  to the desired signal  $s$ . The noise corrupted sinusoid  $x$  is

```
x = s + v1;
```

where  $s$  is the desired signal the the noise is  $v_1$ . Adaptive filter processing seeks to recover  $s$  from  $x$ . To complete the signals needed to perform adaptive filtering, the process requires a reference signal.

### Create a Reference Signal

Define a moving average signal  $v_2$  that is correlated with  $v_1$ . This  $v_2$  is the reference signal for the examples.

```
ma = [1, -0.8, 0.4, -0.2];
v2 = filter(ma,1,v);
```

### Construct Two Adaptive Filters

Two similar adaptive filters—LMS and NLMS—form the basis of this example, both sixth order. Set the order as a variable in MATLAB and create the filters.

```
l = 7; % Seven taps or weights. Order equals 6.
halms=adaptfilt.lms(l)
```

```
halms =
```

```
        Algorithm: 'Direct-Form FIR LMS Adaptive Filter'
        FilterLength: 7
        StepSize: 0.1
        Leakage: 1
        PersistentMemory: false

hanlms=adaptfilt.nlms(1)

hanlms =

        Algorithm: 'Direct-Form FIR Normalized LMS Adaptive Filter'
        FilterLength: 7
        StepSize: 1

        Leakage: 1
        Offset: 0
        PersistentMemory: false
```

### Choose the Step Size

LMS-like algorithms have a step size that determines the amount of correction applied as the filter adapts from one iteration to the next. Choosing the appropriate step size is not always easy, usually requiring experience in adaptive filter design.

- A step size that is too small increases the time for the filter to converge on a set of coefficients. This becomes an issue of speed and accuracy.
- One that is too large may cause the adapting filter to diverge, never reaching convergence. In this case, the issue is stability—the resulting filter might not be stable.

As a rule of thumb, smaller step sizes improve the accuracy of the convergence of the filter to match the characteristics of the unknown, at the expense of the time it takes to adapt.

The toolbox includes an algorithm—`maxstep`—to determine the maximum step size suitable for each LMS adaptive filter algorithm that still ensures that the filter converges to a solution. Often, the notation for the step size is  $\mu$ .

```
[mumaxlms,mumaxmselms] = maxstep(halms,x)
[mumaxnlms,mumaxmsenlms] = maxstep(hanlms) % Always equal to 2.
```

Warning: Step size is not in the range  $0 < \mu < \text{mumaxmse}/2$ :

Erratic behavior might result.

```
mumaxlms =
```

```
0.2270
```

```
mumaxmselms =
```

```
0.1356
```

```
mumaxnlms =
```

```
2
```

```
mumaxmsenlms =
```

```
2
```

## Set the Adapting Filter Step Size

The first output of `maxstep` is the value needed for the mean of the coefficients to converge while the second is the value needed for the mean squared coefficients to converge. Choosing a large step size often causes large variations from the convergence values, so choose smaller step sizes generally.

```
halms.stepsize = mumaxmselms/30; % You can set this graphically.
inspect(halms) % Opens the Property Inspector in MATLAB.
hanlms.stepsize = mumaxmsenlms/20;
inspect(hanlms)
```

If you know the step size to use, set the step size value when you create the filter with the `step` input argument.

```
halms = adaptfilt.lms(n,step); Adds the step input argument.
```

## Filter with the Adaptive Filters

Now you have set up the parameters of the adaptive filters and are ready to filter the noisy signal. The reference signal, `v2` is the input to the adaptive filters, while `x` is the desired signal in this configuration.

Through adaptation  $y$ , the output of the filters, tries to emulate  $x$  as closely as possible.

Since  $v_2$  is correlated only with the noise component  $v_1$  of  $x$ , it can only really emulate  $v_1$ . The error signal, the desired  $x$ , minus the actual output  $y$ , constitutes an estimate of the part of  $x$  that is not correlated with  $v_2$  —  $s$ , the signal to extract from  $x$ .

```
[ylms,elms] = filter(hlms,v2,x);  
[ynlms,enlms] = filter(hnlms,v2,x);
```

### Compute the Optimal Solution

For comparison, compute the optimal FIR Wiener filter.

```
filterbw = firwiener(L-1,v2,x); % Optimal FIR Wiener.  
filteryw = filter(bw,1,v2);    % Estimate of x using Wiener.  
filterew = x-yw;              % Estimate of actual sinusoid.
```

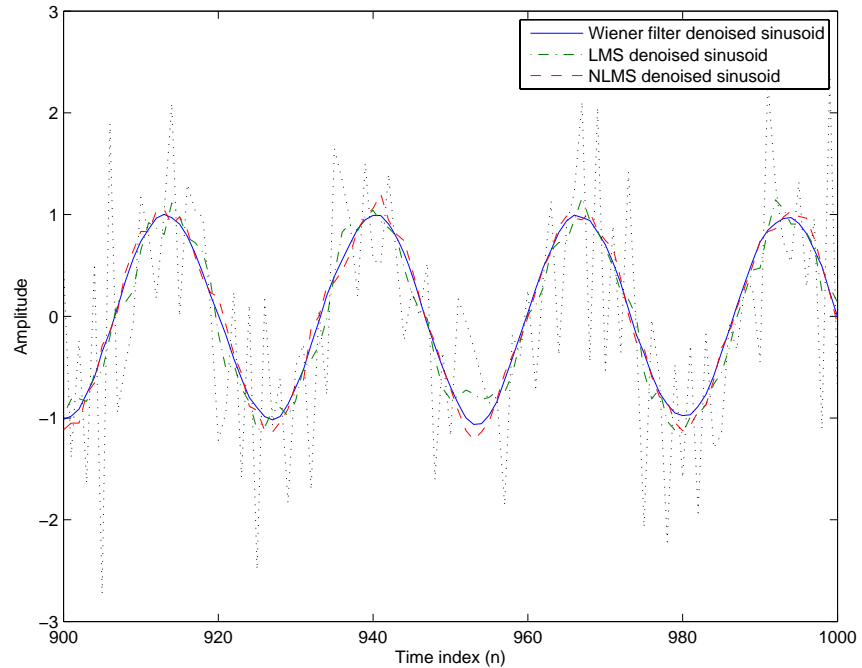
### Plot the Results

Plot the resulting denoised sinusoid for each filter—the Wiener filter, the LMS adaptive filter, and the NLMS adaptive filter—to compare the performance of the various techniques.

```
plot(n(900:end),[ew(900:end), elms(900:end),enlms(900:end)]);  
legend('Wiener filter denoised sinusoid','LMS denoised...  
sinusoid', 'NLMS denoised sinusoid');
```

As a reference point, include the noisy signal as a dotted line in the plot.

```
hold on  
plot(n(900:end),x(900:end),'k:')  
hold off
```



## Compare the Final Coefficients

Finally, compare the Wiener filter coefficients with the coefficients of the adaptive filters. While adapting, the adaptive filters try to converge to the Wiener coefficients.

```
[bw.' hlms.Coefficients.' hnllms.Coefficients.']
ans =
```

1.0221	0.8751	1.0411
0.3345	0.1201	0.3601
0.1217	-0.0118	0.1077
0.0483	-0.0183	0.0081
0.1179	0.0558	0.0420
0.0637	-0.0049	-0.0290

```
0.0216   -0.0235   -0.0222
```

### Reset the Filter Before Filtering

Adaptive filters have a `PersistentMemory` property that you can use to reproduce experiments exactly. By default, the `PersistentMemory` is `false`. The states and the coefficients of the filter are reset before filtering and the filter does not remember the results from previous times you use the filter.

For instance, the following successive calls produce the same output when `PersistentMemory` is `false`.

```
[ylms,elms] = filter(hlms,v2,x);  
[ylms2,elms2] = filter(hlms,v2,x);
```

To keep the history of the filter when filtering a new set of data, enable persistent memory for the filter by setting the `PersistentMemory` property to `true`. In this configuration, the filter uses the final states and coefficients from the previous run as the initial conditions for the next run and set of data.

```
[ylms,elms] = filter(hlms,v2,x);  
hlms.PersistentMemory = true;  
[ylms2,elms2] = filter(hlms,v2,x); % No longer the same.
```

Setting the property value to `true` is useful when you are filtering large amounts of data that you partition into smaller sets and then feed into the filter using a for-loop construction.

### Investigate Convergence Through Learning Curves

To analyze the convergence of the adaptive filters, look at the learning curves. The toolbox provides methods to generate the learning curves, but you need more than one iteration of the experiment to obtain significant results.

This demonstration uses 25 sample realizations of the noisy sinusoids.

```
n = (1:5000)';  
s = sin(0.075*pi*n);  
nr = 25;  
v = 0.8*randn(5000,nr);  
v1 = filter(1,ar,v);  
x = repmat(s,1,nr) + v1;  
v2 = filter(ma,1,v);
```

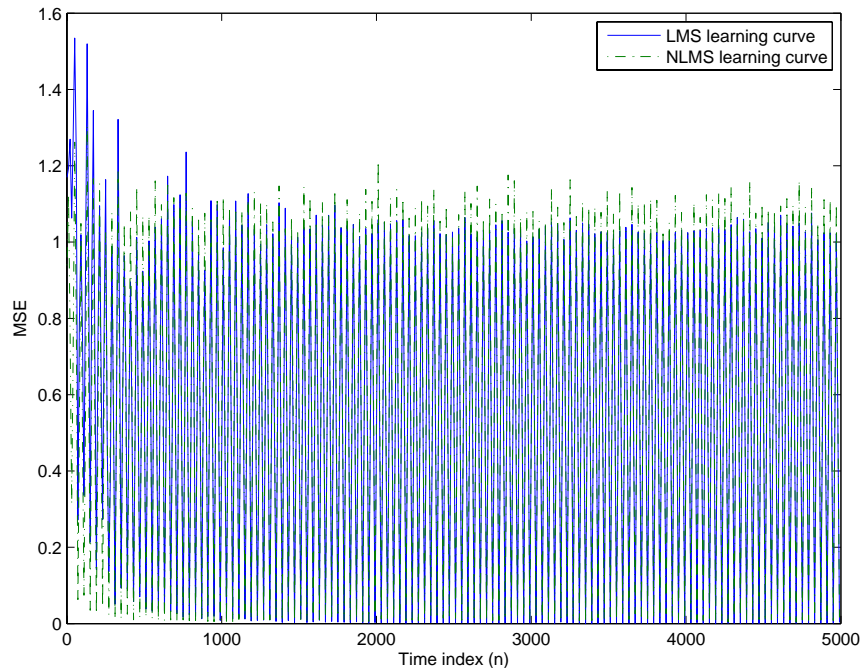
### Compute the Learning Curves

Now compute the mean-square error. To speed things up, compute the error every 10 samples.

First, reset the adaptive filters to avoid using the coefficients it has already computed and the states it has stored.

```
reset(hlms);  
reset(hnlms);  
M = 10; % Decimation factor.  
mse1ms = msesim(hlms,v2,x,M);  
mse1ms = msesim(hnlms,v2,x,M);  
plot(1:M:n(end),[mse1ms,mse1ms])legend('LMS learning...  
curve','NLMS learning curve')
```

In the next plot you see the calculated learning curves for the LMS and NLMS adaptive filters.



## Compute the Theoretical Learning Curves

For the LMS and NLMS algorithms, functions in the toolbox help you compute the theoretical learning curves, along with the minimum mean-square error (MMSE) the excess mean-square error (EMSE) and the mean value of the coefficients.

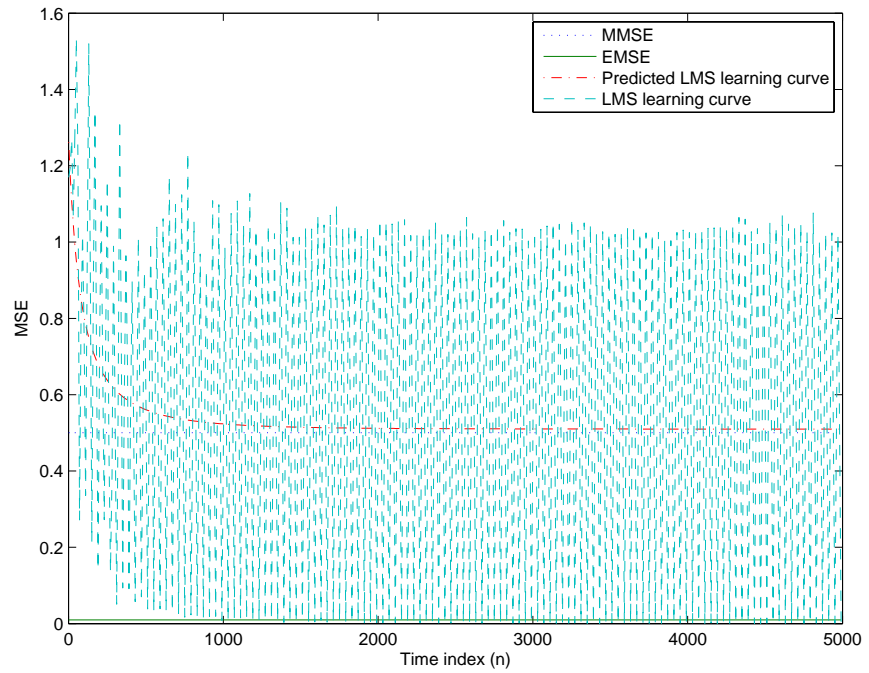
MATLAB may take some time to calculate the curves. The figure shown after the code plots the predicted and actual LMS curves.

```

reset(hlms);
[mmse1ms,emse1ms,meanw1ms,pmse1ms] = msepred(hlms,v2,x,M);
plot(1:M:n(end),[mmse1ms*ones(500,1),emse1ms*ones(500,1),...
pmse1ms,mse1ms])
legend('MMSE','EMSE','Predicted LMS learning curve',...
'LMS learning curve')

```

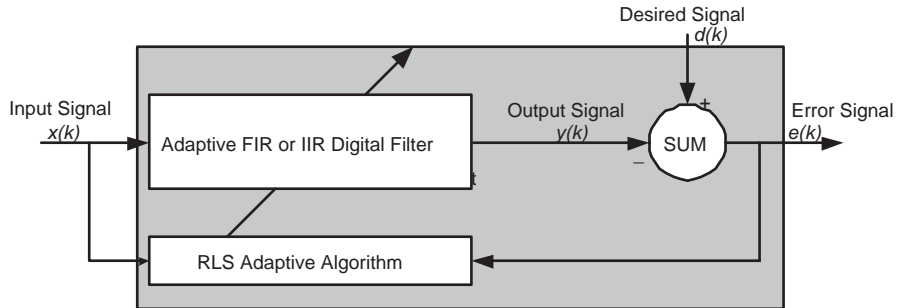




## Overview of Adaptive Filters and Applications

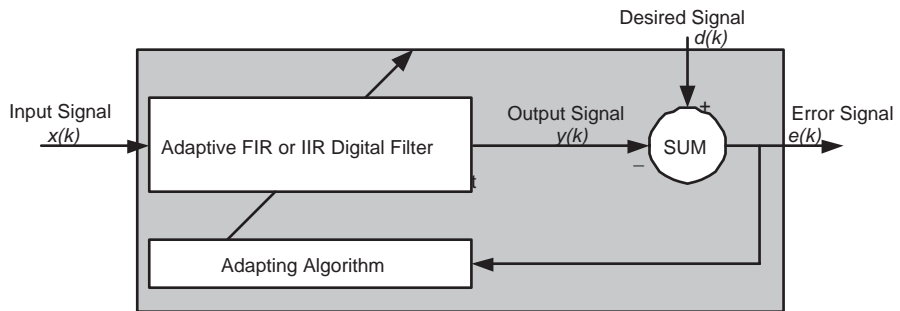
This section presents a brief description of how adaptive filters work and some of the applications where they can be useful.

Adaptive filters self learn. As the signal into the filter continues, the adaptive filter coefficients adjust themselves to achieve the desired result, such as identifying an unknown filter or canceling noise in the input signal. In the figure below, the shaded box represents the adaptive filter, comprising the adaptive filter and the adaptive recursive least squares (RLS) algorithm.



**Block Diagram That Defines the Inputs and Output of a Generic RLS Adaptive Filter**

The next figure provides the general adaptive filter setup with inputs and outputs.



**Block Diagram Defining General Adaptive Filter Algorithm Inputs and Outputs**

Filter Design Toolbox includes adaptive filters of a broad range of forms, all of which can be worthwhile for specific needs. Some of the common ones are:

- Adaptive filters based on least mean squares (LMS) techniques, such as `adaptfilt.lms`, `adaptfilt.filtxlms`, and `adaptfilt.nlms`
- Adaptive filters based on recursive least squares (RLS) techniques. For example, `adaptfilt.rls` and `adaptfilt.swrls`
- Adaptive filters based on sign-data (`adaptfilt.sd`), sign-error (`adaptfilt.se`), and sign-sign (`adaptfilt.ss`) techniques
- Adaptive filters based on lattice filters. For example, `adaptfilt.gal` and `adaptfilt.lsl`
- Adaptive filters that operate in the frequency domain, such as `adaptfilt.fdaf` and `adaptfilt.pbufdaf`.
- Adaptive filters that operate in the transform domain. Two of these are the `adaptfilt.tdafdft` and `adaptfilt.tdafdct` filters

An adaptive filter designs itself based on the characteristics of the input signal to the filter and a signal that represents the desired behavior of the filter on its input.

Designing the filter does not require any other frequency response information or specification. To define the self-learning process the filter uses, you select the adaptive algorithm used to reduce the error between the output signal  $y(k)$  and the desired signal  $d(k)$ .

When the LMS performance criterion for  $e(k)$  has achieved its minimum value through the iterations of the adapting algorithm, the adaptive filter is finished and its coefficients have converged to a solution. Now the output from the adaptive filter matches closely the desired signal  $d(k)$ . When you change the input data characteristics, sometimes called the *filter environment*, the filter adapts to the new environment by generating a new set of coefficients for the new data. Notice that when  $e(k)$  goes to zero and remains there you achieve perfect adaptation, the ideal result but not likely in the real world.

The adaptive filter functions in this toolbox implement the shaded portion of the figures, replacing the adaptive algorithm with an appropriate technique. To use one of the functions, you provide the input signal or signals and the initial values for the filter.

“Adaptive Filters in the Filter Design Toolbox” on page 4-21 offers details about the algorithms available and the inputs required to use them in MATLAB.

### Choosing an Adaptive Filter

Selecting the adaptive filter that best meets your needs requires careful consideration. An exhaustive discussion of the criteria for selecting your approach is beyond the scope of this User’s Guide. However, a few guidelines can help you make your choice.

Two main considerations frame the decision—how you plan to use the filter and the filter algorithm to use.

When you begin to develop an adaptive filter for your needs, most likely the primary concern is whether using an adaptive filter is a cost-competitive approach to solving your filtering needs. Generally many areas determine the suitability of adaptive filters (these areas are common to most filtering and signal processing applications). Four such areas are

- **Filter consistency**—Does your filter performance degrade when the filter coefficients change slightly as a result of quantization, or you switch to fixed-point arithmetic? Will excessive noise in the signal hurt the performance of your filter?
- **Filter performance**—Does your adaptive filter provide sufficient identification accuracy or fidelity, or does the filter provide sufficient signal discrimination or noise cancellation to meet your requirements?
- **Tools**—Do tools exist that make your filter development process easier? Better tools can make it practical to use more complex adaptive algorithms.
- **DSP requirements**—Can your filter perform its job within the constraints of your application? Does your processor have sufficient memory, throughput, and time to use your proposed adaptive filtering approach? Can you trade memory for throughput: use more memory to reduce the throughput requirements or use a faster signal processor?

Of the preceding considerations, characterizing filter consistency or robustness may be the most difficult.

The simulations in the Filter Design Toolbox offers a good first step in developing and studying these issues. LMS algorithm filters provide both

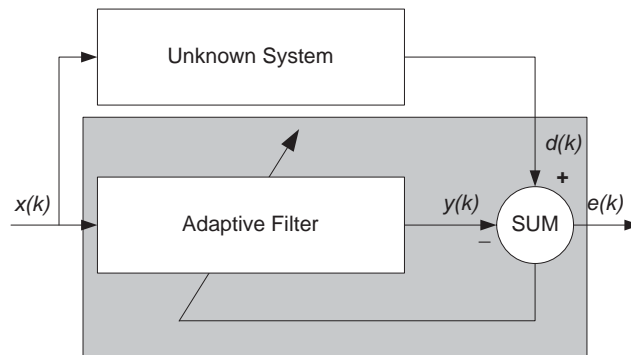
a relatively straightforward filters to implement and sufficiently powerful tool for evaluating whether adaptive filtering can be useful for your problem.

Additionally, starting with an LMS approach can form a solid baseline against which you can study and compare the more complex adaptive filters available in the toolbox. Finally, your development process should, at some time, test your algorithm and adaptive filter with real data. For truly testing the value of your work there is no substitute for actual data.

## System Identification

One common adaptive filter application is to use adaptive filters to identify an unknown system, such as the response of an unknown communications channel or the frequency response of an auditorium, to pick fairly divergent applications. Other applications include echo cancellation and channel identification.

In the figure, the unknown system is placed in parallel with the adaptive filter. This layout represents just one of many possible structures. The shaded area contains the adaptive filter system.

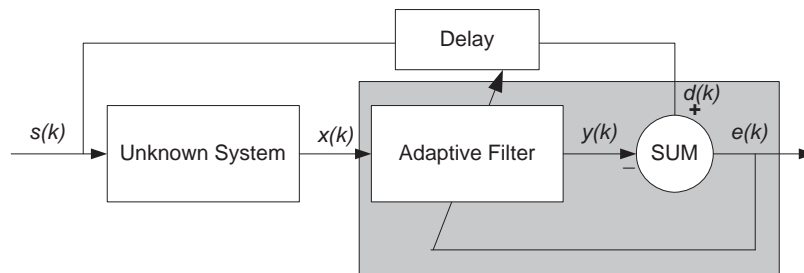


### Using an Adaptive Filter to Identify an Unknown System

Clearly, when  $e(k)$  is very small, the adaptive filter response is close to the response of the unknown system. In this case the same input feeds both the adaptive filter and the unknown. If, for example, the unknown system is a modem, the input often represents white noise, and is a part of the sound you hear from your modem when you log in to your Internet service provider.

## Inverse System Identification

By placing the unknown system in series with your adaptive filter, your filter adapts to become the inverse of the unknown system as  $e(k)$  becomes very small. As shown in the figure the process requires a delay inserted in the desired signal  $d(k)$  path to keep the data at the summation synchronized. Adding the delay keeps the system causal.



### Determining an Inverse Response to an Unknown System

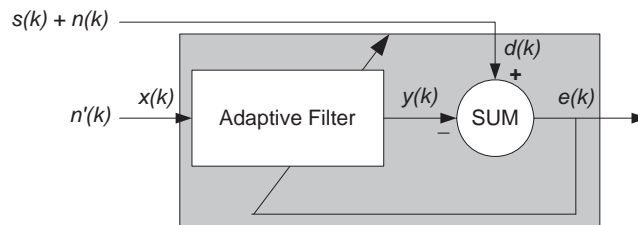
Including the delay to account for the delay caused by the unknown system prevents this condition.

Plain old telephone systems (POTS) commonly use inverse system identification to compensate for the copper transmission medium. When you send data or voice over telephone lines, the copper wires behave like a filter, having a response that rolls off at higher frequencies (or data rates) and having other anomalies as well.

Adding an adaptive filter that has a response that is the inverse of the wire response, and configuring the filter to adapt in real time, lets the filter compensate for the rolloff and anomalies, increasing the available frequency output range and data rate for the telephone system.

## Noise Cancellation (or Interference Cancellation)

In noise cancellation, adaptive filters let you remove noise from a signal in real time. Here, the desired signal, the one to clean up, combines noise and desired information. To remove the noise, feed a signal  $n'(k)$  to the adaptive filter that represents noise that is correlated to the noise to remove from the desired signal.

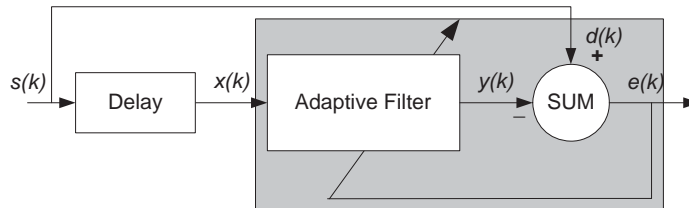


### Using an Adaptive Filter to Remove Noise from an Unknown System

So long as the input noise to the filter remains correlated to the unwanted noise accompanying the desired signal, the adaptive filter adjusts its coefficients to reduce the value of the difference between  $y(k)$  and  $d(k)$ , removing the noise and resulting in a clean signal in  $e(k)$ . Notice that in this application, the error signal actually converges to the input data signal, rather than converging to zero.

### Prediction

Predicting signals requires that you make some key assumptions. Assume that the signal is either steady or slowly varying over time, and periodic over time as well.



### Predicting Future Values of a Periodic Signal

Accepting these assumptions, the adaptive filter must predict the future values of the desired signal based on past values. When  $s(k)$  is periodic and the filter is long enough to remember previous values, this structure with the delay in the input signal, can perform the prediction. You might use this structure to remove a periodic signal from stochastic noise signals.

Finally, notice that most systems of interest contain elements of more than one of the four adaptive filter structures. Carefully reviewing the real structure may be required to determine what the adaptive filter is adapting to.

Also, for clarity in the figures, the analog-to-digital (A/D) and digital-to-analog (D/A) components do not appear. Since the adaptive filters are assumed to be digital in nature, and many of the problems produce analog data, converting the input signals to and from the analog domain is probably necessary.



## Adaptive Filters in the Filter Design Toolbox

Filter Design Toolbox contains many objects for constructing and applying adaptive filters to data. As you see in the tables in the next section, the objects use various algorithms to determine the weights for the filter coefficients of the adapting filter. While the algorithms differ in their detail implementations, the LMS and RLS share a common operational approach—minimizing the error between the filter output and the desired signal.

### Algorithms

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

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

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

<b>Adaptive Filter Method</b>	<b>Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</b>
<code>adaptfilt.adj1ms</code>	Adjoint LMS FIR adaptive filter algorithm
<code>adaptfilt.blms</code>	Block LMS FIR adaptive filter algorithm
<code>adaptfilt.blmsfft</code>	FFT-based Block LMS FIR adaptive filter algorithm
<code>adaptfilt.dlms</code>	Delayed LMS FIR adaptive filter algorithm
<code>adaptfilt.filtxlms</code>	Filtered-x LMS FIR adaptive filter algorithm
<code>adaptfilt.lms</code>	LMS FIR adaptive filter algorithm
<code>adaptfilt.nlms</code>	Normalized LMS FIR adaptive filter algorithm
<code>adaptfilt.sd</code>	Sign-data LMS FIR adaptive filter algorithm
<code>adaptfilt.se</code>	Sign-error LMS FIR adaptive filter algorithm
<code>adaptfilt.ss</code>	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>Adaptive Filter Method</b>	<b>Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</b>
adaptfilt.ftf	Fast transversal least-squares adaptation algorithm
adaptfilt.qrdrls	QR-decomposition RLS adaptation algorithm
adaptfilt.hr1s	Householder RLS adaptation algorithm
adaptfilt.hswrls	Householder SWRLS adaptation algorithm
adaptfilt.rls	Recursive-least squares (RLS) adaptation algorithm
adaptfilt.swrls	Sliding window (SW) RLS adaptation algorithm
adaptfilt.swftf	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>Adaptive Filter Method</b>	<b>Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</b>
adaptfilt.ap	Affine projection algorithm that uses direct matrix inversion
adaptfilt.apru	Affine projection algorithm that uses recursive matrix updating
adaptfilt.bap	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>Adaptive Filter Method</b>	<b>Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</b>
<code>adaptfilt.fdaf</code>	Frequency domain adaptation algorithm
<code>adaptfilt.pbfdaf</code>	Partition block version of the FDAF algorithm
<code>adaptfilt.pbufdaf</code>	Partition block unconstrained version of the FDAF algorithm
<code>adaptfilt.tdafdct</code>	Transform domain adaptation algorithm using DCT
<code>adaptfilt.tdafdft</code>	Transform domain adaptation algorithm using DFT
<code>adaptfilt.ufdaf</code>	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>Adaptive Filter Method</b>	<b>Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</b>
<code>adaptfilt.gal</code>	Gradient adaptive lattice filter adaptation algorithm
<code>adaptfilt.lsl</code>	Least squares lattice adaptation algorithm
<code>adaptfilt.qrdls1</code>	QR decomposition RLS adaptation algorithm

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

Presenting a detailed derivation of the Wiener-Hopf equation and determining solutions to it is beyond the scope of this *User's Guide*. Full descriptions of the

theory appear in the adaptive filter references provided in the “Selected Bibliography” on page 4-52.

## **Using Adaptive Filter Objects**

After you construct an adaptive filter object, how do you apply it to your data or system? Like quantizer objects, adaptive filter objects have a `filter` method that you use to apply the `adaptfilt` object to data. In the following sections, various examples of using LMS and RLS adaptive filters show you how `filter` works with the objects to apply them to data.

- “Examples of Adaptive Filters That Use LMS Algorithms” on page 4-26
- “Example of Adaptive Filter That Uses RLS Algorithm” on page 4-47

## Examples of Adaptive Filters That Use LMS Algorithms

This section provides introductory examples using some of the least mean squares (LMS) adaptive filter functions in the toolbox.

The toolbox provides many adaptive filter design functions that use the LMS algorithms to search for the optimal solution to the adaptive filter, including

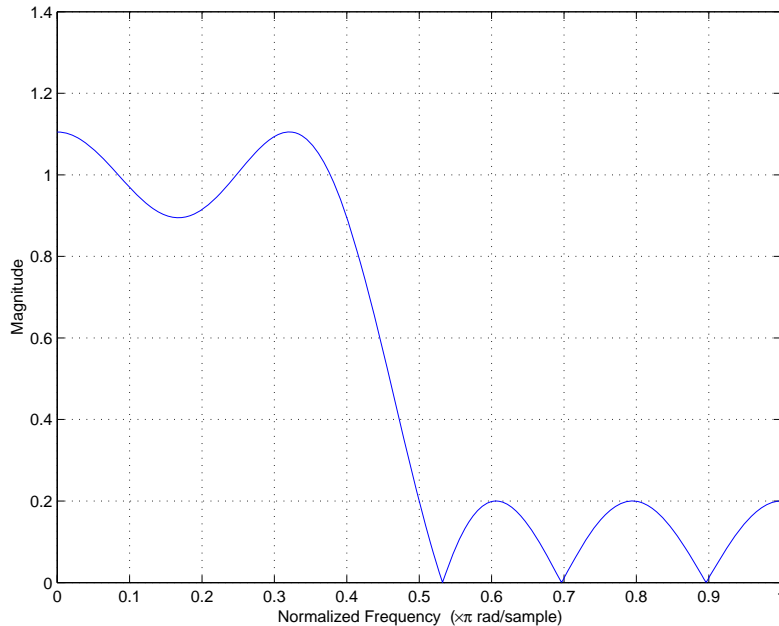
- `adaptfilt.lms`—Implement the LMS algorithm to solve the Wiener-Hopf equation and find the filter coefficients for an adaptive filter.
- `adaptfilt.nlms`—implement the normalized variation of the LMS algorithm to solve the Wiener-Hopf equation and determine the filter coefficients of an adaptive filter.
- `adaptfilt.sd`—Implement the sign-data variation of the LMS algorithm to solve the Wiener-Hopf equation and determine the filter coefficients of an adaptive filter. The correction to the filter weights at each iteration depends on the sign of the input  $x(k)$ .
- `adaptfilt.se`—Implement the sign-error variation of the LMS algorithm to solve the Wiener-Hopf equation and determine the filter coefficients of an adaptive filter. The correction applied to the current filter weights for each successive iteration depends on the sign of the error,  $e(k)$ .
- `adaptfilt.ss`—Implement the sign-sign variation of the LMS algorithm to solve the Wiener-Hopf equation and determine the filter coefficients of an adaptive filter. The correction applied to the current filter weights for each successive iteration depends on both the sign of  $x(k)$  and the sign of  $e(k)$ .

To demonstrate the differences and similarities among the various LMS algorithms supplied in the toolbox, the LMS and NLMS adaptive filter examples use the same filter for the unknown system. The unknown filter is the constrained lowpass filter from “firgr and firband Examples” on page 2-9.

```
[b,err,res]=firgr(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...  
{'w' 'c'});
```

From the figure you see that the filter is indeed lowpass and constrained to 0.2 ripple in the stopband. With this as the baseline, the adaptive LMS filter examples use the adaptive LMS algorithms and their initialization functions to identify this filter in a system identification role.

To review the general model for system ID mode, look at “System Identification” on page 4-17 for the layout.



For the sign variations of the LMS algorithm, the examples use noise cancellation as the demonstration application, as opposed to the system identification application used in the LMS examples.

### **adaptfilt.lms Example—System Identification**

To use the adaptive filter functions in the toolbox you need to provide three things:

- The adaptive LMS function to use. This example uses the LMS adaptive filter function `adaptfilt.lms`.
- An unknown system or process to adapt to. In this example, the filter designed by `firgr` is the unknown system.

- Appropriate input data to exercise the adaptation process. In terms of the generic LMS model, these are the desired signal  $d(k)$  and the input signal  $x(k)$ .

Start by defining an input signal  $x$ .

```
x = 0.1*randn(1,250);
```

The input is broadband noise. For the unknown system filter, use `firgr` to create a twelfth-order lowpass filter:

```
[b,err,res] = firgr(22,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...  
{'w' 'c'});
```

Although you do not need them here, include the `err` and `res` output arguments.

Now filter the signal through the unknown system to get the desired signal.

```
d = filter(b,1,x);
```

With the unknown filter designed and the desired signal in place you construct and apply the adaptive LMS filter object to identify the unknown.

Preparing the adaptive filter object requires that you provide starting values for estimates of the filter coefficients and the LMS step size. You could start with estimated coefficients of some set of nonzero values; this example uses zeros for the 12 initial filter weights.

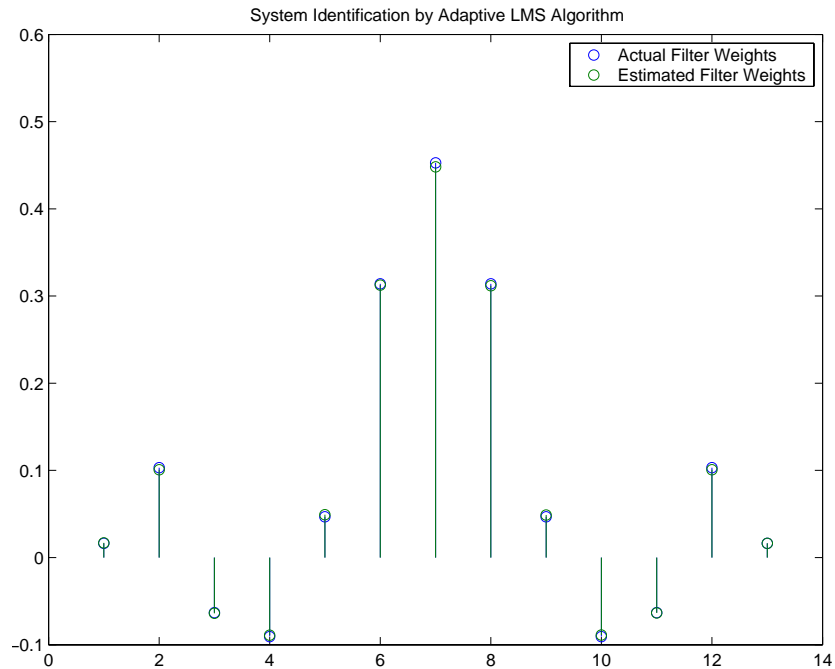
For the step size, 0.8 is a reasonable value—a good compromise between being large enough to converge well within the 250 iterations (250 input sample points) and small enough to create an accurate estimate of the unknown filter.

```
mu = 0.8;  
ha = adaptfilt.lms(13,mu,w0)
```

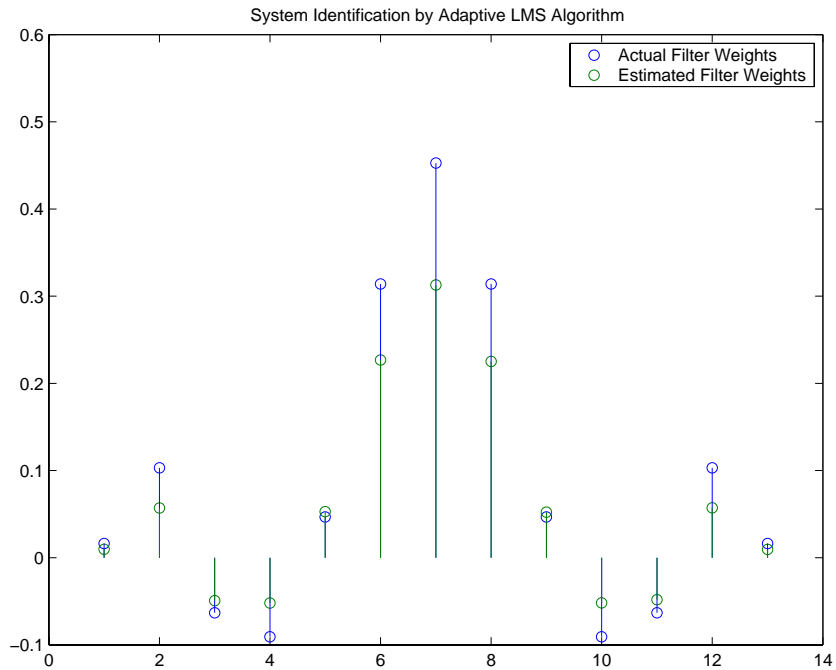
Finally, using the `adaptfilt` object `ha`, desired signal, `d`, and the input to the filter, `x`, run the adaptive filter to determine the unknown system and plot the results, comparing the actual coefficients from `firgr` to the coefficients found by `adaptlms`.

```
[y,e] = filter(ha,x,d);  
stem([b.' ha.coefficients.'])
```

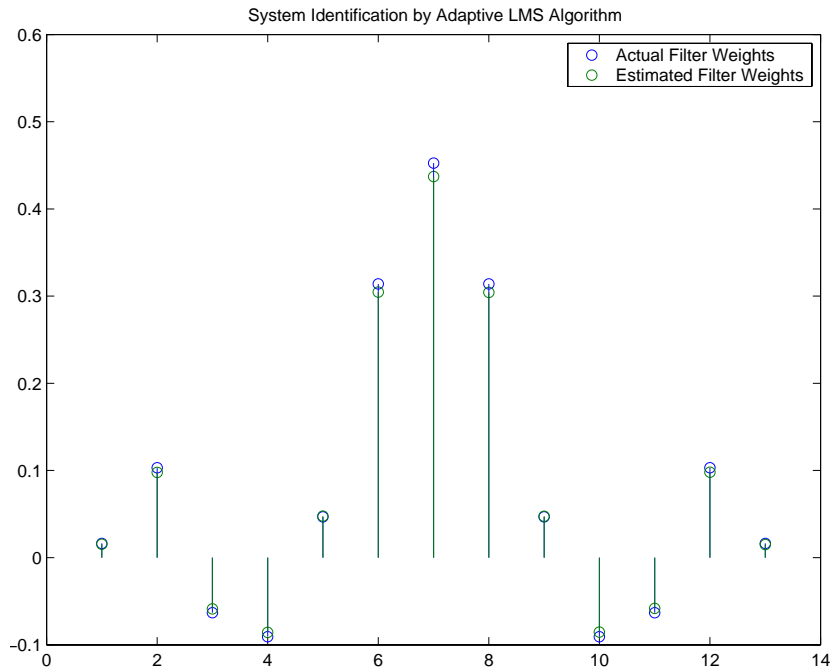




In the stem plot the actual and estimated filter weights are the same. As an experiment, try changing the step size to 0.2. Repeating the example with  $\mu = 0.2$  results in the following stem plot. The estimated weights fail to approximate the actual weights closely.



Since this may be because you did not iterate over the LMS algorithm enough times, try using 1000 samples. With 1000 samples, the stem plot, shown in the next figure, looks much better, albeit at the expense of much more computation. Clearly you should take care to select the step size with both the computation required and the fidelity of the estimated filter in mind.



## adaptfilt.nlms Example—System Identification

To improve the convergence performance of the LMS algorithm, the normalized variant (NLMS) uses an adaptive step size based on the signal power. As the input signal power changes, the algorithm calculates the input power and adjusts the step size to maintain an appropriate value. Thus the step size changes with time.

As a result, the normalized algorithm converges more quickly with fewer samples in many cases. For input signals that change slowly over time, the normalized LMS can represent a more efficient LMS approach.

In the `adaptlms` example, you used `firgr` to create the filter that you would identify. So you can compare the results, you use the same filter, and replace

`adaptlms` with `adaptnlms`, to use the normalized LMS algorithm variation. You should see better convergence with similar fidelity.

First, generate the input signal and the unknown filter.

```
x = 0.1*randn(1,500);  
[b,err,res] = fircband(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...  
{'w' 'c'});  
d = filter(b,1,x);
```

Again  $d$  represents the desired signal  $d(x)$  as you defined it earlier and  $b$  contains the filter coefficients for your unknown filter.

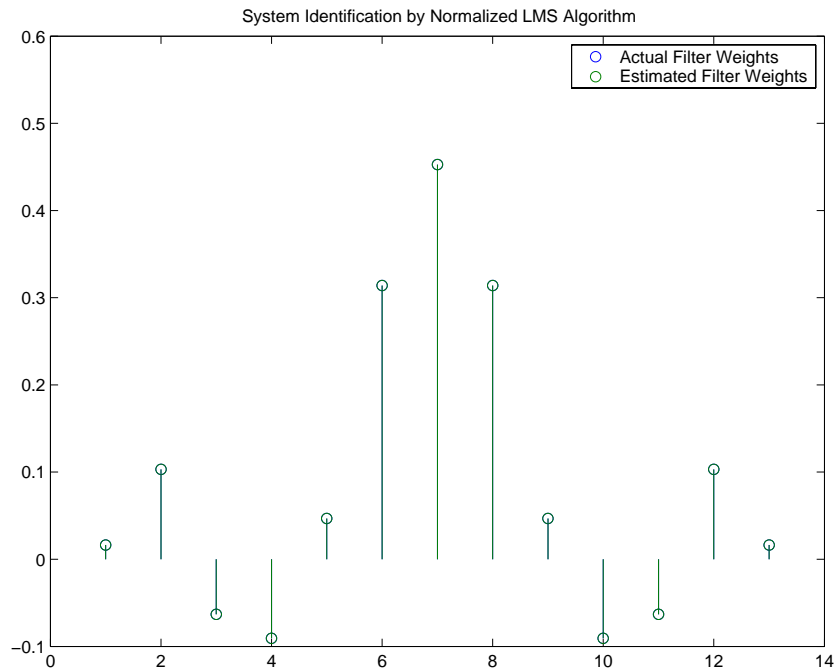
```
mu = 0.8;  
ha = adaptfilt.nlms(13,mu);
```

You use the preceding code to initialize the normalized LMS algorithm. For more information about the optional input arguments, refer to `adaptfilt.nlms` in the reference section of this *User's Guide*.

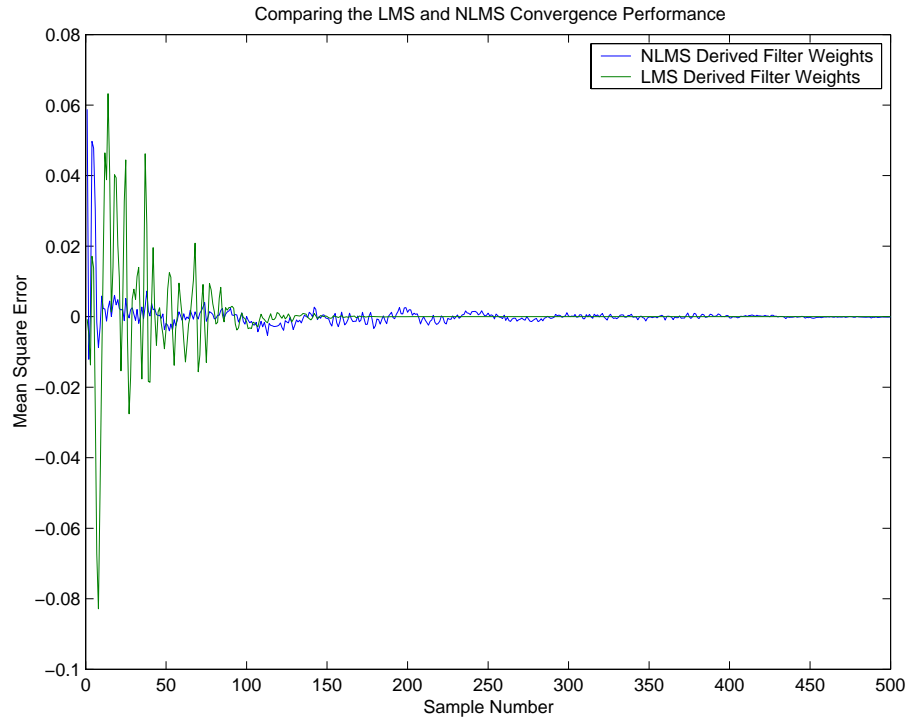
Running the system identification process is a matter of using `adaptfilt.nlms` with the desired signal, the input signal, and the initial filter coefficients and conditions specified in `s` as input arguments. Then plot the results to compare the adapted filter to the actual filter.

```
[y,e] = filter(ha,x,d);  
stem([b.' ha.coefficients.'])
```

As shown in the following stem plot (a convenient way to compare the estimated and actual filter coefficients), the two are nearly identical.



If you compare the convergence performance of the regular LMS algorithm to the normalized LMS variant, you see the normalized version adapts in far fewer iterations to a result almost as good as the nonnormalized version.



### **adaptfilt.sd Example—Noise Cancellation**

When the amount of computation required to derive an adaptive filter drives your development process, the sign-data variant of the LMS (SDLMS) algorithm may be a very good choice as demonstrated in this example.

Fortunately, the current state of digital signal processor (DSP) design has relaxed the need to minimize the operations count by making DSPs whose multiply and shift operations are as fast as add operations. Thus some of the impetus for the sign-data algorithm (and the sign-error and sign-sign variations) has been lost to DSP technology improvements.

In the standard and normalized variations of the LMS adaptive filter, coefficients for the adapting filter arise from the mean square error between the desired signal and the output signal from the unknown system. Using the sign-data algorithm changes the mean square error calculation by using the

sign of the input data to change the filter coefficients.

When the error is positive, the new coefficients are the previous coefficients plus the error multiplied by the step size  $\mu$ . If the error is negative, the new coefficients are again the previous coefficients minus the error multiplied by  $\mu$ —note the sign change.

When the input is zero, the new coefficients are the same as the previous set.

In vector form, the sign-data LMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu e(k) \text{sgn}[\mathbf{x}(k)] , \quad \text{sgn}[\mathbf{x}(k)] = \begin{cases} 1, & \mathbf{x}(k) > 0 \\ 0, & \mathbf{x}(k) = 0 \\ -1, & \mathbf{x}(k) < 0 \end{cases}$$

with vector  $\mathbf{w}$  containing the weights applied to the filter coefficients and vector  $\mathbf{x}$  containing the input data.  $e(k)$  (equal to desired signal - filtered signal) is the error at time  $k$  and is the quantity the SDLMS algorithm seeks to minimize.  $\mu$  ( $\mu$ ) is the step size.

As you specify  $\mu$  smaller, the correction to the filter weights gets smaller for each sample and the SDLMS error falls more slowly. Larger  $\mu$  changes the weights more for each step so the error falls more rapidly, but the resulting error does not approach the ideal solution as closely. To ensure good convergence rate and stability, select  $\mu$  within the following practical bounds

$$0 < \mu < \frac{1}{N\{\text{InputSignalPower}\}}$$

where  $N$  is the number of samples in the signal. Also, define  $\mu$  as a power of two for efficient computing.

---

**Note** How you set the initial conditions of the sign-data algorithm profoundly influences the effectiveness of the adaptation. Because the algorithm essentially quantizes the input signal, the algorithm can become unstable easily.

A series of large input values, coupled with the quantization process may

result in the error growing beyond all bounds. You restrain the tendency of the sign-data algorithm to get out of control by choosing a small step size ( $\mu \ll 1$ ) and setting the initial conditions for the algorithm to nonzero positive and negative values.

---

In this noise cancellation example, `adaptfilt.sd` requires two input data sets:

- Data containing a signal corrupted by noise. In Figure , this is  $d(k)$ , the desired signal. The noise cancellation process removes the noise, leaving the signal.
- Data containing random noise ( $x(k)$  in Figure ) that is correlated with the noise that corrupts the signal data. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*[0:1000-1]');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter, and then add the filtered noise to the signal.

```
noise=randn(1,1000);  
nfilt=fir1(11,0.4); % Eleventh order lowpass filter  
fnoise=filter(nfilt,1,noise); % Correlated noise data  
d=signal.'+fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the `adaptfilt` object for processing, set the input conditions `coeffs` and `mu` for the object. As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path.

In “`adaptfilt.lms` Example—System Identification” on page 4-27, you constructed a default filter that sets the filter coefficients to zeros. In most cases that approach does not work for the sign-data algorithm. The closer you set your initial filter coefficients to the expected values, the more likely it is



that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively.

For this example, start with the coefficients in the filter you used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
coeffs = nfilt.' -0.01; % Set the filter initial conditions.  
mu = 0.05;           % Set the step size for algorithm updating.
```

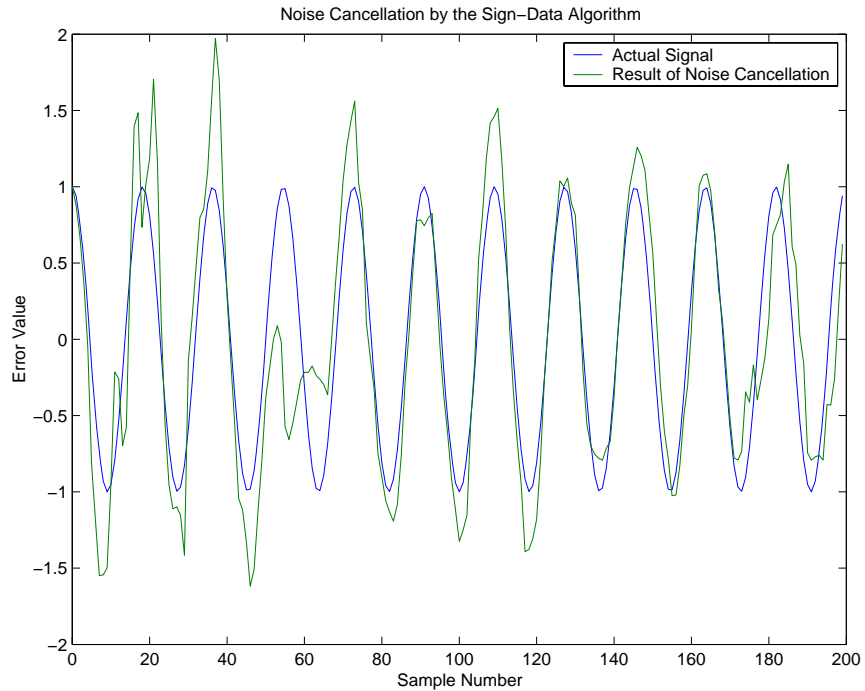
With the required input arguments for `adaptfilt.sd` prepared, construct the `adaptfilt` object, run the adaptation, and view the results.

```
ha = adaptfilt.sd(12,mu)  
set(ha,'coefficients',coeffs);  
[y,e] = filter(ha,noise,d);  
plot(0:199,signal(1:200),0:199,e(1:200));
```

When `adaptfilt.sd` runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-data adaptation requires only bit shifting multiplies when the step size is a power of two.

Although the performance of the sign-data algorithm as shown in the next figure is quite good, the sign-data algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal, but the algorithm could very easily grow without bound rather than achieve good performance.

Changing `coeffs`, `mu`, or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.



## **adaptfilt.se Example—Noise Cancellation**

In some cases, the sign-error variant of the LMS algorithm (SELMS) may be a very good choice for an adaptive filter application.

In the standard and normalized variations of the LMS adaptive filter, the coefficients for the adapting filter arise from calculating the mean square error between the desired signal and the output signal from the unknown system, and applying the result to the current filter coefficients. Using the sign-error algorithm replaces the mean square error calculation by using the sign of the error to modify the filter coefficients.

When the error is positive, the new coefficients are the previous coefficients plus the error multiplied by the step size  $\mu$ . If the error is negative, the new coefficients are again the previous coefficients minus the error multiplied by

$\mu$ —note the sign change. When the input is zero, the new coefficients are the same as the previous set.

In vector form, the sign-error LMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu \operatorname{sgn}[e(k)] \mathbf{x}(k), \quad \operatorname{sgn}[e(k)] = \begin{cases} 1, & e(k) > 0 \\ 0, & e(k) = 0 \\ -1, & e(k) < 0 \end{cases}$$

with vector  $\mathbf{w}$  containing the weights applied to the filter coefficients and vector  $\mathbf{x}$  containing the input data.  $e(k)$  (equal to desired signal - filtered signal) is the error at time  $k$  and is the quantity the SELMS algorithm seeks to minimize.  $\mu$  (mu) is the step size. As you specify  $\mu$  smaller, the correction to the filter weights gets smaller for each sample and the SELMS error falls more slowly.

Larger  $\mu$  changes the weights more for each step so the error falls more rapidly, but the resulting error does not approach the ideal solution as closely. To ensure good convergence rate and stability, select  $\mu$  within the following practical bounds

$$0 < \mu < \frac{1}{N\{\text{InputSignalPower}\}}$$

where  $N$  is the number of samples in the signal. Also, define  $\mu$  as a power of two for efficient computation.

---

**Note** How you set the initial conditions of the sign-data algorithm profoundly influences the effectiveness of the adaptation. Because the algorithm essentially quantizes the error signal, the algorithm can become unstable easily.

A series of large error values, coupled with the quantization process may result in the error growing beyond all bounds. You restrain the tendency of the sign-error algorithm to get out of control by choosing a small step size ( $\mu \ll 1$ ) and setting the initial conditions for the algorithm to nonzero positive and negative values.

---

In this noise cancellation example, `adaptfilt.se` requires two input data sets:

- Data containing a signal corrupted by noise. In Figure , this is  $d(k)$ , the desired signal. The noise cancellation process removes the noise, leaving the signal.
- Data containing random noise ( $x(k)$  in Figure ) that is correlated with the noise that corrupts the signal data. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*[0:1000-1]');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter, then add the filtered noise to the signal.

```
noise=randn(1,1000);  
nfilt=fir1(11,0.4); % Eleventh order lowpass filter.  
fnoise=filter(nfilt,1,noise); % Correlated noise data.  
d=signal.'+fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the `adaptfilt` object for processing, set the input conditions `coeffs` and `mu` for the object. As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path. In “`adaptfilt.lms` Example—System Identification” on page 4-27, you constructed a default filter that sets the filter coefficients to zeros.

Setting the coefficients to zero often does not work for the sign-error algorithm. The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively.

For this example, you start with the coefficients in the filter you used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
coeffs = nfilt.' -0.01; % Set the filter initial conditions.  
mu = 0.05; % Set the step size for algorithm update.
```

With the required input arguments for `adaptfilt.se` prepared, run the adaptation and view the results.

```
ha = adaptfilt.sd(12,mu)
set(ha,'coefficients',coeffs);
set(ha,'persistentmemory',true); % Prevent filter reset.
[y,e] = filter(ha,noise,d);
plot(0:199,signal(1:200),0:199,e(1:200));
```

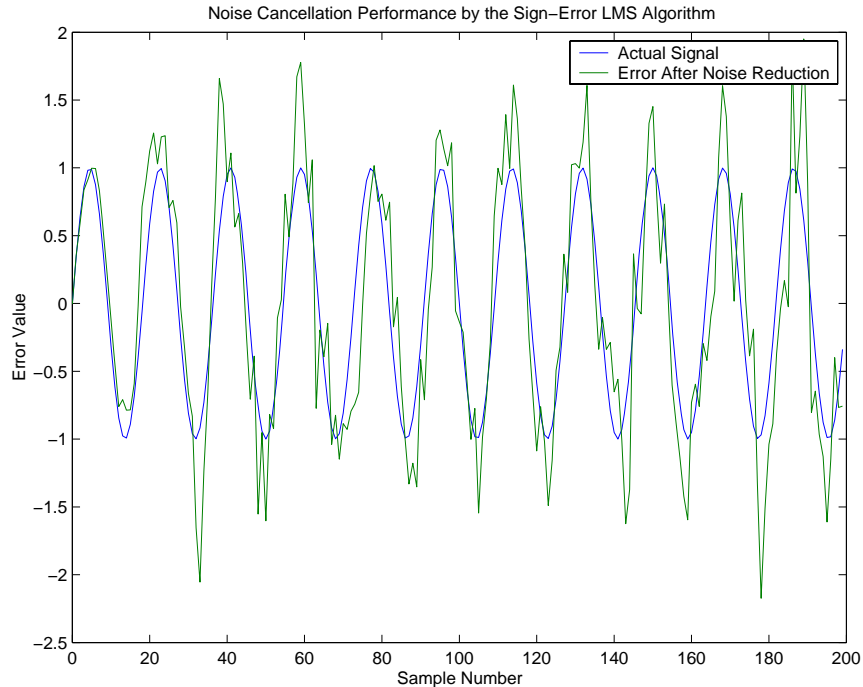
Notice that you have to set the property `PersistentMemory` to `true` when you manually change the settings of object `ha`.

If `PersistentMemory` is left to `false`, the default, when you try to apply `ha` with the method `filter`, the filtering process starts by resetting the object properties to their initial conditions at construction. To preserve the customized coefficients in this example, you set `PersistentMemory` to `true` so the coefficients do not get reset automatically back to zero.

When `adaptfilt.se` runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-error adaptation requires only bit shifting multiplies when the step size is a power of two.

Although the performance of the sign-data algorithm as shown in the next figure is quite good, the sign-data algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal, but the algorithm could very easily become unstable rather than achieve good performance.

Changing `coeffs`, `mu`, or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.



### **adaptfilt.ss Example—Noise Cancellation**

One more example of a variation of the LMS algorithm in the toolbox is the sign-sign variant (SSLMS). The rationale for this version matches those for the sign-data and sign-error algorithms presented in preceding sections. For more details, refer to “adaptfilt.sd Example—Noise Cancellation” on page 4-34.

The sign-sign algorithm (SSLMS) replaces the mean square error calculation with using the sign of the input data to change the filter coefficients. When the error is positive, the new coefficients are the previous coefficients plus the error multiplied by the step size  $\mu$ .

If the error is negative, the new coefficients are again the previous coefficients minus the error multiplied by  $\mu$ —note the sign change. When the input is zero, the new coefficients are the same as the previous set.

In essence, the algorithm quantizes both the error and the input by applying the sign operator to them.

In vector form, the sign-sign LMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu \operatorname{sgn}[e(k)] \operatorname{sgn}[\mathbf{x}(k)], \operatorname{sgn}[z(k)] = \begin{cases} 1, & z(k) > 0 \\ 0, & z(k) = 0 \\ -1, & z(k) < 0 \end{cases}$$

where

$$z(k) = [e(k)] \operatorname{sgn}[\mathbf{x}(k)]$$

Vector  $\mathbf{w}$  contains the weights applied to the filter coefficients and vector  $\mathbf{x}$  contains the input data.  $e(k)$  (= desired signal - filtered signal) is the error at time  $k$  and is the quantity the SSLMS algorithm seeks to minimize.  $\mu$  (mu) is the step size. As you specify  $\mu$  smaller, the correction to the filter weights gets smaller for each sample and the SSLMS error falls more slowly.

Larger  $\mu$  changes the weights more for each step so the error falls more rapidly, but the resulting error does not approach the ideal solution as closely. To ensure good convergence rate and stability, select  $\mu$  within the following practical bounds

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where  $N$  is the number of samples in the signal. Also, define  $\mu$  as a power of two for efficient computation.

---

**Note** How you set the initial conditions of the sign-sign algorithm profoundly influences the effectiveness of the adaptation. Because the algorithm essentially quantizes the input signal and the error signal, the algorithm can become unstable easily.

A series of large error values, coupled with the quantization process may

result in the error growing beyond all bounds. You restrain the tendency of the sign-sign algorithm to get out of control by choosing a small step size ( $\mu \ll 1$ ) and setting the initial conditions for the algorithm to nonzero positive and negative values.

---

In this noise cancellation example, `adaptfilt.ss` requires two input data sets:

- Data containing a signal corrupted by noise. In Figure , this is  $d(k)$ , the desired signal. The noise cancellation process removes the noise, leaving the cleaned signal as the content of the error signal.
- Data containing random noise ( $x(k)$  in Figure ) that is correlated with the noise that corrupts the signal data, called. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*[0:1000-1]');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter, then add the filtered noise to the signal.

```
noise=randn(1,1000);  
nfilt=fir1(11,0.4); % Eleventh order lowpass filter  
fnoise=filter(nfilt,1,noise); % Correlated noise data  
d=signal.'+fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the `adaptfilt` object for processing, set the input conditions `coeffs` and `mu` for the object. As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path. In “`adaptfilt.lms` Example—System Identification” on page 4-27, you constructed a default filter that sets the filter coefficients to zeros. Usually that approach does not work for the sign-sign algorithm.

The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter



solution that removes the noise effectively. For this example, you start with the coefficients in the filter you used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
coeffs = nfilt.' -0.01; % Set the filter initial conditions.
mu = 0.05;           % Set the step size for algorithm updating.
```

With the required input arguments for `adaptfilt.ss` prepared, run the adaptation and view the results.

```
ha = adaptfilt.ss(12,mu)
set(ha,'coefficients',coeffs);
set(ha,'persistentmemory',true); % Prevent filter reset.
[y,e] = filter(ha,noise,d);
plot(0:199,signal(1:200),0:199,e(1:200));
```

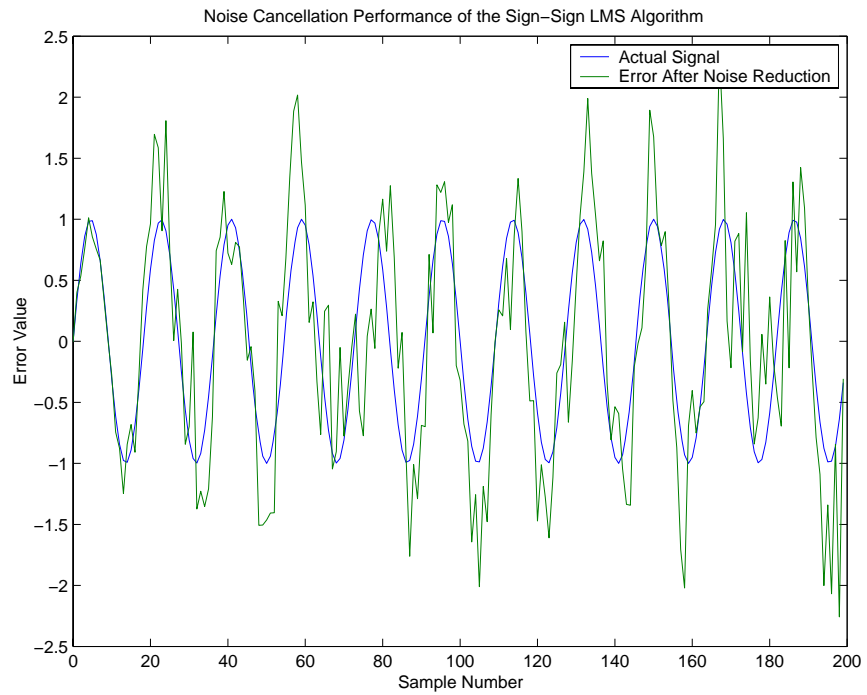
Notice that you have to set the property `PersistentMemory` to `true` when you manually change the settings of object `ha`.

If `PersistentMemory` is left to `false`, when you try to apply `ha` with the method `filter` the filtering process starts by resetting the object properties to their initial conditions at construction. To preserve the customized coefficients in this example, you set `PersistentMemory` to `true` so the coefficients do not get reset automatically back to zero.

When `adaptfilt.ss` runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-sign adaptation requires only bit shifting multiplies when the step size is a power of two.

Although the performance of the sign-sign algorithm as shown in the next figure is quite good, the sign-sign algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal, but the algorithm could very easily become unstable rather than achieve good performance.

Changing `coeffs`, `mu`, or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.



As an aside, the sign-sign LMS algorithm is part of the international CCITT standard for 32 Kb/s ADPCM telephony.

## Example of Adaptive Filter That Uses RLS Algorithm

This section provides an introductory example that uses the RLS adaptive filter function `adaptfilt.rls`.

If LMS algorithms represent the simplest and most easily applied adaptive algorithms, the recursive least squares (RLS) algorithms represents increased complexity, computational cost, and fidelity. In performance, RLS approaches the Kalman filter in adaptive filtering applications, at somewhat reduced required throughput in the signal processor.

Compared to the LMS algorithm, the RLS approach offers faster convergence and smaller error with respect to the unknown system, at the expense of requiring more computations.

In contrast to the least mean squares algorithm, from which it can be derived, the RLS adaptive algorithm minimizes the total squared error between the desired signal and the output from the unknown system.

Referring to Figure , you see the signal flow graph (or model) for the RLS adaptive filter system. Note that the signal paths and identifications are the same whether the filter uses RLS or LMS. The difference lies in the adapting portion.

Within limits, you can use any of the adaptive filter algorithms to solve an adaptive filter problem by replacing the adaptive portion of the application with a new algorithm.

Examples of the sign variants of the LMS algorithms demonstrated this feature to demonstrate the differences between the sign-data, sign-error, and sign-sign variations of the LMS algorithm.

One interesting input option that applies to RLS algorithms is not present in the LMS processes—a forgetting factor,  $\lambda$ , that determines how the algorithm treats past data input to the algorithm.

When the LMS algorithm looks at the error to minimize, it considers only the current error value. In the RLS method, the error considered is the total error from the beginning to the current data point.

Said another way, the RLS algorithm has infinite memory—all error data is given the same consideration in the total error. In cases where the error value might come from a spurious input data point or points, the forgetting factor lets

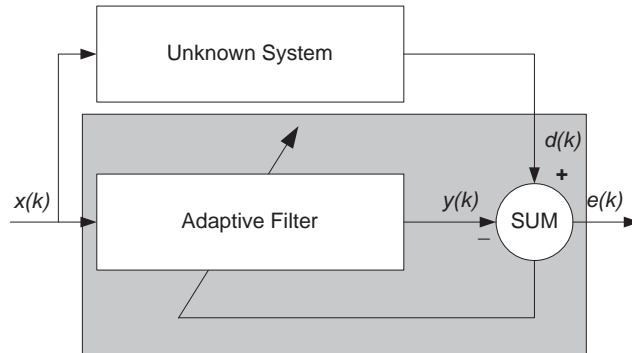
the RLS algorithm reduce the value of older error data by multiplying the old data by the forgetting factor.

Since  $0 \leq \lambda < 1$ , applying the factor is equivalent to weighting the older error. When  $\lambda = 1$ , all previous error is considered of equal weight in the total error.

As  $\lambda$  approaches zero, the past errors play a smaller role in the total. For example, when  $\lambda = 0.9$ , the RLS algorithm multiplies an error value from 50 samples in the past by an attenuation factor of  $0.9^{50} = 5.15 \times 10^{-3}$ , considerably deemphasizing the influence of the past error on the current total error.

## adaptfilt.rls Example – Inverse System Identification

Rather than use a system identification application to demonstrate the RLS adaptive algorithm, or a noise cancellation model, this example use the inverse system identification model shown in here.



Cascading the adaptive filter with the unknown filter causes the adaptive filter to converge to a solution that is the inverse of the unknown system.

If the transfer function of the unknown is  $H(z)$  and the adaptive filter transfer function is  $G(z)$ , the error measured between the desired signal and the signal from the cascaded system reaches its minimum when the product of  $H(z)$  and  $G(z)$  is 1,  $G(z) \cdot H(z) = 1$ . For this relation to be true,  $G(z)$  must equal  $-H(z)$ , the inverse of the transfer function of the unknown system.

To demonstrate that this is true, create a signal to input to the cascaded filter pair.

```
x = randn(1,3000);
```

In the cascaded filters case, the unknown filter results in a delay in the signal arriving at the summation point after both filters. To prevent the adaptive filter from trying to adapt to a signal it has not yet seen (equivalent to predicting the future), delay the desired signal by 32 samples, the order of the unknown system.

Generally, you do not know the order of the system you are trying to identify. In that case, delay the desired signal by the number of samples equal to half the order of the adaptive filter. Delaying the input requires prepending 12 zero-values samples to  $x$ .

```
delay = zeros(1,12);
d = [delay x(1:2988)]; % Concatenate the delay and the signal.
```

You have to keep the desired signal vector  $d$  the same length as  $x$ , hence adjust the signal element count to allow for the delay samples.

Although not generally true, for this example you know the order of the unknown filter, so you add a delay equal to the order of the unknown filter.

For the unknown system, use a lowpass, 12th-order FIR filter.

```
ufilt = fir1(12,0.55,'low');
```

Filtering  $x$  provides the input data signal for the adaptive algorithm function.

```
xdata = filter(ufilt,1,x);
```

To set the input argument values for the `adaptfilt.rls` object, use the constructor `adaptfilt.rls`, providing the needed arguments `l`, `lambda`, and `invcov`.

For more information about the input conditions to prepare the RLS algorithm object, refer to `adaptfilt.rls` in the reference section of this user's guide.

```
p0 = 2*eye(13);
lambda = 0.99;
ha = adaptfilt.rls(13,lambda,p0);
```

Most of the process to this point is the same as the preceding examples. However, since this example seeks to develop an inverse solution, you need to be careful about which signal carries the data and which is the desired signal.

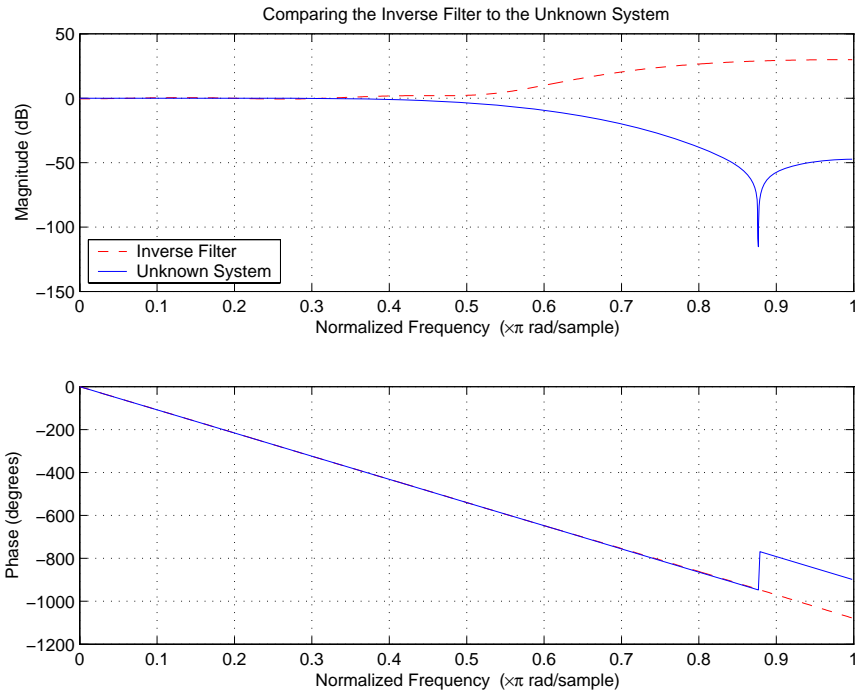
Earlier examples of adaptive filters use the filtered noise as the desired signal. In this case, the filtered noise ( $xdata$ ) carries the unknown system information.

With Gaussian distribution and variance of 1, the unfiltered noise  $d$  is the desired signal. The code to run this adaptive filter example is

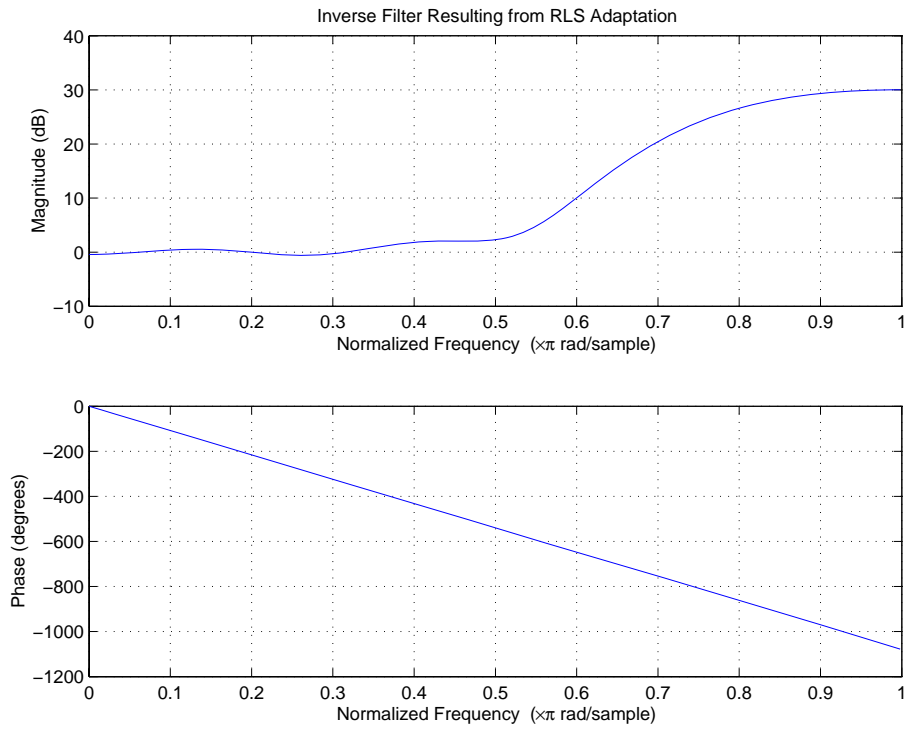
```
[y,e] = filter(ha,xdata,d);
```

where  $y$  returns the coefficients of the adapted filter and  $e$  contains the error signal as the filter adapts to find the inverse of the unknown system. You can review the returned elements of the adapted filter in the properties of  $ha$ .

The next figure presents the results of the adaptation. In the figure, the magnitude response curves for the unknown and adapted filters show. As a reminder, the unknown filter was a lowpass filter with cutoff at 0.55, on the normalized frequency scale from 0 to 1.



Viewed alone (refer to the following figure), the inverse system looks like a fair compensator for the unknown lowpass filter—a high pass filter with linear phase.



### **Selected Bibliography**

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*, John Wiley & Sons, 1996, 493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*, Prentice-Hall, Inc., 1996



---

# User's Guide

---

Digital Frequency Transformations  
(p. 5-1)

Provides tutorial information about performing transformations of discrete-time filters

Using FDATool with the Filter Design Toolbox (p. 6-1)

Presents a detailed reference covering the fixed-point, multirate, and scaling pages of the Filter Design and Analysis Tool

Reference for the Properties of Filter Objects (p. 7-1)

Provides:

- A summary of the filter objects properties
- A detailed filter property reference, including descriptions of the filter structures and properties for `adaptfilt`, `dfilt`, and `mfilt` objects

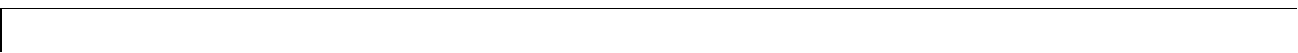
Functions — By Category (p. 8-2)  
(online only)

Provides:

- Tables that include short descriptions of the functions in this toolbox
- A detailed alphabetical function reference

Bibliography (p. A-1)

Lists references for filtering texts and papers



# Digital Frequency Transformations

---

Introduction (p. 5-2)

Provides background about digital frequency transformations for filters

Definition of the Problem (p. 5-3)

Presents and defines the problem of using digital frequency transformation

Frequency Transformations for Real Filters (p. 5-11)

Discusses the functions for transforming real filters to other real filters

Frequency Transformations for Complex Filters (p. 5-26)

Describes the functions for transforming complex filters to other complex filters, or real filters to complex filters

## Introduction

Converting existing FIR or IIR filter designs to a modified IIR form is often done using allpass frequency transformations. Although the resulting designs can be considerably more expensive in terms of dimensionality than the prototype (original) filter, their ease of use in fixed or variable applications is a big advantage.

The general idea of the frequency transformation is to take an existing prototype filter and produce another filter from it that retains some of the characteristics of the prototype, in the frequency domain. Transformation functions achieve this by replacing each delaying element of the prototype filter with an allpass filter carefully designed to have a prescribed phase characteristic for achieving the modifications requested by the designer.

This tutorial gives an overview and interpretation of the frequency transformations, and describes the range of transformations available to the toolbox user. To aid this purpose the tutorial has been arranged into three sections:

- “Definition of the Problem” on page 5-3 introduces the frequency transformation concept and provides its mathematical and intuitive interpretations.
- “Frequency Transformations for Real Filters” on page 5-11 describes the real frequency transformations available in the toolbox. Such transformations start from a real prototype filter and return a real target filter.
- “Frequency Transformations for Complex Filters” on page 5-26 describes complex frequency transformations available in the toolbox. Such transformations start from the any real or complex prototype filter and return a complex target filter.

## Definition of the Problem

The basic form of mapping in common use is

$$H_T(z) = H_o[H_A(z)]$$

The  $H_A(z)$  is an  $N$ th-order allpass mapping filter given by

$$H_A(z) = S \frac{\sum_{i=0}^N \alpha_i z^{-i}}{\sum_{i=0}^N \alpha_i z^{-N+i}} = \frac{N_A(z)}{D_A(z)}$$

$$\alpha_0 = 1$$

where

$H_o(z)$ — Transfer function of the prototype filter

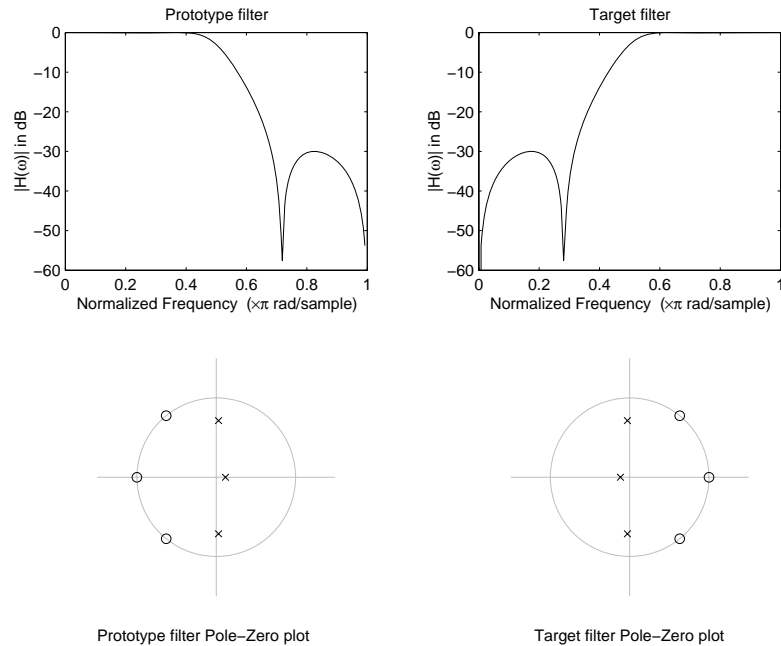
$H_A(z)$ — Transfer function of the allpass mapping filter

$H_T(z)$ — Transfer function of the target filter

Let's look at a simple example of the transformation given by

$$H_T(z) = H_o(-z)$$

The target filter has its poles and zeroes flipped across the origin of the real and imaginary axes. For the real filter prototype, it gives a mirror effect against 0.5, which means that lowpass  $H_o(z)$  gives rise to a real highpass  $H_T(z)$ . This is shown in the following figure for the prototype filter designed as a third-order halfband elliptic filter.



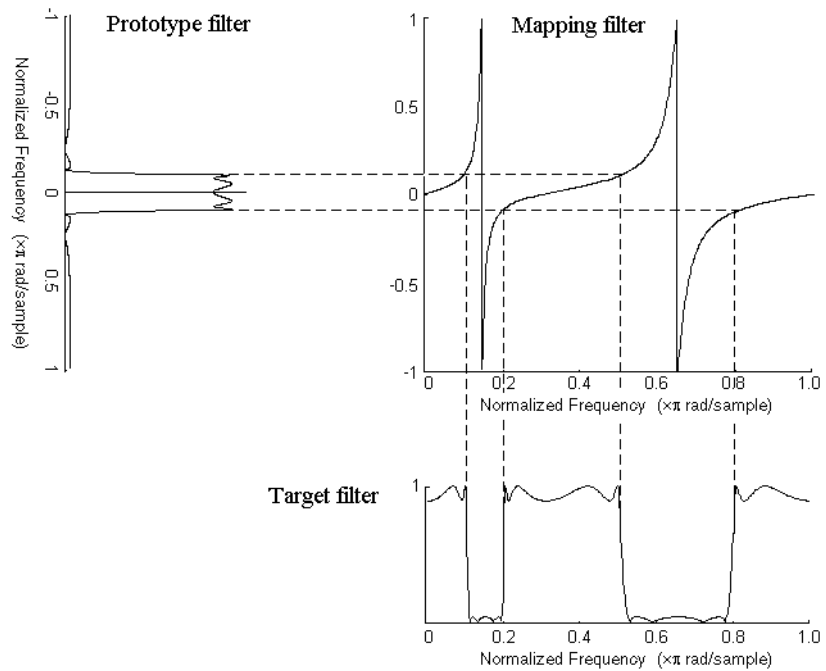
**Figure 5-1: Example of a Simple Mirror Transformation**

The choice of an allpass filter to provide the frequency mapping is necessary to provide the frequency translation of the prototype filter frequency response to the target filter by changing the frequency position of the features from the prototype filter without affecting the overall shape of the filter response.

The phase response of the mapping filter normalized to  $\pi$  can be interpreted as a translation function:

$$H(w_{new}) = \omega_{old}$$

The graphical interpretation of the frequency transformation is shown in the figure below. The complex multiband transformation takes a real lowpass filter and converts it into a number of passbands around the unit circle.



**Figure 5-2: Graphical Interpretation of the Mapping Process**

Most of the frequency transformations are based on the second-order allpass mapping filter:

$$H_A(z) = \pm \frac{1 + \alpha_1 z^{-1} + \alpha_2 z^{-2}}{\alpha_2 + \alpha_1 z^{-1} + z^{-2}}$$

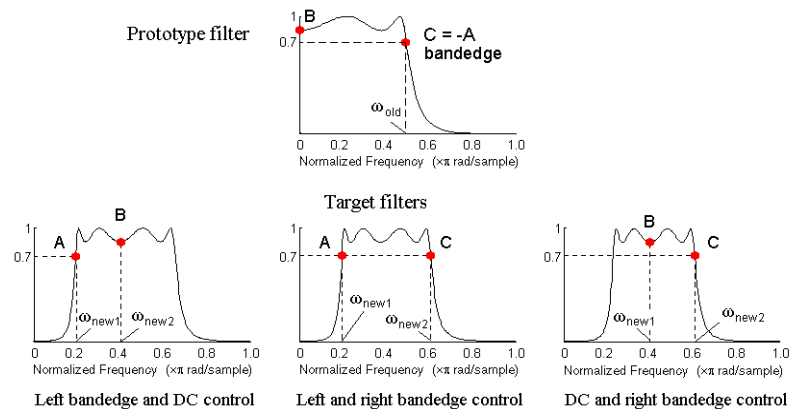
The two degrees of freedom provided by  $\alpha_1$  and  $\alpha_2$  choices are not fully used by the usual restrictive set of “flat-top” classical mappings like lowpass to bandpass. Instead, any two transfer function features can be migrated to (almost) any two other frequency locations if  $\alpha_1$  and  $\alpha_2$  are chosen so as to keep the poles of  $H_A(z)$  strictly outside the unit circle (since  $H_A(z)$  is substituted for  $z$  in the prototype transfer function). Moreover, as first pointed out by Constantinides, the selection of the outside sign influences whether the original feature at zero can be moved (the minus sign, a condition known as

“DC mobility”) or whether the Nyquist frequency can be migrated (the “Nyquist mobility” case arising when the leading sign is positive).

All the transformations forming the package are explained in the next sections of the tutorial. They are separated into those operating on real filters and those generating or working with complex filters. The choice of transformation ranges from standard Constantinides first and second-order ones [19][20] up to the real multiband filter by Mullis and Franchitti [21], and the complex multiband filter and real/complex multipoint ones by Krukowski, Cain and Kale [22].

### Selecting Features Subject to Transformation

Choosing the appropriate frequency transformation for achieving the required effect and the correct features of the prototype filter is very important and needs careful consideration. It is not advisable to use a first-order transformation for controlling more than one feature. The mapping filter will not give enough flexibility. It is also not good to use higher order transformation just to change the cutoff frequency of the lowpass filter. The increase of the filter order would be too big, without considering the additional replica of the prototype filter that may be created in undesired places.

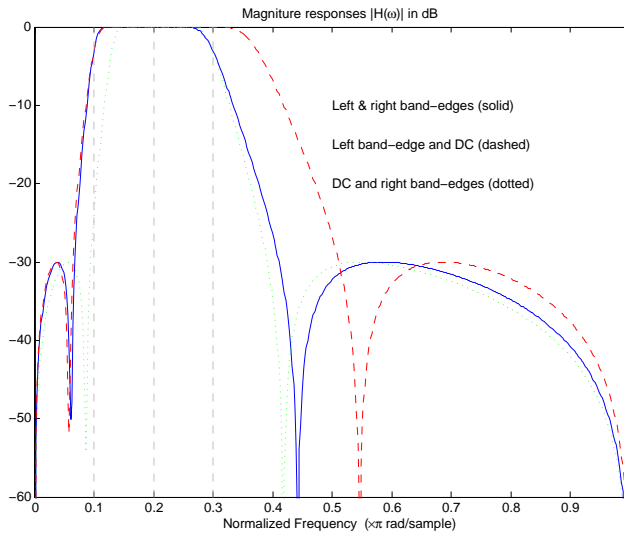


**Figure 5-3: Feature Selection for Real Lowpass to Bandpass Transformation**

To illustrate the idea, the second-order real multipoint transformation was applied three times to the same elliptic halfband filter in order to make it into a bandpass filter. In each of the three cases, two different features of the



prototype filter were selected in order to obtain a bandpass filter with passband ranging from 0.25 to 0.75. The position of the DC feature was not important, but it would be advantageous if it were in the middle between the edges of the passband in the target filter. In the first case the selected features were the left and the right band edges of the lowpass filter passband, in the second case they were the left band edge and the DC, in the third case they were DC and the right band edge.



**Figure 5-4: Result of choosing different features**

The results of all three approaches are completely different. For each of them only the selected features were positioned precisely where they were required. In the first case the DC is moved toward the left passband edge just like all the other features close to the left edge being squeezed there. In the second case the right passband edge was pushed way out of the expected target as the precise position of DC was required. In the third case the left passband edge was pulled toward the DC in order to position it at the correct frequency. The conclusion is that if only the DC can be anywhere in the passband, the edges of the passbands should have been selected for the transformation. For most of the cases requiring the positioning of passbands and stopbands, designers should

always choose the position of the edges of the prototype filter in order to make sure that they get the edges of the target filter in the correct places. Frequency responses for the three cases considered are shown in the figure. The prototype filter was a third-order elliptic lowpass filter with cutoff frequency at 0.5.

The MATLAB code used to generate the figure is given here.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

In the example the requirements are set to create a real bandpass filter with passband edges at 0.1 and 0.3 out of the real lowpass filter having the cutoff frequency at 0.5. This is attempted in three different ways. In the first approach both edges of the passband are selected, in the second approach the left edge of the passband and the DC are chosen, while in the third approach the DC and the right edge of the passband are taken:

```
[num1,den1] = iirlp2xn(b, a, [-0.5, 0.5], [0.1, 0.3]);
[num2,den2] = iirlp2xn(b, a, [-0.5, 0.0], [0.1, 0.2]);
[num3,den3] = iirlp2xn(b, a, [ 0.0, 0.5], [0.2, 0.3]);
```

## Mapping from Prototype Filter to Target Filter

In general the frequency mapping converts the prototype filter,  $H_o(z)$ , to the target filter,  $H_T(z)$ , using the  $N_A$ th-order allpass filter,  $H_A(z)$ . The general form of the allpass mapping filter is given in Equation . The frequency mapping is a mathematical operation that replaces each delay of the prototype filter with an allpass filter. There are two ways of performing such mapping. The choice of the approach is dependent on how prototype and target filters are represented.

When the  $N$ th-order prototype filter is given with pole-zero form

$$H_o(z) = \frac{\prod_{i=1}^N (z - z_i)}{\prod_{i=1}^N (z - p_i)}$$

the mapping will replace each pole,  $p_i$ , and each zero,  $z_i$ , with a number of poles and zeros equal to the order of the allpass mapping filter:

$$H_o(z) = \frac{\sum_{i=1}^N \left( S \sum_{k=0}^{N-1} \alpha_k z^k - z_i \cdot \sum_{k=0}^{N-1} \alpha_k z^{N-k} \right)}{\sum_{i=1}^N \left( S \sum_{k=0}^{N-1} \alpha_k z^k - p_i \cdot \sum_{k=0}^{N-1} \alpha_k z^{N-k} \right)}$$

The root finding needs to be used on the bracketed expressions in order to find the poles and zeros of the target filter.

When the prototype filter is described in the numerator-denominator form:

$$H_T(z) = \frac{\beta_0 z^N + \beta_1 z^{N-1} + \dots + \beta_N}{\alpha_0 z^N + \alpha_1 z^{N-1} + \dots + \alpha_N} \Bigg|_{z = H_A(z)}$$

Then the mapping process will require a number of convolutions in order to calculate the numerator and denominator of the target filter:

$$H_T(z) = \frac{\beta_1 N_A(z)^N + \beta_2 N_A(z)^{N-1} D_A(z) + \dots + \beta_N D_A(z)^N}{\beta_1 N_A(z)^N + \beta_2 N_A(z)^{N-1} D_A(z) + \dots + \beta_N D_A(z)^N}$$

For each coefficient  $\alpha_i$  and  $\beta_i$  of the prototype filter the  $N_A$ th-order polynomials must be convolved  $N$  times. Such approach may cause rounding errors for large prototype filters and/or high order mapping filters. In such a case the user should consider the alternative of doing the mapping using via poles and zeros.

## Summary of Frequency Transformations

### *Advantages*

- Most frequency transformations are described by closed-form solutions or can be calculated from the set of linear equations.
- They give predictable and familiar results.
- Ripple heights from the prototype filter are preserved in the target filter.
- They are architecturally appealing for variable and adaptive filters.

### *Disadvantages*

- There are cases when using optimization methods to design the required filter gives better results.
- High-order transformations increase the dimensionality of the target filter, which may give expensive final results.
- Starting from fresh designs helps avoid locked-in compromises.

## Frequency Transformations for Real Filters

This section discusses real frequency transformations that take the real lowpass prototype filter and convert it into a different real target filter. The target filter has its frequency response modified in respect to the frequency response of the prototype filter according to the characteristic of the applied frequency transformation:

- “Real Frequency Shift” on page 5-12
- “Real Lowpass to Real Lowpass” on page 5-13
- “Real Lowpass to Real Highpass” on page 5-15
- “Real Lowpass to Real Bandpass” on page 5-17
- “Real Lowpass to Real Bandstop” on page 5-19
- “Real Lowpass to Real Multiband” on page 5-21
- “Real Lowpass to Real Multipoint” on page 5-23

## Real Frequency Shift

Real frequency shift transformation uses a second-order allpass mapping filter. It performs an exact mapping of one selected feature of the frequency response into its new location, additionally moving both the Nyquist and DC features. This effectively moves the whole response of the lowpass filter by the distance specified by the selection of the feature from the prototype filter and the target filter. As a real transformation, it works in a similar way for positive and negative frequencies.

$$H_A(z) = z^{-1} \cdot \frac{1 - \alpha z^{-1}}{\alpha - z^{-1}}$$

with  $\alpha$  given by

$$\alpha = \begin{cases} \frac{\cos \frac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\cos \frac{\pi}{2}\omega_{old}} & \text{for } \left| \cos \frac{\pi}{2}(\omega_{old} - 2\omega_{new}) \right| < 1 \\ \frac{\sin \frac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\sin \frac{\pi}{2}\omega_{old}} & \text{otherwise} \end{cases}$$

where

$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new}$  — Position of the feature originally at  $\omega_{old}$  in the target filter

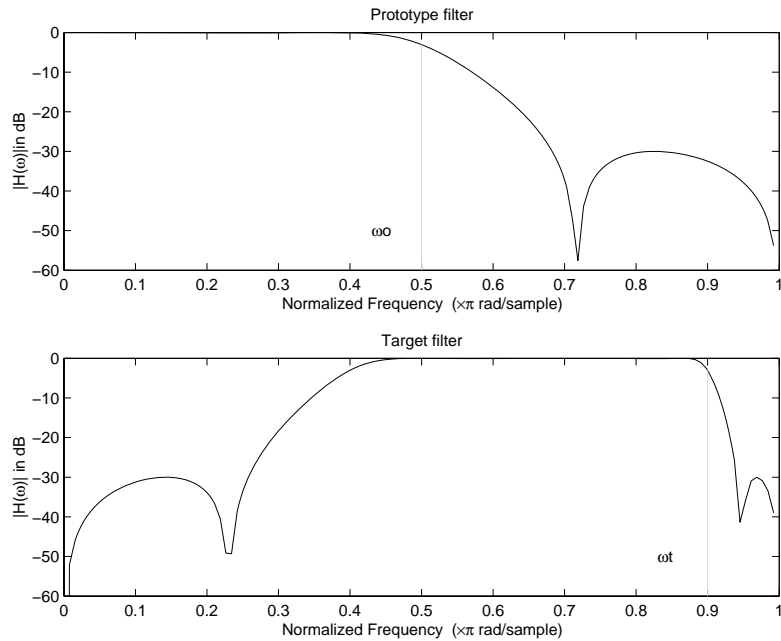
The example below shows how this transformation can be used to move the response of the prototype lowpass filter in either direction. Please note that because the target filter must also be real, the response of the target filter will inherently be disturbed at frequencies close to Nyquist and close to DC. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation moves the feature originally at 0.5 to 0.9:

```
[num,den] = iirshift(b, a, 0.5, 0.9);
```



**Figure 5-5: Example of Real Frequency Shift Mapping**

## Real Lowpass to Real Lowpass

Real lowpass filter to real lowpass filter transformation uses a first-order allpass mapping filter. It performs an exact mapping of one feature of the frequency response into the new location keeping DC and Nyquist features fixed. As a real transformation, it works in a similar way for positive and negative frequencies. It is important to mention that using first-order mapping ensures that the order of the filter after the transformation is the same as it was originally.

$$H_A(z) = -\left(\frac{1 - \alpha z^{-1}}{\alpha - z^{-1}}\right)$$

with  $\alpha$  given by

$$\alpha = \frac{\sin \frac{\pi}{2}(w_{old} - w_{new})}{\sin \frac{\pi}{2}(w_{old} + w_{new})}$$

where

$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new}$  — Frequency location of the same feature in the target filter

The example below shows how to modify the cutoff frequency of the prototype filter. The MATLAB code for this example is shown in the figure below.

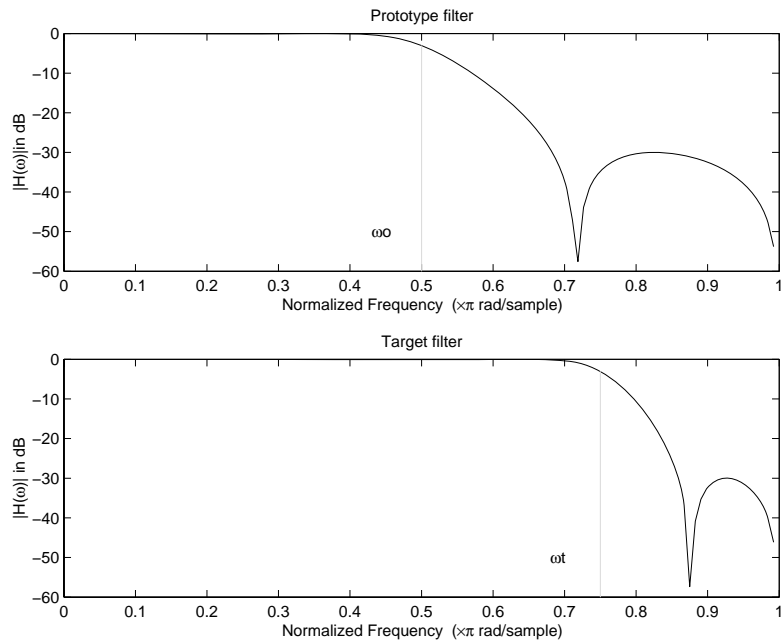
The prototype filter is a halfband elliptic, real, third-order filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The cutoff frequency moves from 0.5 to 0.75:

```
[num,den] = iir1p2lp(b, a, 0.5, 0.75);
```





**Figure 5-6: Example of Real Lowpass to Real Lowpass Mapping**

## Real Lowpass to Real Highpass

Real lowpass filter to real highpass filter transformation uses a first-order allpass mapping filter. It performs an exact mapping of one feature of the frequency response into the new location additionally swapping DC and Nyquist features. As a real transformation, it works in a similar way for positive and negative frequencies. Just like in the previous transformation because of using a first-order mapping, the order of the filter before and after the transformation is the same.

$$H_A(z) = -\left(\frac{1 + \alpha z^{-1}}{\alpha + z^{-1}}\right)$$

with  $\alpha$  given by

$$\alpha = - \left( \frac{\cos \frac{\pi}{2}(w_{old} + w_{new})}{\cos \frac{\pi}{2}(w_{old} - w_{new})} \right)$$

where

$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new}$  — Frequency location of the same feature in the target filter

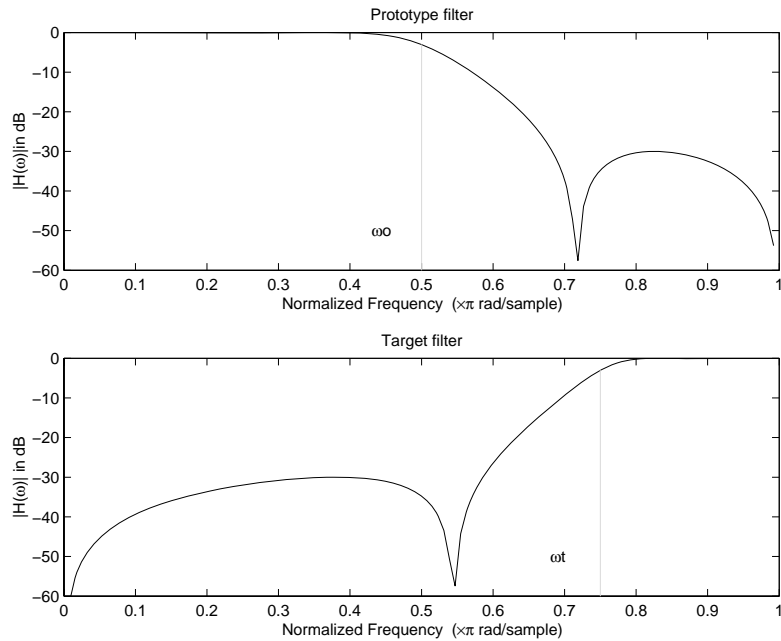
The example below shows how to convert the lowpass filter into a highpass filter with arbitrarily chosen cutoff frequency. In the MATLAB code below, the lowpass filter is converted into a highpass with cutoff frequency shifted from 0.5 to 0.75. Results are shown in the figure.

The prototype filter is a halfband elliptic, real, third-order filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example moves the cutoff frequency from 0.5 to 0.75:

```
[num,den] = iir1p2lp(b, a, 0.5, 0.75);
```



**Figure 5-7: Example of Real Lowpass to Real Highpass Mapping**

## Real Lowpass to Real Bandpass

Real lowpass filter to real bandpass filter transformation uses a second-order allpass mapping filter. It performs an exact mapping of two features of the frequency response into their new location additionally moving a DC feature and keeping the Nyquist feature fixed. As a real transformation, it works in a similar way for positive and negative frequencies.

$$H_A(z) = -\left(\frac{1 - \beta(1 + \alpha)z^{-1} - \alpha z^{-2}}{\alpha - \beta(1 + \alpha)z^{-1} + z^{-2}}\right)$$

with  $\alpha$  and  $\beta$  given by

$$\alpha = \frac{\sin \frac{\pi}{4}(2w_{old} - w_{new,2} + w_{new,1})}{\sin \frac{\pi}{4}(2w_{old} + w_{new,2} - w_{new,1})}$$

$$\beta = \cos \frac{\pi}{2}(w_{new,1} + w_{new,2})$$

where

$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new,1}$  — Position of the feature originally at  $(-\omega_{old})$  in the target filter

$\omega_{new,2}$  — Position of the feature originally at  $(+\omega_{old})$  in the target filter

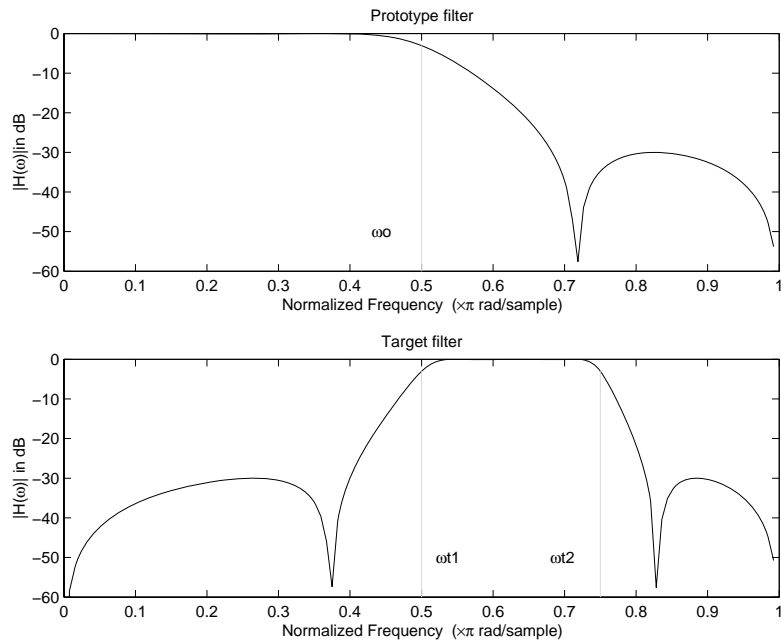
The example below shows how to move the response of the prototype lowpass filter in either direction. Please note that because the target filter must also be real, the response of the target filter will inherently be disturbed at frequencies close to Nyquist and close to DC. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates the passband between 0.5 and 0.75:

```
[num,den] = iirlp2bp(b, a, 0.5, [0.5, 0.75]);
```



**Figure 5-8: Example of Real Lowpass to Real Bandpass Mapping**

## Real Lowpass to Real Bandstop

Real lowpass filter to real bandstop filter transformation uses a second-order allpass mapping filter. It performs an exact mapping of two features of the frequency response into their new location additionally moving a Nyquist feature and keeping the DC feature fixed. This effectively creates a stopband between the selected frequency locations in the target filter. As a real transformation, it works in a similar way for positive and negative frequencies.

$$H_A(z) = \frac{1 - \beta(1 + \alpha)z^{-1} + \alpha z^{-2}}{\alpha - \beta(1 + \alpha)z^{-1} + z^{-2}}$$

with  $\alpha$  and  $\beta$  given by

$$\alpha = \frac{\cos \frac{\pi}{4}(2w_{old} + w_{new,2} - w_{new,1})}{\cos \frac{\pi}{4}(2w_{old} - w_{new,2} + w_{new,1})}$$

$$\beta = \cos \frac{\pi}{2}(w_{new,1} + w_{new,2})$$

where

$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new,1}$  — Position of the feature originally at  $(-\omega_{old})$  in the target filter

$\omega_{new,2}$  — Position of the feature originally at  $(+\omega_{old})$  in the target filter

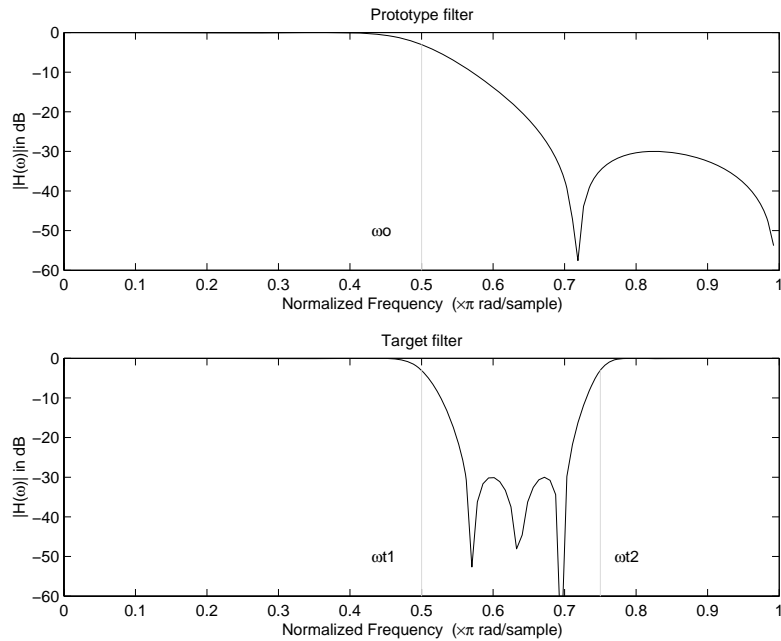
The example below shows how this transformation can be used to convert the prototype lowpass filter with cutoff frequency at 0.5 into a real bandstop filter with the same passband and stopband ripple structure and stopband positioned between 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates a stopband from 0.5 to 0.75:

```
[num,den] = iirlp2bs(b, a, 0.5, [0.5, 0.75]);
```



**Figure 5-9: Example of Real Lowpass to Real Bandstop Mapping**

## Real Lowpass to Real Multiband

This high-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into a number of new locations in the target filter. Its most common use is to convert a real lowpass with predefined passband and stopband ripples into a real multiband filter with  $N$  arbitrary band edges, where  $N$  is the order of the allpass mapping filter.

$$H_A(z) = S \frac{\sum_{i=0}^N \alpha_i z^{-i}}{\sum_{i=0}^{N-1} \alpha_i z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients  $\alpha$  are given by

$$\begin{cases} \alpha_0 = 1 & k = 1, \dots, N \\ \alpha_k = -S \frac{\sin \frac{\pi}{2}(N\omega_{new} + (-1)^k \omega_{old})}{\sin \frac{\pi}{2}((N-2k)\omega_{new} + (-1)^k \omega_{old})} \end{cases}$$

where

$\omega_{old,k}$  – Frequency location of the first feature in the prototype filter

$\omega_{new,k}$  – Position of the feature originally at  $\omega_{old,k}$  in the target filter

The mobility factor,  $S$ , specifies the mobility or either DC or Nyquist feature:

$$S = \begin{cases} 1 & Nyquist \\ -1 & DC \end{cases}$$

The example below shows how this transformation can be used to convert the prototype lowpass filter with cutoff frequency at 0.5 into a filter having a number of bands positioned at arbitrary edge frequencies 1/5, 2/5, 3/5 and 4/5. Parameter  $S$  was such that there is a passband at DC. Here is the MATLAB code for generating the figure.

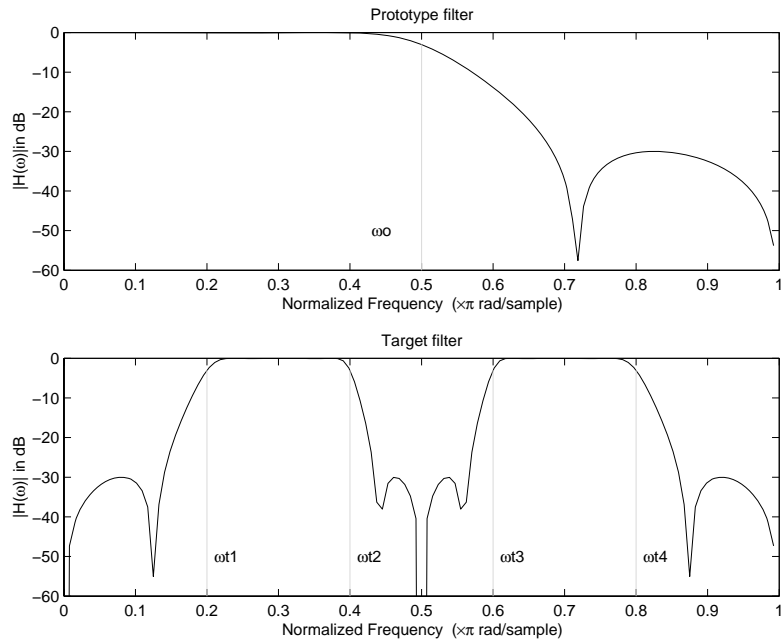
The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates three passbands, from DC to 0.2, from 0.4 to 0.6 and from 0.8 to Nyquist:

```
[num,den] = iir1p2mb(b, a, 0.5, [0.2, 0.4, 0.6, 0.8], 'pass');
```





**Figure 5-10: Example of Real Lowpass to Real Multiband Mapping**

### Real Lowpass to Real Multipoint

This high-order frequency transformation performs an exact mapping of a number of selected features of the prototype filter frequency response to their new locations in the target filter. The mapping filter is given by the general IIR polynomial form of the transfer function as given below.

$$H_A(z) = S \frac{\sum_{i=0}^N \alpha_i z^{-i}}{\sum_{i=0}^N \alpha_i z^{-N+i}}$$

$$\alpha_0 = 1$$

For the  $N$ th-order multipoint frequency transformation the coefficients  $\alpha$  are

$$\begin{cases} \sum_{i=1}^N \alpha_{N-i} z_{old,k}^i \cdot z_{new,k}^{-i} - S \cdot z_{new,k}^{N-i} = -z_{old,k} - S \cdot z_{new,k} \\ z_{old,k} = e^{j\pi\omega_{old,k}} \\ z_{new,k} = e^{j\pi\omega_{new,k}} \\ k = 1, \dots, N \end{cases}$$

where

$\omega_{old,k}$  – Frequency location of the first feature in the prototype filter

$\omega_{new,k}$  – Position of the feature originally at  $\omega_{old,k}$  in the target filter

The mobility factor,  $S$ , specifies the mobility of either DC or Nyquist feature:

$$S = \begin{cases} 1 & Nyquist \\ -1 & DC \end{cases}$$

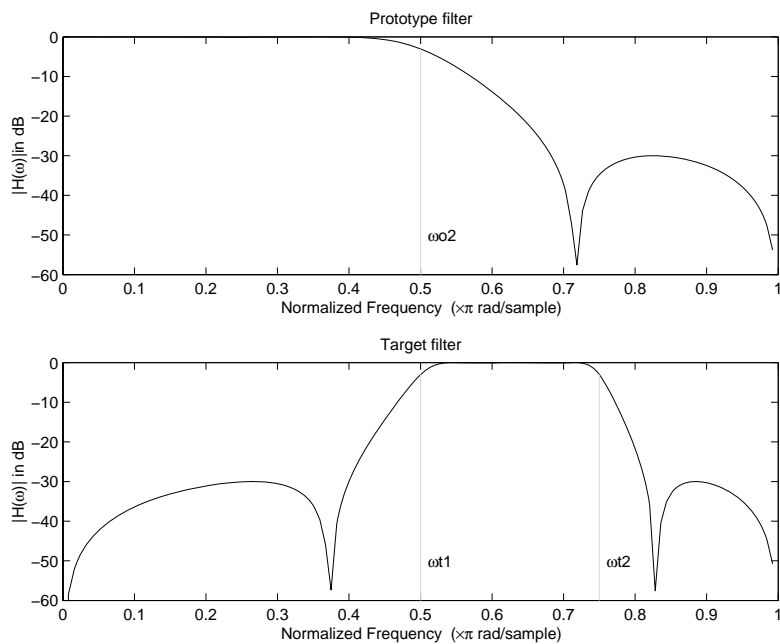
The example below shows how this transformation can be used to move features of the prototype lowpass filter originally at -0.5 and 0.5 to their new locations at 0.5 and 0.75, effectively changing a position of the filter passband. Here is the MATLAB code for generating the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates a passband from 0.5 to 0.75:

```
[num,den] = iir1p2xn(b, a, [-0.5, 0.5], [0.5, 0.75], `pass`);
```



**Figure 5-11: Example of Real Lowpass to Real Multipoint Mapping**

## Frequency Transformations for Complex Filters

This section discusses complex frequency transformation that take the complex prototype filter and convert it into a different complex target filter. The target filter has its frequency response modified in respect to the frequency response of the prototype filter according to the characteristic of the applied frequency transformation from:

- “Complex Frequency Shift” on page 5-26
- “Real Lowpass to Complex Bandpass” on page 5-28
- “Real Lowpass to Complex Bandstop” on page 5-29
- “Real Lowpass to Complex Multiband” on page 5-31
- “Real Lowpass to Complex Multipoint” on page 5-33
- “Complex Bandpass to Complex Bandpass” on page 5-35

### Complex Frequency Shift

Complex frequency shift transformation is the simplest first-order transformation that performs an exact mapping of one selected feature of the frequency response into its new location. At the same time it rotates the whole response of the prototype lowpass filter by the distance specified by the selection of the feature from the prototype filter and the target filter.

$$H_A(z) = \alpha z^{-1}$$

with  $\alpha$  given by

$$\alpha = e^{j2\pi(\nu_{new} - \nu_{old})}$$

where

$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new}$  — Position of the feature originally at  $\omega_{old}$  in the target filter

A special case of the complex frequency shift is a, so called, Hilbert Transformer. It can be designed by setting the parameter to  $|\alpha|=1$ , that is

$$\alpha = \begin{cases} 1 & \text{forward} \\ -1 & \text{inverse} \end{cases}$$

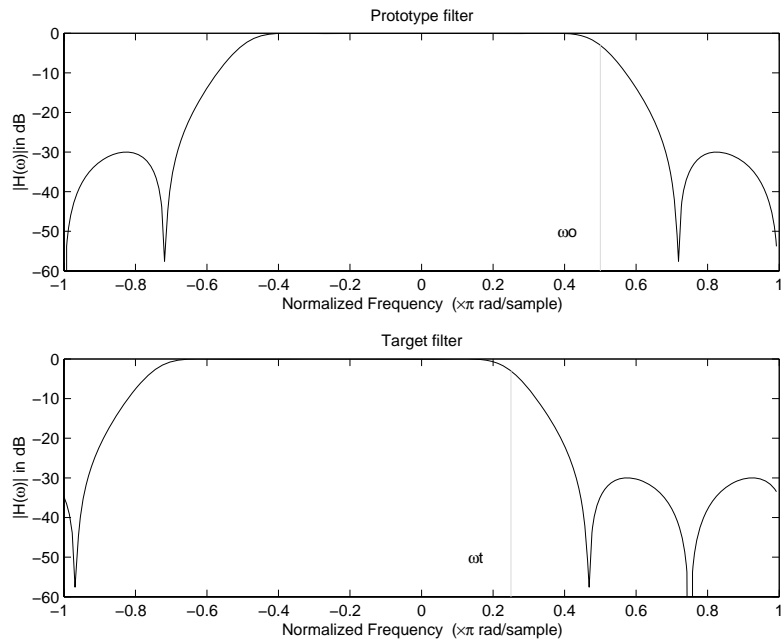
The example below shows how to apply this transformation to rotate the response of the prototype lowpass filter in either direction. Please note that because the transformation can be achieved by a simple phase shift operator, all features of the prototype filter will be moved by the same amount. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation moves the feature originally at 0.5 to 0.9:

```
[num,den] = iirshift(b, a, 0.5, 0.9);
```



**Figure 5-12: Example of Complex Frequency Shift Mapping**

## Real Lowpass to Complex Bandpass

This first-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into two new locations in the target filter creating a passband between them. Both Nyquist and DC features can be moved with the rest of the frequency response.

$$H_A(z) = \frac{\beta - \alpha z^{-1}}{z^{-1} - \alpha\beta}$$

with  $\alpha$  and  $\beta$  are given by

$$\alpha = \frac{\sin \frac{\pi}{4}(2w_{old} - w_{new,2} + w_{new,1})}{\sin \pi(2w_{old} + w_{new,2} - w_{new,1})}$$

$$\beta = e^{-j\pi(w_{new} - w_{old})}$$

where

$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new,1}$  — Position of the feature originally at  $(-\omega_{old})$  in the target filter

$\omega_{new,2}$  — Position of the feature originally at  $(+\omega_{old})$  in the target filter

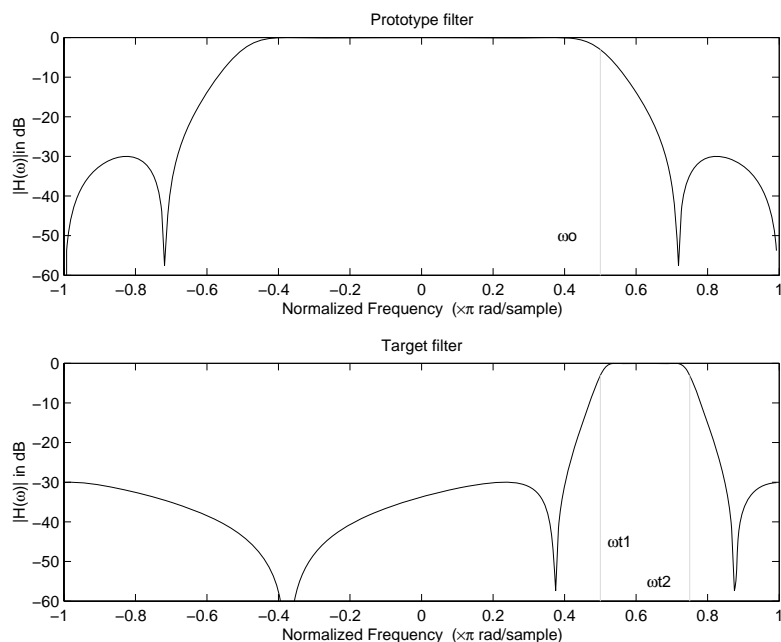
The example below shows the use of such a transformation for converting a real halfband lowpass filter into a complex bandpass filter with band edges at 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a half band elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The transformation creates a passband from 0.5 to 0.75:

```
[num,den] = iirlp2bpc(b, a, 0.5, [0.5 0.75]);
```



**Figure 5-13: Example of Real Lowpass to Complex Bandpass Mapping**

### Real Lowpass to Complex Bandstop

This first-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into two new locations in the target filter creating a stopband between them. Both Nyquist and DC features can be moved with the rest of the frequency response.

$$H_A(z) = \frac{\beta - \alpha z^{-1}}{\alpha\beta - z^{-1}}$$

with  $\alpha$  and  $\beta$  are given by

$$\alpha = \frac{\cos\pi(2w_{old} + v_{new,2} - v_{new,1})}{\cos\pi(2w_{old} - v_{new,2} + v_{new,1})}$$

$$\beta = e^{-j\pi(w_{new} - w_{old})}$$

where

$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new,1}$  — Position of the feature originally at  $(-\omega_{old})$  in the target filter

$\omega_{new,2}$  — Position of the feature originally at  $(+\omega_{old})$  in the target filter

The example below shows the use of such a transformation for converting a real halfband lowpass filter into a complex bandstop filter with band edges at 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

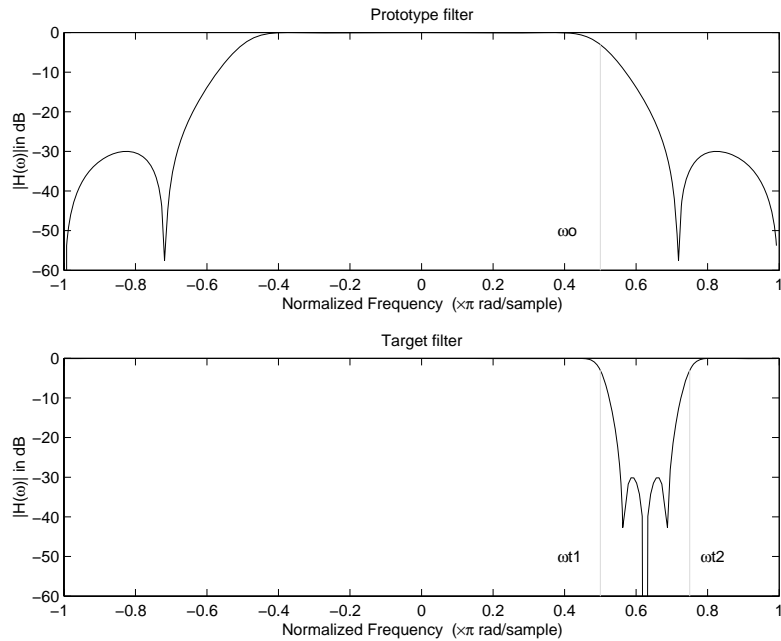
The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The transformation creates a stopband from 0.5 to 0.75:

```
[num,den] = iir1p2bsc(b, a, 0.5, [0.5 0.75]);
```





**Figure 5-14: Example of Real Lowpass to Complex Bandstop Mapping**

## Real Lowpass to Complex Multiband

This high-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into a number of new locations in the target filter. Its most common use is to convert a real lowpass with predefined passband and stopband ripples into a multiband filter with arbitrary band edges. The order of the mapping filter must be even, which corresponds to an even number of band edges in the target filter. The  $N$ th-order complex allpass mapping filter is given by the general transfer function form as shown below.

$$H_A(z) = S \frac{\sum_{i=0}^N \alpha_i z^{-i}}{\sum_{i=0}^N \alpha_i^* z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients  $\alpha$  are calculated from the system of linear equations:

$$\left\{ \begin{array}{l} \sum_{i=1}^N \Re(\alpha_i) \cdot [\cos\beta_{1,k} - \cos\beta_{2,k}] + \Im(\alpha_i) \cdot [\sin\beta_{1,k} + \sin\beta_{2,k}] = \cos\beta_{3,k} \\ \sum_{i=1}^N \Re(\alpha_i) \cdot [\sin\beta_{1,k} - \sin\beta_{2,k}] - \Im(\alpha_i) \cdot [\cos\beta_{1,k} + \cos\beta_{2,k}] = \sin\beta_{3,k} \\ \beta_{1,k} = -\pi[v_{old} \cdot (-1)^k + v_{new,k}(N-k)] \\ \beta_{2,k} = -\pi[\Delta C + v_{new,k}k] \\ \beta_{3,k} = -\pi[v_{old} \cdot (-1)^k + v_{new,k}N] \\ k = 1 \dots N \end{array} \right.$$

where

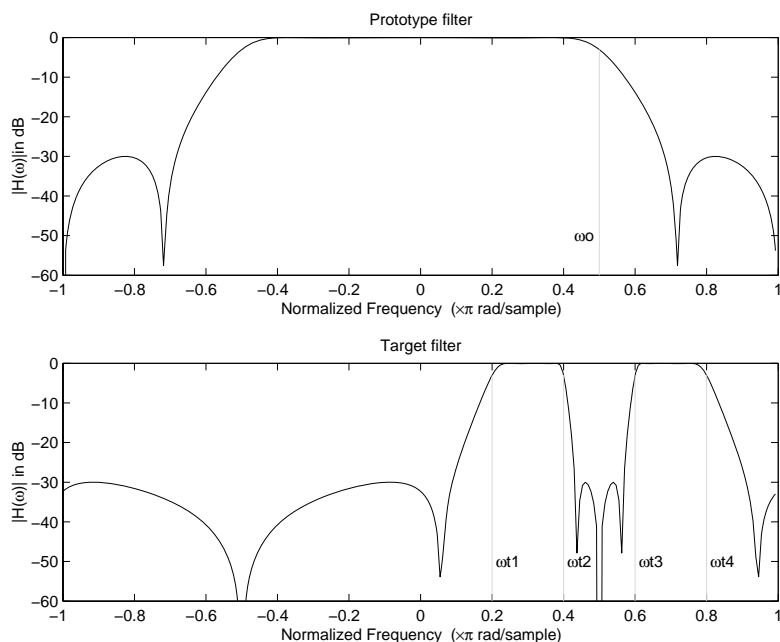
$\omega_{old}$  — Frequency location of the selected feature in the prototype filter

$\omega_{new,i}$  — Position of features originally at  $\pm\omega_{old}$  in the target filter

Parameter  $S$  is the additional rotation factor by the frequency distance  $\Delta C$ , giving the additional flexibility of achieving the required mapping:

$$S = e^{-j\pi\Delta C}$$

The example shows the use of such a transformation for converting a prototype real lowpass filter with the cutoff frequency at 0.5 into a multiband complex filter with band edges at 0.2, 0.4, 0.6 and 0.8, creating two passbands around the unit circle. Here is the MATLAB code for generating the figure.



**Figure 5-15: Example of Real Lowpass to Complex Multiband Mapping**

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates two complex passbands:

```
[num,den] = iirlp2mbc(b, a, 0.5, [0.2, 0.4, 0.6, 0.8]);
```

### Real Lowpass to Complex Multipoint

This high-order transformation performs an exact mapping of a number of selected features of the prototype filter frequency response to their new locations in the target filter. The  $N$ th-order complex allpass mapping filter is given by the general transfer function form as shown below.

$$H_A(z) = S \frac{\sum_{i=0}^N \alpha_i z^{-i}}{\sum_{i=0}^N \alpha_i^* z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients  $\alpha$  can be calculated from the system of linear equations:

$$\left\{ \begin{array}{l} \sum_{i=1}^N \Re(\alpha_i) \cdot [\cos \beta_{1,k} - \cos \beta_{2,k}] + \Im(\alpha_i) \cdot [\sin \beta_{1,k} + \sin \beta_{2,k}] = \cos \beta_{3,k} \\ \sum_{i=1}^N \Re(\alpha_i) \cdot [\sin \beta_{1,k} - \sin \beta_{2,k}] - \Im(\alpha_i) \cdot [\cos \beta_{1,k} + \cos \beta_{2,k}] = \sin \beta_{3,k} \\ \beta_{1,k} = -\frac{\pi}{2} [w_{old,k} + w_{new,k}(N-k)] \\ \beta_{2,k} = -\frac{\pi}{2} [2\Delta C + w_{new,k}k] \\ \beta_{3,k} = -\frac{\pi}{2} [w_{old,k} + w_{new,k}N] \\ k = 1 \dots N \end{array} \right.$$

where

$\omega_{old,k}$  — Frequency location of the first feature in the prototype filter

$\omega_{new,k}$  — Position of the feature originally at  $\omega_{old,k}$  in the target filter

Parameter  $S$  is the additional rotation factor by the frequency distance  $\Delta C$ , giving the additional flexibility of achieving the required mapping:

$$S = e^{-j\pi\Delta C}$$

The example below shows how this transformation can be used to move one selected feature of the prototype lowpass filter originally at -0.5 to two new frequencies -0.5 and 0.1, and the second feature of the prototype filter from 0.5

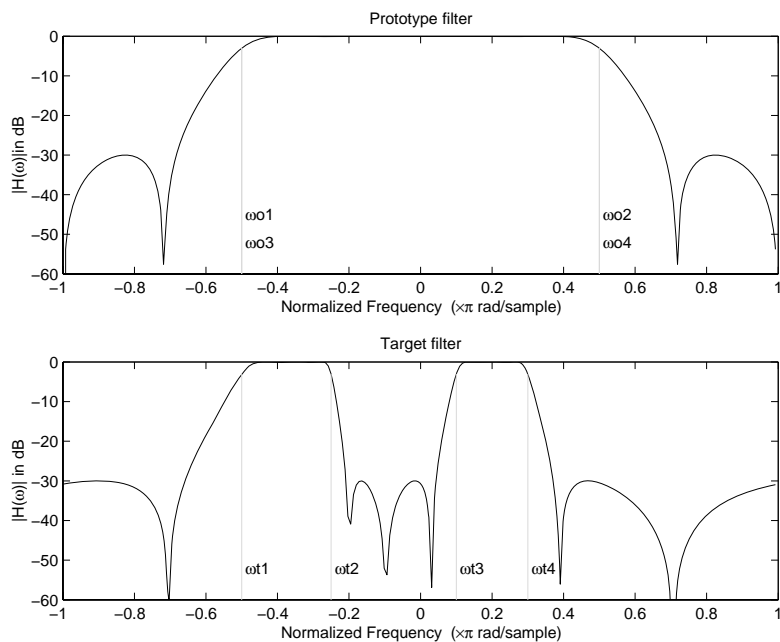
to new locations at -0.25 and 0.3. This creates two nonsymmetric passbands around the unit circle, creating a complex filter. Here is the MATLAB code for generating the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates two nonsymmetric passbands:

```
[num,den] = iir1p2xc(b,a,0.5*[-1,1,-1,1], [-0.5,-0.25,0.1,0.3]);
```



**Figure 5-16: Example of Real Lowpass to Complex Multipoint Mapping**

## Complex Bandpass to Complex Bandpass

This first-order transformation performs an exact mapping of two selected features of the prototype filter frequency response into two new locations in the

target filter. Its most common use is to adjust the edges of the complex bandpass filter.

$$H_A(z) = \frac{\alpha(\gamma - \beta z^{-1})}{z^{-1} - \beta\gamma}$$

with  $\alpha$  and  $\beta$  are given by

$$\alpha = \frac{\sin \frac{\pi}{4}((w_{old,2} - w_{old,1}) - (w_{new,2} - w_{new,1}))}{\sin \frac{\pi}{4}((w_{old,2} - w_{old,1}) + (w_{new,2} - w_{new,1}))}$$

$$\alpha = e^{-j\pi(w_{old,2} - w_{old,1})}$$

$$\gamma = e^{-j\pi(w_{new,2} - w_{new,1})}$$

where

$\omega_{old,1}$  — Frequency location of the first feature in the prototype filter

$\omega_{old,2}$  — Frequency location of the second feature in the prototype filter

$\omega_{new,1}$  — Position of the feature originally at  $\omega_{old,1}$  in the target filter

$\omega_{new,2}$  — Position of the feature originally at  $\omega_{old,2}$  in the target filter

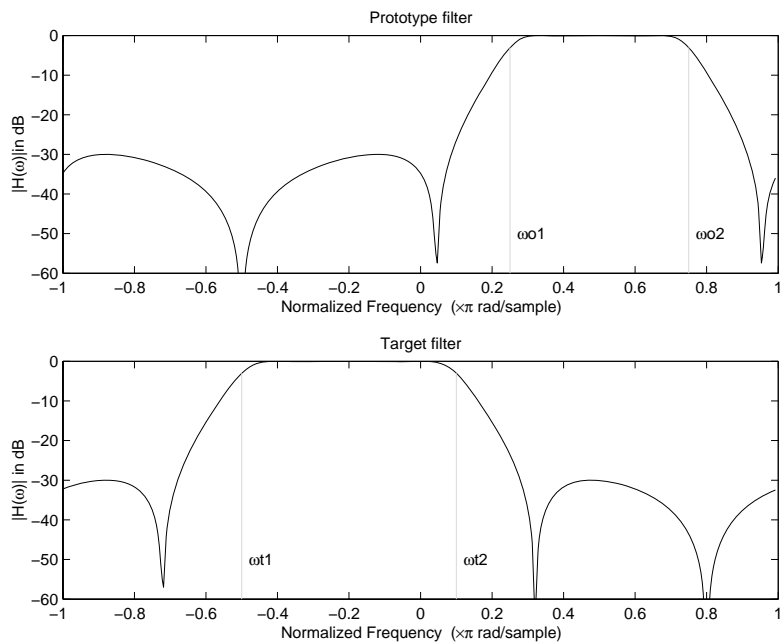
The example below shows how this transformation can be used to modify the position of the passband of the prototype filter, either real or complex. In the example below the prototype filter passband spanned from 0.5 to 0.75. It was converted to having a passband between -0.5 and 0.1. Here is the MATLAB code for generating the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates a passband from 0.25 to 0.75:

```
[num,den] = iirbpc2bpc(b, a, [0.25, 0.75], [-0.5, 0.1]);
```



**Figure 5-17: Example of Complex Bandpass to Complex Bandpass Mapping**





# Using FDATool with the Filter Design Toolbox

---

Designing Advanced Filters in FDATool (p. 6-5)

Using FDATool to design more advanced filters. This section assumes you are familiar with FDATool from the Signal Processing Toolbox.

Switching FDATool to Quantization Mode (p. 6-8)

After you open FDATool, this section explains how to access the quantization features in the tool.

Quantizing Filters in the Filter Design and Analysis Tool (p. 6-12)

Explains how you quantize a filter in FDATool.

Analyzing Filters with a Noise-Based Method (p. 6-23)

FDATool provides a variety of analysis methods for quantized filters; this section explains how to use two of them.

Scaling Second-Order Section Filters (p. 6-30)

You can adjust the way FDATool scales SOS filters. To learn how, read this section.

Reordering the Sections of Second-Order Section Filters (p. 6-38)

Shows you how to change the order of the sections in an SOS filter.

Viewing SOS Filter Sections (p. 6-46)

Shows you how to use the SOS View feature in FDATool to analyze the sections of SOS filters.

Importing and Exporting Quantized Filters (p. 6-53)

Shows you how to import and export filters to and from your MATLAB workspace, as well as to other destinations.

Importing XILINX Coefficient (.COE) Files (p. 6-58)

Import the coefficients from a XILINX .coe file to create a quantized filter in FDATool.

Transforming Filters (p. 6-59)

Describes how you use the filter transformation capability in FDATool to change the magnitude response of your FIR or IIR filters in the tool.

Designing Multirate Filters in FDATool (p. 6-70)

Explains how to use FDATool to design multirate filters. This section assumes you are familiar with FDATool from the Signal Processing Toolbox and you are familiar with `mfilt` objects.

Quantizing Multirate Filters (p. 6-81)

Explains how to use FDATool to quantize multirate filters.

Realizing Filters as Simulink Subsystem Blocks (p. 6-84)

Using the Realize Model feature to create a Simulink model of your quantized filter as a subsystem block.

Getting Help for FDATool (p. 6-89)

Shows you how to get help about the features in FDATool, such as using Help or using the What's This option.

---

The Filter Design Toolbox adds new dialogs and operating modes, and new menu selections, to the Filter Design and Analysis Tool (FDATool) provided by the Signal Processing Toolbox. From the new dialogs, one titled **Set Quantization Parameters** and one titled **Frequency Transformations**, you can:

- Design advanced filters that Signal Processing Toolbox does not provide the design tools to develop.
- View Simulink models of the filter structures available in the toolbox.
- Quantize double-precision filters you design in this GUI using the design mode.
- Quantize double-precision filters you import into this GUI using the import mode.
- Analyze quantized filters.
- Scale second-order section filters.
- Select the quantization settings for the properties of the quantized filter displayed by the tool:
  - Coefficients—select the quantization options applied to the filter coefficients
  - Input/output—control how the filter processes input and output data
  - Filter Internals—specify how the arithmetic for the filter behaves
- Design multirate filters.
- Transform both FIR and IIR filters from one response to another.

After you import a filter in to FDATool, the options on the quantization dialog let you quantize the filter and investigate the effects of various quantization settings.

Options in the frequency transformations dialog let you change the frequency response of your filter, keeping various important features while changing the response shape.

This section presents the following information and procedures for using FDATool:

- “Designing Advanced Filters in FDATool” on page 6-5
- “Switching FDATool to Quantization Mode” on page 6-8
- “Quantizing Filters in the Filter Design and Analysis Tool” on page 6-12

- “Analyzing Filters with a Noise-Based Method” on page 6-23
- “Choosing Quantized Filter Structures” on page 6-28
- “Reordering the Sections of Second-Order Section Filters” on page 6-38
- “Viewing SOS Filter Sections” on page 6-46
- “Importing XILINX Coefficient (.COE) Files” on page 6-58
- “Transforming Filters” on page 6-59
- “Designing Multirate Filters in FDATool” on page 6-70
- “Realizing Filters as Simulink Subsystem Blocks” on page 6-84

## Designing Advanced Filters in FDATool

Adding the Filter Design Toolbox to your tool suite adds a number of filter design techniques to FDATool. Use the new filter responses to develop filters that meet more complex requirements than those you can design in the Signal Processing Toolbox. While the designs in FDATool are available as command line functions, the graphical user interface of FDATool makes the design process more clear and easier to accomplish.

As you select a response type, the options in the panels to the right in FDATool change to let you set the values that define your filter. You also see that the analysis area includes a diagram (called a *design mask*) that describes the options for the filter response you choose.

By reviewing the mask you can see how the options are defined and how to use them. While this is usually straightforward for lowpass or highpass filter responses, setting the options for the arbitrary response types or the peaking/notching filters is more complicated. Having the masks leads you to your result more easily.

Changing the filter design method changes the available response type options. Similarly, the response type you select may change the filter design methods you can choose.

### Example—Design a Notch Filter

Notch filters aim to remove one or a few frequencies from a broader spectrum. You must specify the frequencies to remove by setting the filter design options in FDATool appropriately:

- Response Type
- Design Method
- Frequency Specifications
- Magnitude Specifications

Here is how you design a notch filter that removes concert A (440 Hz) from an input musical signal spectrum.

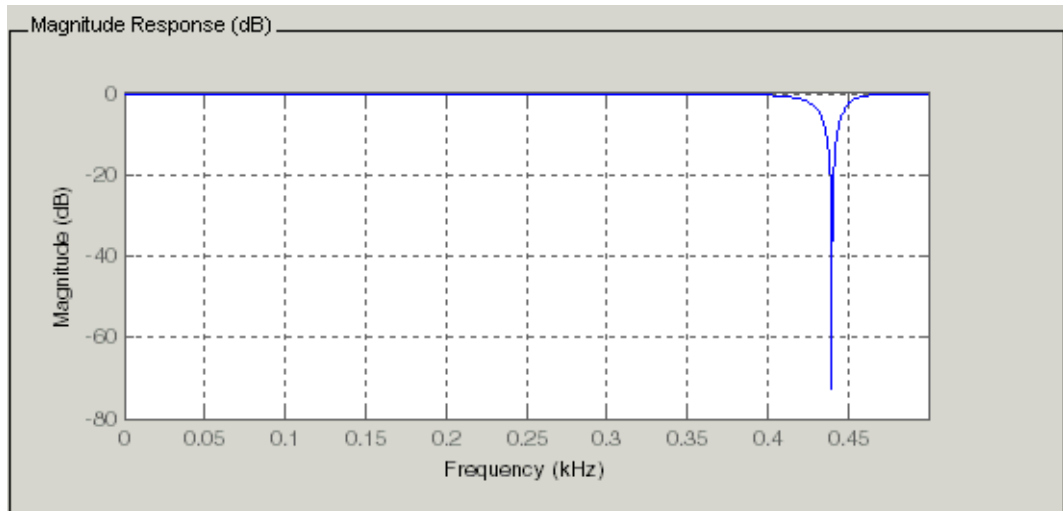
- 1 Select Notching from the **Differentiator** list in **Response Type**.
- 2 Select **IIR** in **Filter Design Method** and choose Single Notch from the list.

- 3 For the **Frequency Specifications**, set **Units** to Hz and **Fs**, the full scale frequency, to 10000.
- 4 Set the location of the center of the notch, in either normalized frequency or Hz. For the notch center at 440 Hz, enter 440.
- 5 To shape the notch, enter the **bandwidth**, bw, to be 40.
- 6 Leave the **Magnitude Specification** in dB (the default) and leave **Apass** as 1.
- 7 Click Design Filter.

FDATool computes the filter coefficients and plots the filter magnitude response in the analysis area for you to review.

When you design a single notch filter, you do not have the option of setting the filter order—the **Filter Order** options are disabled.

Your filter should look about like this:



For more information about a design method, refer to the online Help system. For instance, to get further information about the **Q** setting for the notch filter in FDATool, enter

```
doc iirnotch
```

at the prompt. This opens the Help browser and displays the reference page for function `iirnotch`.

Designing other filters follows a similar procedure, adjusting for different design specification options as each design requires.

Any one of the designs may be quantized in FDATool and analyzed with the available analyses on the **Analysis** menu. For more general information about FDATool, such as the user interface and areas, refer to the FDATool documentation in the Signal Processing Toolbox documentation. One way to do this is to enter

```
doc signal/fdatool
```

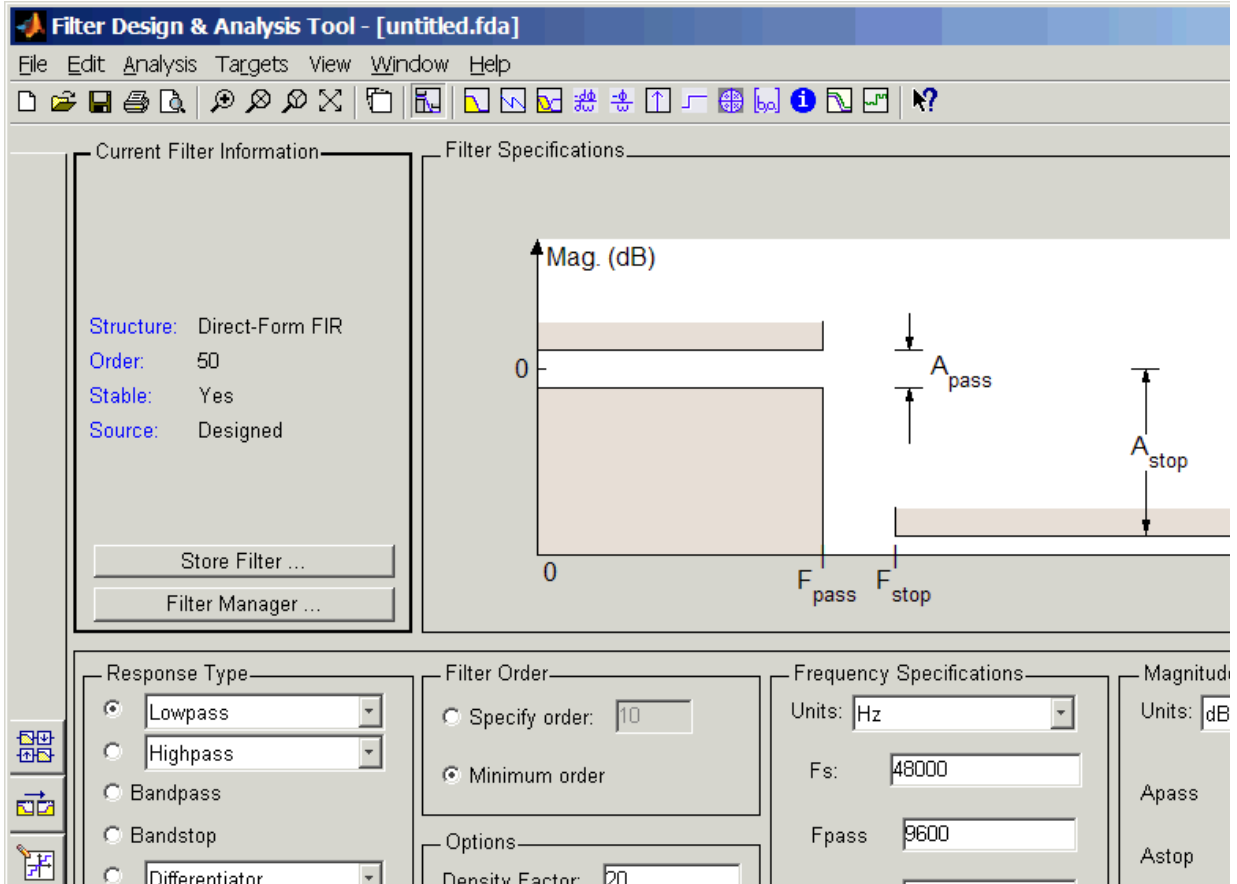
at the prompt. The `signal` qualifier is necessary to open the reference page in the Signal Processing Toolbox documentation, rather than the page in the Filter Design Toolbox documentation. You might also look at the general section on FDATool in the *Signal Processing Toolbox User's Guide*.

## Switching FDATool to Quantization Mode

You use the quantization mode in FDATool to quantize filters. Quantization represents the fourth operating mode for FDATool, along with the filter design, filter transformation, and import modes. To switch to quantization mode, open FDATool from the MATLAB command prompt by entering

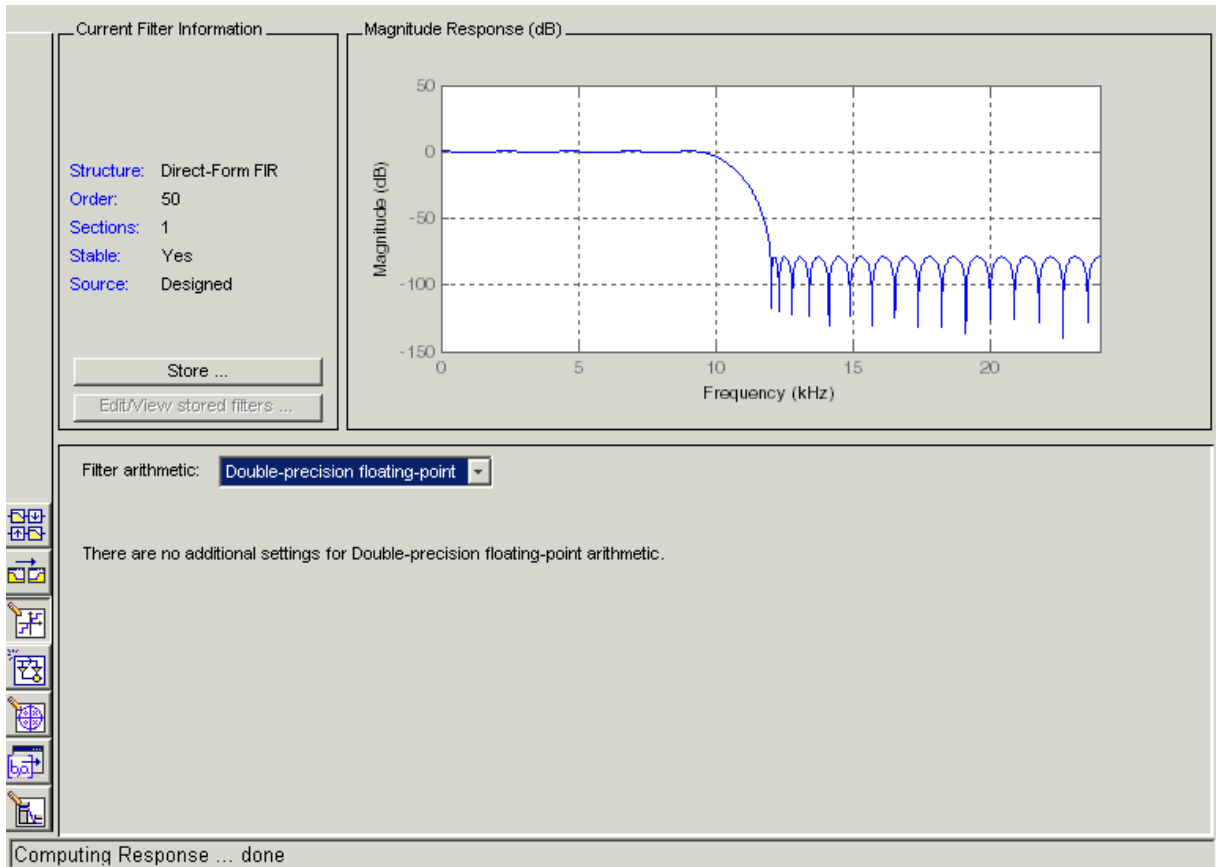
```
fdatool
```

You see FDATool in this configuration.



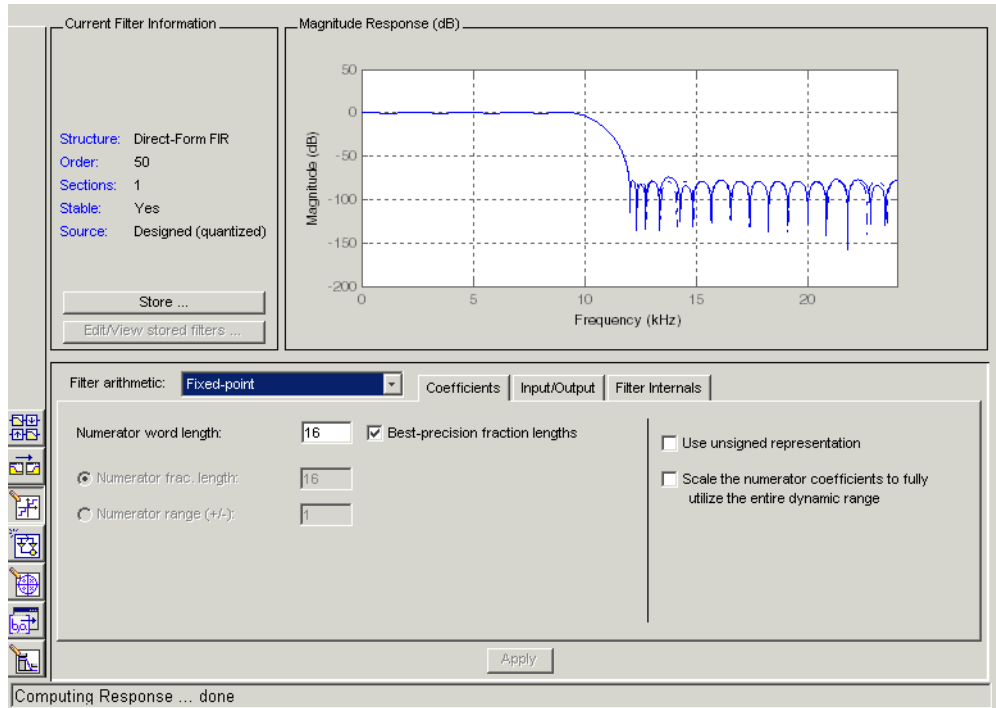


When FDATool opens, click the **Set Quantization Parameters** button on the side bar. FDATool switches to quantization mode and you see the following panel at the bottom of FDATool, with the default double-precision option shown for **Filter arithmetic**.



The **Filter arithmetic** option lets you quantize filters and investigate the effects of changing quantization settings. To enable the quantization settings in FDATool, select Fixed-point from the **Filter Arithmetic**.

The quantization options appear in the lower panel of FDATool. You see tabs that access various sets of options for quantizing your filter.



You use the following tabs in the dialog to perform tasks related to quantizing filters in FDATool:

- **Coefficients** provides access the settings for defining the coefficient quantization. This is the default active panel when you switch FDATool to quantization mode without a quantized filter in the tool. When you import a fixed-point filter into FDATool, this is the active pane when you switch to quantization mode.
- **Input/Output** switches FDATool to the options for quantizing the inputs and outputs for your filter.
- **Filter Internals** lets you set a variety of options for the arithmetic your filter performs, such as how the filter handles the results of multiplication operations or how the filter uses the accumulator.

- **Apply**—applies changes you make to the quantization parameters for your filter.

## Quantizing Filters in the Filter Design and Analysis Tool

Quantized filters have properties that define how they quantize data you filter. Use the **Set Quantization Parameters** dialog in FDATool to set the properties. Using options in the **Set Quantization Parameters** dialog, FDATool lets you perform a number of tasks:

- Create a quantized filter from a double-precision filter after either importing the filter from your workspace, or using FDATool to design the prototype filter.
- Create a quantized filter that has the default structure (Direct form II transposed) or any structure you choose, and other property values you select.
- Change the quantization property values for a quantized filter after you design the filter or import it from your workspace.

When you click **Set Quantization Parameters**, and then change **Filter Arithmetic** to **Fixed-point**, the quantized filter panel opens in FDATool, with the coefficient quantization options set to default values. In this image, you see the options for an SOS filter. Some of the options shown apply only to SOS filters. Other filter structures present a subset of the options you see here.

The screenshot shows the 'Set Quantization Parameters' dialog box in FDATool. The 'Filter arithmetic' dropdown is set to 'Fixed-point'. The 'Coefficients' tab is selected. The dialog contains the following options:

- Coefficient word length: 16
- Best-precision fraction lengths
- Use unsigned representation
- Numerator frac. length: 13
- Scale Values frac. length: 14
- Numerator range (+/-): -2
- Scale Values range (+/-): 1
- Denominator frac. length: 14
- Denominator range (+/-): 1

An 'Apply' button is located at the bottom of the dialog.

## Coefficients Options

To let you set the properties for the filter coefficients that make up your quantized filter, FDATool lists options for numerator word length (and denominator word length if you have an IIR filter). The following table lists each coefficients option and a short description of what the option setting does in the filter.

Option Name	When Used	Description
<b>Numerator Word Length</b>	FIR filters only	Sets the word length used to represent numerator coefficients in FIR filters.
<b>Numerator Frac. Length</b>	FIR/IIR	Sets the fraction length used to interpret numerator coefficients in FIR filters.
<b>Numerator Range (+/-)</b>	FIR/IIR	Lets you set the range the numerators represent. You use this instead of the <b>Numerator Frac. Length</b> option to set the precision. When you enter a value $x$ , the resulting range is $-x$ to $x$ . Range must be a positive integer.
<b>Coefficient Word Length</b>	IIR filters only	Sets the word length used to represent both numerator and denominator coefficients in IIR filters. You cannot set different word lengths for the numerator and denominator coefficients.
<b>Denominator Frac. Length</b>	IIR filters	Sets the fraction length used to interpret denominator coefficients in IIR filters.
<b>Denominator Range (+/-)</b>	IIR filters	Lets you set the range the denominator coefficients represent. You use this instead of the <b>Denominator Frac. Length</b> option to set the precision. When you enter a value $x$ , the resulting range is $-x$ to $x$ . Range must be a positive integer.

Option Name	When Used	Description
<b>Best-precision fraction lengths</b>	All filters	Directs FDATool to select the fraction lengths for numerator (and denominator where available) values to maximize the filter performance. Selecting this option disables all of the fraction length options for the filter.
<b>Scale Values frac. length</b>	SOS IIR filters	Sets the fraction length used to interpret the scale values in SOS filters.
<b>Scale Values range (+/-)</b>	SOS IIR filters	Lets you set the range the SOS scale values represent. You use this with SOS filters to adjust the scaling used between filter sections. Setting this value disables the <b>Scale Values frac. length</b> option. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
<b>Use unsigned representation</b>	All filters	Tells FDATool to interpret the coefficients as unsigned values.
<b>Scale the numerator coefficients to fully utilize the entire dynamic range</b>	All filters	Directs FDATool to scale the numerator coefficients to effectively use the dynamic range defined by the numerator word length and fraction length format.

### Input/Output Options

The options that specify how the quantized filter uses input and output values are listed in the table below. In the following picture you see the options for an SOS filter.

Filter arithmetic: Fixed-point

Coefficients | Input/Output | Filter Internals

Input word length: 16

Output word length: 16

Stage input word length: 16

Input fraction length: 15

Avoid Overflow

Avoid overflow

Input range (+/-): 1

Output fraction length: 11

Output range (+/-): 1

Stage input fraction length: 9

Stage output word length: 16

Avoid overflow

Stage output fraction length: 11

Apply

Option Name	When Used	Description
<b>Input Word Length</b>	All filters	Sets the word length used to represent the input to a filter.
<b>Input fraction length</b>	All filters	Sets the fraction length used to interpret input values to filter.
<b>Input range (+/-)</b>	All filters	Lets you set the range the inputs represent. You use this instead of the <b>Input fraction length</b> option to set the precision. When you enter a value $x$ , the resulting range is $-x$ to $x$ . Range must be a positive integer.
<b>Output word length</b>	All filters	Sets the word length used to represent the output from a filter.

<b>Option Name</b>	<b>When Used</b>	<b>Description</b>
<b>Avoid overflow</b>	All filters	Directs the filter to set the fraction length for the input to prevent the output values from exceeding the available range as defined by the word length. Clearing this option lets you set <b>Output fraction length</b> .
<b>Output fraction length</b>	All filters	Sets the fraction length used to represent output values from a filter.
<b>Output range (+/-)</b>	All filters	Lets you set the range the outputs represent. You use this instead of the <b>Output fraction length</b> option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
<b>Stage input word length</b>	SOS filters only	Sets the word length used to represent the input to an SOS filter section.
<b>Avoid overflow</b>	SOS filters only	Directs the filter to use a fraction length for stage inputs that prevents overflows in the values. When you clear this option, you can set <b>Stage input fraction length</b> .
<b>Stage input fraction length</b>	SOS filters only	Sets the fraction length used to represent input to a section of an SOS filter.
<b>Stage output word length</b>	SOS filters only	Sets the word length used to represent the output from an SOS filter section.



Option Name	When Used	Description
<b>Avoid overflow</b>	SOS filters only	Directs the filter to use a fraction length for stage outputs that prevents overflows in the values. When you clear this option, you can set <b>Stage output fraction length</b> .
<b>Stage output fraction length</b>	SOS filters only	Sets the fraction length used to represent the output from a section of an SOS filter.

## Filter Internals Options

The options that specify how the quantized filter performs arithmetic operations are listed in the table after the figure. In the following picture you see the options for an SOS filter.

The screenshot shows the 'Filter Internals' tab of a configuration window. At the top, 'Filter arithmetic' is set to 'Fixed-point'. Below this, there are tabs for 'Coefficients', 'Input/Output', and 'Filter Internals'. The 'Filter Internals' section contains the following settings:

- Round towards: Nearest (convergent)
- Overflow Mode: Wrap
- Product mode: Full precision
- Accum. mode: Keep MSB
- State word length: 16
- Product word length: 32
- Accum. word length: 40
- Num. fraction length: 29
- Num. fraction length: 29
- Den. fraction length: 29
- Den. fraction length: 29
- State fraction length: 15
- Avoid overflow
- Cast signals before accum.

An 'Apply' button is located at the bottom center of the dialog.

<b>Option</b>	<b>Equivalent Filter Property (using wildcard *)</b>	<b>Description</b>
<b>Round towards</b>	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). Choose from one of:</p> <ul style="list-style-type: none"> <li>▪ <b>Ceiling</b>—round up to the nearest allowable quantized value.</li> <li>▪ <b>Floor</b>—round down to the next allowable quantized value.</li> <li>▪ <b>Nearest</b>—round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</li> <li>▪ <b>Nearest (convergent)</b>—round to the next allowable quantized value. For numbers that lie halfway between the two nearest allowable values, round up to the nearest value only when the least significant bit after rounding would be a 1.</li> <li>▪ <b>Zero</b>—round negative numbers and positive numbers towards zero to the next allowable quantized value</li> </ul>

<b>Option</b>	<b>Equivalent Filter Property (using wildcard *)</b>	<b>Description</b>
<b>Overflow Mode</b>	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).
<b>Filter Product (Multiply) Options</b>		
<b>Product Mode</b>	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 word length. Specify all lets you set the fraction length applied to the results of product operations.
<b>Product word length</b>	*ProdWordLength	Sets the word length applied to interpret the results of multiply operations.
<b>Num. fraction length</b>	NumProdFracLength	Sets the fraction length used to interpret the results of product operations that involve numerator coefficients.
<b>Den. fraction length</b>	DenProdFracLength	Sets the fraction length used to interpret the results of product operations that involve denominator coefficients.

<b>Option</b>	<b>Equivalent Filter Property (using wildcard *)</b>	<b>Description</b>
<b>Filter Sum Options</b>		
<b>Accum. mode</b>	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 this to Specify all.
<b>Accum. word length</b>	*AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
<b>Num. fraction length</b>	NumAccumFracLength	Sets the fraction length used to interpret the numerator coefficients.
<b>Den. fraction length</b>	DenAccumFracLength	Sets the fraction length the filter uses to interpret denominator coefficients.
<b>Cast signals before sum</b>	CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams for each filter structure) before performing sum operations.
<b>Filter State Options</b>		
<b>State word length</b>	*StateWordLength	Sets the word length used to represent the filter states. Applied to both numerator- and denominator-related states

<b>Option</b>	<b>Equivalent Filter Property (using wildcard *)</b>	<b>Description</b>
<b>Avoid overflow</b>	None	Prevent overflows in arithmetic calculations by setting the fraction length appropriately.
<b>State fraction length</b>	*StateFracLength	Lets you set the fraction length applied to interpret the filter states. Applied to both numerator- and denominator-related states

## Filter Internals Options for CIC Filters

CIC filters use slightly different options for specifying the fixed-point arithmetic in the filter. The next table shows and describes the options.

### Example—Quantize Double-Precision Filters

When you are quantizing a double-precision filter by switching to fixed-point or single-precision floating point arithmetic, follow these steps.

- 1** Click **Set Quantization Parameters** to display the **Set Quantization Parameters** pane in FDATool.
- 2** Select Single-precision floating point or Fixed-point from **Filter arithmetic**.

When you select one of the optional arithmetic settings, FDATool quantizes the current filter according to the settings of the options in the Set Quantization Parameter panes, and changes the information displayed in the analysis area to show quantized filter data.

- 3** In the quantization panes, set the options for your filter. Set options for **Coefficients**, **Input/Output**, and **Filter Internals**.
- 4** Click **Apply**.

FDATool quantizes your filter using your new settings.

- 5 Use the analysis features in FDATool to determine whether your new quantized filter meets your requirements.

### **Example—Change the Quantization Properties of Quantized Filters**

When you are changing the settings for the quantization of a quantized filter, or after you import a quantized filter from your MATLAB workspace, follow these steps to set the property values for the filter:

- 1 Verify that the current filter is quantized.
- 2 Click **Set Quantization Parameters** to display the **Set Quantization Parameters** panel.
- 3 Review and select property settings for the filter quantization: **Coefficients**, **Input/Output**, and **Filter Internals**. Settings for options on these panes determine how your filter quantizes data during filtering operations.
- 4 Click **Apply** to update your current quantized filter to use the new quantization property settings from Step 3.
- 5 Use the analysis features in FDATool to determine whether your new quantized filter meets your requirements.

## Analyzing Filters with a Noise-Based Method

One technique for estimating the frequency response for quantized filters is the magnitude response estimate. FDATool offers this noise-based method as a filter analysis tool accessible from the toolbar.

### Using the Magnitude Response Estimate Method

After you design and quantize your filter, the **Magnitude Response Estimate** option on the **Analysis** menu lets you apply the noise loading method to your filter. When you select **Analysis -> Magnitude Response Estimate** from the menubar, FDATool immediately starts the Monte Carlo trials that form the basis for the method and runs the analysis, ending by displaying the results in the analysis area in FDATool.

With the noise-based method, you estimate the complex frequency response for your filter as determined by applying a noise- like signal to the filter input. **Magnitude Response Estimate** uses the Monte Carlo trials to generate a noise signal that contains complete frequency content across the range 0 to  $F_s$ . The first time you run the analysis, magnitude response estimate uses default settings for the various conditions that define the process, such as the number of test points and the number of trials.

<b>Analysis Parameter</b>	<b>Default Setting</b>	<b>Description</b>
<b>Number of Points</b>	512	Number of equally spaced points around the upper half of the unit circle.
<b>Frequency Range</b>	0 to $F_s/2$	Frequency range of the plot x-axis.
<b>Frequency Units</b>	Hz	Units for specifying the frequency range.
<b>Sampling Frequency</b>	48000	Inverse of the sampling period.

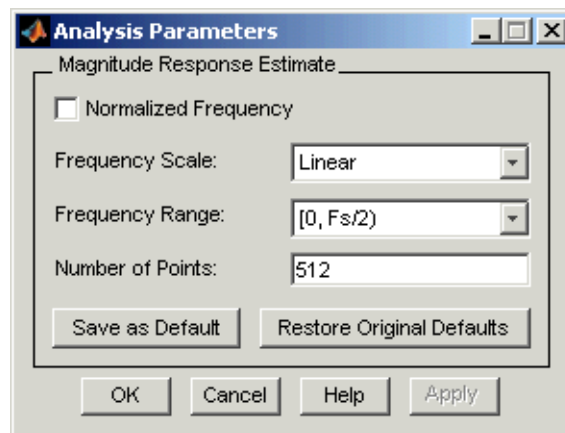
Analysis Parameter	Default Setting	Description
Frequency Scale	dB	Units used for the y-axis display of the output.
Normalized Frequency	Off	Use normalized frequency for the display.

After your first analysis run ends, open the **Analysis Parameters** dialog and adjust your settings appropriately, such as changing the number of trials or number of points.

To open the **Analysis Parameters** dialog, use either of the next procedures when you have a quantized filter in FDATool:

- Select **Analysis -> Analysis Parameters** from the menu bar
- Right-click in the filter analysis area and select **Analysis Parameters** from the context menu

Whichever option you choose opens the dialog as shown in the figure. Notice that the settings for the options reflect the defaults.






### Example—Noise Method Applied to a Filter

To demonstrate the magnitude response estimate method, start by creating a quantized filter. For this example, use FDATool to design a sixth-order Butterworth IIR filter.

#### To Use Noise-Based Analysis in FDATool

- 1 Enter `fdatool` at the MATLAB prompt to launch FDATool.
- 2 Under **Response Type**, select **Highpass**.
- 3 Select IIR in **Design Method**. Then select Butterworth.
- 4 To set the filter order to 6, select **Specify order** under **Filter Order**. Enter 6 in the text box.
- 5 Click **Design Filter**.

In FDATool, the analysis area changes to display the magnitude response for your filter.

- 6 To generate the quantized version of your filter, using default quantizer settings, click  on the side bar.

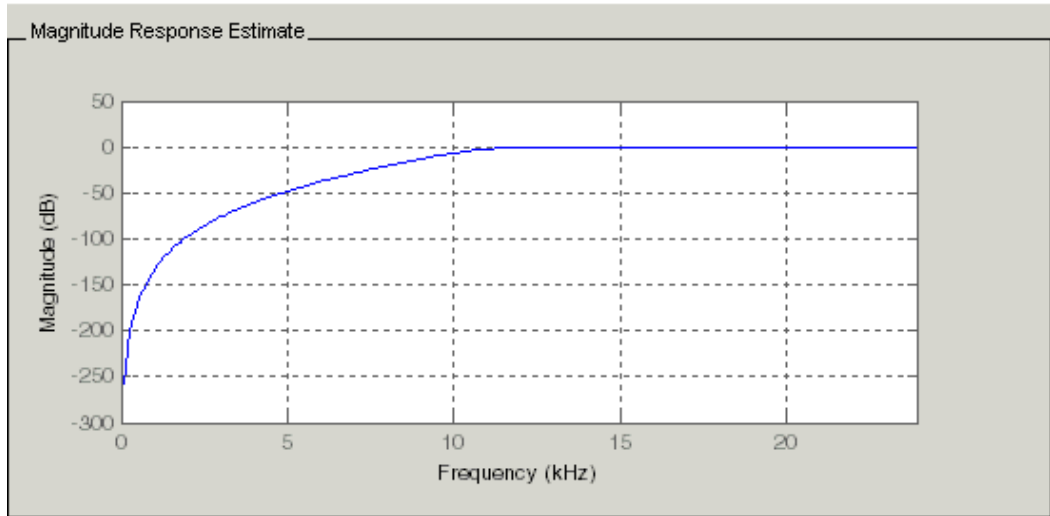
FDATool switches to quantization mode and displays the quantization panel.

- 7 From **Filter arithmetic**, select fixed-point.

Now the analysis areas shows the magnitude response for both filters—your original filter and the fixed-point arithmetic version.

- 8 Finally, to use noise-based estimation on your quantized filter, select **Analysis -> Magnitude Response Estimate** from the menubar.

FDATool runs the trial, calculates the estimated magnitude response for the filter, and displays the result in the analysis area as shown in this figure.



In the figure you see the magnitude response as estimated by the analysis method.

### To View the Noise Power Spectrum

When you use the noise method to estimate the magnitude response of a filter, FDATool simulates and applies a spectrum of noise values to test your filter response. While the simulated noise is essentially white, you might want to see the actual spectrum that FDATool used to test your filter.

From the **Analysis** menu bar option, select **Round-off Noise Power Spectrum**. In the analysis area in FDATool, you see the spectrum of the noise used to estimate the filter response. The details of the noise spectrum, such as the range and number of data points, appear in the **Analysis Parameters** dialog.

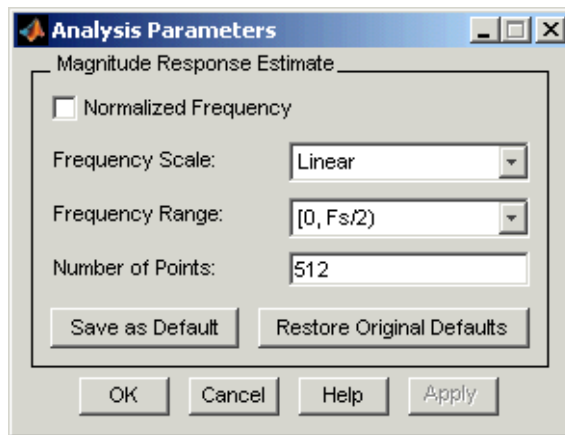
### To Change Your Noise Analysis Parameters

In “Example—Noise Method Applied to a Filter”, you used synthetic white noise to estimate the magnitude response for a fixed-point highpass Butterworth filter. Since you ran the estimate only once in FDATool, your noise analysis used the default analysis parameters settings shown in “Using the Magnitude Response Estimate Method”.

To change the settings, follow these steps after the first time you use the noise estimate on your quantized filter.

- 1 With the results from running the noise estimating method displayed in the FDATool analysis area, select **Analysis->Analysis Parameters** from the menubar.

To give you access to the analysis parameters, the **Analysis Parameters** dialog opens as shown here (with default settings).



- 2 To use more points in the spectrum to estimate the magnitude response, change **Number of Points** to 1024 and click **OK** to run the analysis.

FDATool closes the **Analysis Parameters** dialog and reruns the noise estimate, returning the results in the analysis area.

To rerun the test without closing the dialog, press **Enter** after you type your new value into a setting, then click **Apply**. Now FDATool runs the test without closing the dialog. When you want to try many different settings for the noise-based analysis, this is a useful shortcut.

### Comparing the Estimated and Theoretical Magnitude Responses

An important measure of the effectiveness of the noise method for estimating the magnitude response of a quantized filter is to compare the estimated response to the theoretical response.

One way to do this comparison is to overlay the theoretical response on the estimated response. While you have the Magnitude Response Estimate displaying in FDATool, select **Analysis->Overlay Analysis** from the menu bar. Then select **Magnitude Response** to show both response curves plotted together in the analysis area.

### Choosing Quantized Filter Structures

FDATool lets you change the structure of any quantized filter. Use the **Convert structure** option to change the structure of your filter to one that meets your needs.

To learn about changing the structure of a filter in FDATool, refer to “Converting to a New Structure” in your Signal Processing Toolbox documentation.

### Converting the Structure of a Quantized Filter

You use the **Convert structure** option to change the structure of filter. When the **Source** is **Designed(Quantized)** or **Imported(Quantized)**, **Convert structure** lets you recast the filter to one of the following structures:

- “Direct Form II Transposed Filter Structure” on page 7-52
- “Direct Form I Transposed Filter Structure” on page 7-48
- “Direct Form II Filter Structure” on page 7-49
- “Direct Form I Filter Structure” on page 7-47
- “Direct Form Finite Impulse Response (FIR) Filter Structure” on page 7-57
- “Direct Form FIR Transposed Filter Structure” on page 7-58
- “Lattice Autoregressive Moving Average (ARMA) Filter Structure” on page 7-64
- “dfilt.calattice” on page 8-305
- “dfilt.calatticepc” on page 8-308

- “Direct Form Symmetric FIR Filter Structure (Any Order)” on page 7-66

Starting from any quantized filter, you can convert to one of the following representation:

- Direct form I
- Direct form II
- Direct form I transposed
- Direct form II transposed
- Lattice ARMA

Additionally, FDATool lets you do the following conversions:

- Minimum phase FIR filter to Lattice MA minimum phase
- Maximum phase FIR filter to Lattice MA maximum phase
- Allpass filters to Lattice allpass

Refer to “FilterStructure” on page 7-43 for details about each of these structures.

## Converting Filters to Second-Order Sections Form

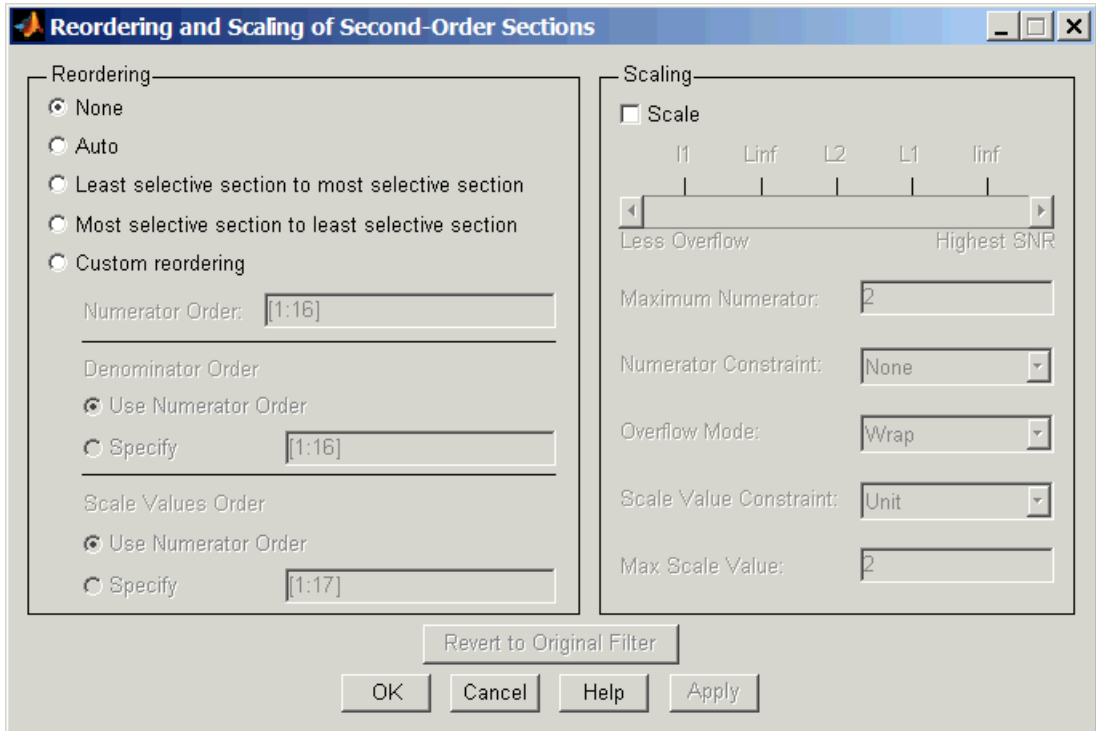
To learn about using FDATool to convert your quantized filter to use second-order sections, refer to “Converting to Second-Order Sections” in your Signal Processing Toolbox documentation. You might notice that filters you design in FDATool, rather than filters you imported, are implemented in SOS form.

### To View Filter Structures in FDATool

To open the demonstration, click **Help -> Show filter structures**. After the Help browser opens, you see the reference page for the current filter. You find the filter structure signal flow diagram on this reference page, or you can navigate to reference pages for other filter.

## Scaling Second-Order Section Filters

FDATool provides the ability to scale SOS filters after you create them. Using options on the Reordering and Scaling Second-Order Sections dialog, FDATool scales either or both the filter numerators and filter scale values according to your choices for the scaling options.



<b>Parameter</b>	<b>Description and Valid Value</b>
<b>Scale</b>	Apply any scaling options to the filter. Select this when you are reordering your SOS filter and you want to scale it at the same time. Or when you are scaling your filter, with or without reordering. Scaling is disabled by default.
<b>No Overflow—High SNR slider</b>	<p>Lets you set whether scaling favors reducing arithmetic overflow in the filter or maximizing the signal-to-noise ratio (SNR) at the filter output. Moving the slider to the right increases the emphasis on SNR at the expense of possible overflows.</p> <p>The markings indicate the P-norm applied to achieve the desired result in SNR or overflow protection. For more information about the P-norm settings, refer to norm for details.</p>
<b>Maximum Numerator</b>	Maximum allowed value for numerator coefficients after scaling.
<b>Numerator Constraint</b>	Specifies whether and how to constrain numerator coefficient values. Options are none, normalize, power of 2, and unit. Choosing none lets the scaling use any scale value for the numerators by removing any constraints on the numerators. Normalize. The power of 2 option forces scaling to use numerator values that are powers of 2, such as 2 or 0.5.

Parameter	Description and Valid Value
<b>Overflow Mode</b>	Sets the way the filter handles arithmetic overflow situations during scaling. 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).
<b>Scale Value Constraint</b>	Specify whether to constrain the filter scale values, and how to constrain them. Valid options are none, power of 2, and unit. Choosing unit for the constraint disables the <b>Max. Scale Value</b> setting and limits scale values to one. Power of 2 constrains the scale values to be powers of 2, such as 2 or 0.5, while none removes any constraint on the scale values.
<b>Max. Scale Value</b>	Sets the maximum allowed scale values. SOS filter scaling applies the <b>Max. Scale Value</b> limit only when you set <b>Scale Value Constraint</b> to a value other than unit (the default setting). Note that setting a maximum scale value removes any other limits on the scale values.
<b>Revert to Original Filter</b>	Returns your filter to the original scaling. Being able to revert to your original filter makes it easier to assess the results of scaling your filter.

Various combinations of settings let you scale filter numerators without changing the scale values, or adjust the filter scale values without changing the numerators. There is no scaling control for denominators.

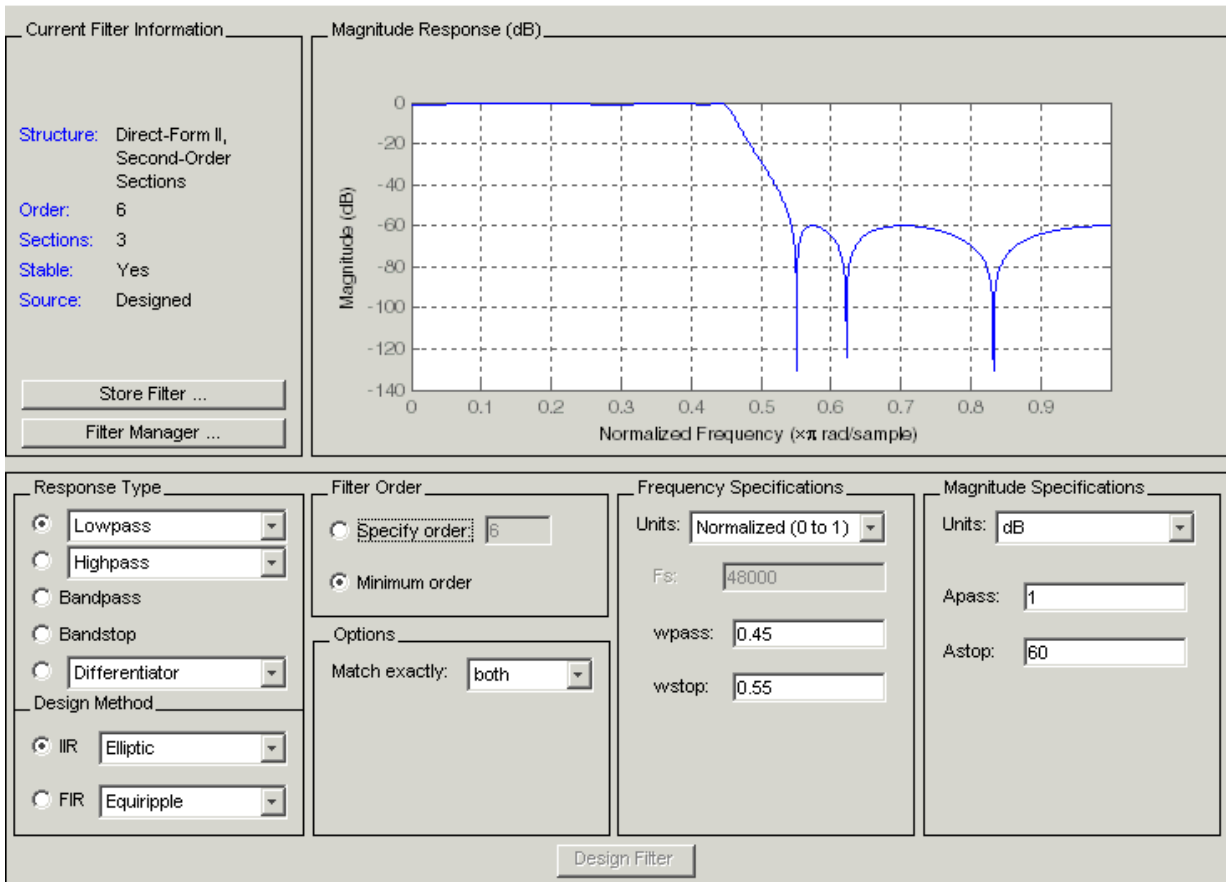


### Example—Scale An SOS Filter

Start the process by designing a lowpass elliptical filter in FDATool.

- 1 Launch FDATool.
- 2 In **Response Type**, select **Lowpass**.
- 3 In Design Method, select **IIR** and **Elliptic** from the IIR design methods list.
- 4 Select **Minimum Order** for the filter.
- 5 Switch the frequency units by choosing **Normalized(0 to 1)** from the **Units** list.
- 6 To set the passband specifications, enter 0.45 for **wpass** and 0.55 for **wstop**. Finally, in **Magnitude Specifications**, set **Astop** to 60.
- 7 Click **Design Filter** to design the filter.

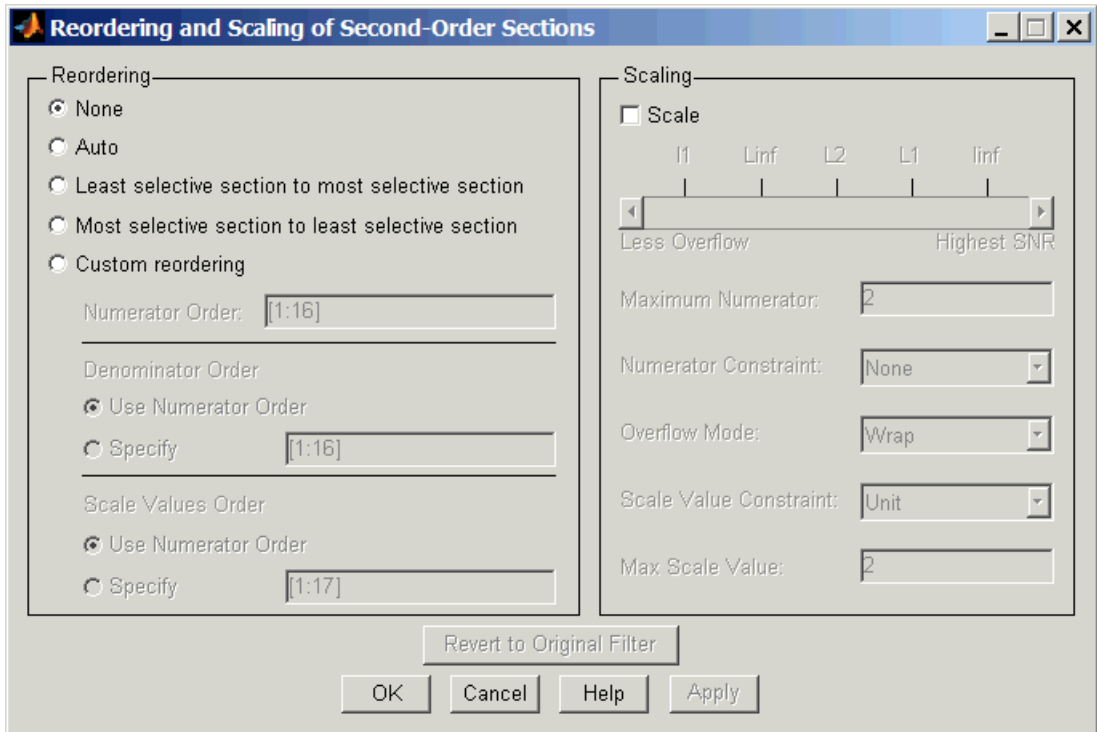
After FDATool finishes designing the filter, you see the following plot and settings in the tool.



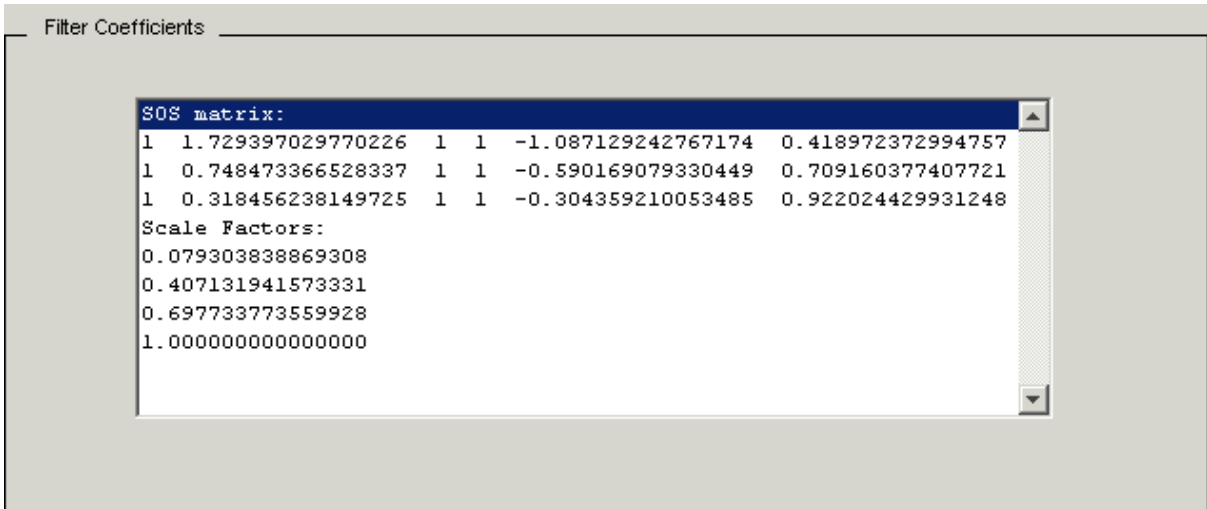
You kept the **Options** setting for **Match exactly** as both, meaning the filter design matches the specification for the passband and the stopband.

- To switch to scaling the filter, select **Edit**—>**Reorder and Scale Second-Order Sections** from the menu bar.

Your selection opens the **Reordering and Scaling Second-Order Sections** dialog shown here.



- To see the filter coefficients, return to **FDATool** and select **Filter Coefficients** from the **Analysis** menu. **FDATool** displays the coefficients and scale values in **FDATool**.



With the coefficients displayed you can see the effects of scaling your filter directly in the scale values and filter coefficients.

Now try scaling the filter in a few different ways. First scale the filter to maximize the SNR.

- 1 Return to the **Reordering and Scaling Second-Order Sections** dialog and select **None** for **Reordering** in the left pane. This prevents FDATool from reordering the filter sections when you rescale the filter.
- 2 Move the **No Overflow—High SNR** slider from **No Overflow** to **High SNR**.
- 3 Click **Apply** to scale the filter and leave the dialog open.

After a few moments, FDATool updates the coefficients displayed so you see the new scaling, as shown here.

```
Coefficients
SOS matrix:
0.426561323134070  0.853122906018389  0.426553138389891  1  -0.160114400
0.299288054987959  0.599907675766906  0.300625546185459  1  -0.184213800
0.141045994796363  0.281464374410923  0.140421171709000  1  -0.249172362
Scale Factors:
1.000000000000000
1.000000000000000
1.000000000000000
1.000000000000000
```

All of the scale factors are now 1, and the SOS matrix of coefficients shows that none of the numerator coefficients are 1 and the first denominator coefficient of each section is 1.

- 4 Click **Revert to Original Filter** to restore the filter to the original settings for scaling and coefficients.

## Reordering the Sections of Second-Order Section Filters

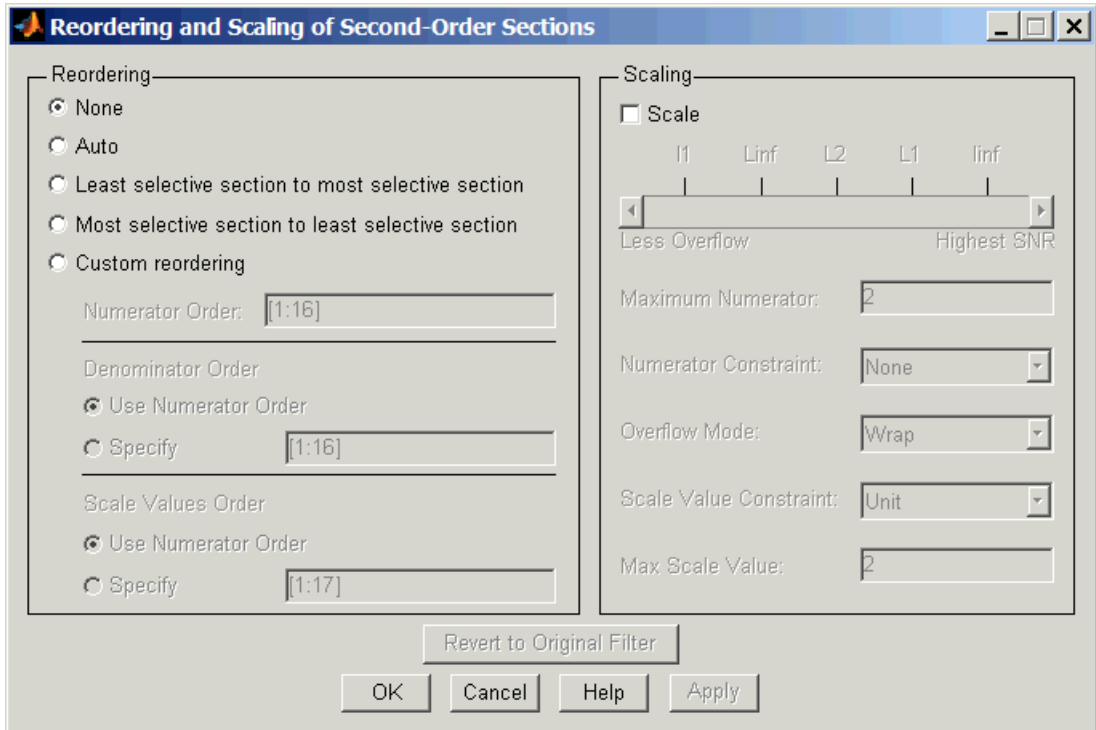
FDATool design most discrete-time filters in second-order sections. Generally, SOS filters resist the effects of quantization changes when you create fixed-point filters. After you have a second-order section filter in FDATool, either one you designed in the tool, or one you imported, FDATool provides the capability to change the order of the sections that compose the filter.

Any SOS filter in FDATool allows reordering of the sections.

### Switching FDATool to Reorder Filters

To reorder the sections of a filter, you access the Reorder and Scaling of Second-Order Sections dialog in FDATool.

With your SOS filter in FDATool, select **Edit—>Reorder and Scale Second-Order Sections** from the menu bar. FDATool returns the reordering dialog shown here with the default settings.



## Controls on the Reordering and Scaling of Second-Order Sections Dialog

In this dialog, the left-hand side contains options for reordering SOS filters. On the right you see the scaling options. These are independent—reordering your filter does not require scaling (note the **Scale** option) and scaling does not require that you reorder your filter (note the **None** option under **Reordering**). For more about scaling SOS filters, refer to “Scaling Second-Order Section Filters” on page 6-30 and to scale in the reference section.

Reordering SOS filters involves using the options in the **Reordering and Scaling of Second-Order Sections** dialog. The following table lists each reorder option and provides a description of what the option does.

<b>Control Option</b>	<b>Description</b>
<b>Auto</b>	Reorders the filter sections to minimize the output noise power of the filter. Note that different ordering applies to each specification type, such as lowpass or highpass. Automatic ordering adapts to the specification type of your filter.
<b>None</b>	Does no reordering on your filter. Selecting <b>None</b> lets you scale your filter without applying reordering at the same time. When you access this dialog with a current filter, this is the default setting—no reordering is applied.
<b>Least selective section to most selective section</b>	Rearranges the filter sections so the least restrictive (lowest Q) section is the first section and the most restrictive (highest Q) section is the last section.
<b>Most selective section to least selective section</b>	Rearranges the filter sections so the most restrictive (highest Q) section is the first section and the least restrictive (lowest Q) section is the last section.
<b>Custom reordering</b>	Lets you specify the section ordering to use by enabling the Numerator Order and Denominator Order options
<b>Numerator Order</b>	Specify new ordering for the sections of your SOS filter. Enter a vector of the indices of the sections in the order in which to rearrange them. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].



<b>Control Option</b>	<b>Description</b>
<b>Use Numerator Order</b>	Rearranges the denominators in the order assigned to the numerators.
<b>Specify</b>	Lets you specify the order of the denominators, rather than using the numerator order. Enter a vector of the indices of the sections to specify the order of the denominators to use. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].
<b>Use Numerator Order</b>	Reorders the scale values according to the order of the numerators.
<b>Specify</b>	Lets you specify the order of the scale values, rather than using the numerator order. Enter a vector of the indices of the sections to specify the order of the denominators to use. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].
<b>Revert to Original Filter</b>	Returns your filter to the original section ordering. Being able to revert to your original filter makes comparing the results of changing the order of the sections easier to assess.

### Example—Reorder an SOS Filter

With FDATool open and a second-order filter as the current filter, you use the following process to access the reordering capability and reorder your filter. Start by launching FDATool from the command prompt.

- 1 Enter `fdatool` at the command prompt to launch FDATool.
- 2 Design a lowpass Butterworth filter with order 10 and the default frequency specifications by entering the following settings:
  - Under **Response Type** select Lowpass.

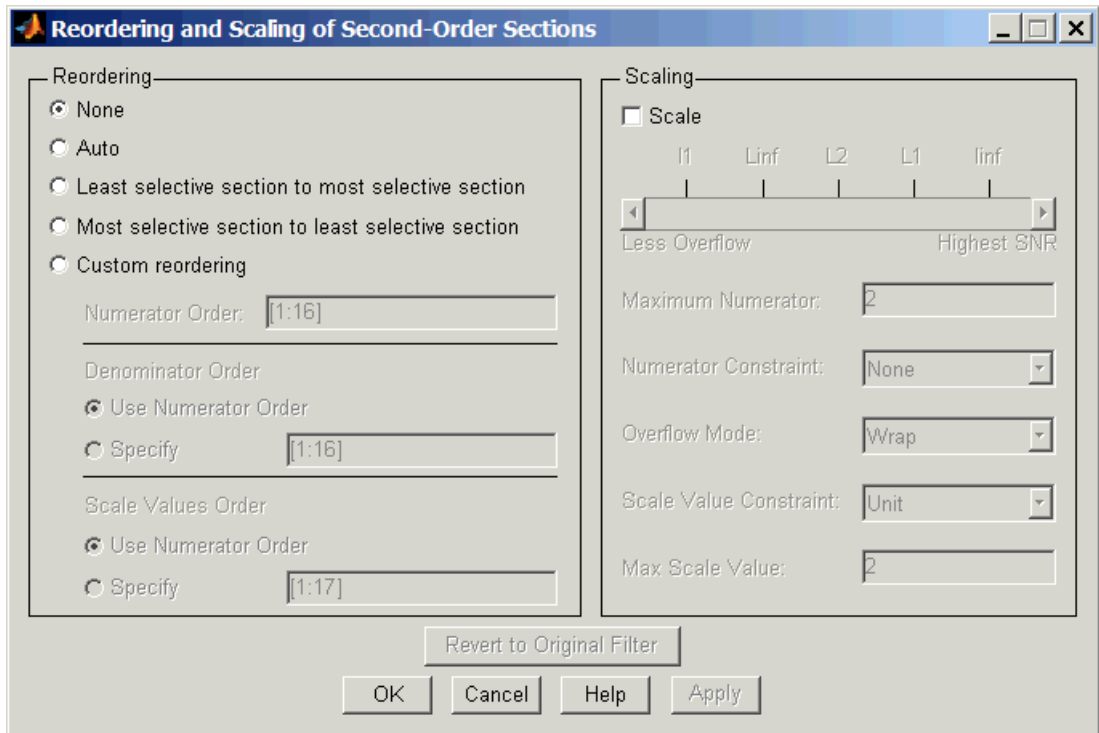
- Under **Design Method**, select **IIR** and Butterworth from the list.
- Specify the order equal to 10 in **Specify order** under **Filter Order**.
- Keep the default **F<sub>s</sub>** and **F<sub>c</sub>** values in **Frequency Specifications**.

### 3 Click **Design Filter**.

FDATool design the Butterworth filter and returns your filter as a Direct-Form II filter implemented with second-order sections. You see the specifications in the **Current Filter Information** area.

With the second-order filter in FDATool, reordering the filter uses the **Reordering and Scaling of Second-Order Sections** feature in FDATool (also available in Filter Visualization Tool, fvtool).

- ### 4 To reorder your filter, select **Edit—>Reorder and Scale Second-Order Sections** from the FDATool menus. FDATool opens the following dialog that controls reordering of the sections of your filter.



Now you are ready to reorder the sections of your filter. Note that FDATool performs the reordering on the current filter in the session.

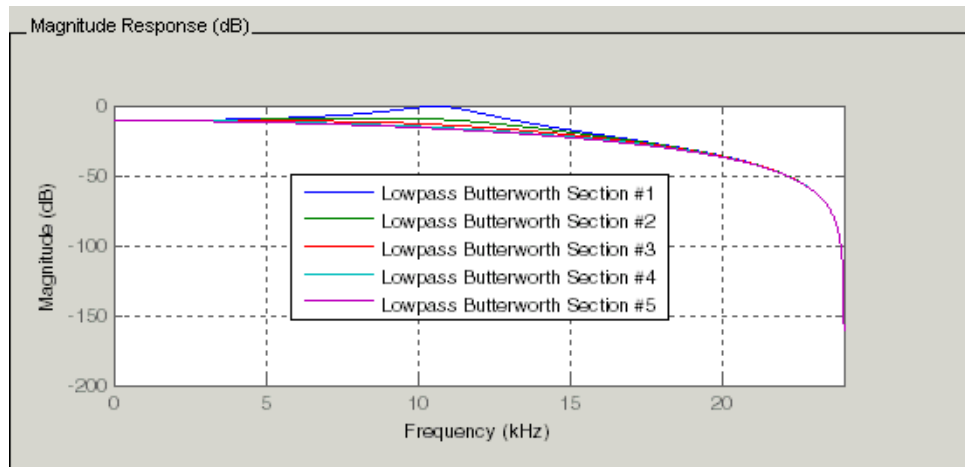
### Use Least Selective to Most Selective Section Reordering

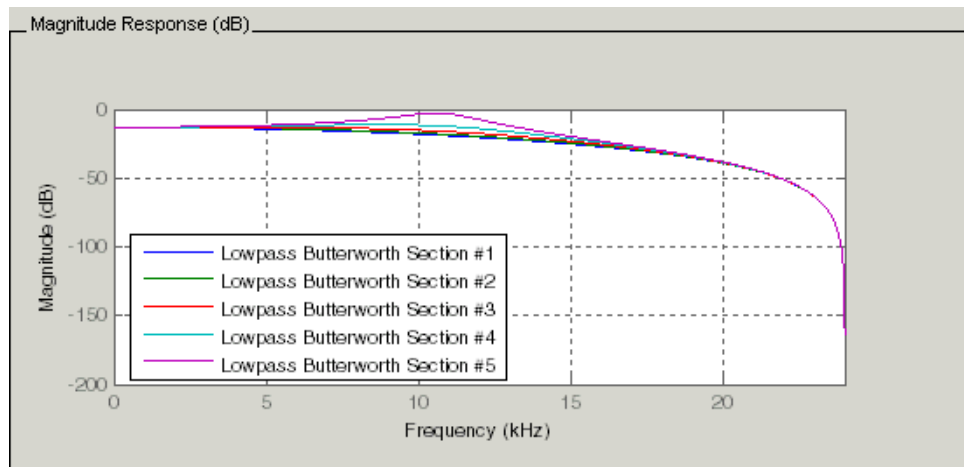
To let FDATool reorder your filter so the least selective section is first and the most selective section is last, perform the following steps in the **Reordering and Scaling of Second-Order Sections** dialog.

- 1** In **Reordering**, select **Least selective section to most selective section**.
- 2** To prevent filter scaling at the same time, clear **Scale** in **Scaling**.
- 3** In FDATool, select **View**—>**SOS View** from the menu bar so you see the sections of your filter displayed in FDATool.

- 4 In the **SOS View** dialog, select **Individual sections**. Making this choice configures FDATool to show the magnitude response curves for each section of your filter in the analysis area.
- 5 Back in the **Reordering and Scaling of Second-Order Sections** dialog, click **Apply** to reorder your filter according to the Qs of the filter sections, and keep the dialog open. In response, FDATool presents the responses for each filter section (there should be five sections) in the analysis area.

In the next two figures you can compare the ordering of the sections of your filter. In the first figure, your original filter sections appear. In the second figure, the sections have been rearranged from least selective to most selective.





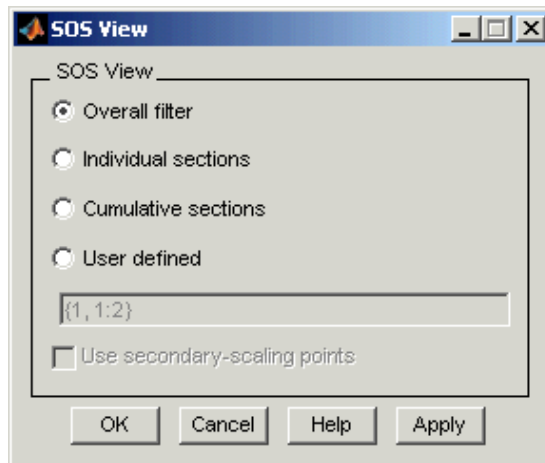
You see what reordering does, although the result is a bit subtle. Now try custom reordering the sections of your filter or using the most selective to least selective reordering option.

## Viewing SOS Filter Sections

Since you can design and reorder the sections of SOS filters, FDATool provides the ability to view the filter sections in the analysis area—SOS View. Once you have a second-order section filter as your current filter in FDATool, you turn on the SOS View option to see the filter sections individually, or cumulatively, or even only some of the sections. Enabling SOS View puts FDATool in a mode where all second-order section filters display sections until you disable the SOS View option. SOS View mode applies to any analysis you display in the analysis area. For example, if you configure FDATool to show the phase responses for filters, enabling SOS View means FDATool displays the phase response for each section of SOS filters.

### Controls on the SOS View Dialog

SOS View uses a few options to control how FDATool displays the sections, or which sections to display. When you select **View**→**SOS View** from the FDATool menu bar, you see this dialog containing options to configure SOS View operation.



By default, SOS View shows the overall response of SOS filters. Options in the SOS View dialog let you change the display. This table lists all the options and describes the effects of each.

<b>Option</b>	<b>Description</b>
<b>Overall Filter</b>	This is the familiar display in FDATool. For a second-order section filter you see only the overall response rather than the responses for the individual sections. This is the default configuration.
<b>Individual sections</b>	When you select this option, FDATool displays the response for each section as a curve. If your filter has five sections you see five response curves, one for each section, and they are independent. Compare to <b>Cumulative sections</b> .
<b>Cumulative sections</b>	<p>When you select this option, FDATool displays the response for each section as the accumulated response of all prior sections in the filter. If your filter has five sections you see five response curves:</p> <ul style="list-style-type: none"> <li>• The first curve plots the response for the first filter section.</li> <li>• The second curve plots the response for the combined first and second sections.</li> <li>• The third curve plots the response for the first, second, and third sections combined.</li> </ul> <p>And so on until all filter sections appear in the display. The final curve represents the overall filter response. Compare to <b>Cumulative sections</b> and <b>Overall Filter</b>.</p>

Option	Description
<p><b>User defined</b></p>	<p>Here you define which sections to display, and in which order. Selecting this option enables the text box where you enter a cell array of the indices of the filter sections.</p> <p>Each index represents one section. Entering one index plots one response. Entering something like {1:2} plots the combined response of sections 1 and 2. If you have a filter with four sections, the entry {1:4} plots the combined response for all four sections, whereas {1,2,3,4} plots the response for each section.</p> <p>Note that after you enter the cell array, you need to click <b>OK</b> or <b>Apply</b> to update the FDATool analysis area to the new SOS View configuration.</p>
<p><b>Use secondary-scaling points</b></p>	<p>This directs FDATool to use the secondary scaling points in the sections to determine where to split the sections. This option applies only when the filter is a <code>df2sos</code> or <code>df1tsos</code> filter. For these 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). By default, secondary -scaling points is not enabled.</p> <p>You use this with the <b>Cumulative sections</b> option only.</p>

**Example—View the Sections of SOS Filters**

After you design or import an SOS filter in to FDATool, the SOS view option lets you see the per section performance of your filter. Enabling SOS View from



the View menu in FDATool configures the tool to display the sections of SOS filters whenever the current filter is an SOS filter.

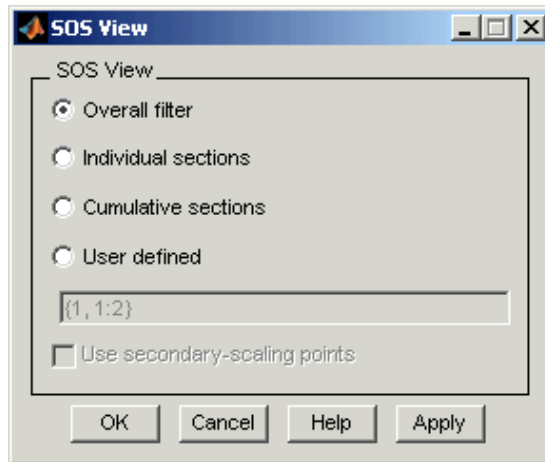
These next steps demonstrate using SOS View to see your filter sections displayed in FDATool.

- 1** Launch FDATool.
- 2** Create a lowpass SOS filter using the Butterworth design method. Specify the filter order to be 6. Using a low order filter makes seeing the sections more clear.
- 3** Design your new filter by clicking **Design Filter**.

FDATool design your filter and show you the magnitude response in the analysis area. In Current Filter Information you see the specifications for your filter. You should have a sixth-order Direct-Form II, Second-Order Sections filter with three sections.

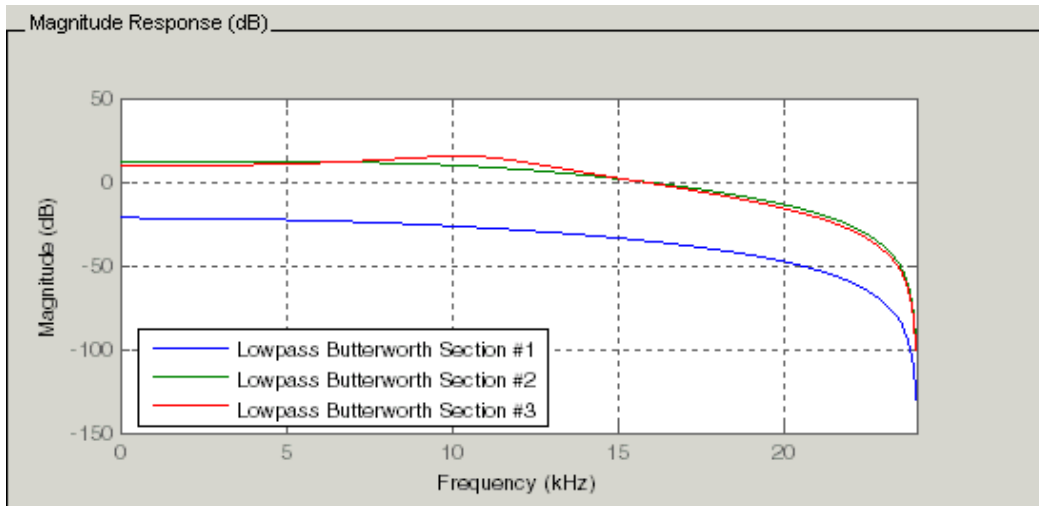
- 4** To enable SOS View, select **View—>SOS View** from the menu bar.

Now you see the **SOS View** dialog in FDATool. Options here let you specify how to display the filter sections.

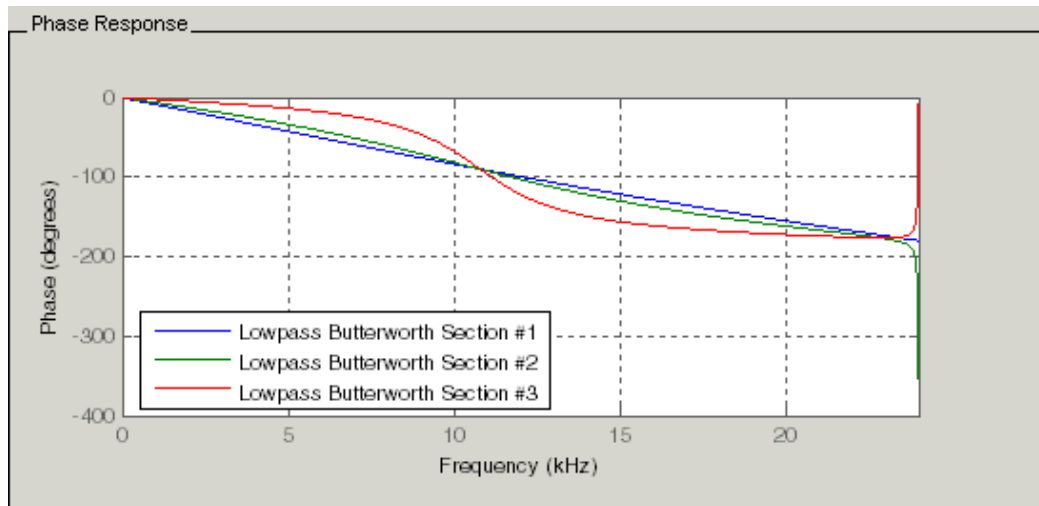


By default the analysis area in FDATool shows the overall filter response, not the individual filter section responses. This dialog lets you change the display configuration to see the sections.

- 5 To see the magnitude responses for each filter section, select **Individual sections**.
- 6 Click **Apply** to update FDATool to display the responses for each filter section. The analysis area changes to show you something like the following figure.

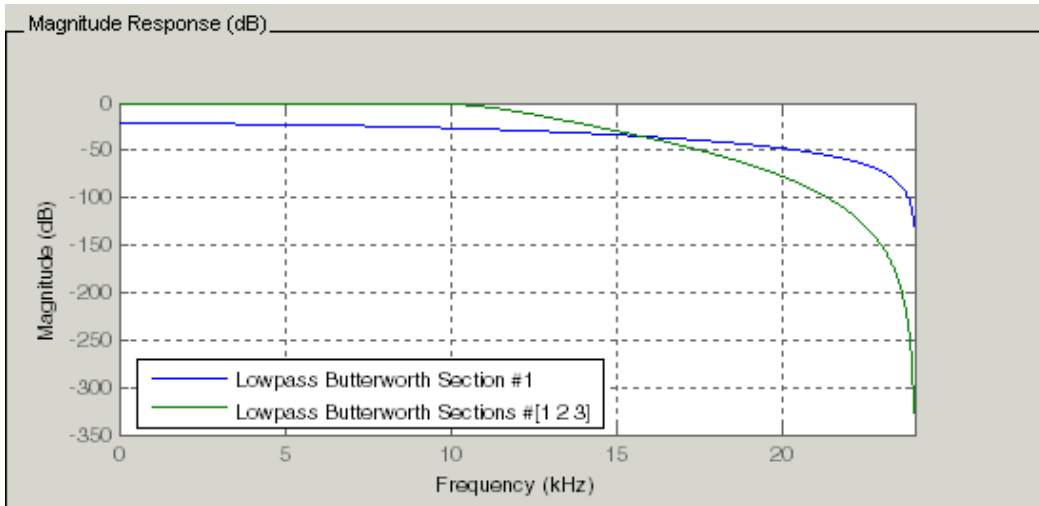


If you switch FDATool to display filter phase responses, you see the phase response for each filter section in the analysis area.



- 7 To define your own display of the sections, you use the **User defined** option and enter a vector of section indices to display. Now we display the first section response, and the cumulative first, second, and third sections response:
  - Select **User defined** to enable the text entry box in the dialog.
  - Enter the cell array `{1, 1:3}` to specify that FDATool should display the response of the first section and the cumulative response of the first three sections of the filter.
- 8 To apply your new SOS View selection, click **Apply** or **OK** (which closes the **SOS View** dialog).

In the FDATool analysis area you see two curves—one for the response of the first filter section and one for the combined response of sections 1, 2, and 3.



## Importing and Exporting Quantized Filters

When you import a quantized filter into FDATool, or export a quantized filter from FDATool to your workspace, the import and export functions use objects and you specify the filter as a variable. This contrasts with importing and exporting nonquantized filters, where you select the filter structure and enter the filter numerator and denominator for the filter transfer function.

You have the option of exporting quantized filters to your MATLAB workspace, exporting them to text files, or exporting them to MAT-files.

This section includes:

- “Example—Import Quantized Filters”
- “To Export Quantized Filters”

For general information about importing and exporting filters in FDATool, refer to “Filter Design and Analysis Tool” section in your *Signal Processing Toolbox User’s Guide*.

FDATool imports quantized filters having the following structures:

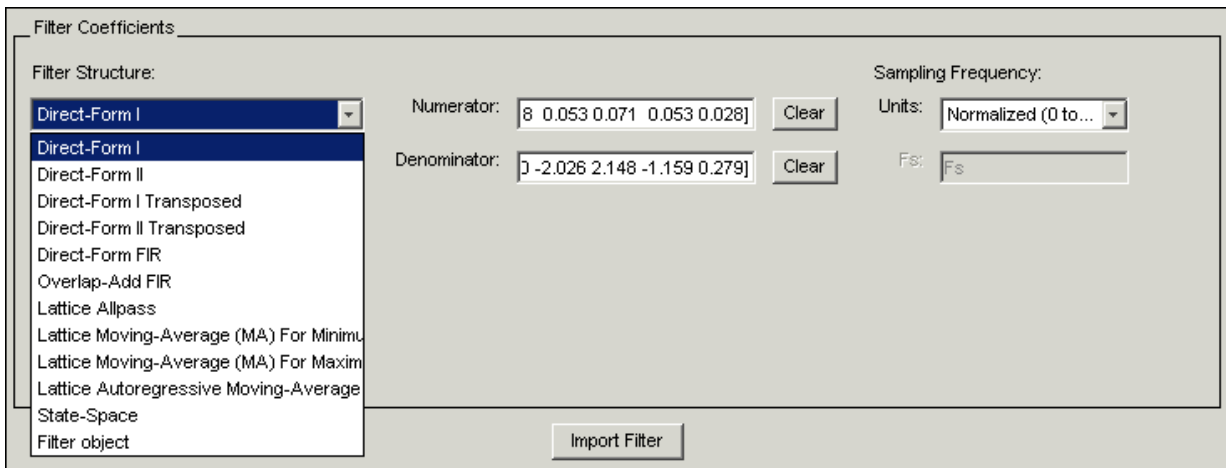
- Direct form I
- Direct form II
- Direct form I transposed
- Direct form II transposed
- Direct form symmetric FIR
- Direct form antisymmetric FIR
- Lattice allpass
- Lattice AR
- Lattice MA minimum phase
- Lattice MA maximum phase
- Lattice ARMA
- Lattice coupled-allpass
- Lattice coupled-allpass power complementary

### Example—Import Quantized Filters

After you design or open a quantized filter in your MATLAB workspace, FDATool lets you import the filter for analysis. Follow these steps to import your filter in to FDATool:

- 1 Open FDATool.
- 2 Select **Filter->Import Filter** from the menu bar.

In the lower region of FDATool, the **Design Filter** pane becomes **Import Filter**, and options appear for importing quantized filters, as shown.



- 3 From the **Filter Structure** list, select **Filter object**.

The options for importing filters change to include:

- **Discrete filter**—Enter the variable name for the discrete-time, fixed-point filter in your workspace.
- **Frequency units**—Select the frequency units from the **Units** list under **Sampling Frequency**, and specify the sampling frequency value in **F<sub>s</sub>** if needed. Your sampling frequency must correspond to the units you select. For example, when you select Normalized (0 to 1), **F<sub>s</sub>** defaults to one. But if you choose one of the frequency options, enter the sampling

frequency in your selected units. If you have the sampling frequency defined in your workspace as a variable, enter the variable name for the sampling frequency.

**4** Click **Import** to import the filter.

FDATool checks your workspace for the specified filter. It imports the filter if it finds it, displaying the magnitude response for the filter in the analysis area. If it cannot find the filter it returns an **FDATool Error** dialog.

---

**Note** If, during any FDATool session, you switch to quantization mode and create a fixed-point filter, FDATool remains in quantization mode. If you import a double-precision filter, FDATool automatically quantizes your imported filter applying the most recent quantization parameters.

When you check the current filter information for your imported filter, it will indicate that the filter is **Source:** imported (quantized) even though you did not import a quantized filter.

---

## To Export Quantized Filters

To save your filter design, FDATool lets you export the quantized filter to your MATLAB workspace (or you can save the current session in FDATool). When you choose to save the quantized filter by exporting it, you select one of these options:

- Export to your MATLAB workspace
- Export to a text file
- Export to a MAT-file

### Example—Export Coefficients or Objects to the Workspace

You can save the filter as filter coefficients variables or as a `dfilt` filter object variable. To save the filter to the MATLAB workspace:

- 1** Select **Export** from the **File** menu. The **Export** dialog appears.
- 2** Select Workspace from the **Export To** list.

- 3 Select **Coefficients** from the **Export As** list to save the filter coefficients or select **Objects** to save the filter in a filter object.
- 4 For coefficients, assign variable names using the **Numerator** and **Denominator** options under **Variable Names**. For objects, assign the variable name in the **Discrete** or **Quantized filter** option. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** box.
- 5 Click the **OK** button

If you try to export the filter to a variable name that exists in your workspace, and you did not select **Overwrite existing variables**, FDATool stops the export operation and returns a warning that the variable you specified as the quantized filter name already exists in the workspace. To continue to export the filter to the existing variable, click **OK** to dismiss the warning dialog, select the **Overwrite existing variables** check box and click **OK** or **Apply**.

### Getting Filter Coefficients after Exporting

To extract the filter coefficients from your quantized filter after you export the filter to MATLAB, use the `celldisp` function in MATLAB. For example, create a quantized filter in FDATool and export the filter as `Hq`. To extract the filter coefficients for `Hq`, use

```
celldisp(Hq.referencecoefficients)
```

which returns the cell array containing the filter reference coefficients, or

```
celldisp(Hq.quantizedcoefficients)
```

to return the quantized coefficients.

### Example—Exporting as a Text File

To save your quantized filter as a text file, follow these steps:

- 1 Select **Export** from the **File** menu.
- 2 Select **Text-file** under **Export to**.



- 3 Click **OK** to export the filter and close the dialog. Click **Apply** to export the filter without closing the **Export** dialog. Clicking **Apply** lets you export your quantized filter to more than one name without leaving the **Export** dialog.

The **Export Filter Coefficients to Text-file** dialog appears. This is the standard Microsoft Windows save file dialog.

- 4 Choose or enter a directory and filename for the text file and click **OK**.

FDATool exports your quantized filter as a text file with the name you provided, and the MATLAB editor opens, displaying the file for editing.

### **Example—Exporting as a MAT-File**

To save your quantized filter as a MAT-file, follow these steps:

- 1 Select **Export** from the **File** menu.
- 2 Select **MAT-file** under **Export to**.
- 3 Assign a variable name for the filter.
- 4 Click **OK** to export the filter and close the dialog. Click **Apply** to export the filter without closing the **Export** dialog. Clicking **Apply** lets you export your quantized filter to more than one name without leaving the **Export** dialog.

The **Export Filter Coefficients to MAT-file** dialog appears. This is the standard Microsoft Windows save file dialog.

- 5 Choose or enter a directory and filename for the text file and click **OK**.

FDATool exports your quantized filter as a MAT-file with the specified name.

## Importing XILINX Coefficient (.COE) Files

You can import XILINX coefficients (.coe) files into FDATool to create quantized filters directly using the imported filter coefficients.

### Example—Import XILINX .COE Files

To use the new import file feature:

- 1** Select **File->Import Filter From XILINX Coefficient (.COE) File** in FDATool.
- 2** In the **Import Filter From XILINX Coefficient (.COE) File** dialog, find and select the .coe file to import.
- 3** Click **Open** to dismiss the dialog and start the import process.

FDATool imports the coefficient file and creates a quantized, single-section, direct-form FIR filter.

## Transforming Filters

The toolbox provides functions for transforming filters between various forms. When you use FDATool with the Toolbox installed, a side bar button and a menu bar option enable you to use the **Transform Filter** panel to transform filters as well as using the command line functions.

From the selection on the FDATool menu bar—**Transformations**—you can transform lowpass FIR and IIR filters to a variety of passband shapes.

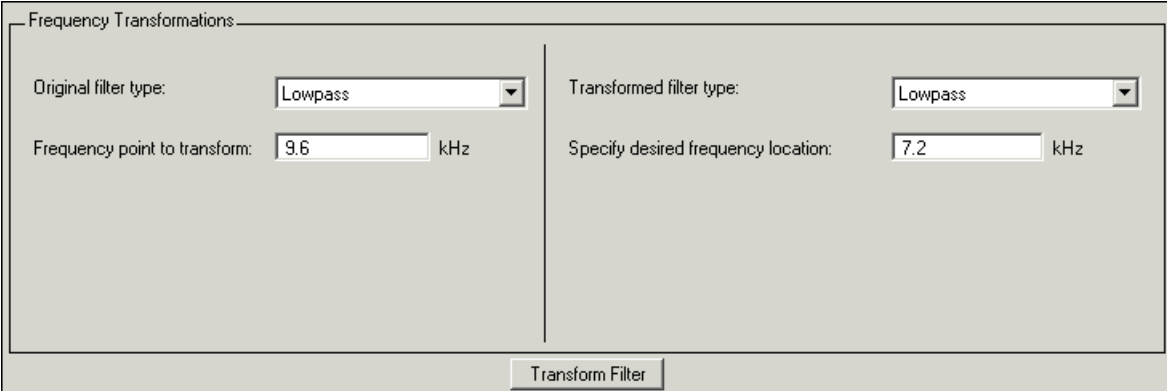
You can convert your FIR filters from:

- Lowpass to lowpass.
- Lowpass to highpass.

For IIR filters, you can convert from:

- Lowpass to lowpass.
- Lowpass to highpass.
- Lowpass to bandpass.
- Lowpass to bandstop.

When you click the **Transform Filter** button, , on the side bar, the **Transform Filter** panel opens in FDATool, as shown here.



Frequency Transformations

Original filter type: Lowpass

Transformed filter type: Lowpass

Frequency point to transform: 9.6 kHz

Specify desired frequency location: 7.2 kHz

Transform Filter

Your options for **Original filter type** refer to the type of your current filter to transform. If you select lowpass, you can transform your lowpass filter to another lowpass filter or to a highpass filter, or to numerous other filter formats, real and complex.

---

**Note** When your original filter is an FIR filter, both the FIR and IIR transformed filter type options appear on the **Transformed filter type** list. Both options remain active because you can apply the IIR transforms to an FIR filter. If your source is as IIR filter, only the IIR transformed filter options show on the list.

---

### Original Filter Type

Select the magnitude response of the filter you are transforming from the list. Your selection changes the types of filters you can transform to. For example:

- When you select **Lowpass** with an IIR filter, your transformed filter type can be
  - **Lowpass**
  - **Highpass**
  - **Bandpass**
  - **Bandstop**
  - **Multiband**
  - **Bandpass (complex)**
  - **Bandstop (complex)**
  - **Multiband (complex)**
- When you select **Lowpass** with an FIR filter, your transformed filter type can be
  - **Lowpass**
  - **Lowpass (FIR)**
  - **Highpass**
  - **Highpass (FIR) narrowband**
  - **Highpass (FIR) wideband**
  - **Bandpass**
  - **Bandstop**

- **Multiband**
- **Bandpass (complex)**
- **Bandstop (complex)**
- **Multiband (complex)**

In the following table you see each available original filter type and all the types of filter to which you can transform your original.

<b>Original Filter</b>	<b>Available Transformed Filter Types</b>
Lowpass FIR	<ul style="list-style-type: none"> <li>• <b>Lowpass</b></li> <li>• <b>Lowpass (FIR)</b></li> <li>• <b>Highpass</b></li> <li>• <b>Highpass (FIR) narrowband</b></li> <li>• <b>Highpass (FIR) wideband</b></li> <li>• <b>Bandpass</b></li> <li>• <b>Bandstop</b></li> <li>• <b>Multiband</b></li> <li>• <b>Bandpass (complex)</b></li> <li>• <b>Bandstop (complex)</b></li> <li>• <b>Multiband (complex)</b></li> </ul>
Lowpass IIR	<ul style="list-style-type: none"> <li>• <b>Lowpass</b></li> <li>• <b>Highpass</b></li> <li>• <b>Bandpass</b></li> <li>• <b>Bandstop</b></li> <li>• <b>Multiband</b></li> <li>• <b>Bandpass (complex)</b></li> <li>• <b>Bandstop (complex)</b></li> <li>• <b>Multiband (complex)</b></li> </ul>

<b>Original Filter</b>	<b>Available Transformed Filter Types</b>
Highpass FIR	<ul style="list-style-type: none"> <li>• <b>Lowpass</b></li> <li>• <b>Lowpass (FIR) narrowband</b></li> <li>• <b>Lowpass (FIR) wideband</b></li> <li>• <b>Highpass (FIR)</b></li> <li>• <b>Highpass</b></li> <li>• <b>Bandpass</b></li> <li>• <b>Bandstop</b></li> <li>• <b>Multiband</b></li> <li>• <b>Bandpass (complex)</b></li> <li>• <b>Bandstop (complex)</b></li> <li>• <b>Multiband (complex)</b></li> </ul>
Highpass IIR	<ul style="list-style-type: none"> <li>• <b>Lowpass</b></li> <li>• <b>Highpass</b></li> <li>• <b>Bandpass</b></li> <li>• <b>Bandstop</b></li> <li>• <b>Multiband</b></li> <li>• <b>Bandpass (complex)</b></li> <li>• <b>Bandstop (complex)</b></li> <li>• <b>Multiband (complex)</b></li> </ul>
Bandpass FIR	<ul style="list-style-type: none"> <li>• <b>Bandpass</b></li> <li>• <b>Bandpass (FIR)</b></li> </ul>
Bandpass IIR	<b>Bandpass</b>
Bandstop FIR	<ul style="list-style-type: none"> <li>• <b>Bandstop</b></li> <li>• <b>Bandstop (FIR)</b></li> </ul>
Bandstop IIR	<b>Bandstop</b>

Note also that the transform options change depending on whether your original filter is FIR or IIR. Starting from an IIR filter, you can transform to IIR or FIR forms. With an IIR original filter, you are limited to IIR target filters.

After selecting your response type, use **Frequency point to transform** to specify the magnitude response point in your original filter to transfer to your target filter. Your target filter inherits the performance features of your original filter, such as passband ripple, while changing to the new response form.

For more information about transforming filters, refer to “Frequency Transformations for Real Filters” on page 5-11 and “Frequency Transformations for Complex Filters” on page 5-26.

## Frequency Point to Transform

The frequency point you enter in this field identifies a magnitude response value (in dB) on the magnitude response curve.

When you enter frequency values in the **Specify desired frequency location** option, the frequency transformation tries to set the magnitude response of the transformed filter to the value identified by the frequency point you enter in this field.

While you can enter any location, generally you should specify a filter passband or stopband edge, or a value in the passband or stopband.

The **Frequency point to transform** sets the magnitude response at the values you enter in **Specify desired frequency location**. Specify a value that lies at either the edge of the stopband or the edge of the passband.

If, for example, you are creating a bandpass filter from a highpass filter, the transformation algorithm sets the magnitude response of the transformed filter at the **Specify desired frequency location** to be the same as the response at the **Frequency point to transform** value. Thus you get a bandpass filter whose response at the low and high frequency locations is the same. Notice that the passband between them is undefined. In the next two figures you see the original highpass filter and the transformed bandpass filter.

For more information about transforming filters, refer to “Digital Frequency Transformations” on page 5-1.

## Transformed Filter Type

Select the magnitude response for the target filter from the list. The complete list of transformed filter types is:

- **Lowpass**
- **Lowpass (FIR)**
- **Highpass**
- **Highpass (FIR) narrowband**
- **Highpass (FIR) wideband**
- **Bandpass**
- **Bandstop**
- **Multiband**
- **Bandpass (complex)**
- **Bandstop (complex)**
- **Multiband (complex)**

Not all types of transformed filters are available for all filter types on the **Original filter types** list. You can transform bandpass filters only to bandpass filters. Or bandstop filters to bandstop filters. Or IIR filters to IIR filters.

For more information about transforming filters, refer to “Frequency Transformations for Real Filters” on page 5-11 and “Frequency Transformations for Complex Filters” on page 5-26.

## Specify Desired Frequency Location

The frequency point you enter in **Frequency point to transform** matched a magnitude response value. At each frequency you enter here, the transformation tries to make the magnitude response the same as the response identified by your **Frequency point to transform** value.


While you can enter any location, generally you should specify a filter passband or stopband edge, or a value in the passband or stopband.

For more information about transforming filters, refer to “Digital Frequency Transformations” on page 5-1.

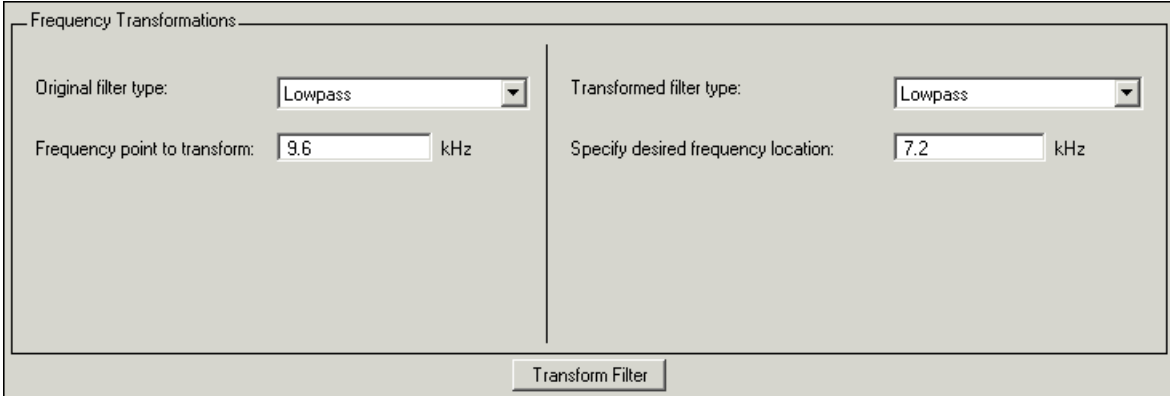
## Example—Transform Filters

To transform the magnitude response of your filter, use the **Transform Filter** option on the side bar.



- 1 Design or import your filter into FDATool.
- 2 Click **Transform Filter**, , on the side bar.

FDATool opens the **Transform Filter** panel in FDATool.



- 3 From the **Original filter type** list, select the response form of the filter you are transforming.

When you select the type, whether is **lowpass**, **highpass**, **bandpass**, or **bandstop**, FDATool recognizes whether your filter form is FIR or IIR. Using both your filter type selection and the filter form, FDATool adjusts the entries on the **Transformed filter type** list to show only those that apply to your original filter.

- 4 Enter the frequency point to transform value in **Frequency point to transform**. Notice that the value you enter must be in KHz; for example, enter 0.1 for 100 Hz or 1.5 for 1500 Hz.
- 5 From the **Transformed filter type** list, select the type of filter you want to transform to.

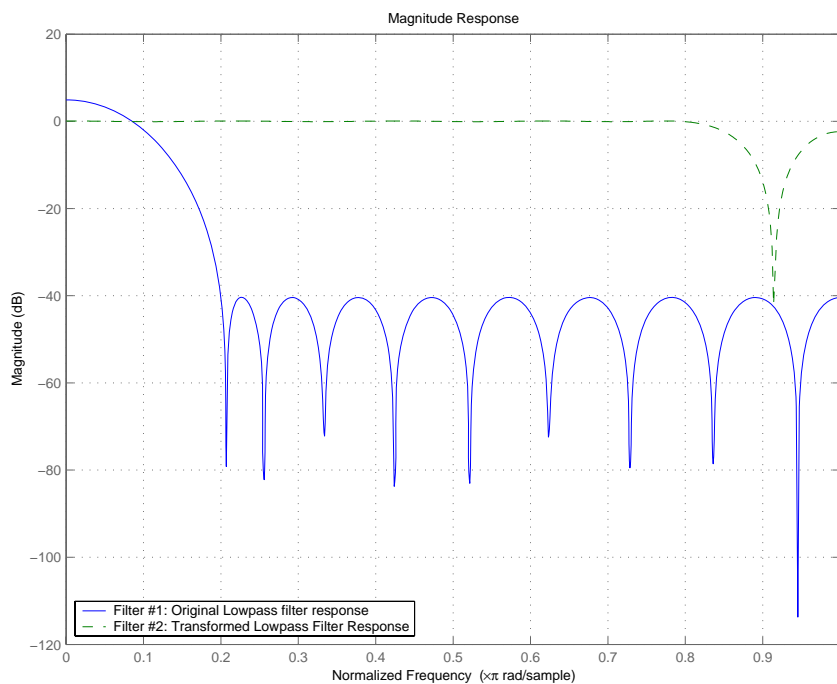
Your filter type selection changes the options here.

- When you pick a lowpass or highpass filter type, you enter one value in **Specify desired frequency location**.

- When you pick a bandpass or bandstop filter type, you enter two values—one in **Specify desired low frequency location** and one in **Specify desired high frequency location**. Your values define the edges of the passband or stopband.
- When you pick a multiband filter type, you enter values as elements in a vector in **Specify a vector or desired frequency locations**— one element for each desired location. Your values define the edges of the passbands and stopbands.

After you click **Transform Filter**, FDATool transforms your filter, displays the magnitude response of your new filter, and updates the **Current Filter Information** to show you that your filter has been transformed. In the filter information, the **Source** is **Transformed**.

For example, the figure shown here includes the magnitude response curves for two filter. The original filter is a lowpass filter with rolloff between 0.2 and 0.25. The transformed filter is a lowpass filter with rolloff region between 0.8 and 0.85.

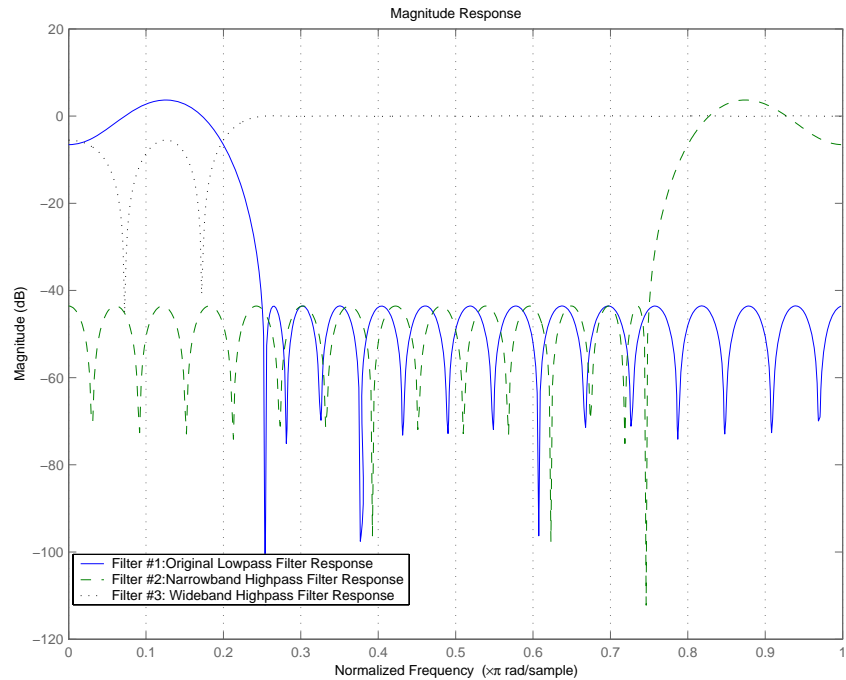


- To transform your lowpass filter to a highpass filter, select **Lowpass to Highpass**.

When you select **Lowpass to Highpass**, FDATool returns the dialog shown here. More information about the **Select Transform...** dialog follows the figure.



To demonstrate the effects of selecting **Narrowband Highpass** or **Wideband Highpass**, the next figure presents the magnitude response curves for a source lowpass filter after it is transformed to both narrow- and wideband highpass filters. For comparison, the response of the original filter appears as well.




For the narrowband case, the transformation algorithm essentially reverses the magnitude response, like reflecting the curve around the  $y$ -axis, then translating the curve to the right until the origin lies at 1 on the  $x$ -axis. After reflecting and translating, the passband at high frequencies is the reverse of the passband of the original filter at low frequencies with the same rolloff and ripple characteristics.

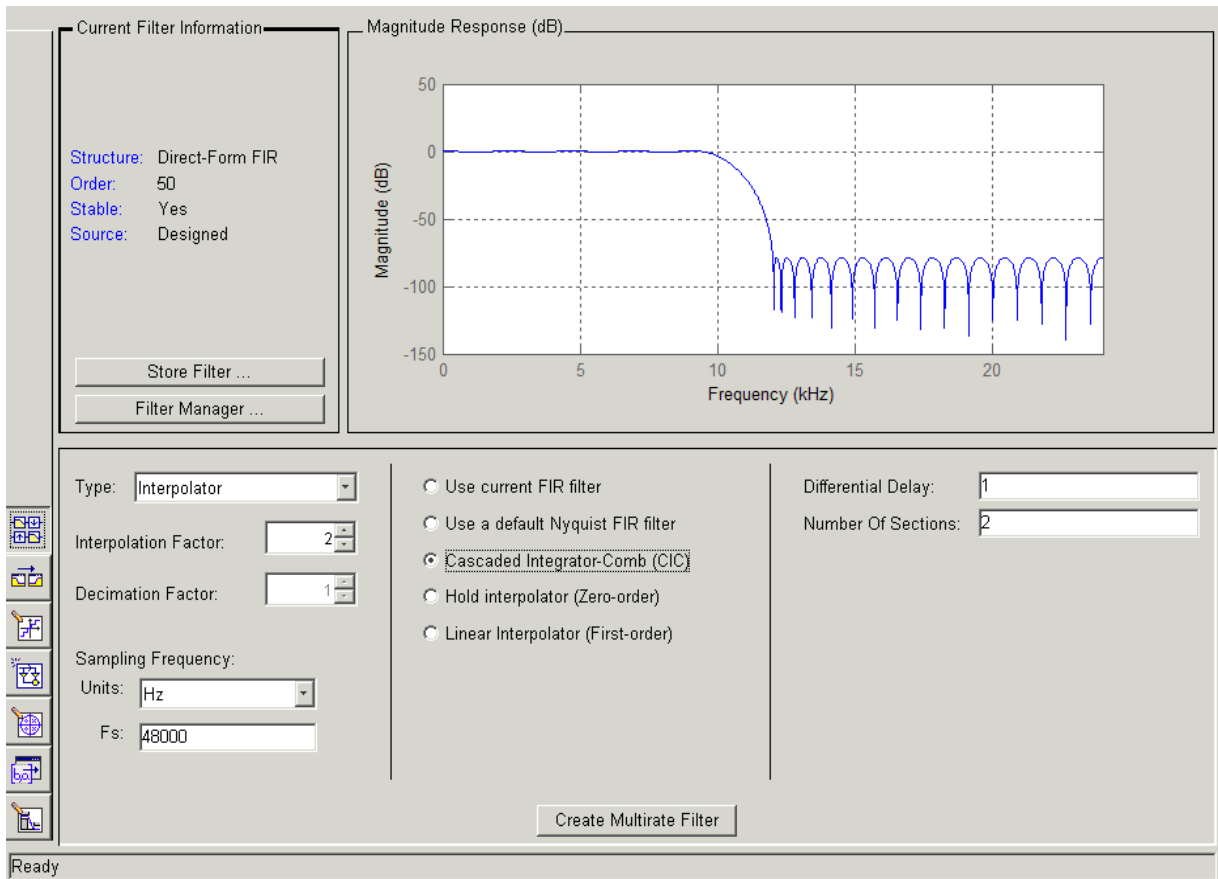
# Designing Multirate Filters in FDATool

Not only can you design multirate filters from the MATLAB command prompt, FDATool provides the same design capability in a graphical user interface tool. By starting FDATool and switching to the multirate filter design mode you have access to all of the multirate design capabilities in the toolbox—decimators, interpolators, and fractional rate changing filters, among others.

## Switching FDATool to Multirate Filter Design Mode

The multirate filter design mode in FDATool lets you specify and design a wide range of multirate filters, including decimators and interpolators.

With FDATool open, click **Create a Multirate Filter**, , on the side bar. You see FDATool switch to the design mode showing the multirate filter design options. Shown in the figure below is the default multirate design configuration that designs an interpolating filter with an interpolation factor of 2. The design uses the current FIR filter in FDATool.



When the current filter in FDATool is not an FIR filter, the multirate filter design panel removes the **Use current FIR filter** option and selects the **Use default Nyquist FIR filter** option instead as the default setting.

## Controls on the Multirate Design Panel

You see the options that allow you to design a variety of multirate filters. The Type option is your starting point. From this list you select the multirate filter to design. Based on your selection, other options change to provide the controls you need to specify your filter.

Notice the separate sections of the design panel. On the left is the filter type area where you choose the type of multirate filter to design and set the filter performance specifications.

In the center section FDATool provides choices that let you pick the filter design method to use.

The rightmost section offers options that control filter configuration when you select **Cascaded-Integrator Comb (CIC)** as the design method in the center section. Both the Decimator type and Interpolator type filters let you use the **Cascaded-Integrator Comb (CIC)** option to design multirate filters.



Here are all the options available when you switch to multirate filter design mode. Each option listed includes a brief description of what the option does when you use it.

<b>Option for Selecting and Configuring Your Filter</b>	<b>Description</b>
<b>Type</b>	<p>Specifies the type of multirate filter to design. Choose from Decimator, Interpolator, or Fractional-rate convertor.</p> <ul style="list-style-type: none"> <li>• When you choose Decimator, set <b>Decimation Factor</b> to specify the decimation to apply.</li> <li>• When you choose Interpolator, set <b>Interpolation Factor</b> to specify the interpolation amount applied.</li> <li>• When you choose Fractional-rate convertor, set both <b>Interpolation Factor</b> and <b>Decimation Factor</b>. FDATool uses both to determine the fractional rate change by dividing <b>Interpolation Factor</b> by <b>Decimation Factor</b> to determine the fractional rate change in the signal.</li> </ul> <p>You should select values for interpolation and decimation that are relatively prime. When your interpolation factor and decimation factor are not relatively prime, FDATool reduces the interpolation/decimation fractional rate to the lowest common denominator and issues a message in the status bar in FDATool.</p> <p>For example, if the interpolation factor is 6 and the decimation factor is 3, FDATool reduces <math>6/3</math> to <math>2/1</math> when you design the rate changer. But if the interpolation factor is 8 and the decimation factor is 3, FDATool designs the filter without change.</p>
<b>Interpolation Factor</b>	Use the up-down control arrows to specify the amount of interpolation to apply to the signal. Factors range upwards from 2.
<b>Decimation Factor</b>	Use the up-down control arrows to specify the amount of decimation to apply to the signal. Factors range upwards from 2.
<b>Sampling Frequency</b>	No settings here. Just <b>Units</b> and <b>Fs</b> below.

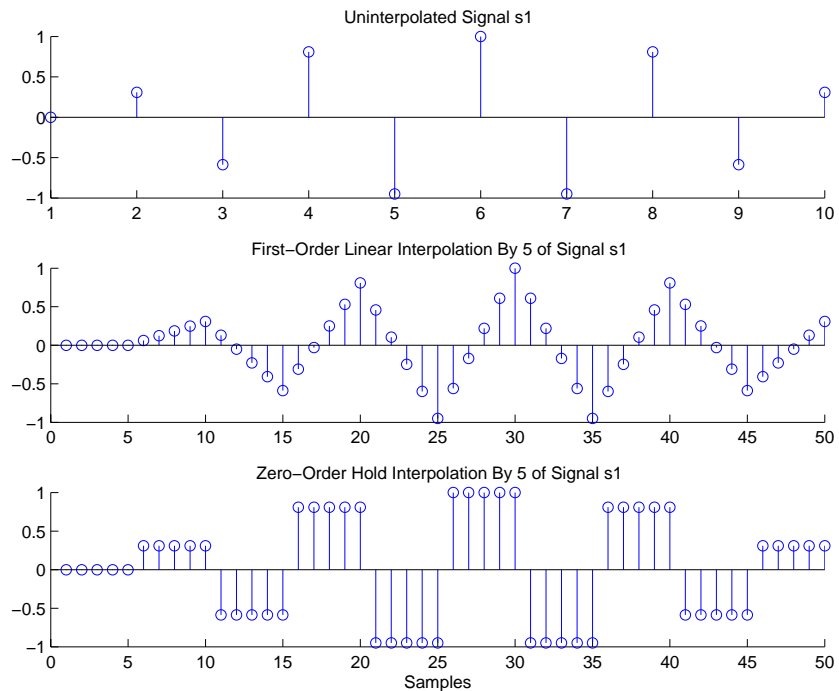
<b>Option for Selecting and Configuring Your Filter</b>	<b>Description</b>
<b>Units</b>	Specify whether $F_s$ is specified in Hz, kHz, MHz, GHz, or Normalized (0 to 1) units.
<b><math>F_s</math></b>	Set the full scale sampling frequency in the frequency units you specified in <b>Units</b> . When you select Normalized for <b>Units</b> , you do not enter a value for <b><math>F_s</math></b> .

<b>Options for Designing Your Filter</b>	<b>Description</b>
<b>Use current FIR filter</b>	Directs FDATool to use the current FIR filter to design the multirate filter. If the current filter is an IIR form, you cannot select this option. You cannot design multirate filters with IIR structures.
<b>Use a default Nyquist Filter</b>	Tells FDATool to use the default Nyquist design method when the current filter in FDATool is not an FIR filter.
<b>Cascaded Integrator-Comb (CIC)</b>	Design CIC filters using the options provided in the right-hand area of the multirate design panel.
<b>Hold Interpolator (Zero-order)</b>	When you design an interpolator, you can specify how the filter sets interpolated values between signal values. When you select this option, the interpolator applies the most recent signal value for each interpolated value until it processes the next signal value. This is similar to sample-and-hold techniques. Compare to the <b>Linear Interpolator</b> option.
<b>Linear Interpolator (First-order)</b>	When you design an interpolator, you can specify how the filter sets interpolated values between signal values. When you select this option, the interpolator applies linear interpolation between signal value to set the interpolated value until it processes the next signal value. Compare to the <b>Linear Interpolator</b> option.

To see the difference between hold interpolation and linear interpolation, the following figure presents a sine wave signal  $s_1$  in three forms:

- The top subplot in the figure presents  $s1$  without interpolation.
- The middle subplot shows signal  $s1$  interpolated by a linear interpolator with an interpolation factor of 5.
- The bottom subplot shows signal  $s1$  interpolated by a hold interpolator with an interpolation factor of 5.

You see in the bottom figure the sample and hold nature of hold interpolation, and the first-order linear interpolation applied by the linear interpolator.



We used FDATool to create interpolators similar to the following code for the figure:


- Linear interpolator—`hm=mfilt.linearinterp(5)`

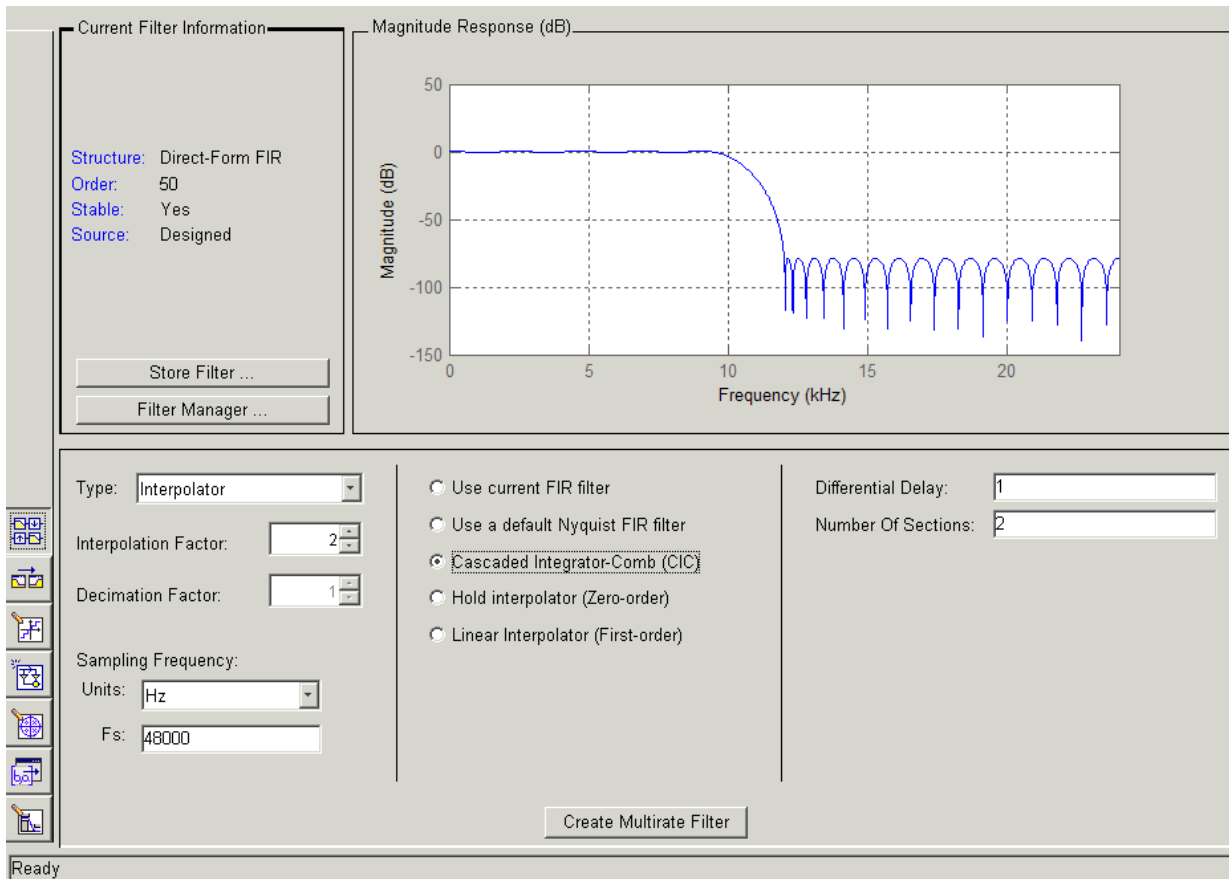
- Hold interpolator—`hm=mfilt.holdinterp(5)`

<b>Options for Designing CIC Filters</b>	<b>Description</b>
<b>Differential Delay</b>	Sets the differential delay for the CIC filter. Usually a value of one or two is appropriate.
<b>Number of Sections</b>	Specifies the number of sections in a CIC decimator. The default number of sections is 2 and the range is any positive integer.

### Example—Design a Fractional Rate Convertor

To introduce the process you use to design a multirate filter in FDATool, this example uses the options to design a fractional rate convertor which uses  $7/3$  as the fractional rate. Begin the design by creating a default lowpass FIR filter in FDATool. You do not have to begin with this FIR filter, but the default filter works fine.

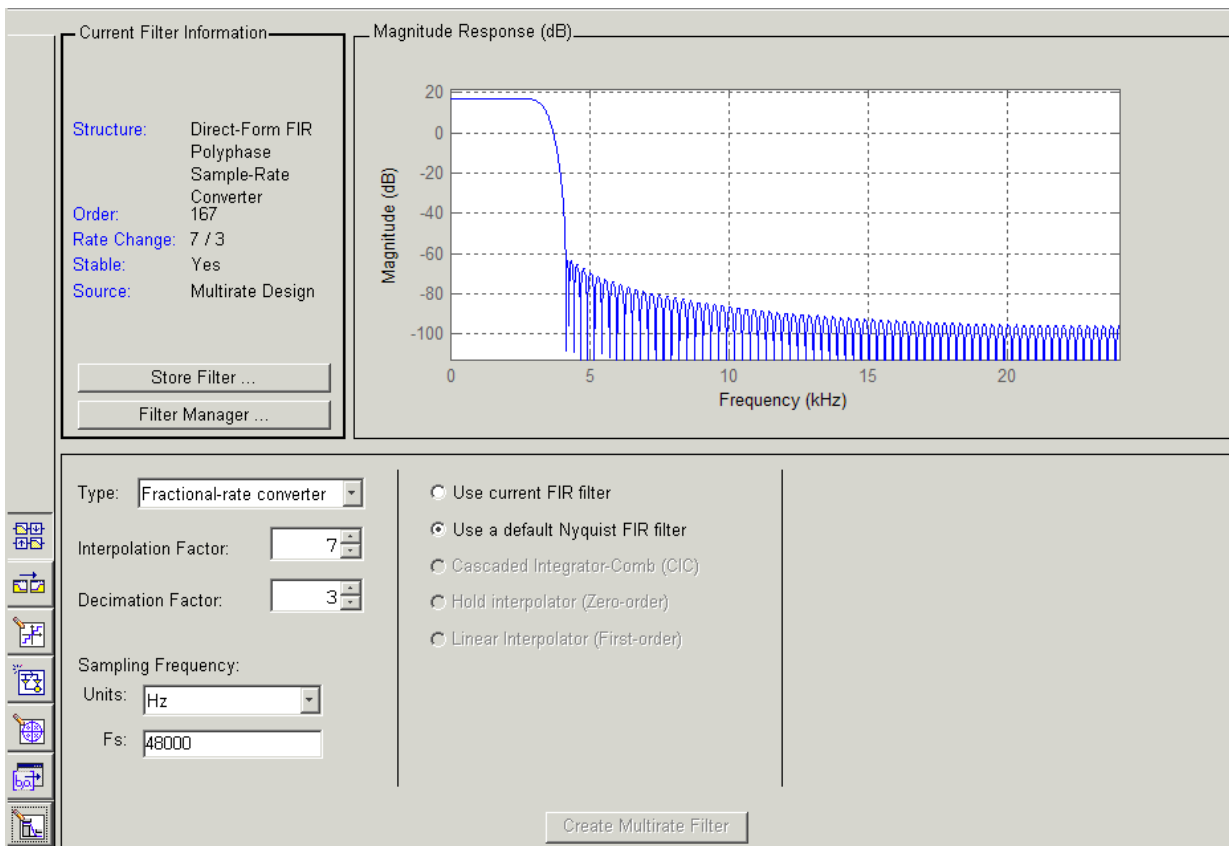
- 1 Launch FDATool.
- 2 Select the settings for a minimum-order lowpass FIR filter, using the Equiripple design method.
- 3 When FDATool displays the magnitude response for the filter, click  in the side bar. FDATool switches to multirate filter design mode, showing the multirate design panel, shown here.



- 4 To design a fractional rate filter, select Fractional-rate convertor from the **Type** list. The **Interpolation Factor** and **Decimation Factor** options become available.
- 5 In **Interpolation Factor**, use the up arrow to set the interpolation factor to 7.
- 6 Using the up arrow in **Decimation Factor**, set 3 as the decimation factor.

- 7 Select Use a default Nyquist FIR filter. You could design the rate convertor with the current FIR filter as well.
- 8 Enter 24000 to set  $F_s$ .
- 9 Click **Create Multirate Filter**.


After designing the filter, FDATool returns with the specifications for your new filter displayed in **Current Filter Information**, and shows the magnitude response of the filter.

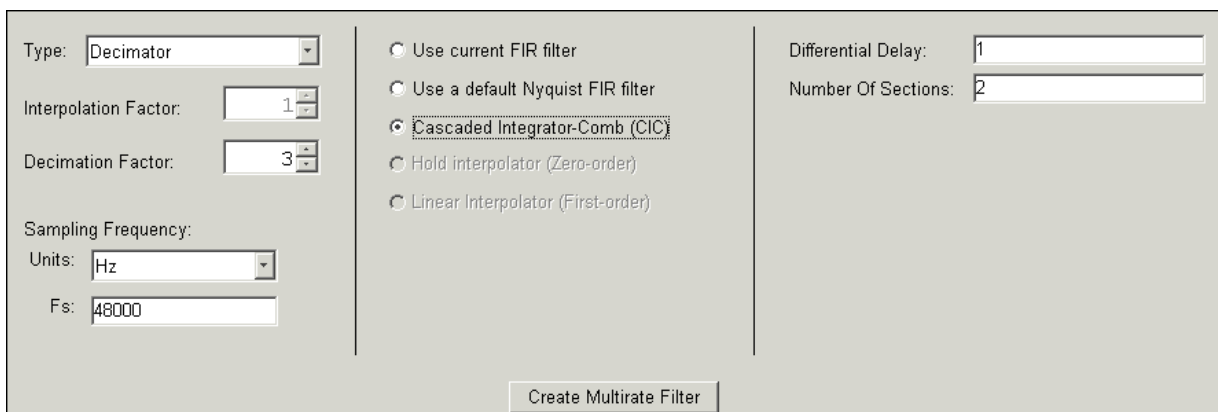


You can test the filter by exporting it to your workspace and using it to filter a signal. For information about exporting filters, refer to “Importing and Exporting Quantized Filters” on page 6-53.

### Example—Design a CIC Decimator for 8 Bit Input/Output Data

Another kind of filter you can design in FDATool is Cascaded-Integrator Comb (CIC) filters. FDATool provides the options needed to configure your CIC to meet your needs.

- 1 Launch FDATool and design the default FIR lowpass filter. Designing a filter at this time is an optional step.
- 2 Switch FDATool to multirate design mode by clicking  on the side bar.
- 3 For **Type**, select Decimator, and set **Decimation Factor** to 3.
- 4 To design the decimator using a CIC implementation, select **Cascaded-Integrator Comb (CIC)**. This enables the CIC-related options on the right of the panel.
- 5 Set Differential Delay to 2. Generally, 1 or 2 are good values to use.
- 6 Enter 2 for the **Number of Sections**. Settings in the multirate design panel should look like this.

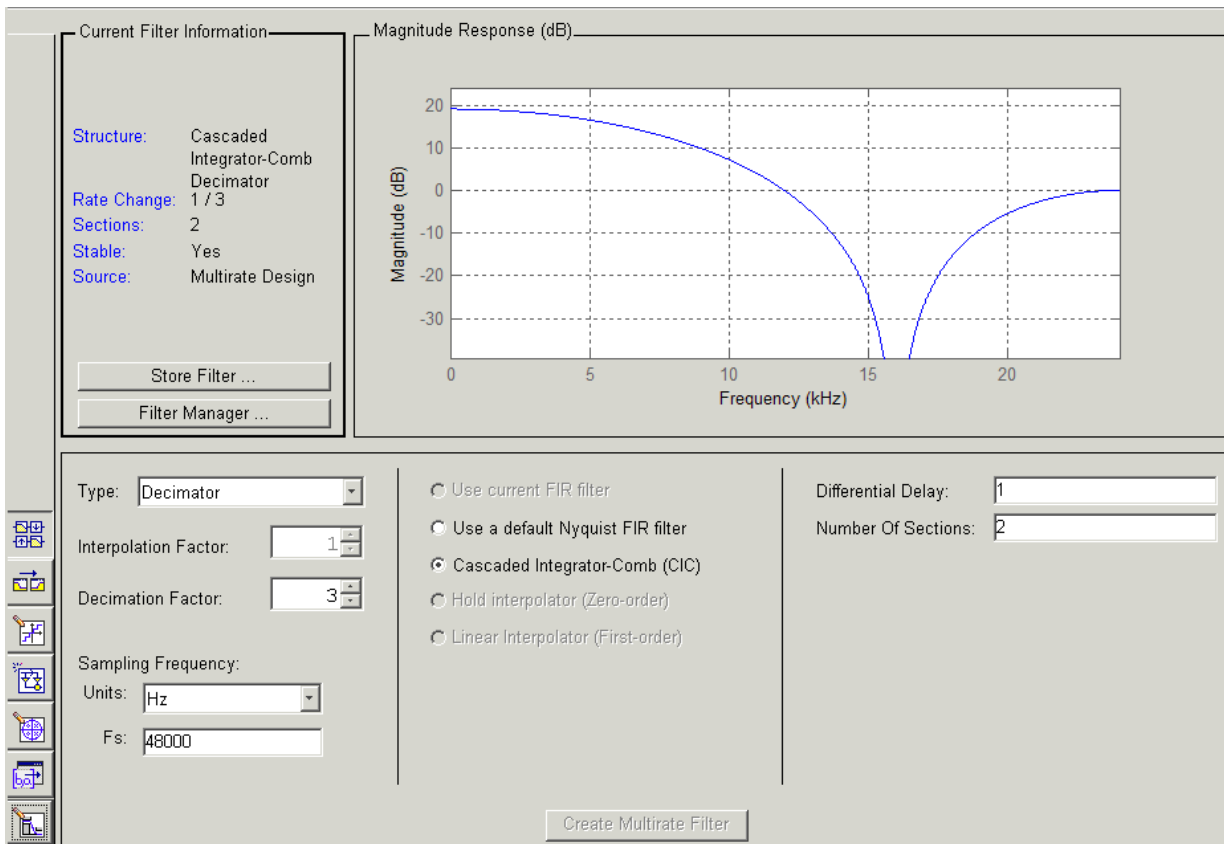


The screenshot shows the FDATool multirate design panel with the following settings:

- Type:** Decimator (selected in the dropdown)
- Interpolation Factor:** 1
- Decimation Factor:** 3
- Sampling Frequency:** Units: Hz, Fs: 48000
- Filter Type Options:**
  - Use current FIR filter
  - Use a default Nyquist FIR filter
  - Cascaded Integrator-Comb (CIC)
  - Hold interpolator (Zero-order)
  - Linear Interpolator (First-order)
- Differential Delay:** 1
- Number Of Sections:** 2
- Create Multirate Filter** button

**7 Click Create Multirate Filter.**

FDATool designs the filter, shows the magnitude response in the analysis area, and updates the current filter information to show that you designed a tenth-order cascaded-integrator comb decimator with two sections. Notice the source is Multirate Design, indicating you used the multirate design mode in FDATool to make the filter. FDATool should look like this now.



Designing other multirate filters follows the same pattern.

To design other multirate filters, do one of the following depending on the filter to design:



- To design an interpolator, select one of these options.
  - **Use a default Nyquist FIR filter**
  - **Cascaded-Integrator Comb (CIC)**
  - **Hold Interpolator (Zero-order)**
  - **Linear Interpolator (First-order)**
- To design a decimator, select from these options.
  - **Use a default Nyquist FIR filter**
  - **Cascaded-Integrator Comb (CIC)**
- To design a fractional-rate convertor, select **Use a default Nyquist FIR filter**.

## Quantizing Multirate Filters

After you design a multirate filter in FDATool, the quantization features enable you to convert your floating-point multirate filter to fixed-point arithmetic.

---

**Note** CIC filters are always fixed-point.

---

With your multirate filter as the current filter in FDATool, you can quantize your filter and use the quantization options to specify the fixed-point arithmetic the filter uses.

### To Quantize and Configure Multirate Filters

Follow these steps to convert your multirate filter to fixed-point arithmetic and set the fixed-point options.

- 1** Design or import your multirate filter and make sure it is the current filter in FDATool.
- 2** Click the **Set Quantization Parameters** button on the side bar.
- 3** From the **Filter Arithmetic** list on the Filter Arithmetic pane, select **Fixed-point**. If your filter is a CIC filter, the **Fixed-point** option is enabled by default and you do not set this option.

4 In the quantization panes, set the options for your filter. Set options for **Coefficients**, **Input/Output**, and **Filter Internals**.

5 Click **Apply**.

When your current filter is a CIC filter, the options on the **Input/Output** and **Filter Internals** panes change to provide specific features for CIC filters.

### Input/Output

The options that specify how your CIC filter uses input and output values are listed in the table below.

Option Name	Description
<b>Input Word Length</b>	Sets the word length used to represent the input to a filter.
<b>Input fraction length</b>	Sets the fraction length used to interpret input values to filter.
<b>Input range (+/-)</b>	Lets you set the range the inputs represent. You use this instead of the <b>Input fraction length</b> option to set the precision. When you enter a value $x$ , the resulting range is $-x$ to $x$ . Range must be a positive integer.
<b>Output word length</b>	Sets the word length used to represent the output from a filter.
<b>Avoid overflow</b>	Directs the filter to set the fraction length for the input to prevent the output values from exceeding the available range as defined by the word length. Clearing this option lets you set <b>Output fraction length</b> .

Option Name	Description
<b>Output fraction length</b>	Sets the fraction length used to represent output values from a filter.
<b>Output range (+/-)</b>	Lets you set the range the outputs represent. You use this instead of the <b>Output fraction length</b> option to set the precision. When you enter a value $x$ , the resulting range is $-x$ to $x$ . Range must be a positive integer.

The available options change when you change the **Filter precision** setting. Moving from **Full** to **Specify all** adds increasing control by enabling more input and output word options.

### Filter Internals

With a CIC filter as your current filter, the **Filter precision** option on the **Filter Internals** pane includes modes for controlling the filter word and fraction lengths.

There are four usage modes for this (the same mode you select for the `FilterInternals` property in CIC filters at the MATLAB prompt).


- **Full**—All word and fraction lengths set to  $B_{\max} + 1$ , called  $B_{\text{accum}}$  by harris in [14]. Full Precision is the default setting.
- **Minimum section word lengths**—Set the section word lengths to minimum values that meet roundoff noise and output requirements as defined by Hogenauer in “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.” on page A-3.
- **Specify word lengths**—Enables the **Section word length** option for you to enter word lengths for each section. Enter either a scalar to use the same value for every section, or a vector of values, one for each section.
- **Specify all**—Enables the **Section fraction length** option in addition to **Section word length**. Now you can provide both the word and fraction lengths for each section, again using either a scalar or a vector of values.

## Realizing Filters as Simulink Subsystem Blocks

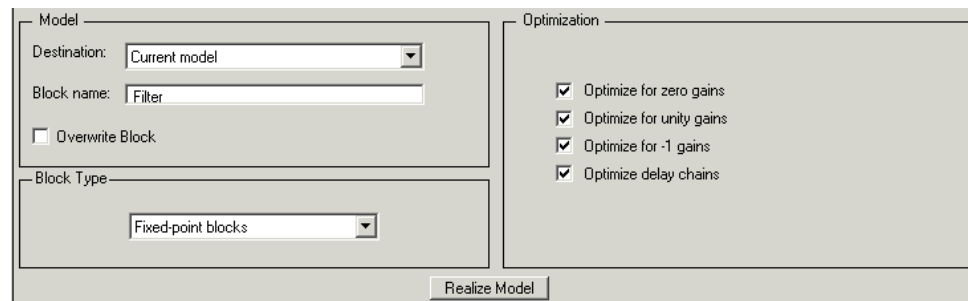
After you design or import a filter in FDATool, the realize model feature lets you create a Simulink subsystem block that implements your filter. The generated filter subsystem block uses the delay, gain, and sum blocks in fixed-point mode from Simulink. If you do not own Simulink Fixed Point, FDATool still realizes your model using blocks in fixed-point mode from Simulink, but you cannot run any model that includes your filter subsystem block in Simulink.

The block you realize from FDATool accepts only individual sample-based input, not vectors or frames as input. If you have input data in frames, consider unbuffering the input or converting the frames to sample-by-sample input in some other way.

### About the Realize Model Panel in FDATool

Switching FDATool to realize model mode, by clicking  on the sidebar, gives you access to the Realize Model panel and the options for realizing your quantized filter as a Simulink subsystem block.

On the panel, as shown here, are the options provided for configuring how FDATool realizes your model.



### Model Options

Under **Model**, you set options that direct FDATool where to put your new subsystem block and what to name the block.

**Destination.** Tells FDATool whether to put the new block in your current Simulink model or open a new Simulink model and add the block to that window. Select `Current model` to add the block to your current model, or select `New model` to create a new model for the block.

**Block name.** Provides FDATool with a name to assign to your block. When you realize your filter as a subsystem, the resulting block shows the name you enter here as the block name, positioned just below the block.

**Overwrite block.** Directs FDATool whether to overwrite an existing block with this block in the destination model. The result is that the new filter realization subsystem block replaces the existing filter subsystem block. Selecting this option replaces your existing filter realization subsystem block with the one you create when you click **Realize Model**. Clearing **Overwrite block** causes FDATool to create a new block in the destination model, rather than replacing the existing block.

### Block Type Option

To realize your quantized filter as a subsystem block, the most appropriate choice is to select `Fixed-point` blocks from the list. When you are licensed to use the fixed-point blocks in Signal Processing Blockset, you have the option of realizing your model as either fixed- or floating-point blocks. Since your filter is designed to use quantized coefficients, the fixed-point blocks option usually matches your needs most closely.

You can elect to realize your filter using floating-point blocks, with the understanding that while the coefficients and gains of your filter retain their fixed-point values (the filter uses the fixed-point values for both gain and coefficients, in floating-point format), the math performed during filtering uses floating-point arithmetic and does not truly match the output of your filter running in fixed-point mode. Although realizing your quantized filter with floating-point blocks is not recommended, selecting `Floating-point` blocks from the list creates your filter from blocks in Simulink and the Signal Processing Blockset.

If you do not own a license for Simulink Fixed Point, realizing your quantized filter as a subsystem generates a subsystem block that uses fixed-point blocks, but you cannot run or edit the block. If you use the filter subsystem in a Simulink model, you cannot run the model.

### Optimization Options

Four options enable you to tailor the way the realized model optimizes various filter features such as delays and gains. When you open the Realize Model panel, these options are selected by default.

**Optimize for zero gains.** Specify whether to remove zero-gain blocks from the realized filter.

**Optimize for unity gains.** Specify whether to replace unity-gain blocks with direct connections in the filter subsystem.

**Optimize for -1 gains.** Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block in the filter.

**Optimize delay chains.** Specify whether to replace cascaded chains of delay blocks with a single integer delay block to provide an equivalent delay.

Each of these options can optimize the way your filter performs in simulation and in code you might generate from your model.

### Example—Realize a Filter Using FDATool

After your quantized filter in FDATool is performing the way you want, with your desired phase and magnitude response, and with the right coefficients and form, follow these steps to realize your filter as a subsystem that you can use in a Simulink model.

- 1 Click **Realize Model** on the sidebar to change FDATool to realize model mode.
- 2 From the **Destination** list under **Model**, select either:
  - **Current model**—to add the realized filter subsystem to your current model
  - **New model**—to open a new Simulink model window and add your filter subsystem to the new window
- 3 Provide a name for your new filter subsystem in the **Name** field.
- 4 Decide whether to overwrite an existing block with this new one, and select or clear **Overwrite block** to direct FDATool which way to go—overwrite or not.

- 5 Select **Fixed-point** blocks from the list in **Block Type**.
- 6 Select or clear the optimizations to apply.
  - **Optimize for zero gains**—removes zero gain blocks from the model realization
  - **Optimize for unity gains**—replaces unity gain blocks with direct connections to adjacent blocks
  - **Optimize for -1 gains**—replaces negative gain blocks by a change of sign at the nearest sum block
  - **Optimize delay chains**—replaces cascaded delay blocks with a single delay block that produces the equivalent gain
- 7 Click **Realize Model** to realize your quantized filter as a subsystem block according to the settings you selected.

If you double-click the filter block subsystem created by FDATool, you see the filter implementation in Simulink model form. Depending on the options you chose when you realized your filter, and the filter you started with, you might see one or more sections, or different architectures based on the form of your quantized filter. From this point on, the subsystem filter block acts like any other block that you use in Simulink models.

### Supported Filter Structures

FDATool lets you realize discrete-time and multirate filters from the following forms:

Structure	Description
firdecim	Decimators based on FIR filters
firtdecim	Decimators based on transposed FIR filters
linearinterp	Linear interpolators
firinterp	Interpolators based on FIR filters
multirate polyphase	Multirate filters

<b>Structure</b>	<b>Description</b>
<code>holdinterp</code>	Interpolators that use the hold interpolation algorithm
<code>dfilt.allpass</code>	Discrete-time filters with allpass structure
<code>dfilt.cascadeallpass</code>	
<code>dfilt.cascadewdfallpass</code>	
<code>mfilt.iirdecim</code>	Decimators based on IIR filters
<code>mfilt.iirwdfdecim</code>	
<code>mfilt.iirinterp</code>	Interpolators based on IIR filters
<code>mfilt.iirwdfinterp</code>	
<code>dfilt.wdfallpass</code>	





## Getting Help for FDATool

To find out more about the buttons or options in the FDATool dialogs, use the **What's This?** button to access context-sensitive help.

### The What's This? Option

To find information on a particular option or region of the dialog:

- 1 Click the **What's This?** button .

Your cursor changes to .

- 2 Click the region or option of interest.

For example, click **Turn quantization on** to find out what this option does.

You can also select **What's this?** from the **Help** menu to launch context-sensitive help.

### Additional Help for FDATool

For help about importing filters into FDATool, or for details about using FDATool to create and analyze double-precision filters, refer to the “Filter Design and Analysis Tool Overview” in your Signal Processing Toolbox documentation.



# Reference for the Properties of Filter Objects

---

Fixed-Point Filter Properties (p. 7-3)	Provides an overview and details of the properties of fixed-point filters
Adaptive Filter Properties (p. 7-103)	Summarizes and details the properties of adaptive filters
Multirate Filter Properties (p. 7-117)	Provides a summary and the details of the properties of multirate filters

## Overview

This chapter presents all of the properties for adaptive filters (`adaptfilt` objects), discrete-time filters (both floating-point and fixed-point `dfilt` objects), and multirate filters (`mfilt` objects).

- “Fixed-Point Filter Properties” on page 7-3
- “Adaptive Filter Properties” on page 7-103
- “Multirate Filter Properties” on page 7-117

## Fixed-Point Filter Properties

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.

This section presents 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 the Fixed-Point Toolbox. 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 the Fixed Point Toolbox.

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 we often call 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.

<b>Property Name</b>	<b>Valid Values [default value]</b>	<b>Brief Description</b>
AccumFracLength	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—DenAccumFracLength and NumAccumFracLength—that let you set the precision for numerator and denominator operations separately.
AccumWordLength	Any positive integer number of bits [40]	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	[Double], single, fixed	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	[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 false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.

<b>Property Name (Continued)</b>	<b>Valid Values [default value]</b>	<b>Brief Description</b>
CoeffFracLength	Any positive or negative integer number of bits [14]	Set the fraction length the filter uses to interpret coefficients. CoeffFracLength is not available until you set CoeffAutoScale to false. 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. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Any filter coefficient value [1]	Holds the denominator coefficients for IIR filters.
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 ProductMode to SpecifyPrecision.
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.
DenStateWordLength	Any positive integer number of bits [16]	Specifies the word length used to represent the states associated with denominator coefficients in the filter.

<b>Property Name (Continued)</b>	<b>Valid Values [default value]</b>	<b>Brief Description</b>
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 (Continued)</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.

<b>Property Name (Continued)</b>	<b>Valid Values [default value]</b>	<b>Brief Description</b>
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
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.

Property Name (Continued)	Valid Values [default value]	Brief Description
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> <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—they maintain full precision.

<b>Property Name (Continued)</b>	<b>Valid Values [default value]</b>	<b>Brief Description</b>
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.
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.



Property Name (Continued)	Valid Values [default value]	Brief Description
RoundMode	[Convergent], ceil, fix, floor, 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>convergent</code>—Round up to the next allowable quantized value.</li> <li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li> <li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li> <li>• <code>floor</code>—Round down to the next allowable quantized value.</li> <li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

<b>Property Name (Continued)</b>	<b>Valid Values [default value]</b>	<b>Brief Description</b>
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 CoeffAutoScale by setting it to false.
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.
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.
SectionInputAutoScale	[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.

<b>Property Name (Continued)</b>	<b>Valid Values [default value]</b>	<b>Brief Description</b>
SectionInputFracLength	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.
SectionInputWordLength	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.
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 false 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 false.
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.

<b>Property Name (Continued)</b>	<b>Valid Values [default value]</b>	<b>Brief Description</b>
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 <i>f i</i> ]	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 <i>f i</i> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in your Signal Processing Toolbox documentation or in the Help system.
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 <code>TapSumMode</code> to <code>false</code> . Symmetric and antisymmetric FIR filters include this property.

Property Name (Continued)	Valid Values [default value]	Brief Description
TapSumMode	FullPrecision, KeepLSB, [KeepMSB], SpecifyPrecision	Determines how the accumulator outputs stored that involve filter tap weights. Choose from full precision ( <code>FullPrecision</code> ) to prevent overflows, or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when outputting results from the accumulator. To let you set the precision (the fraction length) used by the output from the accumulator, set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .  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, we use `filter`, `dfilt` object, and `filter` object 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`.

<b>Arithmetic Property String</b>	<b>Brief Description of Effect on the Filter</b>
<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 Arithmetic 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 the Fixed-Point Toolbox 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 the Signal Processing Toolbox.

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 the Fixed-Point Toolbox as well as the Filter Design Toolbox.

---

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.dfiir]
      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 we 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 us we 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 the Fixed-Point Toolbox 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 “Working with Fixed-Point Direct-Form FIR Filters” on page 2-14.

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.

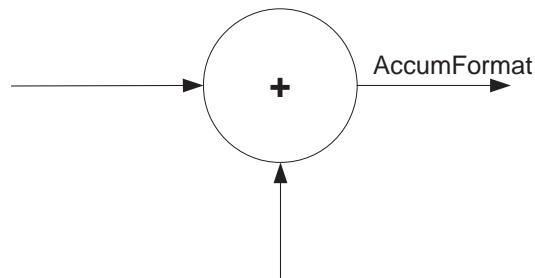
Notice that the output format for every sum operation reflects the value of the output property specified in the filter structure diagram. Which input property is referenced by `CastBeforeSum` depends on the structure.

<b>Property Value</b>	<b>Description</b>
<code>false</code>	Configures filter summation operations to retain the addends in the format carried from the previous operation.
<code>true</code>	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.

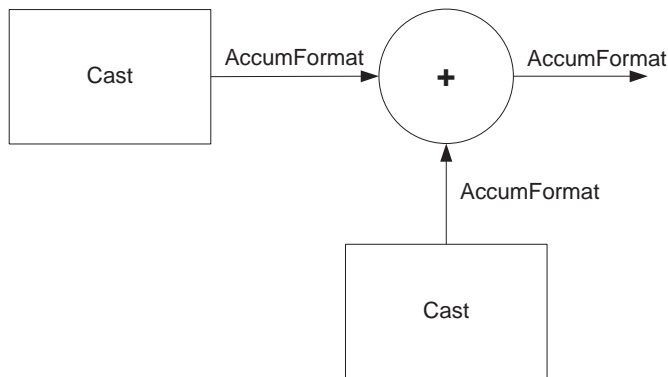
### **Diagrams of `CastBeforeSum` Settings**

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.

<b>Filter Constructor Name</b>	<b>FilterStructure Property String and Filter Type</b>
'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
'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, Figure 7-1 presents the structure for a Direct Form 2 filter, including the quantizations (fixed-point formats) that compose part of



When the leading denominator coefficient  $a(1)$  in your filter is not 1, choose it to be a power of two so that a shift replaces the multiply that would otherwise be used.

### Fixed-Point Arithmetic Filter Structures

You choose among several filter structures when you create fixed-point filters. You can also specify filters with single or multiple cascaded sections of the same type. Because quantization is a nonlinear process, different fixed-point filter structures produce different results.

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.

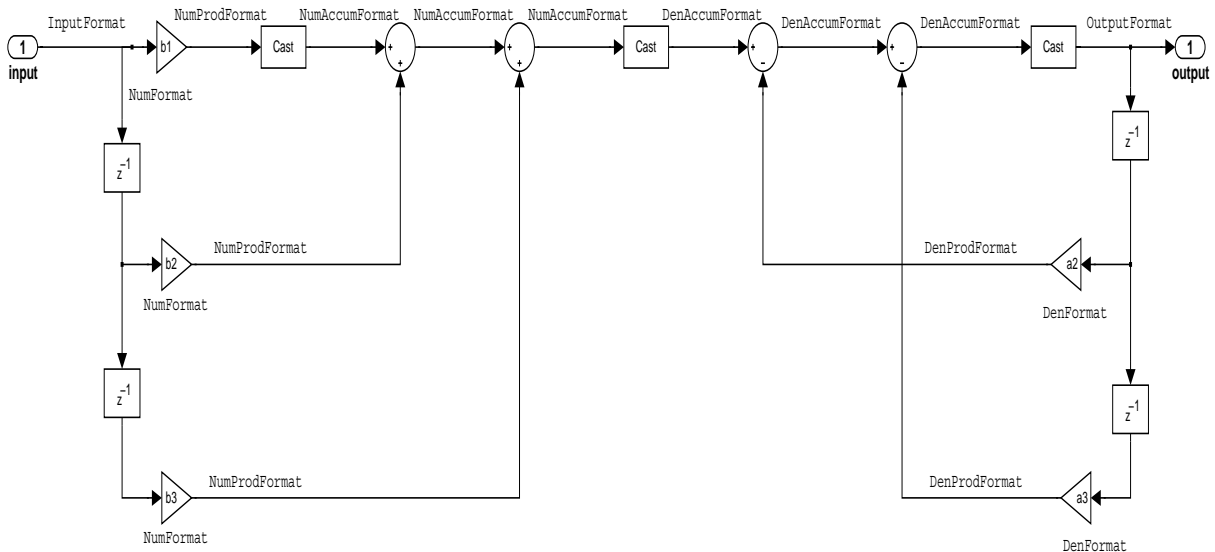
For example, in the `dfilt.df2` filter shown on page 7-45, the entries `InputFormat` and `OutputFormat` are the formats applied, that is the word length and fraction length, to the filter input and output data. For example, filter properties like `OutputWordLength` and `InputWordLength` specify values 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.

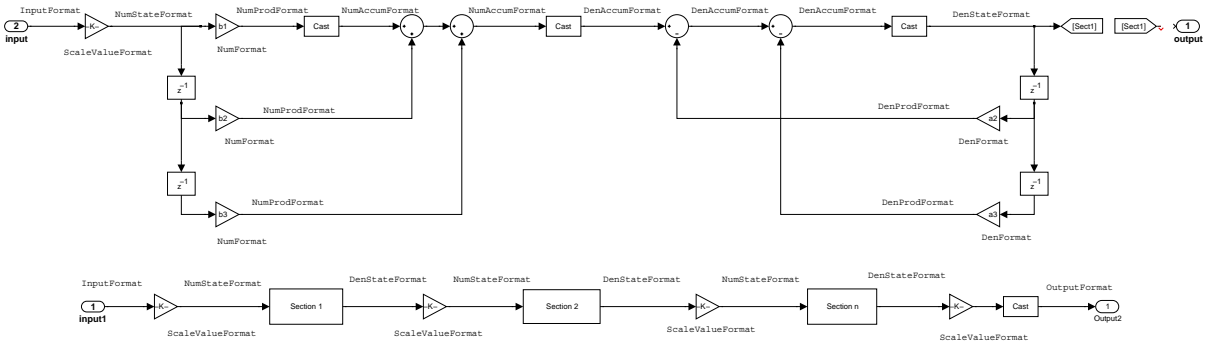
```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1(b,a);
```

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);
```

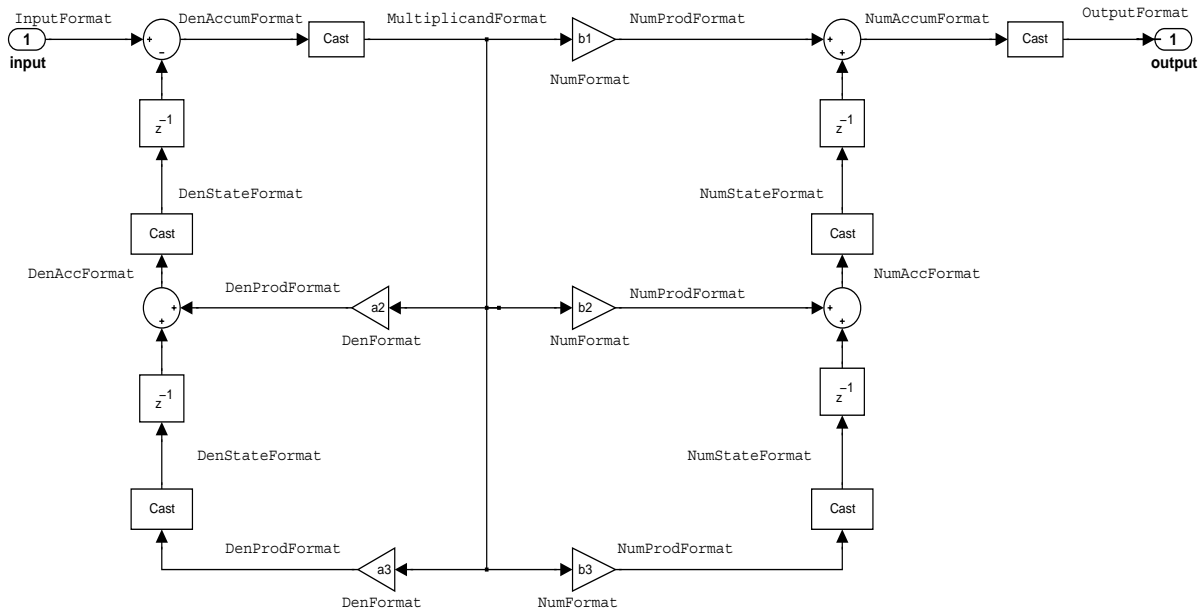
To create the fixed-point filter, set the Arithmetic property to fixed, as shown here.

```
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.



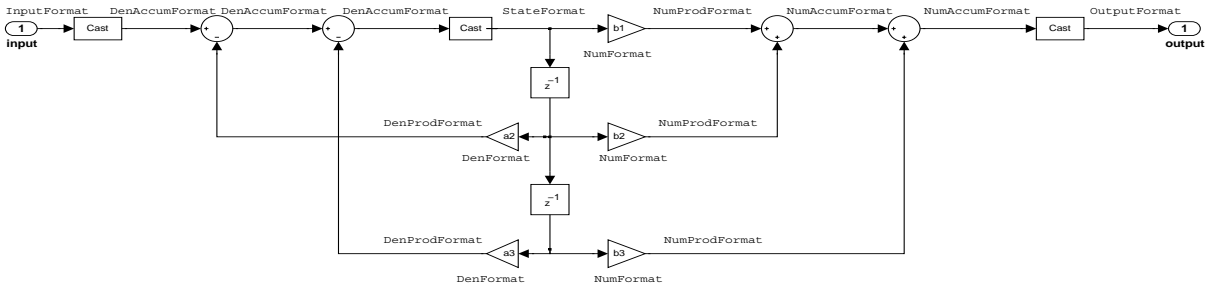
**Example—Specifying a Direct Form I Transposed Filter.** You can specify a second-order direct form I transposed filter structure for a quantized filter `hq` with the following code.

```
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.

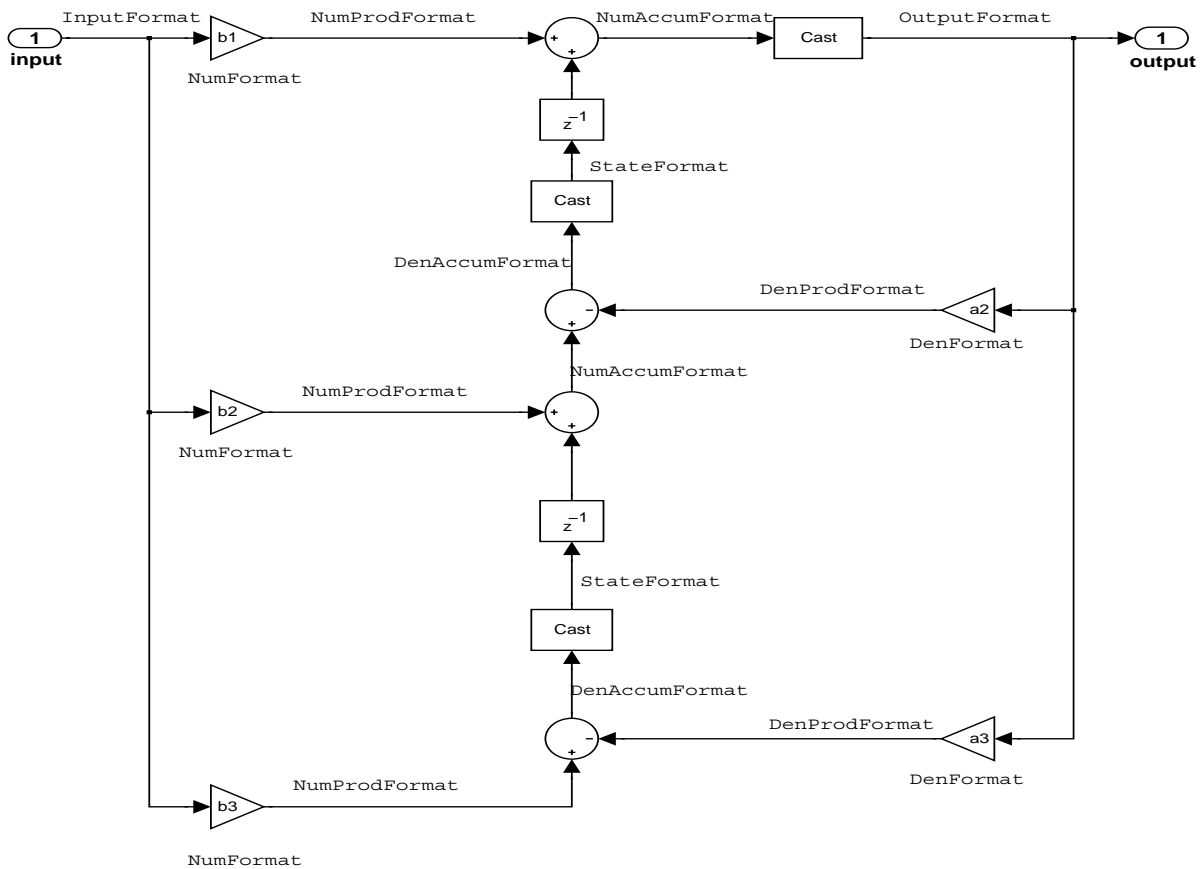
```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df2({b,a});
hq.arithmetic = 'fixed'
```

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 Transposed Filter Structure

The following figure depicts the *direct form II transposed* filter structure that directly realizes transfer functions with a second-order numerator and denominator. The numerator coefficients are labeled  $b(i)$ , the denominator coefficients are labeled  $a(i)$ ,  $i = 1, 2, 3$ , and the states (used for initial and final state values in filtering) are labeled  $z(i)$ . In the first figure, the Arithmetic property value is fixed.



Use the constructor `dfilt.df2t` to specify the value of the `FilterStructure` property for a filter with this structure that you can convert to fixed-point filtering.

**Example—Specifying a Direct Form II Transposed Filter.** Specifying or constructing a second-order direct form II transposed filter for a fixed-point filter `hq` starts with the following code to define the coefficients and construct the filter.

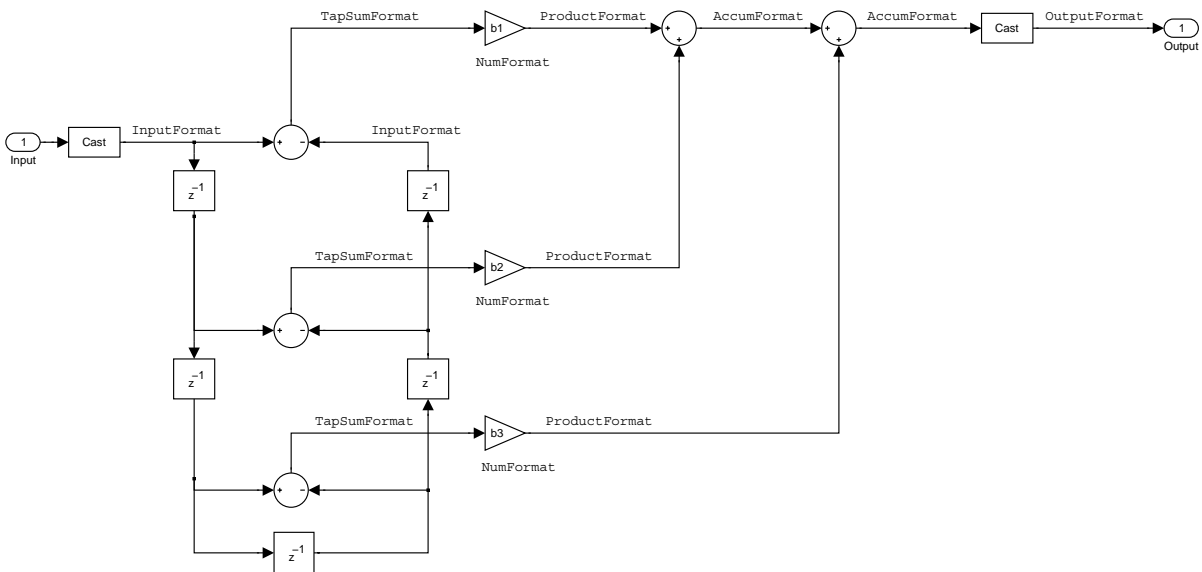
```
b = [0.3 0.6 0.3];  
a = [1 0 0.2];  
hd = dfilt.df2t({b,a});
```

Now create the fixed-point filtering version of the filter from `hd`, which is floating point.

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

### 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.dfasymfir` 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.dfasymfir(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.

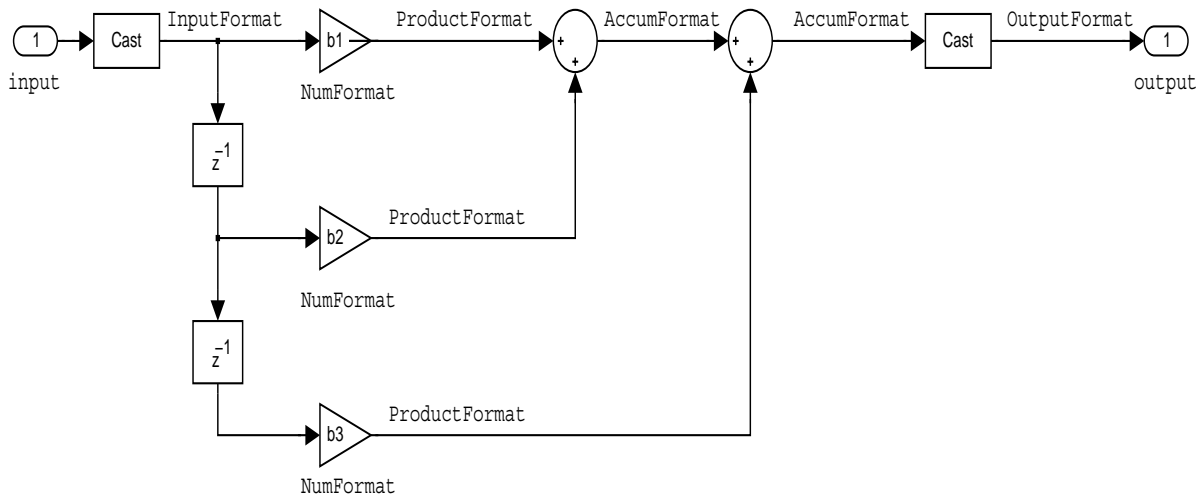
```
b = [-0.01 0.1 0.0 -0.1 0.01];  
hq = dfilt.dfasymfir(b);  
hq.arithmetic='fixed'
```

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.



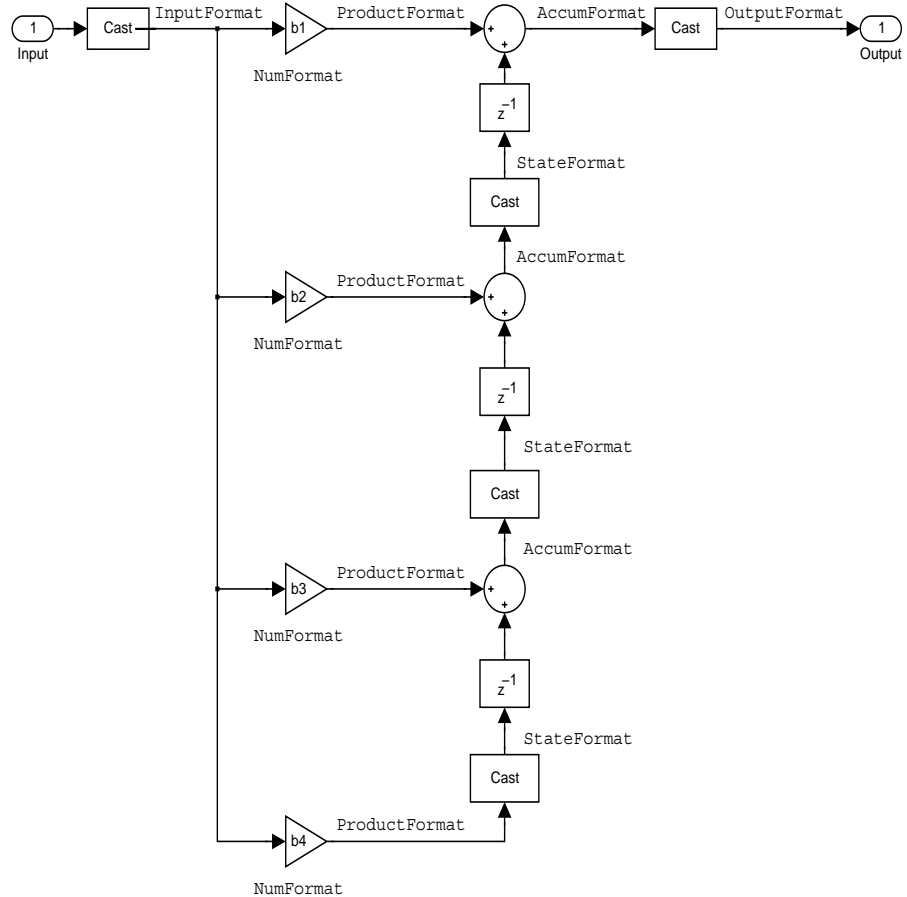
Use the `dfilt.dffir` method to generate a filter that uses this structure.

**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';
```

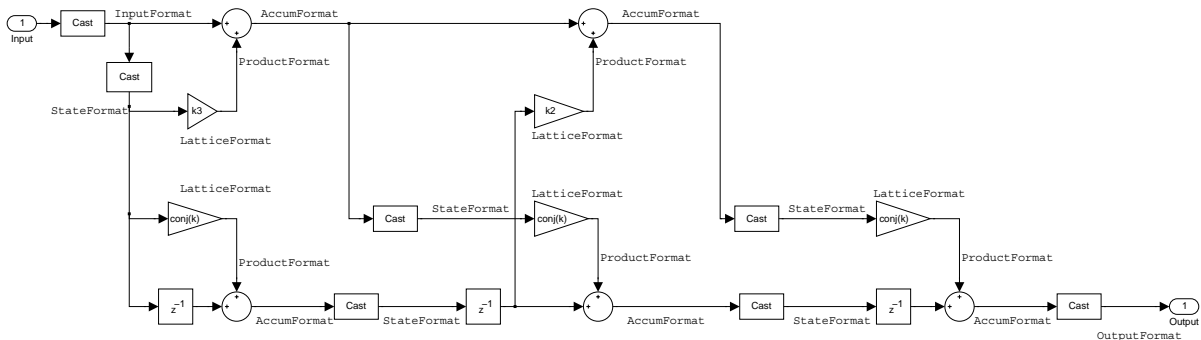


## Lattice Moving Average Maximum Phase Filter Structure

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.

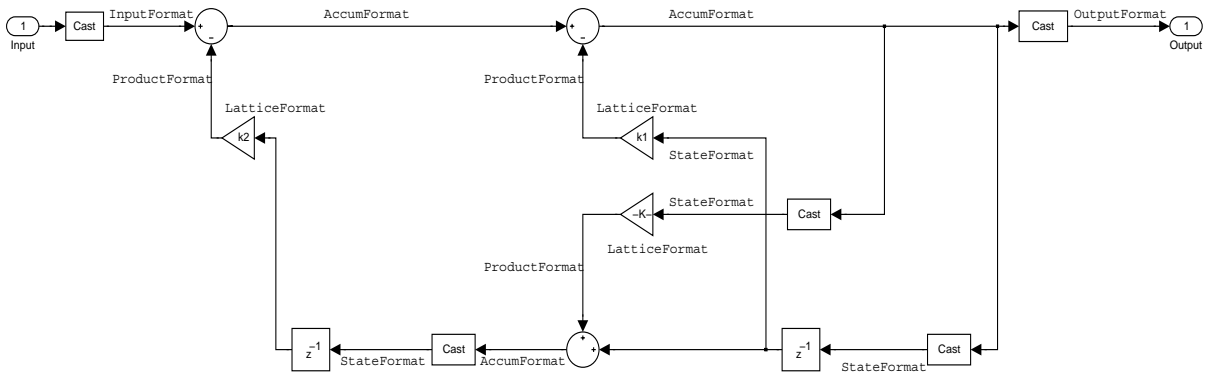


**Example—Constructing a Lattice Moving Average Maximum Phase Filter.** Constructing a fourth-order lattice MA maximum phase filter structure for a quantized filter `hq` begins with the following code.

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

### Lattice Autoregressive (AR) Filter Structure

The method `dfilt.latticear` directly realizes lattice autoregressive filters in the toolbox. The following figure depicts the third-order *lattice autoregressive (AR)* filter structure—with the Arithmetic property equal to fixed. The filter reflection coefficients are labeled  $k(i)$ ,  $i = 1, 2, 3$ , and the states (used for initial and final state values in filtering) are labeled  $z(i)$ .



**Example—Specifying a Lattice AR Filter.** You can specify a third-order lattice AR filter structure for a quantized filter `hq` with the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticear({k});
hq = copy(hd);
hq.arithmetic = 'custom';
```



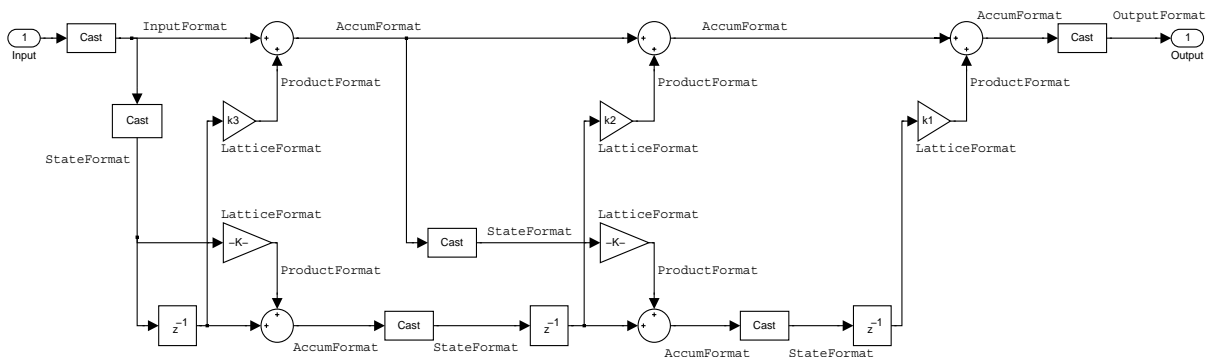
### Lattice Moving Average (MA) Filter Structure for Minimum Phase

The following figures depict *lattice moving average (MA)* filter structures that directly realize third-order lattice MA filters for minimum phase. The filter reflection coefficients are labeled  $k(i)$ ,  $i = 1, 2, 3$ , and the states (used for initial and final state values in filtering) are labeled  $z(i)$ . Setting the Arithmetic property of the filter to fixed results in a fixed-point filter that matches the figure.

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 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  $z(i)$ . 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.

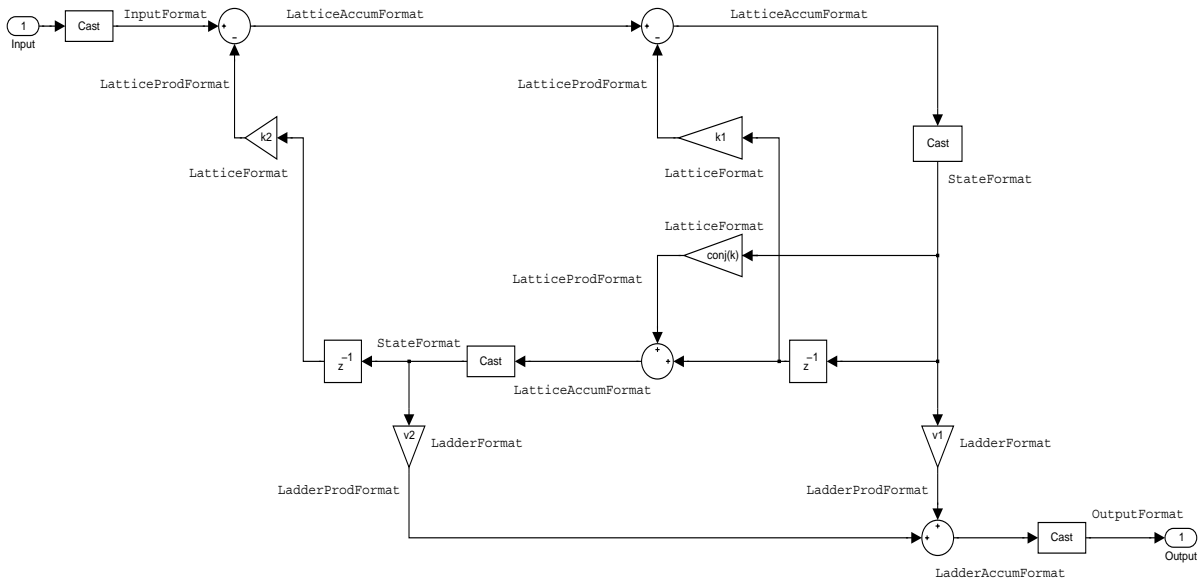


**Example—Specifying a Minimum Phase Lattice MA Filter.** You can specify a third-order lattice MA filter structure for minimum phase applications using variations of the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticemamin({k});
hq = copy(hd);
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  $z(i)$ .

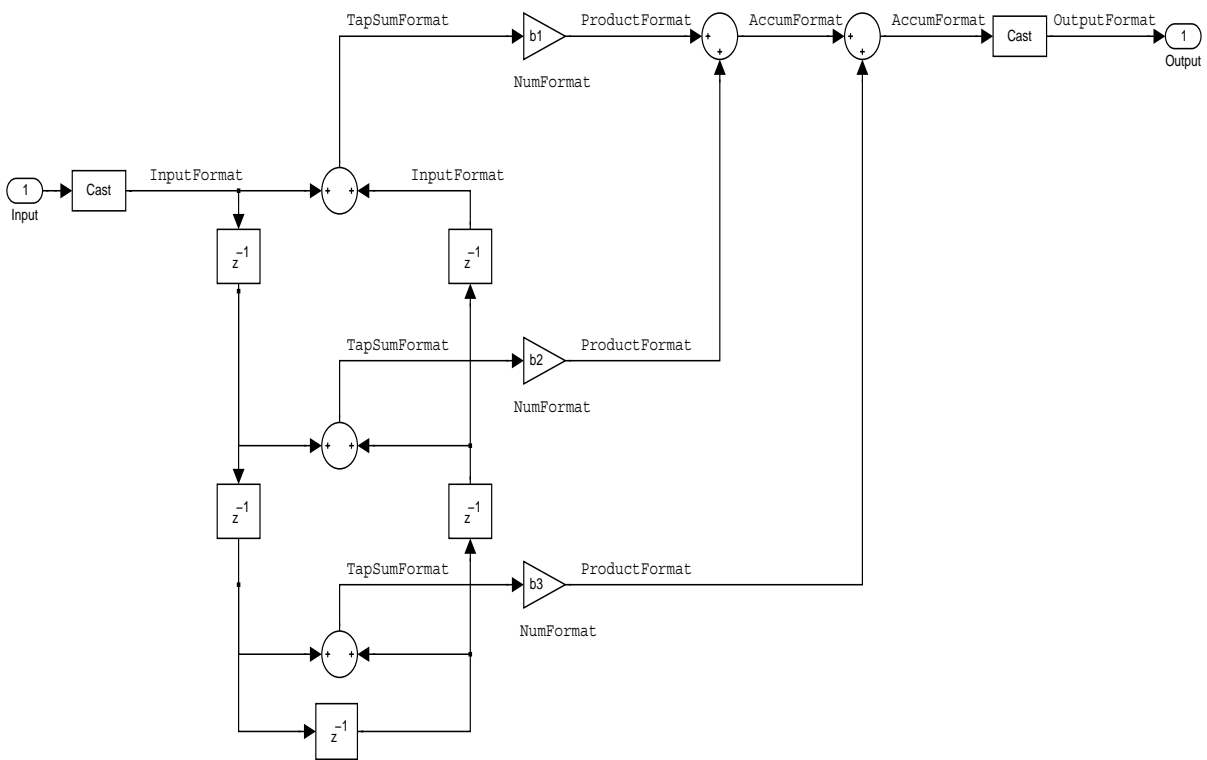


**Example—Specifying an Lattice ARMA Filter.** The following code specifies a fourth-order lattice ARMA filter structure for a quantized filter hq, starting from hd, a floating-point version of the filter.

```
k = [.66 .7 .44 .66];  
v = [1 0 0];  
hd=dfilt.latticearma({k,v});  
hq = copy(hd);  
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});
```

```
hq = copy(hd);
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 the Signal Processing Toolbox 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.

Conversion Function	Description
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.

Conversion Function	Description
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.

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.

- '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 the [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  $\pm\text{inf}$ .

---

### 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 7-30.

## 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 we 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.

RoundMode String	Description of Rounding Algorithm
'ceil'	Round up to the next representable quantized value.
'convergent'	Round to the nearest representable quantized value. Numbers that are exactly halfway between the two nearest representable quantized values are rounded up when the least significant bit would be set to 1 after rounding. Otherwise, the number is rounded down. Filter objects use convergent rounding by default.
'fix'	Round negative numbers up and positive numbers down to the next representable quantized value.
'floor'	Round down to the next representable quantized value.
'round'	Round to the nearest representable quantized value. Numbers that are halfway between the two nearest representable quantized values are rounded up.

**Default value:** 'convergent'

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 interpretation of this property is described below with diagrams in “Interpreting the `ScaleValues` Property”.

**Default value:** 0

**Remarks:** 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.

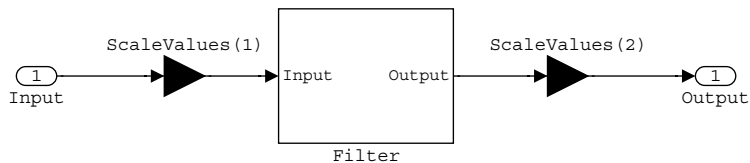
### Interpreting the `ScaleValues` Property

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.

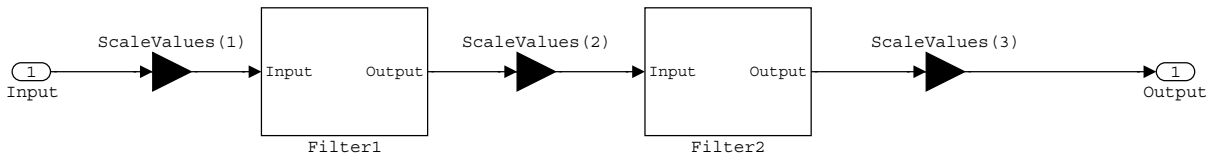
The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with only one section.

#### 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 = 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.

By looking at one of the SOS transposed filter structures, such as this one for the transposed direct-form I filter implemented using second-order sections, you see the filter sections at the bottom of the figure.



- `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`.

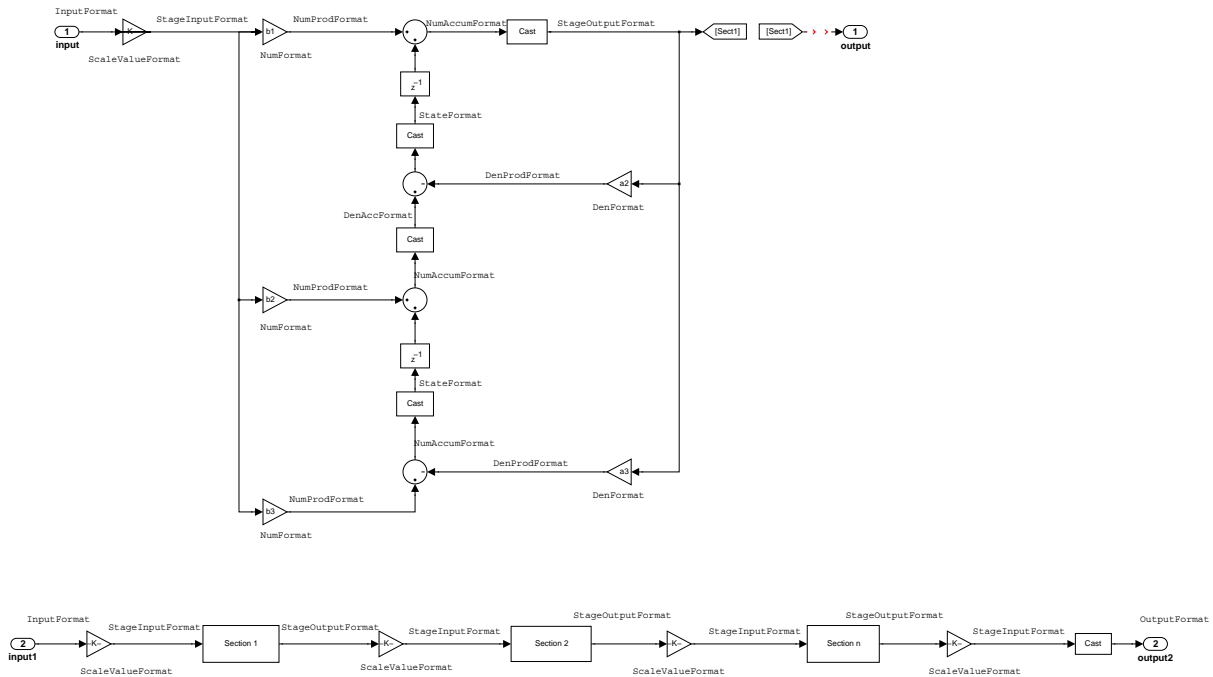
In combination with `CoeffWordLength`, `SectionOutputFracLength` determines how the filter interprets and uses the state values stored in the property States. As with all fraction length properties, `SectionOutputFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive 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.

By looking at one of the SOS transposed filter structures, such as this one for the transposed direct-form I filter implemented using second-order sections, you see the filter sections at the bottom of the figure.





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 as using them while filtering in loops to maintain the filter state between loop iterations. In the toolbox we assume zero-valued initial conditions (the dynamic system is at rest) by default when we 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

The following table summarizes the adaptive filter properties and provides a brief description of each. Full descriptions of each property, in alphabetical order, follow the table.

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 $(1 - \text{step})$ . lambda is the input argument that represents AvgFactor
BkwdPredErrorPower	Returns the minimum mean-squared prediction error. Refer to [12] in the bibliography for details about linear prediction.
BkwdPrediction	Returns the predicted samples generated during adaptation. Refer to [12] in the bibliography for details about linear prediction.

<b>Property (Continued)</b>	<b>Description</b>
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.

Property (Continued)	Description
ErrorStates	Vector of the adaptive filter error states. ErrorStates defaults to a zero vector with length equal to (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.
FwdPredErrorPower	Returns the minimum mean-squared prediction error in the forward direction. Refer to [12] 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.

<b>Property (Continued)</b>	<b>Description</b>
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.
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.

<b>Property (Continued)</b>	<b>Description</b>
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. projectord 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.

<b>Property (Continued)</b>	<b>Description</b>
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 <code>projectord</code> .

<b>Property (Continued)</b>	<b>Description</b>
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.

### 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

### 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 [12] 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.

## Multirate Filter Properties

The following table summarizes the multirate filter properties and provides a brief description of each. Full descriptions of each property follow in the next 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> exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.

<b>Name</b>	<b>Values</b>	<b>Default</b>	<b>Description</b>
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.
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.
Numerator	Array of double values	No default values	Vector containing the coefficients of the FIR lowpass filter used for interpolation.



Name	Values	Default	Description
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	<p>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.</p> <p>PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.</p>
RateChangeFactors	[1,m]	[2,3] or [3,2]	<p>Reports the decimation (m) and interpolation (1) 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.</p>
States	Any m+1-by-n matrix of double values	2-by-2 matrix, int32	<p>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.</p>

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 [1]. When you choose SpecifyWordLengths, you provide the word length for each section. In addition, choosing SpecifyWordLengths exposes the SectionWordLengths property for you to modify as needed.
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 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. On the next pages, we list the `mfilt` object properties in alphabetical order.

### **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  $[BlockLength + length(filter\ order)]$  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 Modes

There are four modes of usage for this which are set using the `FilterInternals` property in multirate filters.

- `FullPrecision`—All word and fraction lengths set to  $B_{\max} + 1$ , called  $B_{\text{accum}}$  by fred harris in [14]. 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  $B_{\text{accum}}$  as defined by

$$B_{\text{accum}} = \text{ceil}(N_{\text{secs}}(\text{Log}_2(D \times M)) + \text{InputWordLength})$$

where  $N_{\text{secs}}$  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  $B_{\text{accum}}$ , roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [15]) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than  $B_{\text{accum}}$  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  $B_{\text{accum}}$  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  $2*(\text{NumberOfSections})$ , where each vector element represents the word length for a section. If you specify a scalar, such as  $B_{\text{accum}}$ , 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  $2*(\text{NumberOfSections})$  with elements that represent the word length for each section. When you enter a scalar such as  $B_{\text{accum}}$ , 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  $2*(\text{NumberOfSections})$  with elements that represent the fraction length for each section as well. When you enter a scalar such as  $B_{\text{accum}}$ , 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

```
InputOffset = mod(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 the 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)`

# Function Reference

---

Functions — By Category (p. 8-2)	Lists the functions in the toolbox, by category, such as object constructors or analysis functions
Adaptive Filter Constructors (p. 8-3)	Lists all for the functions for designing adaptive filters
Discrete-Time Filter Constructors (p. 8-6)	Lists all of the functions for designing discrete-time filters
Multirate Filter Constructors (p. 8-10)	Lists the multirate filter design functions
Filter Analysis Methods (p. 8-12)	Lists the analysis functions provided for working with adaptive, discrete-time, and multirate filters

## Functions — By Category

Adaptive Filter Constructors (p. 8-3)	Functions for designing adaptive filters
Discrete-Time Filter Constructors (p. 8-6)	Functions for designing FIR and IIR discrete-time filter objects
Filter Specification Objects — Response Types (p. 8-8)	Methods for creating objects that specify filter responses, such as lowpass or bandstop
Filter Specification Objects — Design Methods (p. 8-9)	Methods for designing filter objects from specification objects
Multirate Filter Constructors (p. 8-10)	Functions for designing many types of multirate filter objects
Filter Analysis Methods (p. 8-12)	Methods for analyzing filters and filter objects
Fixed-Point Filter Construction and Property Functions (p. 8-17)	Methods and functions for creating fixed-point filters
Quantized Filter Analysis Functions (p. 8-17)	Functions for analyzing fixed-point filters
SOS Conversion Functions (p. 8-19)	Functions for working with second-order section filters
Filter Design Functions (p. 8-19)	Functions for designing filters (not object-based)
Filter Conversion Functions (p. 8-20)	Functions that let you transform filters to other forms, or use features in a filter to develop another filter



## Adaptive Filter Constructors

Least Mean Squares (LMS) Based FIR Adaptive Filters (p. 8-3)	Lists the filter functions that rely on the LMS technique
Recursive Least Squares (RLS) Based FIR Adaptive Filters (p. 8-4)	Lists the filter functions that rely on the RLS technique
Affine Projection (AP) FIR Adaptive Filters (p. 8-4)	Lists the filter functions that affine projection
FIR Adaptive Filters in the Frequency Domain (FD) (p. 8-4)	Lists the filter functions that work in the frequency domain
Lattice Based (L) FIR Adaptive Filters (p. 8-6)	Lists the filter functions that rely on lattice filters

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

<code>adaptfilt.adjlms</code>	Adjoint least mean square (LMS) FIR adaptive filter that adapts using adjoint LMS algorithm
<code>adaptfilt.blms</code>	Construct Block LMS (BLMS) FIR adaptive filter
<code>adaptfilt.blmsfft</code>	Construct FFT-based block LMS FIR adaptive filter
<code>adaptfilt.dlms</code>	Create delayed LMS FIR adaptive filter object
<code>adaptfilt.filtxlms</code>	Create filtered-x LMS FIR adaptive filter
<code>adaptfilt.lms</code>	Construct least-mean-square (LMS) FIR adaptive filter object
<code>adaptfilt.nlms</code>	Construct normalized least mean squares (LMS) FIR adaptive filter object
<code>adaptfilt.sd</code>	Construct FIR adaptive filter object that uses sign-data algorithm
<code>adaptfilt.se</code>	Construct sign-error algorithm FIR adaptive filter object
<code>adaptfilt.ss</code>	Construct adaptive FIR filter object that uses sign-sign algorithm

**Recursive Least Squares (RLS) Based FIR Adaptive Filters**

<code>adaptfilt.ftf</code>	Construct fast transversal least squares adaptive filter object
<code>adaptfilt.hrls</code>	Construct a householder recursive least squares (RLS) FIR adaptive filter object
<code>adaptfilt.hswrls</code>	Construct householder sliding window recursive least squares (RLS) FIR adaptive filter
<code>adaptfilt.qdrls</code>	Create QR-decomposition-based recursive least squares (RLS) FIR adaptive filter object
<code>adaptfilt.rls</code>	Construct direct form recursive least squares (RLS) FIR adaptive filter object
<code>adaptfilt.swftf</code>	Construct sliding window fast transversal least squares adaptive filter object
<code>adaptfilt.swrls</code>	Construct sliding window recursive least squares (RLS) FIR adaptive filter

**Affine Projection (AP) FIR Adaptive Filters**

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

**FIR Adaptive Filters in the Frequency Domain (FD)**

<code>adaptfilt.fdaf</code>	Construct frequency-domain FIR adaptive filter with bin step size normalization
<code>adaptfilt.pbfdaf</code>	Construct partitioned block frequency-domain (PBFDAF) FIR adaptive filter with bin step size normalization

<code>adaptfilt.pbufdaf</code>	Construct partitioned block unconstrained frequency-domain (PBUFD AF) FIR adaptive filter with bin step size normalization
<code>adaptfilt.tdafdct</code>	Construct transform-domain (TDAFDCT) adaptive filter object that uses discrete cosine transform
<code>adaptfilt.tdafdft</code>	Create transform-domain (TDAFDFT) adaptive filter object that uses discrete Fourier transform
<code>adaptfilt.ufdaf</code>	Construct unconstrained frequency-domain (UFD AF) FIR adaptive filter with quantized step size normalization

**Lattice Based (L) FIR Adaptive Filters**

<code>adaptfilt.gal</code>	Construct gradient adaptive lattice FIR filter
<code>adaptfilt.lsl</code>	Construct least squares lattice (LSL) adaptive filter
<code>adaptfilt.qrdls1</code>	QR-decomposition-based least squares lattice (LSL) adaptive filter object

**Discrete-Time Filter Constructors**

<code>dfilt.allpass</code>	Construct allpass filter object
<code>dfilt.calattice</code>	Construct discrete-time, coupled-allpass, lattice filter object
<code>dfilt.calatticepc</code>	Construct discrete-time, coupled-allpass, power-complementary lattice filter object
<code>dfilt.cascade</code>	Construct cascade of discrete-time filter objects
<code>dfilt.cascadeallpass</code>	Construct cascade of allpass discrete-time filter objects
<code>dfilt.cascadewdfallpass</code>	Construct allpass wave digital filter (WDF) object by cascading allpass WDF filter objects
<code>dfilt.df1</code>	Construct discrete-time, direct-form I filter object
<code>dfilt.df1sos</code>	Construct discrete-time, direct-form I filter object that uses second-order sections
<code>dfilt.df1t</code>	Construct discrete-time, direct-form I transposed filter object
<code>dfilt.df1tsos</code>	Construct discrete-time, second-order section, direct-form I transposed filter object
<code>dfilt.df2</code>	Construct discrete-time, direct-form II filter object
<code>dfilt.df2sos</code>	Construct discrete-time, second-order section, direct-form II filter object

---

<code>dfilt.df2t</code>	Construct discrete-time, direct-form II transposed filter object
<code>dfilt.df2tsos</code>	Construct discrete-time, second-order section direct-form II transposed filter object
<code>dfilt.dfasymfir</code>	Construct discrete-time, direct-form antisymmetric FIR filter object
<code>dfilt.dffir</code>	Construct discrete-time direct-form FIR filter object
<code>dfilt.dffirt</code>	Construct discrete-time, direct-form FIR transposed filter object
<code>dfilt.dfsymfir</code>	Construct discrete-time, direct-form symmetric FIR filter object
<code>dfilt.latticeallpass</code>	Construct discrete-time, lattice allpass filter object
<code>dfilt.latticear</code>	Construct discrete-time, lattice, autoregressive filter object
<code>dfilt.latticearma</code>	Construct discrete-time, lattice, autoregressive, moving-average filter object
<code>dfilt.latticemamax</code>	Construct discrete-time, lattice, moving-average filter object with maximum phase
<code>dfilt.latticemamin</code>	Construct discrete-time, lattice, moving-average filter object with minimum phase
<code>dfilt.parallel</code>	Construct discrete-time, parallel structure filter object
<code>dfilt.scalar</code>	Construct discrete-time, scalar filter object
<code>dfilt.wdfallpass</code>	Construct wave digital allpass filter object

## Filter Specification Objects – Response Types

<code>fdesign.arbmag</code>	Construct filter specification object for designing arbitrary response magnitude filters
<code>fdesign.arbmagnphase</code>	Design discrete-time filter specification object for arbitrary magnitude and phase response
<code>fdesign.bandpass</code>	Construct bandpass filter specification object
<code>fdesign.bandstop</code>	Construct bandstop filter specification object
<code>fdesign.ciccomp</code>	Construct filter cascaded-integrator comb (CIC) compensator filter specification object
<code>fdesign.decimator</code>	Construct decimator filter specification object
<code>fdesign.differentiator</code>	Construct differentiator filter specification object
<code>fdesign.halfband</code>	Construct halfband filter specification object
<code>fdesign.highpass</code>	Construct highpass filter specification object
<code>fdesign.hilbert</code>	Construct Hilbert filter specification object
<code>fdesign.interpolator</code>	Construct interpolator filter specification object
<code>fdesign.isinclp</code>	Construct inverse-sinc filter specification object
<code>fdesign.lowpass</code>	Construct lowpass filter specification object
<code>fdesign.nyquist</code>	Construct Nyquist filter specification object
<code>fdesign.rsrc</code>	Construct rational-factor sample-rate converter specifications object

## Filter Specification Objects — Design Methods

<code>butter</code>	Design Butterworth IIR digital filter using the specifications in filter specification object
<code>cheby1</code>	Design Chebyshev Type I digital filter using filter specification object
<code>cheby2</code>	Design Chebyshev Type II digital filter using filter specification object
<code>designmethods</code>	Design methods available for designing filter from filter specification object
<code>designopts</code>	Input arguments and default values applicable to filter specification object and method
<code>ellip</code>	Design elliptical or Causer digital filter using filter specification object
<code>equiripple</code>	Design equiripple single-rate or multirate FIR filter from filter specification object
<code>firls</code>	Design filter from filter specification object and least-square minimization technique
<code>ifir</code>	Use interpolated FIR method to design FIR filter from specification object
<code>iirlinphase</code>	Design quasi-linear phase IIR filter from halfband filter specification object
<code>kaiserwin</code>	Use Kaiser window to design filter from filter specification object
<code>multistage</code>	Design multistage filter from filter specification object
<code>window</code>	Use window design method to construct filter from specification object

## Multirate Filter Constructors

<code>mfilt.cascade</code>	Cascade <code>dfilt</code> and <code>mfilt</code> object(s) into filter
<code>mfilt.cicdecim</code>	Construct fixed-point cascaded integrator-comb (CIC) decimator filter object
<code>mfilt.cicinterp</code>	Construct fixed-point cascaded integrator-comb (CIC) interpolator filter object
<code>mfilt.fftfirinterp</code>	Construct overlap-add FIR polyphase interpolator filter object
<code>mfilt.firdecim</code>	Construct direct-form FIR polyphase decimator filter
<code>mfilt.firfracdecim</code>	Construct direct-form FIR polyphase fractional decimator filter object
<code>mfilt.firfracinterp</code>	Construct direct-form FIR polyphase fractional interpolator filter object
<code>mfilt.firinterp</code>	Construct FIR filter-based interpolator
<code>mfilt.firsrc</code>	Construct direct-form FIR polyphase sample rate converters
<code>mfilt.firtdecim</code>	Construct direct-form transposed FIR filter
<code>mfilt.holdinterp</code>	Construct FIR hold interpolator
<code>mfilt.iirdecim</code>	Construct IIR decimator filter object
<code>mfilt.iirinterp</code>	Construct IIR interpolator filter object
<code>mfilt.iirwdfdecim</code>	Construct IIR wave digital filter decimator object



`mfilt.iirwdfinterp`Construct IIR wave digital  
interpolator filter`mfilt.linearinterp`

Construct linear interpolator filter

## Filter Analysis Methods

<code>block</code>	Multirate (some)	Generate a Signal Processing Blockset block from floating-point or fixed-point multirate ( <code>mfilt</code> ) filter objects. Works only when Signal Processing Blockset is installed.
<code>coefficients</code>	Multirate	Filter coefficients for adaptive, discrete-time, and multirate filter.
<code>cumsec</code>	Discrete-time filters	Vector of filters for cumulative sections
<code>denormalize</code>	Discrete-time filters	Reverse filter coefficient and gain changes caused by function <code>normalize</code>
<code>disp</code>	All filters	Filter object with properties and values
<code>double</code>	Fixed-point filters	Cast fixed-point filter to filter that uses double-precision arithmetic
<code>euclidfactors</code>	Multirate	Use Euclid's theorem to return integer factors for multirate filter
<code>filter</code>	All filters	Apply filter objects to data and access states and filtering information
<code>filtmsb</code>	Multirate filters	$B_{\max}$ , most significant bit, of cascaded integrator-comb (CIC) filter
<code>filtstates.cic</code>	CIC filters	Object for storing states of cascaded-integrator comb (CIC) filters

<code>firtype</code>	Multirate filters	Determine type of linear phase FIR filter, either discrete-time or multirate
<code>freqsamp</code>	Discrete-time filters	Design real or complex frequency-sampled FIR filter from filter specification object
<code>freqz</code>	All filters	Compute frequency response of discrete-time filters, adaptive filters, or multirate filters
<code>fftcoeffs</code>	Single-rate and multirate filters	Frequency-domain coefficients used when filtering with discrete-time and adaptive filter object
<code>grpdelay</code>	All filters	Group delay for filter
<code>help</code>	All filters	Help text for design algorithm in Command Window
<code>impz</code>	All filters	Compute impulse response for filter
<code>isfir</code>	All filters	Determine whether filter is FIR filter
<code>islinphase</code>	All filters	Determine whether filter is linear phase
<code>ismaxphase</code>	All filters	Determine whether filter is maximum phase
<code>isminphase</code>	All filters	Determine whether filter is minimum phase
<code>isreal</code>	All filters	Determine whether filter is real
<code>isstable</code>	All filters	Determine whether filter is stable
<code>limitcycle</code>	Discrete-time filters	Explore steady-state response of single rate, fixed-point IIR filter to zero-valued input

<code>maxstep</code>	Adaptive filter	Maximum step size that allows adaptive filter to converge
<code>measure</code>	Adaptive and discrete-time filter objects	Magnitude response measurement for discrete-time or multirate filter created from filter specification object
<code>msepred</code>	Adaptive filter	Calculate predicted mean-squared error for selected adaptive filter
<code>msesim</code>	Adaptive filter	Calculate measured mean-squared error for adaptive filter
<code>noisepsd</code>	Single-rate filter objects	Compute power spectral density (PSD) of filter output caused by roundoff noise during quantization
<code>noisepsdopts</code>	Single-rate objects	Create object containing options for running output noise power spectral density (PSD) computation <code>noisepsd</code> on filter
<code>norm</code>	All filter objects	P-norm of <code>adaptfilt</code> , <code>dfilt</code> , and <code>mfilt</code> objects
<code>normalize</code>	Discrete-time filters	Normalize filter numerator or feed-forward coefficients to between -1 and 1
<code>normalizefreq</code>	Single-rate and multirate filter specification objects	Normalize filter numerator or feed-forward coefficients to values between -1 and 1
<code>nstates</code>	Single-rate and multirate filter objects	Number of filter states in discrete-time or multirate filter
<code>order</code>	Fixed-point filters	Order of quantized filter
<code>phasedelay</code>	Single-rate and multirate filters	Phase delay of discrete-time or multirate filter

<code>phasez</code>	All filters	Unwrapped phase response for filter
<code>polyphase</code>	Multirate filter	Polyphase decomposition of multirate filter
<code>qreport</code>	All fixed-point filters	Results of most recent fixed-point filtering operation
<code>realizemdl</code>	Fixed-point filters	Realize Simulink subsystem block for quantized filter
<code>reffilter</code>	Discrete-time filters	Double-precision floating-point reference filter that corresponds to fixed-point or single-precision floating-point filter
<code>reorder</code>	SOS discrete-time filters	Rearrange sections in second-order sections (SOS) filter
<code>reset</code>	Adaptive and Multirate filters	Reset filter properties to initial conditions
<code>scale</code>	SOS discrete-time filters	Scale the sections of an SOS filter
<code>scalecheck</code>	SOS discrete-time filters	Check the scaling of an SOS filter
<code>set2int</code>	Single-rate and multirate filters	Configure single-rate and multirate filters for integer filtering
<code>setspecs</code>	<code>fdesign</code> objects	Set specifications for filter specification object
<code>specifyall</code>	Discrete-time filters	Access fixed-point scaling modes and features in direct-form FIR filter object
<code>stepz</code>	Adaptive and Multirate filters	Step response for filter

<code>zerophase</code>	All filters	Return the zerophase response for a filter
<code>zplane</code>	All filters	Return the pole-zero plot for a filter

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

## Fixed-Point Filter Construction and Property Functions

<code>cell2sos</code>	Convert a cell array to a second-order sections matrix
<code>get</code>	Get properties of a quantized filter
<code>isreal</code>	Test if filter coefficients are real
<code>reset</code>	Reset the properties of a quantized filter to their initial values
<code>scale</code>	Scale the sections of second-order section filters
<code>scalecheck</code>	Check the scaling of a second-order sections filter
<code>scalegpts</code>	Create an object that contains scaling options for second-order section scaling
<code>set</code>	Set properties of a quantized filter
<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

## Quantized Filter Analysis Functions

<code>freqz</code>	Compute the frequency response for a quantized filter
<code>impz</code>	Compute the impulse response for a quantized filter
<code>isallpass</code>	Test quantized filters to determine if they are allpass structures
<code>isfir</code>	Test quantized filters to see if they are FIR filters
<code>islinphase</code>	Test quantized filters to see if they are linear phase
<code>ismaxphase</code>	Test quantized filters to see if they are maximum phase filters
<code>isminphase</code>	Test quantized filters to see if they are minimum phase filters
<code>isreal</code>	Test quantized filters for purely real coefficients

<code>issos</code>	Test whether quantized filters are composed of second-order sections
<code>isstable</code>	Test for stability of quantized filters
<code>noisepsd</code>	Compute the power spectral density (PSD) of filter output caused by round-off noise during the quantization process
<code>noisepsdopts</code>	Create an object that contains options for running the output noise PSD computation <code>noisepsd</code> on a filter
<code>zplane</code>	Compute a pole-zero plot for a quantized filter



## SOS Conversion Functions

<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 Functions

<code>farrow</code>	Implement Farrow filter
<code>firband</code>	Perform constrained-band equiripple FIR filter design
<code>firfilt</code>	Design equiripple FIR interpolators
<code>firceqrip</code>	Design constrained, equiripple FIR filter
<code>firgr</code>	Use Parks-McClellan technique to design digital FIR filter
<code>firhalfband</code>	Design halfband FIR filter
<code>firlpnm</code>	Least P-norm optimal FIR filter design
<code>firminphase</code>	Compute minimum-phase FIR spectral factor
<code>firnyquist</code>	Design lowpass Nyquist (Lth-band) FIR filter
<code>ifir</code>	Design interpolated FIR filters
<code>iircomb</code>	Design comb IIR filters with periodic frequency response
<code>iirgrpdelay</code>	Design least-pth norm IIR filters with given group delay
<code>iirlpnm</code>	Design least-pth norm IIR filters
<code>iirlpnmc</code>	Design constrained least-pth norm IIR filters
<code>iirnotch</code>	Design notch IIR filters to attenuate a fixed frequency
<code>iirpeak</code>	Design peaking IIR filters for boosting or cutting specific frequencies

## Filter Conversion Functions

<code>ca2tf</code>	Convert coupled allpass filters to transfer function form
<code>cl2tf</code>	Convert lattice coupled allpass filters to transfer function form
<code>convert</code>	Convert <code>dfilt</code> objects from one structure to another
<code>firlp2lp</code>	Transform lowpass FIR filters to lowpass filters with different passband specifications
<code>firlp2hp</code>	Transform lowpass FIR filters to highpass FIR filters
<code>iirlp2bp</code>	Transform lowpass IIR filters to bandpass filters
<code>iirlp2bs</code>	Transform lowpass IIR filters to bandstop filters
<code>iirlp2hp</code>	Transform lowpass IIR filters to highpass filters
<code>iirlp2lp</code>	Transform lowpass IIR filters to lowpass filters
<code>iirpowcomp</code>	Compute the power complementary IIR filter
<code>set2int</code>	Scale the real filter coefficients to integer values for discrete-time and multirate filter objects
<code>tf2ca</code>	Convert transfer function form to coupled allpass form
<code>tf2cl</code>	Convert transfer function form to lattice coupled allpass form

## Functions — Alphabetical List

This following pages provide the reference information for each function in the toolbox, in alphabetical order by the name of the function.

# adaptfilt

---

**Purpose** Construct adaptive filter object

**Syntax** `ha = adaptfilt.algorithm(input1,input2, )`

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

## Algorithms

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

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

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

<b>adaptfilt.algorithm String</b>	<b>Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</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>Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</b>
<code>adaptfilt.ftf</code>	Use the fast transversal least squares adaptation algorithm
<code>adaptfilt.qdrls</code>	Use the QR-decomposition RLS adaptation algorithm
<code>adaptfilt.hrsls</code>	Use the householder RLS adaptation algorithm
<code>adaptfilt.hswrls</code>	Use the householder SWRLS adaptation algorithm
<code>adaptfilt.rls</code>	Use the recursive-least squares (RLS) adaptation algorithm
<code>adaptfilt.swrls</code>	Use the sliding window (SW) RLS adaptation algorithm
<code>adaptfilt.swftf</code>	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>Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</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>Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</b>
<code>adaptfilt.fdaf</code>	Use the frequency domain adaptation algorithm
<code>adaptfilt.pbfdaf</code>	Use the partition block version of the FDAF algorithm
<code>adaptfilt.pbufdaf</code>	Use the partition block unconstrained version of the FDAF algorithm
<code>adaptfilt.tdafdct</code>	Use the transform domain adaptation algorithm using DCT
<code>adaptfilt.tdafdft</code>	Use the transform domain adaptation algorithm using DFT
<code>adaptfilt.ufdaf</code>	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>Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation</b>
<code>adaptfilt.gal</code>	Use the gradient adaptive lattice filter adaptation algorithm
<code>adaptfilt.lsl</code>	Use the least squares lattice adaptation algorithm
<code>adaptfilt.qrdsl</code>	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 the Signal Processing Toolbox, 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.

<b>Method</b>	<b>Description</b>
adaptfilt/coefficients	Return the instantaneous adaptive filter coefficients
adaptfilt/filter	Apply an adaptfilt object to your signal
adaptfilt/freqz	Plot the instantaneous adaptive filter frequency response
adaptfilt/grpdelay	Plot the instantaneous adaptive filter group delay
adaptfilt/impz	Plot the instantaneous adaptive filter impulse response.
adaptfilt/info	Return the adaptive filter information.
adaptfilt/isfir	Test whether an adaptive filter is a finite impulse response (FIR) filter.
adaptfilt/islinphase	Test whether an adaptive filter is linear phase
adaptfilt/ismaxphase	Test whether an adaptive filter is maximum phase
adaptfilt/isminphase	Test whether an adaptive filter is minimum phase
adaptfilt/isreal	True whether an adaptive filter has real coefficients
adaptfilt/isstable	Test whether an adaptive filter is stable

<b>Method</b>	<b>Description</b>
<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.
<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.

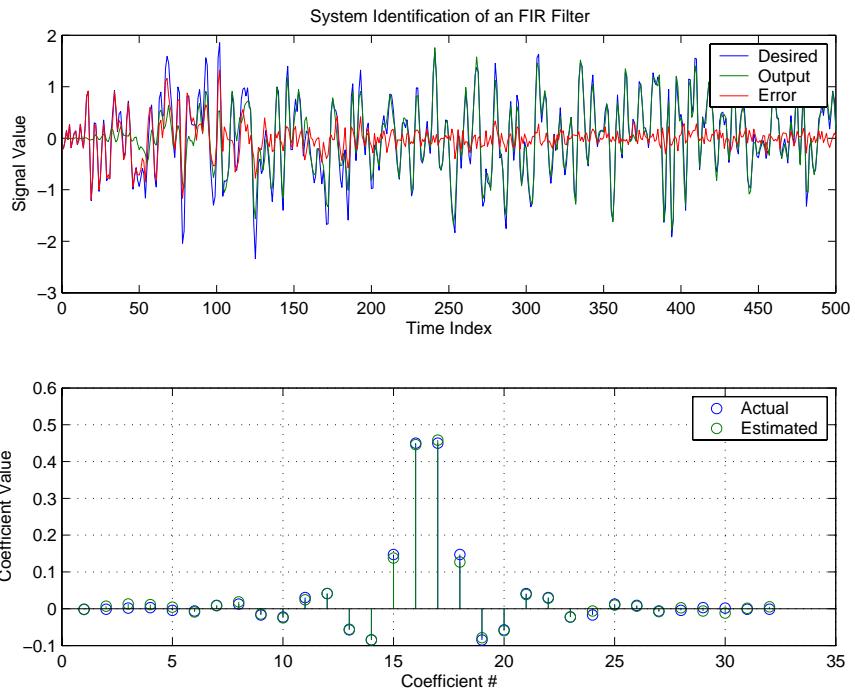
## Examples

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`

<b>Purpose</b>	Adjoint least mean square (LMS) FIR adaptive filter that adapts using adjoint LMS algorithm
<b>Syntax</b>	<pre>ha = adaptfilt.adjlims(l,step,leakage,pathcoeffs,pathest, errstates,pstates,coeffs,states)</pre>
<b>Description</b>	<p><code>ha = adaptfilt.adjlims(l,step,leakage,pathcoeffs,pathest, errstates,pstates,coeffs,states)</code> constructs object <code>ha</code>, an FIR adjoint LMS adaptive filter. <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 when you omit the argument. <code>step</code> is the adjoint LMS step size. It must be a nonnegative scalar. When you omit the <code>step</code> argument, <code>step</code> defaults to 0.1.</p> <p><code>leakage</code> is the adjoint LMS leakage factor. It must be a scalar between 0 and 1. When <code>leakage</code> is less than one, you implement a leaky version of the <code>adjlims</code> algorithm to determine the filter coefficients. <code>leakage</code> defaults to 1 specifying no leakage in the algorithm.</p> <p><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.</p> <p><code>pathest</code> is the estimate of the secondary path filter model. <code>pathest</code> defaults to the values in <code>pathcoeffs</code>.</p> <p><code>errstates</code> 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. <code>errstates</code> defaults to a vector of zeros of appropriate length. <code>pstates</code> contains the secondary path FIR filter states. It must be a vector of length equal to the filter order of the secondary path model. <code>pstates</code> defaults to a vector of zeros of appropriate length. The initial filter coefficients for the secondary path filter compose vector <code>coeffs</code>. It must be a length <code>l</code> vector. <code>coeffs</code> defaults to a length <code>l</code> vector of zeros. <code>states</code> is a vector containing the initial filter states. It must be a vector of length <code>l+ne-1</code>, where <code>ne</code> is the length of <code>errstates</code>. When you omit <code>states</code>, it defaults to an appropriate length vector of zeros.</p>

# adaptfilt.adjlms

## Properties

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

Property	Default Value	Description
Algorithm	None	Specifies the adaptive filter algorithm the object uses during adaptation
Coefficients	Length $l$ vector with zeros for all elements	Adjoint LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need $l$ entries in coefficients. Updated filter coefficients are returned in coefficients when you use <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



<b>Property</b>	<b>Default Value</b>	<b>Description</b>
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
SecondaryPathCoeffs	No default	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor
SecondaryPathEstimate	pathcoeffs values	An estimate of the secondary path filter model
SecondaryPathStates	Length of the secondary path filter. All elements are zeros.	The states of the secondary path filter, the unknown system

## adaptfilt.adj1ms

---

Property	Default Value	Description
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.adj1ms</code> in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.

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

### Example

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

```

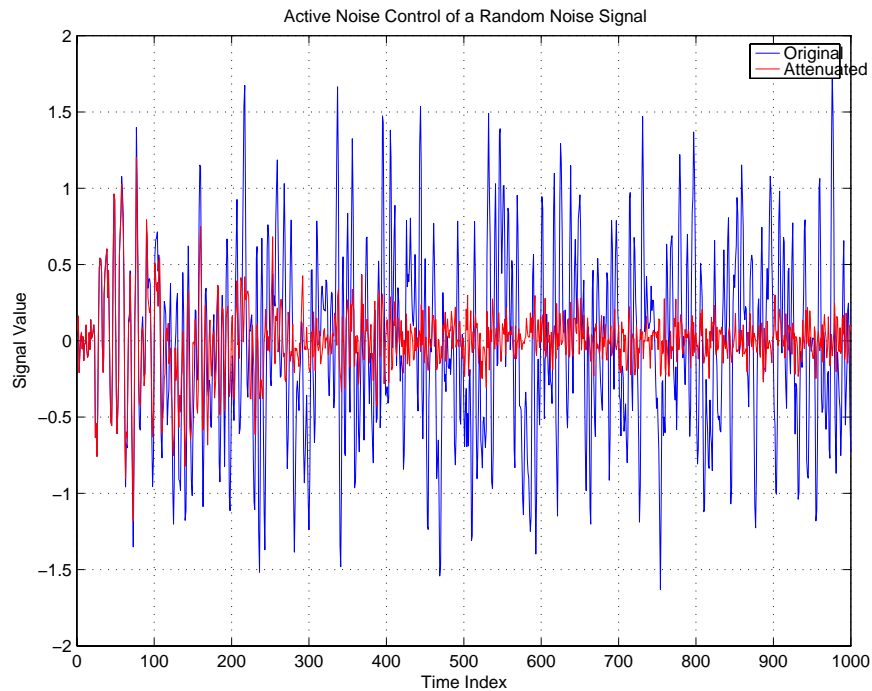
x = randn(1,1000);      % Noise source
g = fir1(47,0.4);      % FIR primary path system model
n = 0.1*randn(1,1000); % Observation noise signal
d = filter(g,1,x)+n;   % Signal to be canceled (desired)
b = fir1(31,0.5);     % FIR secondary path system model
mu = 0.008;           % Adjoint LMS step size

```

# adaptfilt.adjilms

```
ha = adaptfilt.adjilms(32,mu,1,b);  
[y,e] = filter(ha,x,d);  
plot(1:1000,d,'b',1:1000,e,'r');  
title('Active Noise Control of a Random Noise Signal');  
legend('Original','Attenuated');  
xlabel('Time Index'); ylabel('Signal Value'); grid on;
```

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



## See Also

`adaptfilt.dlms`, `adaptfilt.filtxllms`

## 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** Construct affine projection FIR adaptive filter object 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`.

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>	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.

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

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

<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
% QPSK signal
n = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D)); % Noise signal
r = filter(b,a,s)+n; % Received signal
x = r(1+D:ntr+D); % Input signal (received signal)
d = s(1:ntr); % Desired signal (delayed QPSK signal)
mu = 0.1; % Step size
po = 4; % Projection order
offset = 0.05; % Offset for covariance matrix
ha = adaptfilt.ap(32,mu,po,offset);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
```

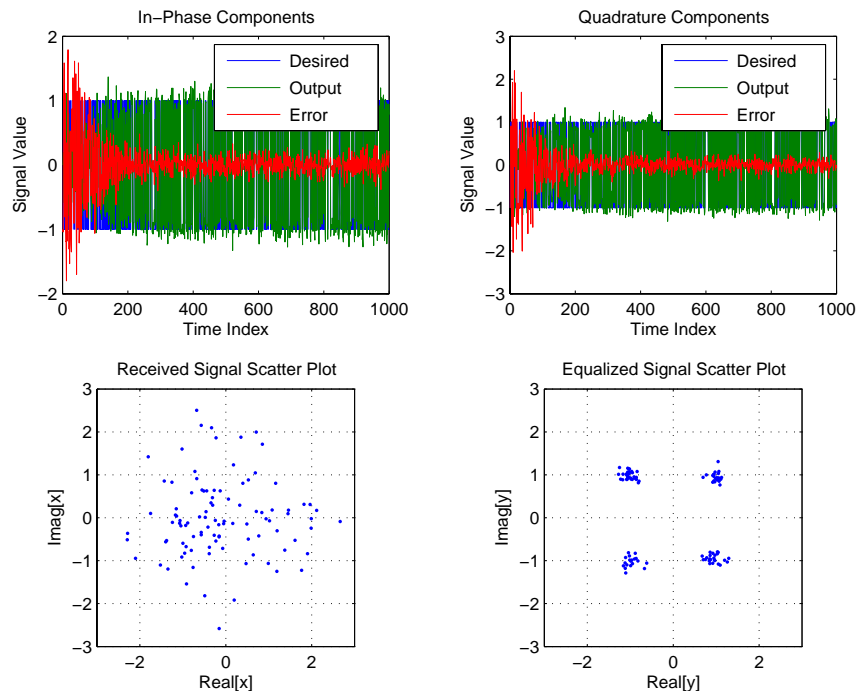


```

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

```

The four plots shown reveal the QPSK process at work.



# adaptfilt.ap

---

## See also

mssim

## References

- [1] K. Ozeki 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] Y. Maruyama, "A Fast Method of Projection Algorithm," *Proc. 1990 IEICE Spring Conf.*, B-744

**Purpose** Affine projection FIR adaptive filter object that uses recursive matrix updating

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

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

### Input Arguments

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

Input Argument	Description
<code>l</code>	Adaptive filter length (the number of coefficients or taps). It must be a positive integer. <code>l</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.

# adaptfilt.apru

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

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

<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.

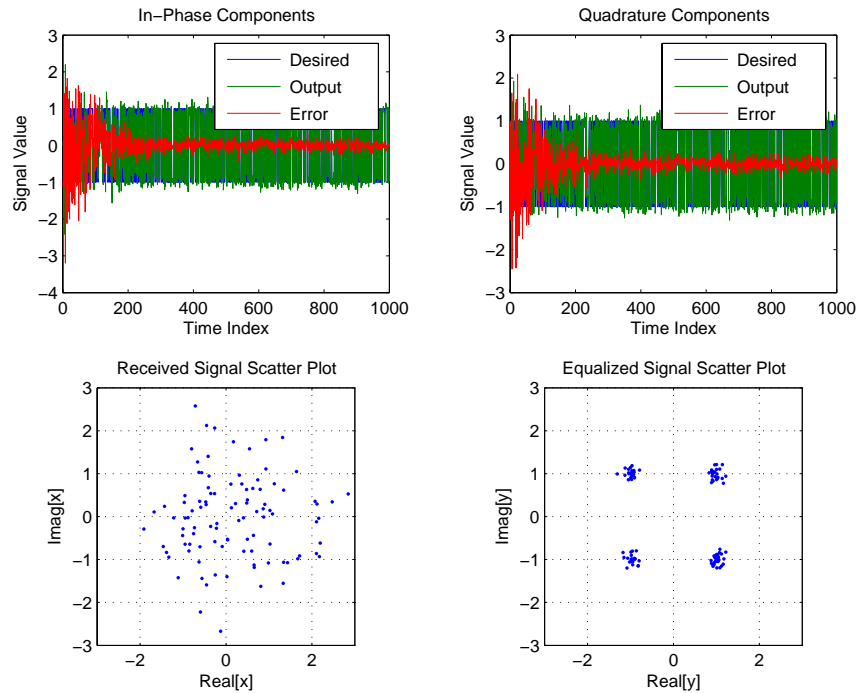
## Example

Demonstrate quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. In this example we run the adaptation for 1000 iterations.

```
D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1000; % Number of iterations
s = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
% QPSK sig
n = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D)); % Noise signal
r = filter(b,a,s)+n; % Received signal
x = r(1+D:ntr+D); % Input signal (received signal)
d = s(1:ntr); % Desired signal (delayed QPSK signal)
mu = 0.1; % Step size
po = 4; % Projection order
del = 0.05; % Offset
ha = adaptfilt.apru(32,mu,po,offset);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
```

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

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



## See Also

adaptfilt, adaptfilt.ap, adaptfilt.bap

## References

- [1] K. Ozeki, Omeda, 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] Y. Maruyama, "A Fast Method of Projection Algorithm," *Proceedings 1990 IEICE Spring Conference*, B-744



**Purpose** Block affine projection FIR adaptive filter object

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

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

### Input Arguments

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

Input Argument	Description
<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>	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.

# adaptfilt.bap

Input Argument	Description
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 $(1 + \text{projectord} - 2)$ .

## Properties

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

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

Name	Range	Description
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length 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.

```

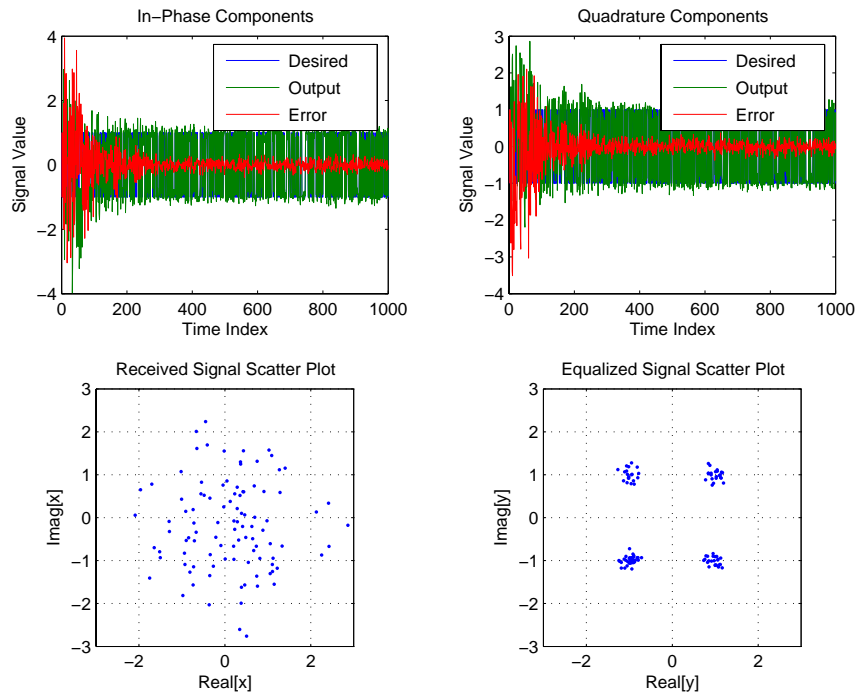
D = 16;                                % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1];              % Numerator coefficients of
                                        % channel

```

# adaptfilt.bap

---

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



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

### See Also

adaptfilt, adaptfilt.ap, adaptfilt.apru

### References

- [1] K. Ozeki, Omeda, 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] M. Montazeri, M, Duhamel, P, "A Set of Algorithms Linking NLMS and Block RLS Algorithms," *IEEE Transactions Signal Processing*, vol. 43, no. 2, pp, 444-453, February 1995

# adaptfilt.blms

---

**Purpose** Construct Block LMS (BLMS) FIR adaptive filter

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

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

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

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

`blocklen` is the block length used. It must be a positive integer and the signal vectors `d` and `x` should be divisible by `blocklen`. Larger block lengths result in faster per-sample execution times but with poor adaptation characteristics. When you choose `blocklen` such that `blocklen + length(coeffs)` is a power of 2, use `adaptfilt.blmsfft`. `blocklen` defaults to 1.

`coeffs` is a vector of initial filter coefficients. it must be a length 1 vector. `coeffs` defaults to length 1 vector of zeros.

`states` contains a vector of your initial filter states. It must be a length 1 vector and defaults to a length 1 vector of zeros when you do not include it in your calling function.

## Properties

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

Property	Default Value	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length 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

# adaptfilt.blms

---

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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 maxstep to determine the maximum usable step size.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

## Example

Use an adaptive filter to identify an unknown 32nd-order FIR filter. In this example we input 500 samples to 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.

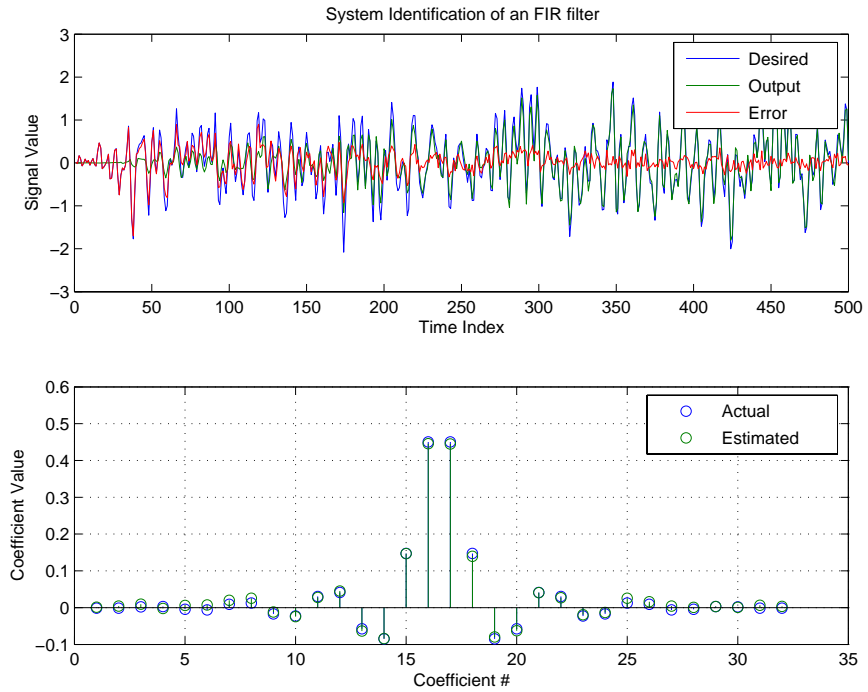
```
x = randn(1,500);           % Input to the filter
```

# adaptfilt.blms

---

```
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

**Reference**

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

# adaptfilt.blmsfft

---

**Purpose** Construct FFT-based block LMS FIR adaptive filter

**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.

Property	Default Value	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length 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

# adaptfilt.blmsfft

---

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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 maxstep to determine the maximum usable step size.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

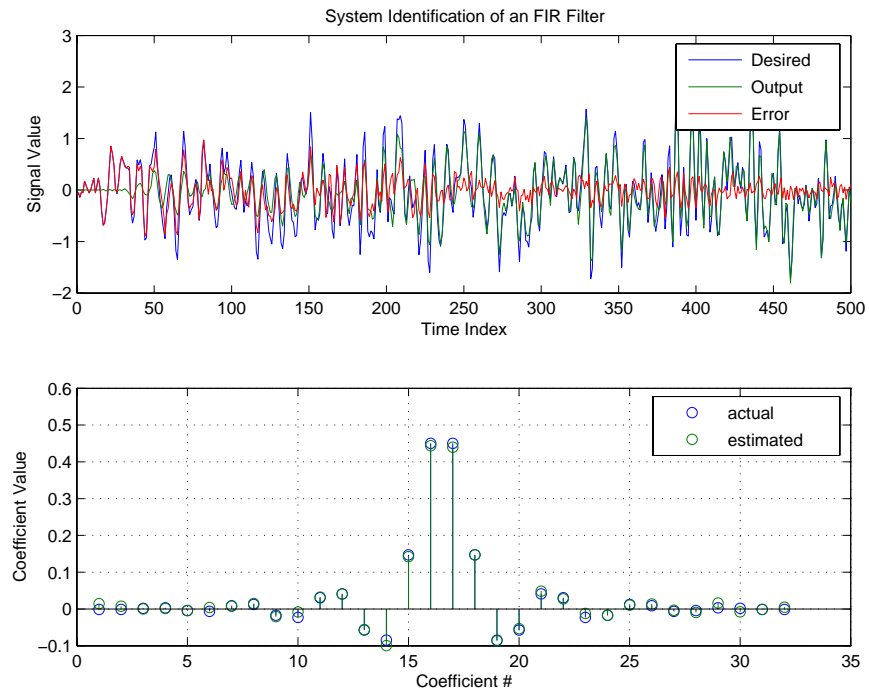
## Example

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

```
x = randn(1,512);      % Input to the filter
b = fir1(31,0.5);     % FIR system to be identified
no = 0.1*randn(1,512); % Observation noise signal
```

# adaptfilt.blmsfft

```
d = filter(b,1,x)+no;    % Desired signal
mu = 0.008;            % Step size
n = 16;                % Block length
ha = adaptfilt.blmsfft(32,mu,1,n);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d(1:500);y(1:500);e(1:500)]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.';ha.coefficients.']);
legend('actual','estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value'); grid on;
```





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

**See Also**

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

**Reference**

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

# adaptfilt.dlms

---

**Purpose** Create delayed LMS FIR adaptive filter object

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

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

## Input Arguments

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

Input Argument	Description
<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.
<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.

<b>Input Argument</b>	<b>Description</b>
<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.
<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.

## Properties

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

Property	Default Value	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length <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. LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need <code>1</code> entries in <code>coeffs</code> .
Delay	1	Specifies the update delay for the adaptive algorithm.
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.

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.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

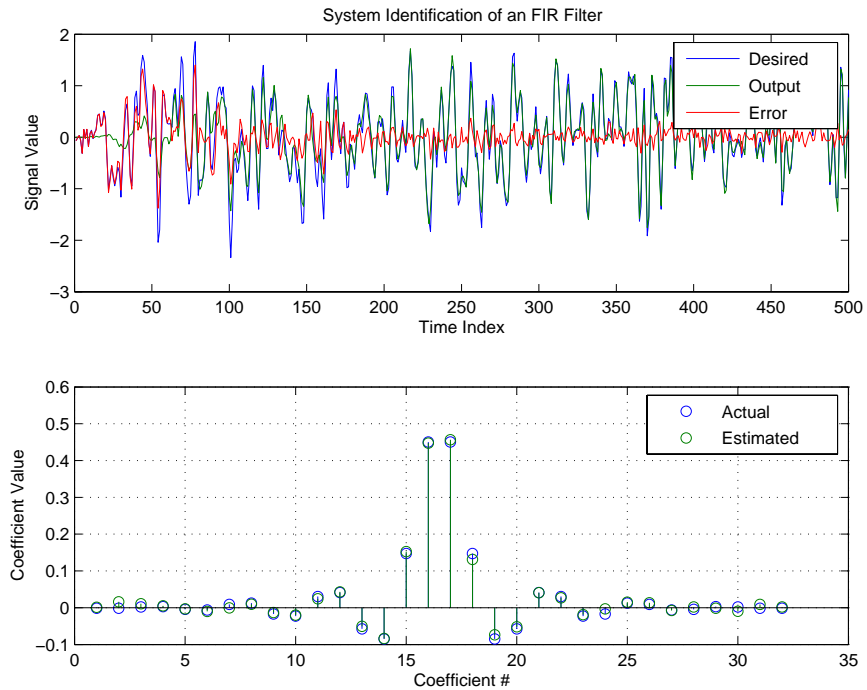
Property	Default Value	Description
StepSize	0.1	Sets the LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .

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

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



**See Also**

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

**Reference**

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

# adaptfilt.fdaf

---

**Purpose** Construct frequency-domain FIR adaptive filter 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.



Input Argument	Description
<code>blocklen</code>	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.
<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

# adaptfilt.fdaf

---

that apply, this table lists and describes each property for the `adaptfilt.fdaf` filter object.

<b>Name</b>	<b>Range</b>	<b>Description</b>
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, <code>(blocklen + 1)</code> 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.

Name	Range	Description
Leakage		Leakage parameter of the adaptive filter. if this parameter is set to a value between zero and one, you implement a leaky FDAF algorithm. leakage defaults to 1—no leakage provided in the algorithm.
Offset	Any positive real value	Offset for the normalization terms in the coefficient updates. Use this to avoid dividing by zero or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
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.

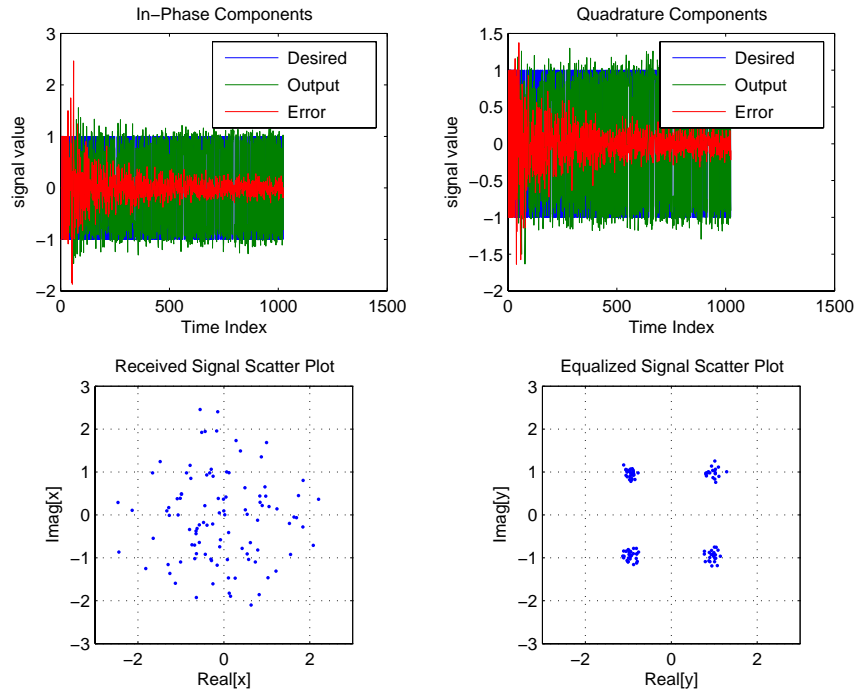
Name	Range	Description
Power		A vector of 2*1 elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates

## Examples

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

```
D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1024; % Number of iterations
s = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
% QPSK signal
n = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D)); % Noise signal
r = filter(b,a,s)+n; % Received signal
x = r(1+D:ntr+D); % Input signal (received signal)
d = s(1:ntr); % Desired signal (delayed QPSK
% signal)
del = 1; % Initial FFT input powers
mu = 0.1; % Step size
lam = 0.9; % Averaging factor
ha = adaptfilt.fdaf(32,mu,1,del,lam);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
```

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



**See Also**

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

**Reference**

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

# adaptfilt.filtxLms

---

**Purpose** Create filtered-x LMS FIR adaptive filter

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

**Description** `ha = adaptfilt.filtxLms(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.filtxLms`.

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	Filtered LMS step size. it must be a nonnegative scalar. step defaults to 0.1.
leakage	is the filtered-x LMS leakage factor. it must be a scalar between 0 and 1. If it is less than one, a leaky version of <code>adaptfilt.filtxLms</code> is implemented. leakage defaults to 1 (no leakage).
pathcoeffs	is the secondary path filter model. this vector should contain the coefficient values of the secondary path from the output actuator to the error sensor.
pathest	is the estimate of the secondary path filter model. pathest defaults to the values in pathcoeffs.
fstates	is a vector of filtered input states of the adaptive filter. fstates defaults to a zero vector of length equal to $(1 - 1)$ .

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.

# adaptfilt.filtx1ms

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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. states defaults to a vector of zeros which has length equal to (1 + projectord - 2)
SecondaryPathCoeffs	No default	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor
SecondaryPathEstimate	pathcoeffs values	An estimate of the secondary path filter model



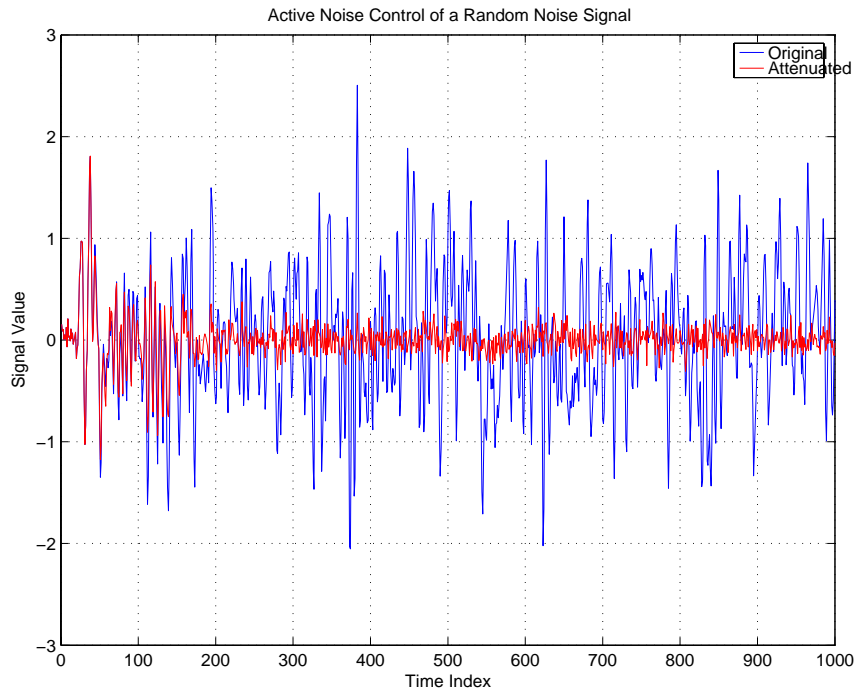
Property	Default Value	Description
SecondaryPathStates	Vector of size (length(pathcoeffs)-1) with all elements equal to zero.	The states of the secondary path FIR filter—the unknown system
StepSize	0.1	Sets the filtered-x algorithm step size used for each iteration of the adapting algorithm. Determines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

## Example

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

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

```
x = randn(1,1000);      % Noise source
g = fir1(47,0.4);      % FIR primary path system model
n = 0.1*randn(1,1000); % Observation noise signal
d = filter(g,1,x)+n;   % Signal to be cancelled (desired)
b = fir1(31,0.5);      % FIR secondary path system model
mu = 0.008;            % Filtered-X LMS step size
ha = adaptfilt.filtxllms(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`

## Reference

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

**Purpose** Construct fast transversal least squares adaptive filter object

**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.

<b>Name</b>	<b>Range</b>	<b>Description</b>
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
BkwdPrediction		Returns the predicted samples generated during adaptation. Refer to [12] 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 <code>(1-0.5/l, 1]</code> . <code>lambda</code> defaults to 1.

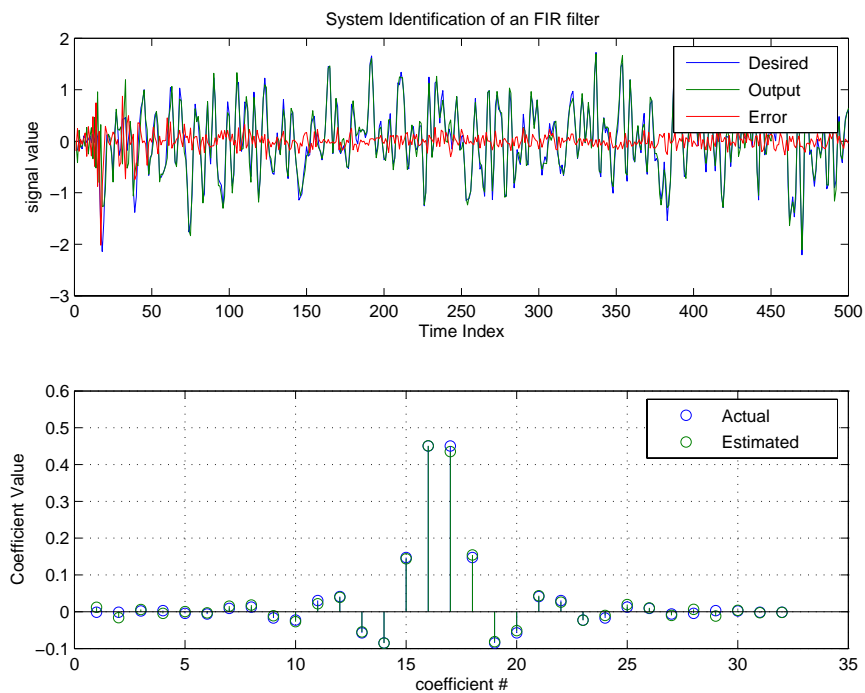
Name	Range	Description
FwdPrediction		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
InitFactor		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. delta defaults to one.
KalmanGain		Empty when you construct the object, this gets populated after you run the filter.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .

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

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



**See Also**

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

**Reference**

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** Construct gradient adaptive lattice FIR filter

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

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

## Input Arguments

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

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

## Properties

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

apply, this table lists and describes each property for the affine projection filter object.

<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 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 [12] 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

Name	Range	Description
FwdPredErrorPower		Returns the minimum mean-squared prediction error in the forward direction. Refer to [12] in the bibliography for details about linear prediction.
Leakage	0 to 1	Leakage parameter of the adaptive filter. If this parameter is set to a value between zero and one, you implement a leaky GAL algorithm. leakage defaults to 1—no leakage provided in the algorithm.
Offset		Offset for the normalization terms in the coefficient updates. Use this to avoid dividing by zero or by very small numbers when input signal amplitude becomes very small. offset defaults to one.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

Name	Range	Description
ReflectionCoeffs		Coefficients determined for the reflection portion of the filter during adaptation.
ReflectionCoeffsStep		Size of the steps used to determine the reflection coefficients.
States	Vector of elements	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to $(1 + \text{projectord} - 2)$ .
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.

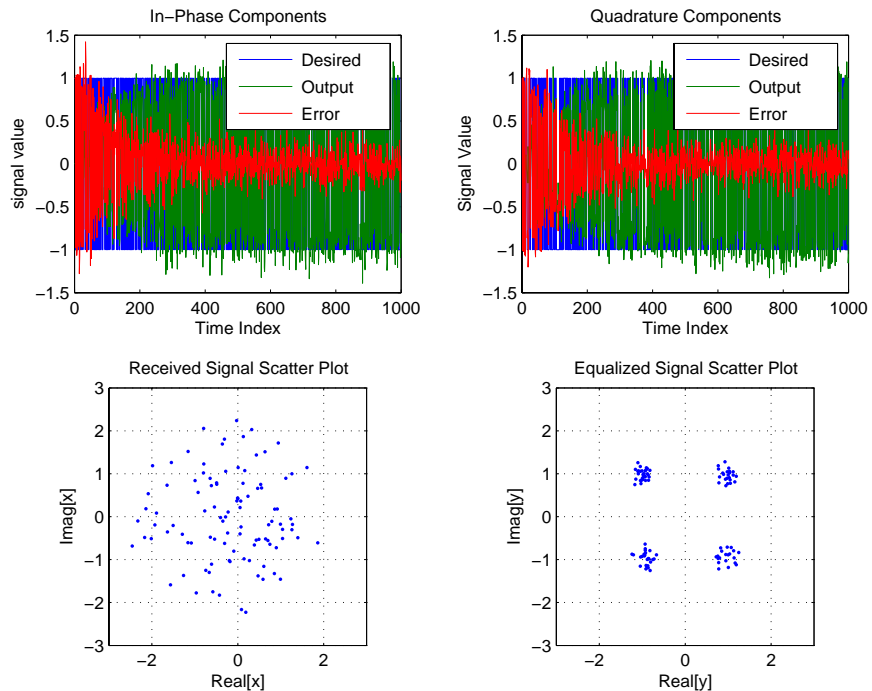
```
D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1000; % Number of iterations
s = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
% QPSK signal
n = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D)); % 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 = 0.007; % 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.qrdls1, adaptfilt.lsl, adaptfilt.tdafdf

## References

L.J. Griffiths, "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

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

**Purpose** Construct a householder recursive least squares (RLS) FIR adaptive filter object

**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).

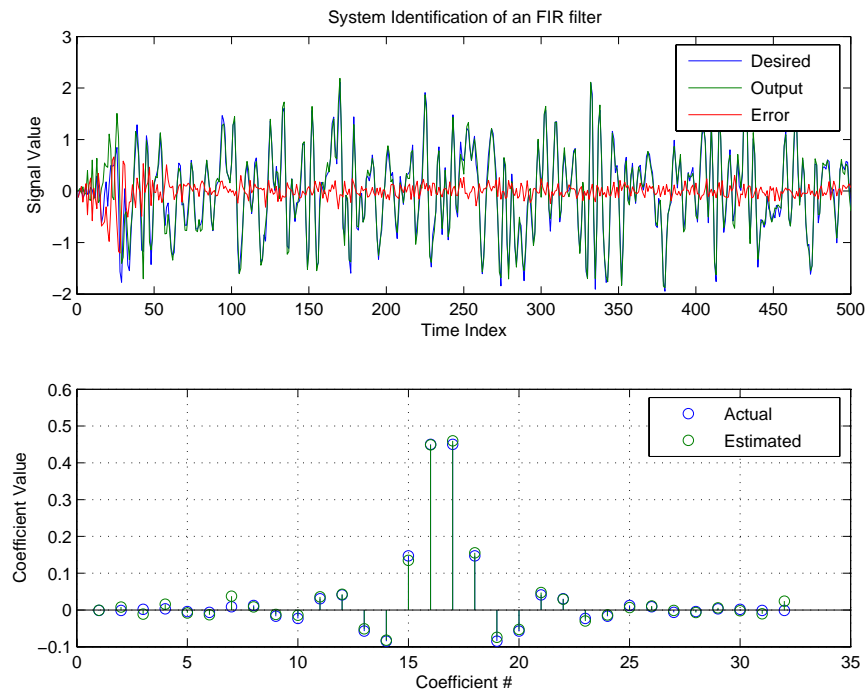
### Examples

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);
```

# adaptfilt.hrls

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



## See Also

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

**Purpose** Construct householder sliding window recursive least squares (RLS) FIR adaptive filter

**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 + swblocklen -2) vector. states defaults to a length (1 + 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 (swblocklen - 1).

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

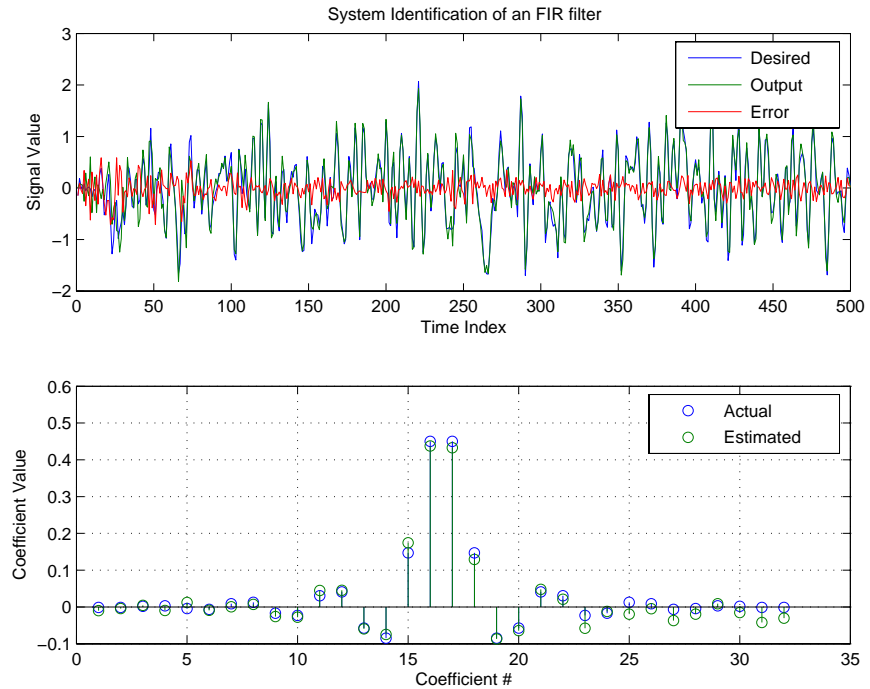
Name	Range	Description
SqrtInvCov	1-by-1 Matrix	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + projectord - 2).
SwBlockLength	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.

## Examples

System Identification of a 32-coefficient FIR filter.

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

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



## See Also

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

# adaptfilt.lms

---

**Purpose** Construct least-mean-square (LMS) FIR adaptive filter object

**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.

<b>Property</b>	<b>Range</b>	<b>Property Description</b>
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 a length 1 vector of zeros when you do not provide the vector as an input argument.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
Leakage	0 to 1	LMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. leakage defaults to 1 (no leakage).

Property	Range	Property Description
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (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 an unknown 32nd-order FIR filter.

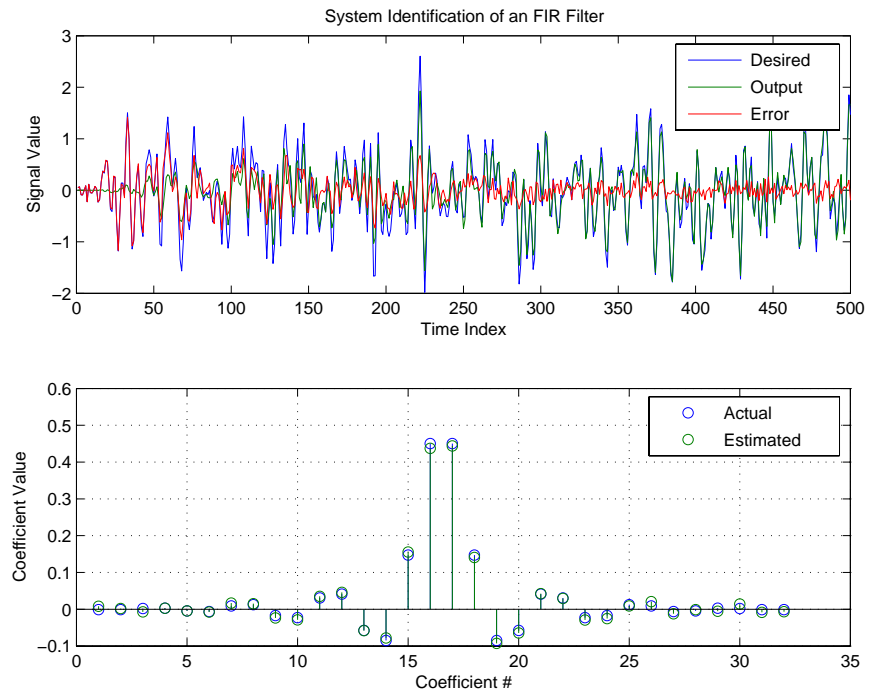
```
x = randn(1,500); % Input to the filter
b = fir1(31,0.5); % FIR system to be identified
n = 0.1*randn(1,500); % Observation noise signal
d = filter(b,1,x)+n; % Desired signal
mu = 0.008; % LMS step size.
ha = adaptfilt.lms(32,mu);
[y,e] = filter(ha,x,d);
```

```

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.



# adaptfilt.lms

---

## See Also

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

## Reference

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

**Purpose** Construct least squares lattice (LSL) adaptive filter

**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 also input

# adaptfilt.lsl

arguments for creating `adaptfilt.lsl` 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
BkwdPrediction		Returns the predicted samples generated during adaptation. Refer to [12] in the bibliography for details about linear prediction.
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		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.

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

## Examples

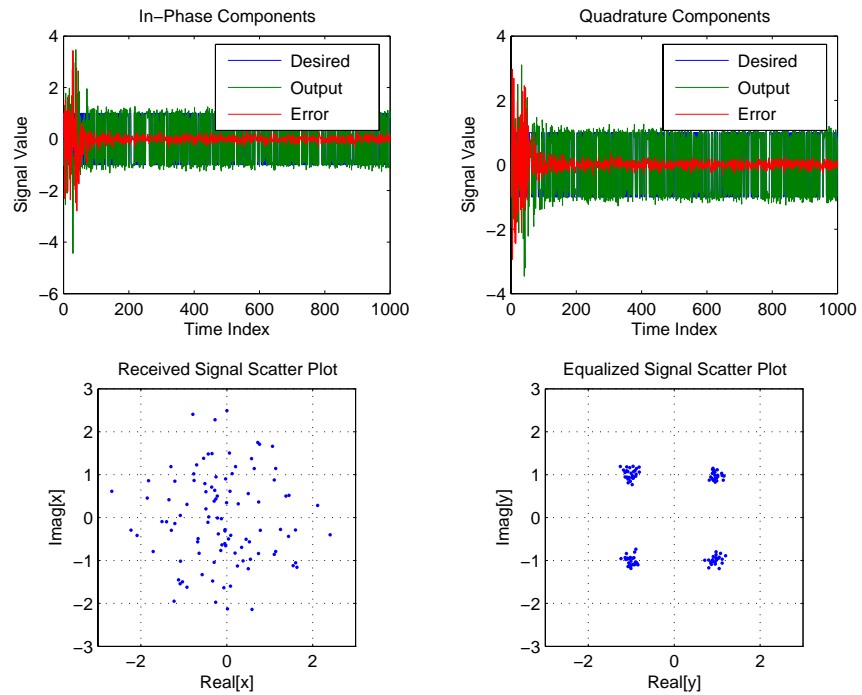
Demonstrate Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter running for 1000 iterations. After you

review the example code, the figure shows the results of running the example to use QPSK adaptive equalization with a 32nd-order FIR filter. Notice that the error between the in-phase and quadrature components, as shown by the errors plotted in the upper plots, falls to near zero. Also, the equalized signal shows the clear quadrature nature.

```
D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1000; % Number of iterations
s = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
% QPSK signal
n = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D)); % Noise signal
r = filter(b,a,s)+n; % Received signal
x = r(1+D:ntr+D); % Input signal (received signal)
d = s(1:ntr); % Desired signal (delayed QPSK
% signal)

lam = 0.995; % Forgetting factor
del = 1; % Soft-constrained initialization
factor
ha = adaptfilt.lsl(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```





**See Also**

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

**References**

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

# adaptfilt.nlms

---

**Purpose** Construct normalized least mean squares (LMS) FIR adaptive filter object

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

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

## Input Arguments

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

Input Argument	Description
<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>	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

# adaptfilt.nlms

---

<b>Property</b>	<b>Range</b>	<b>Property Description</b>
Leakage	0 to 1	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. leakage defaults to 1 (no leakage).
Offset	0 or greater	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors. Use this to avoid dividing by zero (or by very small numbers) when the square of the input data norm becomes very small (when the input signal amplitude becomes very small). When you omit it, offset defaults to zero.

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	NLMS step size. It must be a scalar between zero and one. Setting this step size value to one provides the fastest convergence. step defaults to one.

## Example

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

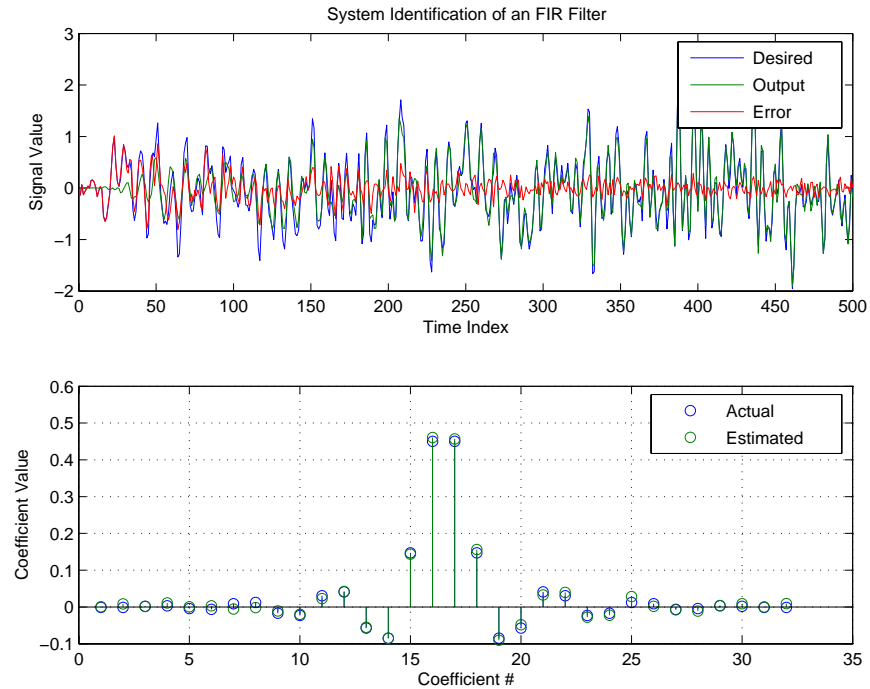
```

x = randn(1,500);    % Input to the filter
b = fir1(31,0.5);   % FIR system to be identified
n = 0.1*randn(1,500); % Observation noise signal
d = filter(b,1,x)+n; % Desired signal
mu = 1;             % NLMS step size

```

# adaptfilt.nlms

```
offset = 50;           % NLMS offset
ha = adaptfilt.nlms(32,mu,1,offset);
[y,e] = filter(ha,x,d);
```



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

## See Also

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

**Purpose** Construct partitioned block frequency-domain (PBFDAF) FIR adaptive filter 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>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.

# adaptfilt.pbfdaf

---

<b>Input Argument</b>	<b>Description</b>
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, blocklen should be a power of two. blocklen defaults to two.
offset	Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. offset defaults to zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a vector of length 1. The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
states	Specifies the filter initial conditions. states defaults to a zero vector of length 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.pbfda` 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 <code>lambda</code> as an input argument.
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, <code>blocklen</code> should be a power of two. <code>blocklen</code> 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 <code>coeffs</code> .
FFTStates		States for the FFT operation.

# adaptfilt.pbfdaf

---

<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, 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.

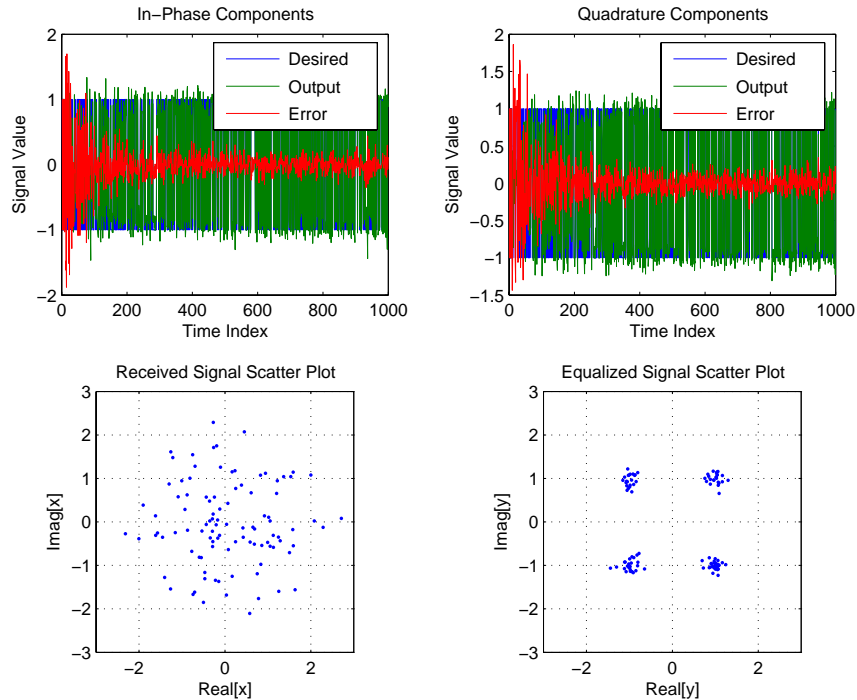
```

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]');
```

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**

J.S. So 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

J.M. Paez Borrillo 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** Construct partitioned block unconstrained frequency-domain (PBUFDAF) FIR adaptive filter 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
<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 <code>l</code> . 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 <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

# adaptfilt.pbufdaf

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.



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.

# adaptfilt.pbufdaf

Name	Range	Description
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.
StepSize	0 to 1	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. step defaults to 1.

## Examples

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

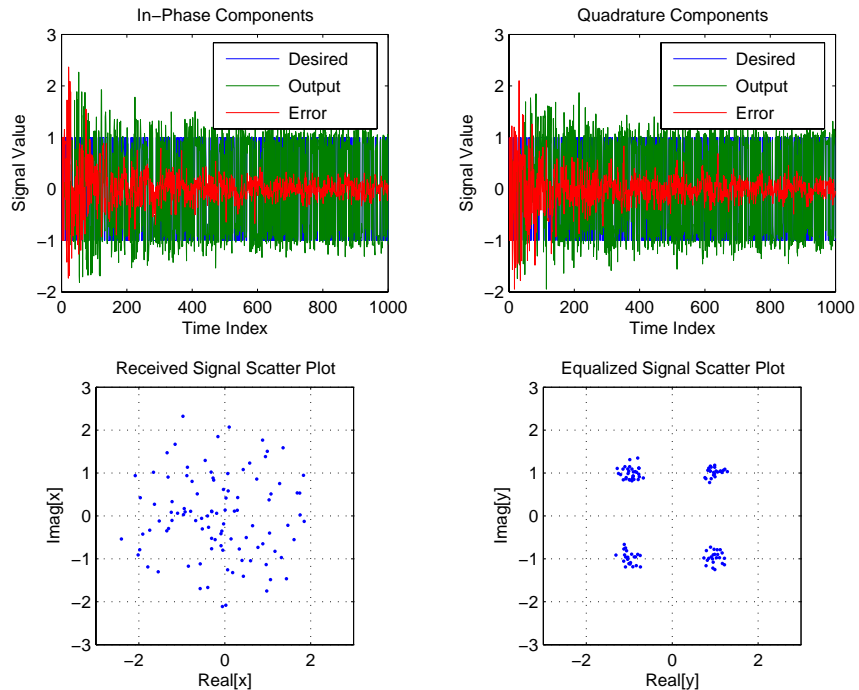
```
D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1000; % Number of iterations
s = sign(randn(1,ntr+D))+j*sign(randn(1,ntr+D)); % Baseband QPSK
% signal
n = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));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

```

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

```

To allow you to compare this algorithm to another, such as the pbfdaf version, we use the same example of QPSK adaptation. The figure shows the results.



## See Also

[adaptfilt.ufdaf](#), [adaptfilt.pbfdaf](#), [adaptfilt.blmsfft](#)

## References

J.S. So 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

J.M. Paez Borrillo 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** QR-decomposition-based least squares lattice (LSL) adaptive filter object

**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>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 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

# adaptfilt.qrdls1

arguments for creating `adaptfilt.qrdls1` 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
BkwdPrediction		Returns the predicted samples generated during adaptation. Refer to [12] 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
ForgettingFactor		Forgetting factor of the adaptive filter. This is a scalar and should lie in the range $(0, 1]$ . It defaults to <code>1</code> . 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 [12] in the bibliography for details about linear prediction.
InitFactor		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. delta defaults to one.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to 1 - 1

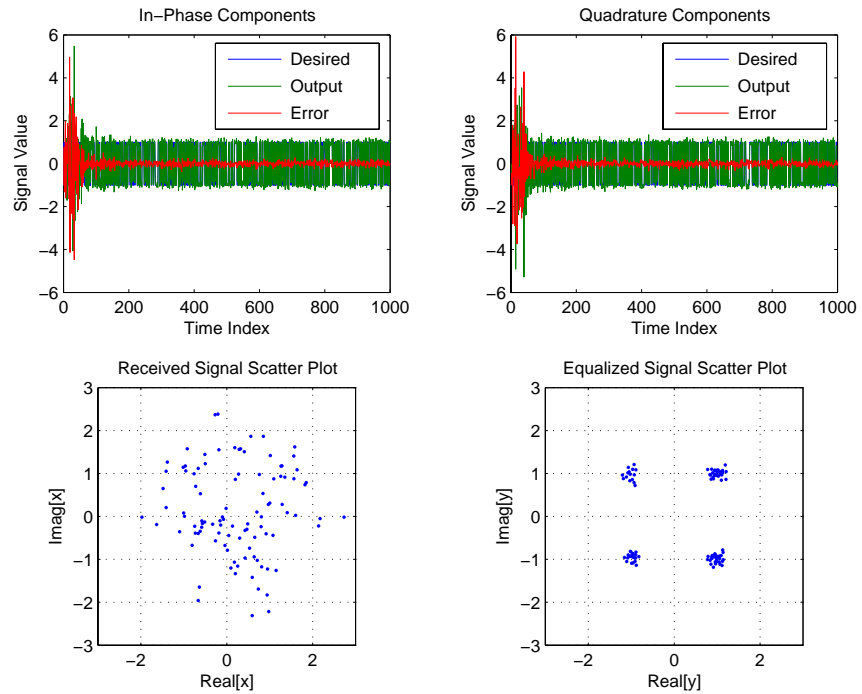
## Examples

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.

# adaptfilt.qrdlsl

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





## See Also

`adaptfilt.qrdls`, `adaptfilt.gal`, `adaptfilt.ftf`, `adaptfilt.lsl`

## References

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

# adaptfilt.qdr1s

---

**Purpose** Create QR-decomposition-based recursive least squares (RLS) FIR adaptive filter object

**Syntax** `ha = adaptfilt.qdr1s(1,lambda,sqrtcov,coeffs,states)`

**Description** `ha = adaptfilt.qdr1s(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.qdr1s`.

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 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>1</code> vector. <code>coeffs</code> defaults to length <code>1</code> vector whose elements are 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.qdr1s` 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.qdr1s` 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
Coefficients	Vector of length <code>l</code>	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	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.

# adaptfilt.qdr1s

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).

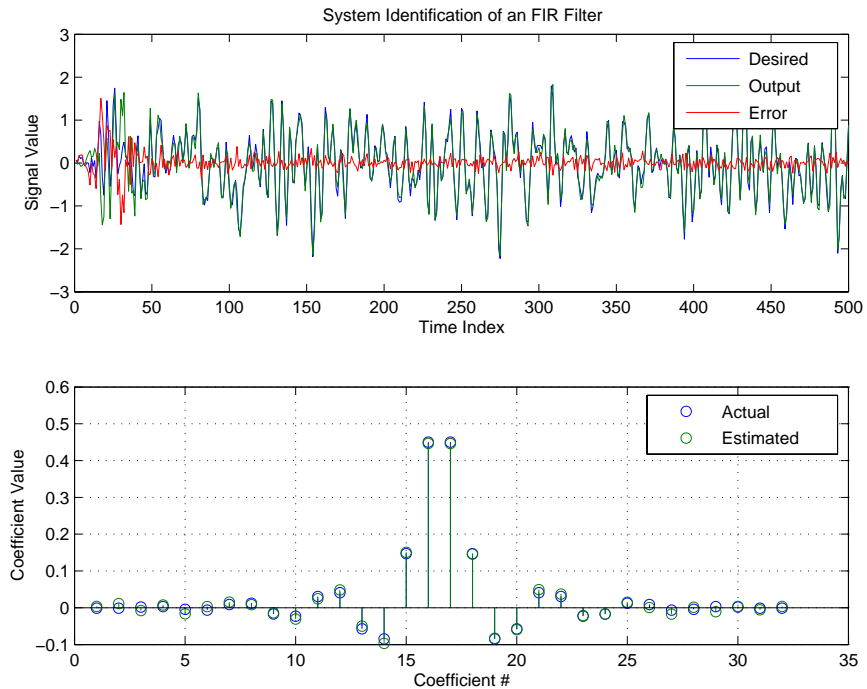
## Examples

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.qdr1s(32,lam,G0);
```

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

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** Construct direct form recursive least squares (RLS) FIR adaptive filter object

**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 <code>l</code> 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. 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.

Name	Range	Description
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.
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 + 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
```

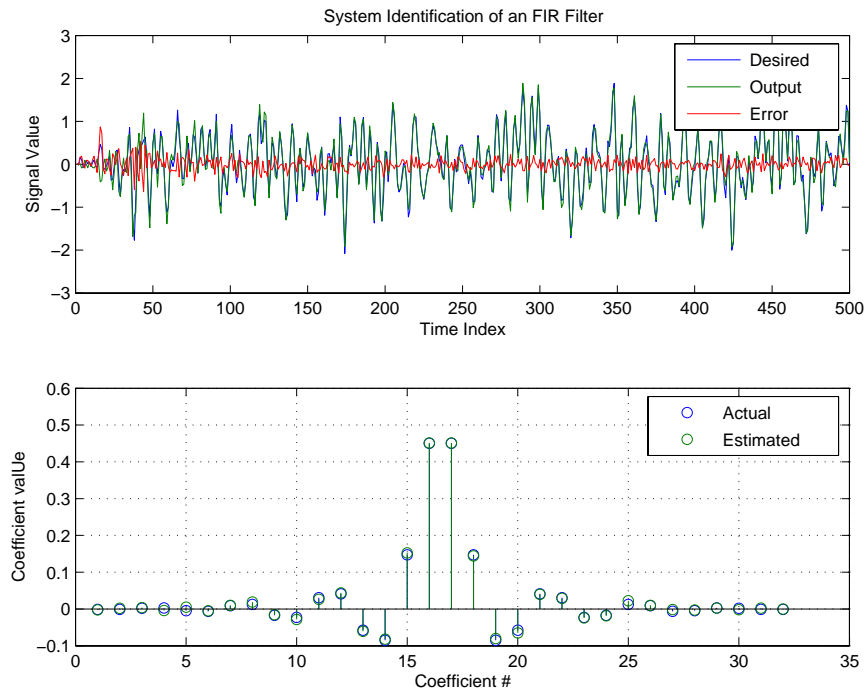


```

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.



# adaptfilt.rls

---

## See Also

`adaptfilt.hrls`, `adaptfilt.hswrls`, `adaptfilt.qdr1s`

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

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

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

### Input Arguments

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

Input Argument	Description
<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
Algorithm	Sign-data	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 <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. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need <code>1</code> entries in coefficients.
FilterLength	10	Reports the length of the filter, the number of coefficients or taps

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.

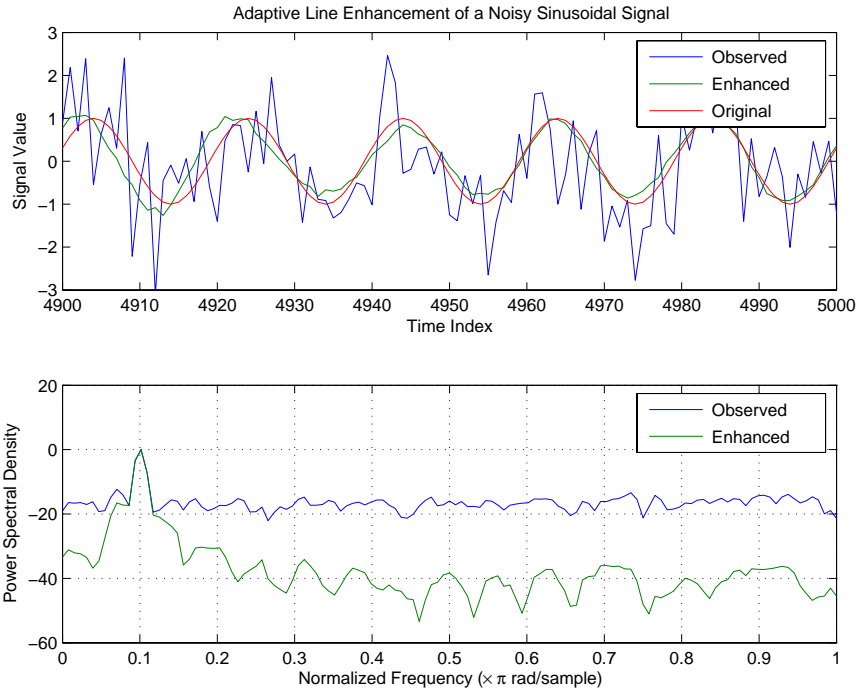
Property	Default Value	Description
States	zeros(1-1,1)	Vector of the adaptive filter states. <code>states</code> defaults to a vector of zeros which has length equal to $(1 - 1)$ .
StepSize	0.1	Sets the SD algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly 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 (delayed desired
% signal)
d = v(1+d:ntr+d)+n(1+d:ntr+d); % Desired signal
mu = 0.0001; % Sign-data step size.
ha = adaptfilt.sd(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:ntr,[d;y;v(1+d:ntr+d)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of a Noisy Sinusoidal Signal');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
[pxx,om] = pwelch(x(ntr-1000:ntr));
pyy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),pyy/max(pyy)]));
```

```
axis([0 1 -60 20]);
legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```



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

**See Also**

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

## References

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

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



**Purpose** Construct sign-error algorithm FIR adaptive filter object

**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
<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>	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
<code>leakage</code>	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.
<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.

**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.

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
Algorithm	Sign-error	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.
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.

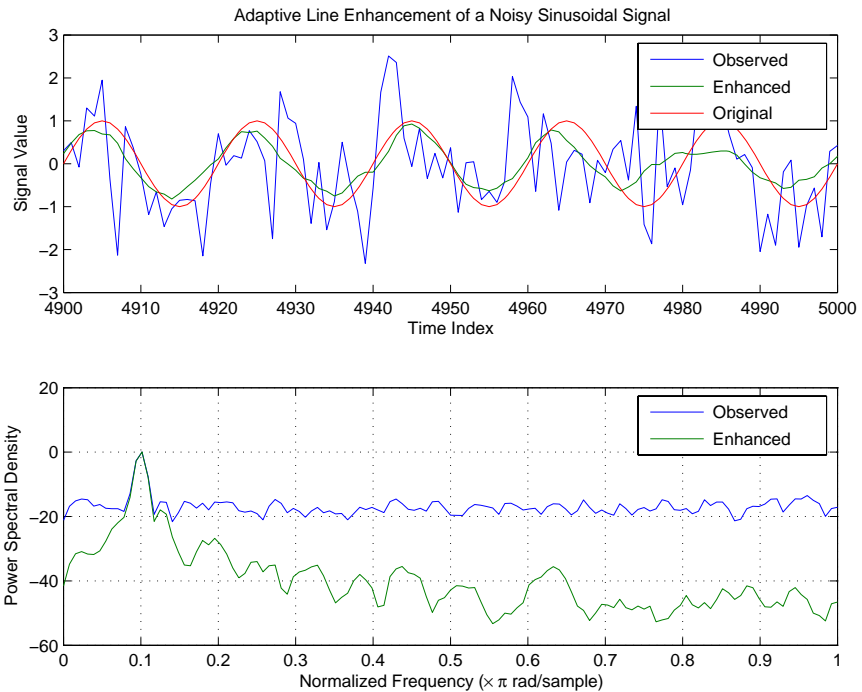
## Examples

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+d:ntr+d)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of a Noisy Sinusoidal Signal');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
[pxx,om] = pwelch(x(ntr-1000:ntr));
pyy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),pyy/max(pyy)]));
axis([0 1 -60 20]);
legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

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** Construct adaptive FIR filter object that uses sign-sign algorithm

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

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

## Input Arguments

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

Input Argument	Description
<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>	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>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.

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

```
for n = 1:length(x)
    ha = adaptfilt.ss(25,0.9);
```

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

## Properties

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

Property	Default Value	Description
Algorithm	Sign-sign	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	zeros(1,1)	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

<b>Property</b>	<b>Default Value</b>	<b>Description</b>
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.



Property	Default Value	Description
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

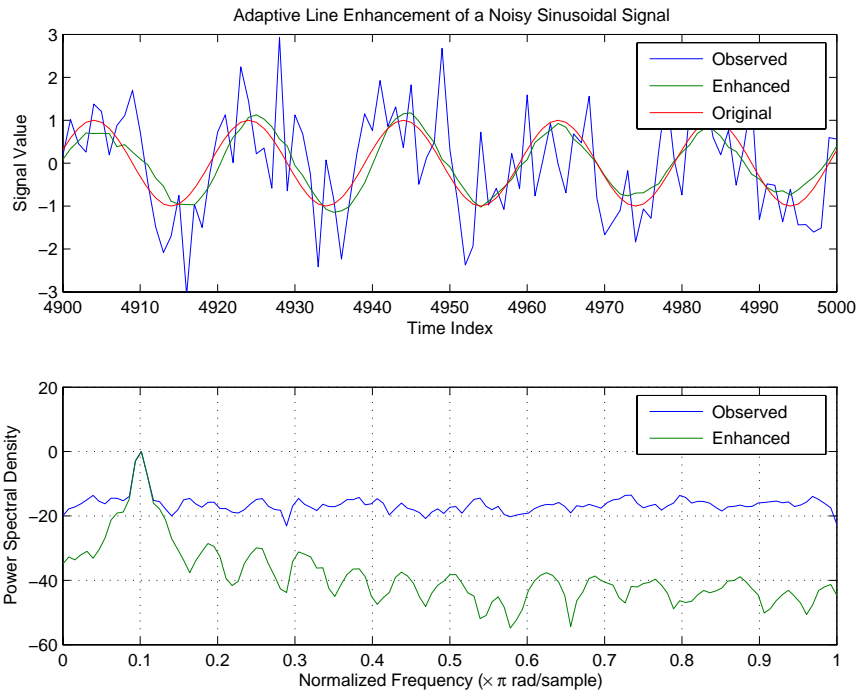
Demonstrating adaptive line enhancement using a 32-coefficient FIR filter provides a good introduction to the sign-sign algorithm.

```

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+d:ntr+d)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of a Noisy Sinusoidal Signal');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
[pxx,om] = pwelch(x(ntr-1000:ntr));
pyy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),pyy/max(pyy)]));
axis([0 1 -60 20]);
legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;

```

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** Construct sliding window fast transversal least squares adaptive filter object

**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>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>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>l</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 $(1-1)$ .

# adaptfilt.swftf

Input Argument	Description
coeffs	Vector of initial filter coefficients. It must be a length 1 vector. coeffs defaults to length 1 vector of all zeros.
states	Vector of initial filter states. states defaults to a zero vector of length equal to $(1 + \text{blocklen} - 2)$ .

## Properties

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

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
BkwdPredictions		Returns the predicted samples generated during adaptation. Refer to [12] 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 1, which is the default value.

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

# adaptfilt.swtf

---

<b>Name</b>	<b>Range</b>	<b>Description</b>
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 $(1 + \text{projectord} - 2)$ .

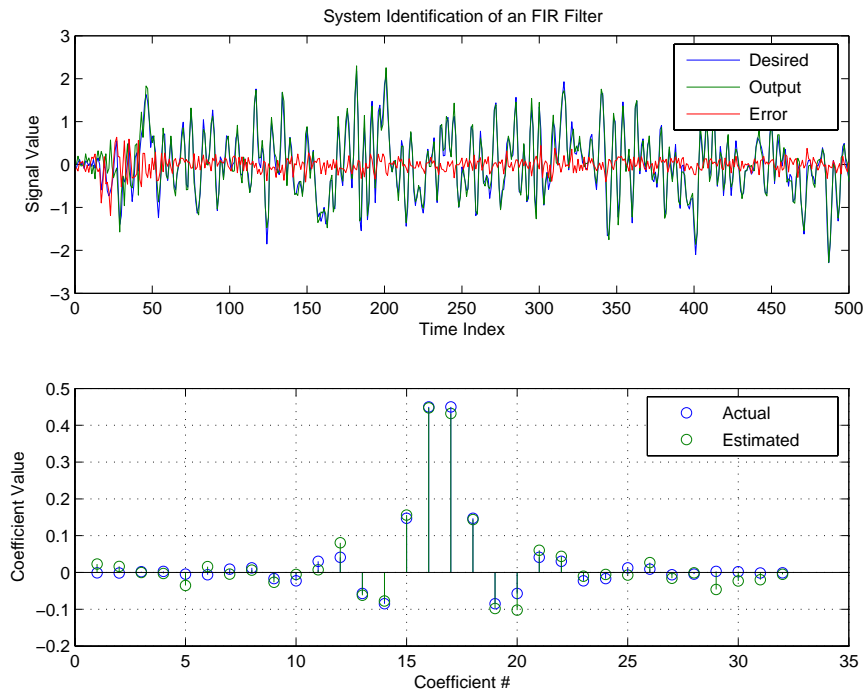
### Examples

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**

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

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

# adaptfilt.swrls

---

**Purpose** Construct sliding window recursive least squares (RLS) FIR adaptive filter

**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 $(swblocklen - 1)$ .

<b>Input Argument</b>	<b>Description</b>
coeffs	Vector of initial filter coefficients. It must be a length 1 vector. coeffs defaults to length 1 vector of all zeros.
states	Vector of initial filter states. states defaults to a zero vector of length equal to $(1 + \text{swblocklen} - 2)$ .

## 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.

<b>Name</b>	<b>Range</b>	<b>Description</b>
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Any vector of 1 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 $(\text{swblocklen} - 1)$ .
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

# adaptfilt.swrls

---

<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 (1,1)	Empty when you construct the object, this gets populated after you run the filter.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. Defaults to false.

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

## Examples

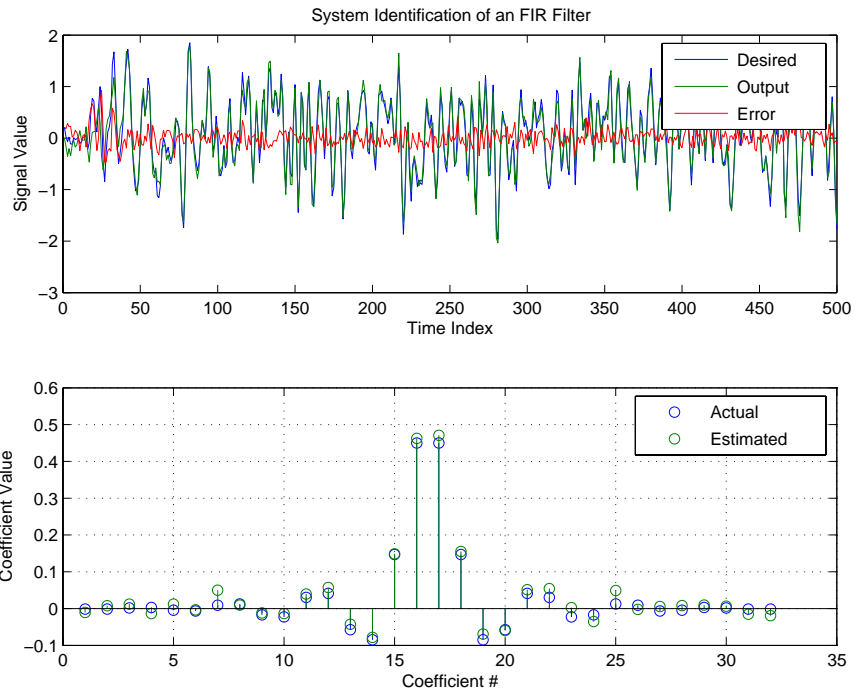
System Identification of a 32-coefficient FIR filter. Use 500 iterations to adapt to the unknown filter. After the example code, you see a figure that plots the results of the running the code.

```

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

```

In the figure you see clearly that the adaptive filter process successfully identified the coefficients of the unknown FIR filter. But then you knew it had to or many things we take for granted, such as modems on computers, would not work.



## See Also

[adaptfilt.rls](#), [adaptfilt.qdr1s](#), [adaptfilt.hswrls](#)

**Purpose** Construct transform-domain (TDAFDCT) adaptive filter object 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.

# adaptfilt.tdafdct

---

<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 should lie between zero and one. For default filter objects, AvgFactor equals $(1 - \text{step})$ . lambda is the input argument that represent AvgFactor.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length $l$ vector where $l$ is the number of filter coefficients. coeffs defaults to length $l$ vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

# adaptfilt.tdafdct

Name	Range	Description
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the 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.
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 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.

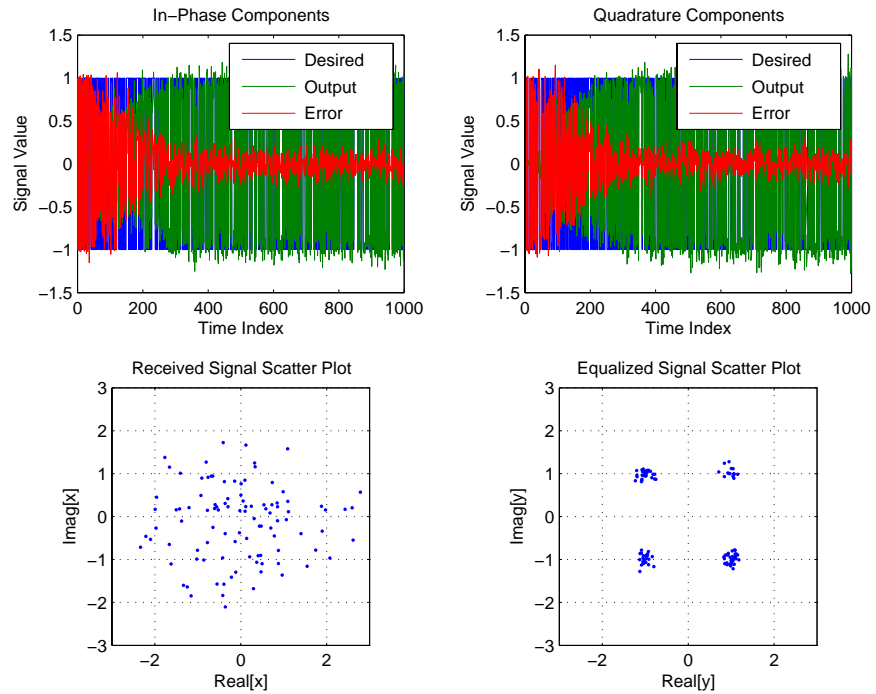
```
D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1000; % Number of iterations
s = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
% QPSK signal
```

## adaptfilt.tdafdct

---

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

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



## See Also

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

## References

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

# adaptfilt.tdafdft

---

**Purpose** Create transform-domain (TDAFDFT) adaptive filter object 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>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>	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.tdafdf` 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.tdafdf` 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 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



Name	Range	Description
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the 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.
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.

Name	Range	Description
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');

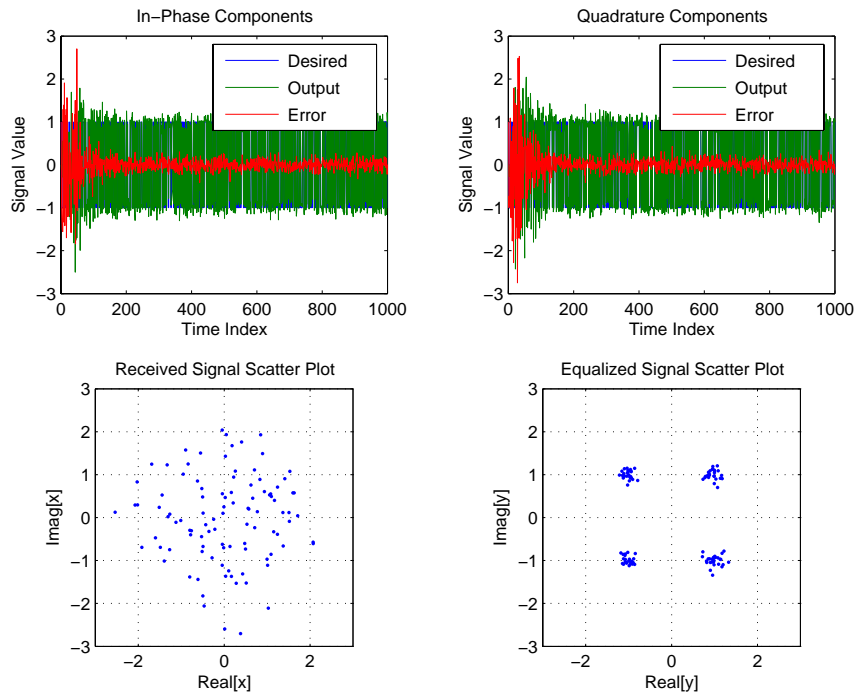
```

```
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.

# adaptfilt.tdafdft



## See Also

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

## References

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

**Purpose** Construct unconstrained frequency-domain (UFDAF) FIR adaptive filter with quantized step size normalization

**Syntax** `ha = adaptfilt.ufdaf(l,step,leakage,delta,lambda,blocklen,offset,c  
oeffs,states)`

**Description** `ha = adaptfilt.ufdaf(l,step,leakage,delta,lambda,blocklen,offset,c  
oeffs,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
<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>	Adaptive filter step size. It must be a nonnegative scalar. <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 UFDAF algorithm. <code>leakage</code> defaults to 1—no leakage.
<code>delta</code>	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.
<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 UFDAF filter objects, <code>lambda</code> defaults to 0.9.

# adaptfilt.ufdaf

---

Input Argument	Description
blocklen	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.
offset	Offset for the normalization terms in the coefficient updates. This can help you avoid divide by zero conditions, or divide by very small numbers conditions, when any of the FFT input signal powers become very small. Default value is zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a length 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> .
states	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 coeffs.
FFTStates		States for the FFT operation.

# adaptfilt.ufdaf

---

<b>Name</b>	<b>Range</b>	<b>Description</b>
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 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 delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	0 to 1	Adaptive filter step size. It must be a nonnegative scalar. You can use maxstep to determine a reasonable range of step size values for the signals being processed. step defaults to 0.

## Examples

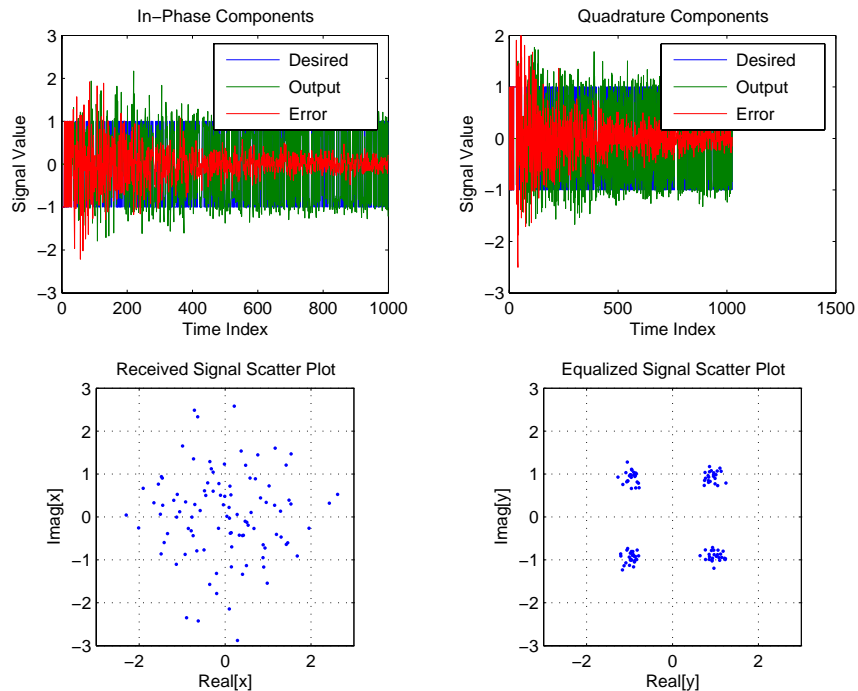
Show an example of Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter. For fidelity, use 1024 iterations. The figure that follows the code provides the information you need to assess the performance of the equalization process.

```

D = 16; % Number of samples of delay
b = exp(j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr= 1024; % Number of iterations
s = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D)); % Baseband
% QPSK signal
n = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D)); % Noise signal
r = filter(b,a,s)+n; % Received signal
x = r(1+D:ntr+D); % Input signal (received signal)
d = s(1:ntr); % Desired signal (delayed QPSK signal)
del = 1; % Initial FFT input powers
mu = 0.1; % Step size
lam = 0.9; % Averaging factor
ha = adaptfilt.ufdaf(32,mu,1,del,lam);
[y,e] = filter(ha,x,d);
subplot(2,2,1);
plot(1:1000,real([d(1:1000);y(1:1000);e(1:1000)]));

```

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



**See Also**

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

**References**

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

# allpassbpc2bpc

---

**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:

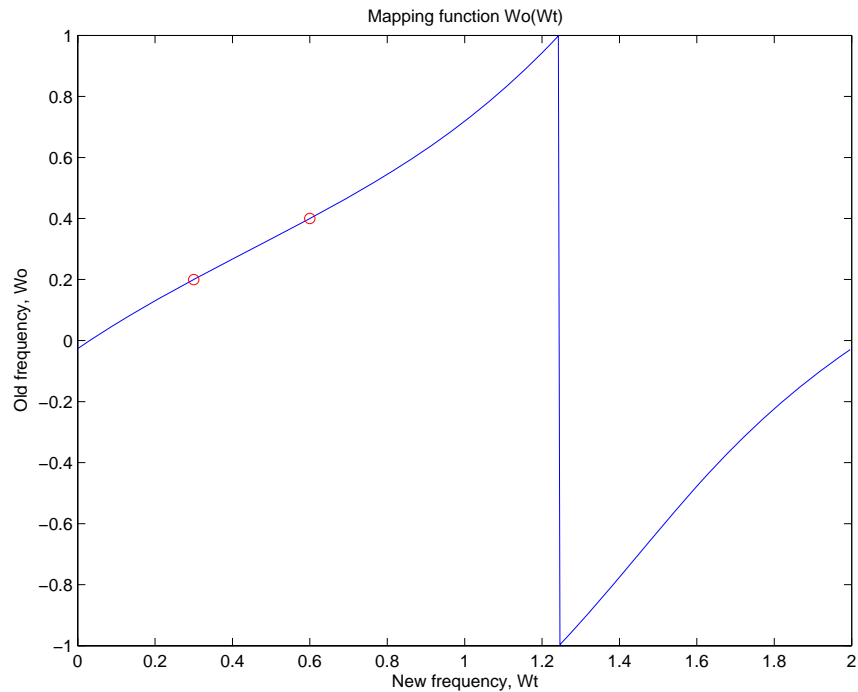
```
[ha, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

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

Wo  
Frequency values to be transformed from the prototype filter

Wt  
Desired frequency locations in the transformed target filter

AllpassNum  
Numerator of the mapping filter

# allpassbpc2bpc

---

AllpassDen  
Denominator of the mapping filter

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

## See Also

iirbpc2bpc, zpkbpc2bpc

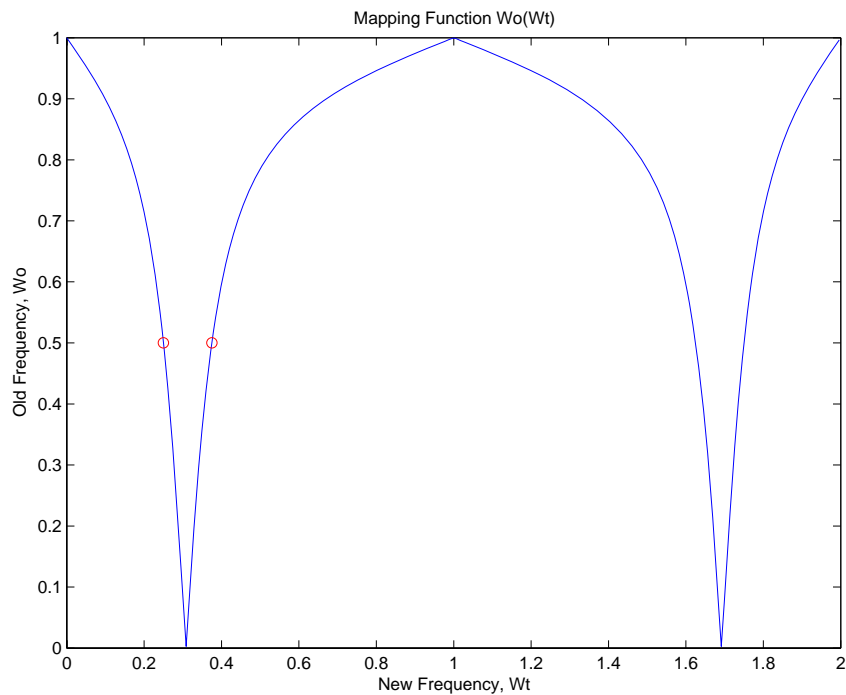
<b>Purpose</b>	Allpass filter for lowpass to bandpass transformation
<b>Syntax</b>	<code>[AllpassNum,AllpassDen] = allpass1p2bp(Wo,Wt)</code>
<b>Description</b>	<p><code>[AllpassNum,AllpassDen] = allpass1p2bp(Wo,Wt)</code> returns the numerator, AllpassNum, and the denominator, AllpassDen, of the second-order 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 <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>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.</p>
<b>Examples</b>	<p>Design the allpass filter changing the lowpass filter with cutoff frequency at <math>W_o=0.5</math> to the real bandpass filter with cutoff frequencies at <math>W_{t1}=0.25</math> and <math>W_{t2}=0.375</math>:</p> <pre> Wo = 0.5; Wt = [0.25, 0.375]; [AllpassNum, AllpassDen] = allpass1p2bp(Wo, Wt); </pre> <p>Calculate the frequency response of the mapping filter in the full range:</p> <pre> [h, f] = freqz(AllpassNum, AllpassDen, 'whole'); </pre>

# allpasslp2bp

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

$W_o$   
Frequency value to be transformed from the prototype filter



Wt

Desired frequency locations in the transformed target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

### See Also

iirlp2bp, zpklp2bp

### References

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

[2] 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.

[3] 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.

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

# allpasslp2bpc

---

**Purpose** Allpass filter for lowpass to complex bandpass transformation

**Syntax** [AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)

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

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

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

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

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

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

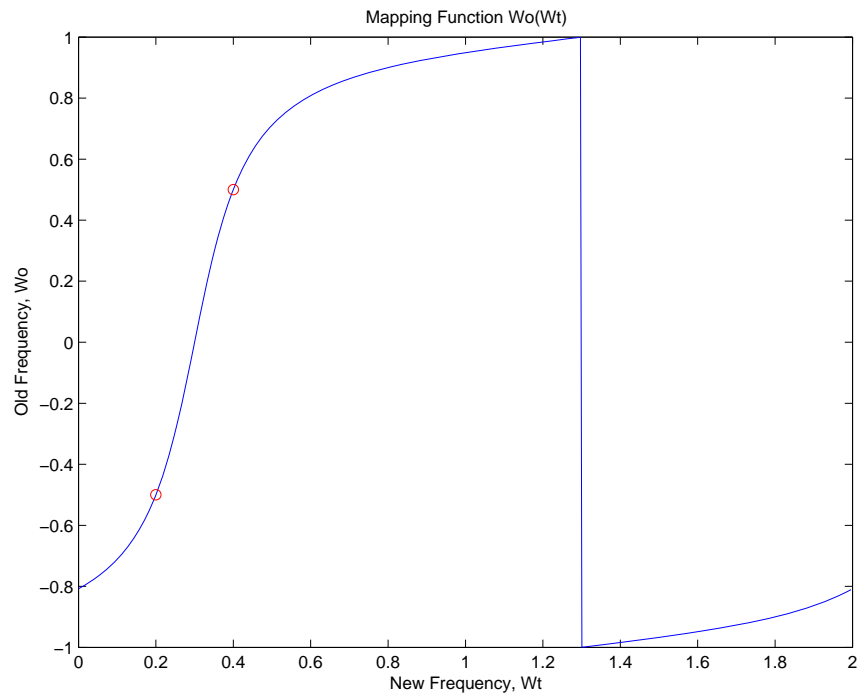
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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

$W_o$

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.

# allpasslp2bpc

---

Wt

Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

## See Also

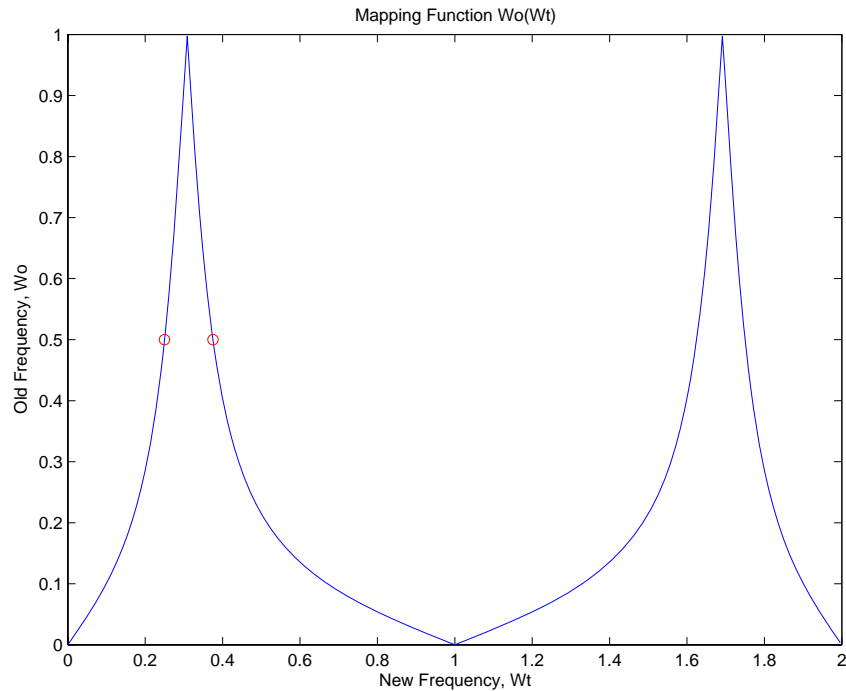
iirlp2bpc, zpk1p2bpc

<b>Purpose</b>	Allpass filter for lowpass to bandstop transformation
<b>Syntax</b>	<code>[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)</code>
<b>Description</b>	<p><code>[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)</code> 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 <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>.</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>
<b>Examples</b>	<p>Design the allpass filter changing the lowpass filter with cutoff frequency at <math>W_o=0.5</math> to the real bandstop filter with cutoff frequencies at <math>W_{t1}=0.25</math> and <math>W_{t2}=0.375</math>:</p> <pre> Wo = 0.5; Wt = [0.25, 0.375]; [AllpassNum, AllpassDen] = allpasslp2bs(Wo, Wt); </pre> <p>Calculate the frequency response of the mapping filter in the full range:</p> <pre> [h, f] = freqz(AllpassNum, AllpassDen, 'whole'); </pre> <p>Plot the phase response normalized to <math>\pi</math>, which is in effect the mapping function <math>W_o(W_t)</math>. Please note that the transformation works in the same way for both positive and negative frequencies:</p>

# allpasslp2bs

```
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

$W_o$   
Frequency value to be transformed from the prototype filter

$W_t$   
Desired frequency locations in the transformed target filter

AllpassNum  
Numerator of the mapping filter

AllpassDen  
Denominator of the mapping filter

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

## See Also

iirlp2bs, zpklp2bs

## References

- [1] Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.
- [2] 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.
- [3] 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.
- [4] Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

# allpasslp2bsc

---

**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:

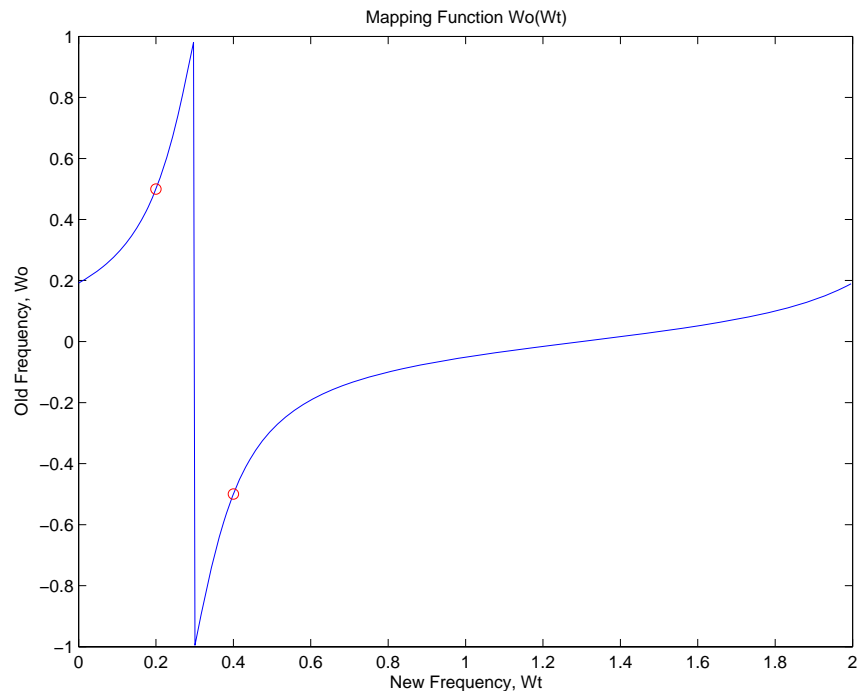


```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

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

$W_o$

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.

# allpasslp2bsc

---

Wt

Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

## See Also

iirlp2bsc, zpk1p2bsc

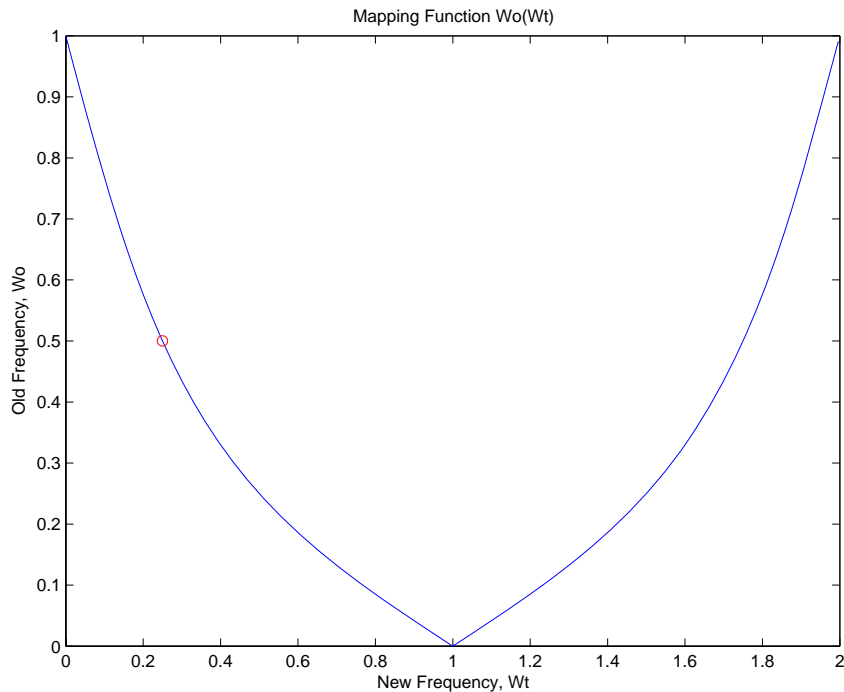
<b>Purpose</b>	Allpass filter for lowpass to highpass transformation
<b>Syntax</b>	<code>[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)</code>
<b>Description</b>	<p><code>[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)</code> returns the numerator, AllpassNum, and the denominator, AllpassDen, of the first-order 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, <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>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, the deep minimum in the stopband.</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 by using the lowpass to highpass transformation.</p>
<b>Examples</b>	<p>Design the allpass filter changing the lowpass filter to the highpass filter with its cutoff frequency moved from <math>W_o=0.5</math> to <math>W_t=0.25</math>:</p> <pre> Wo = 0.5; Wt = 0.25; [AllpassNum, AllpassDen] = allpasslp2hp(Wo, Wt); </pre> <p>Calculate the frequency response of the mapping filter in the full range:</p> <pre> [h, f] = freqz(AllpassNum, AllpassDen, 'whole'); </pre>

# allpasslp2hp

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');
```

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



## Arguments

$W_o$   
Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

### See Also

iirlp2hp, zpklp2hp

### References

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

[2] 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.

[3] 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.

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

# allpasslp2lp

---

**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);
```

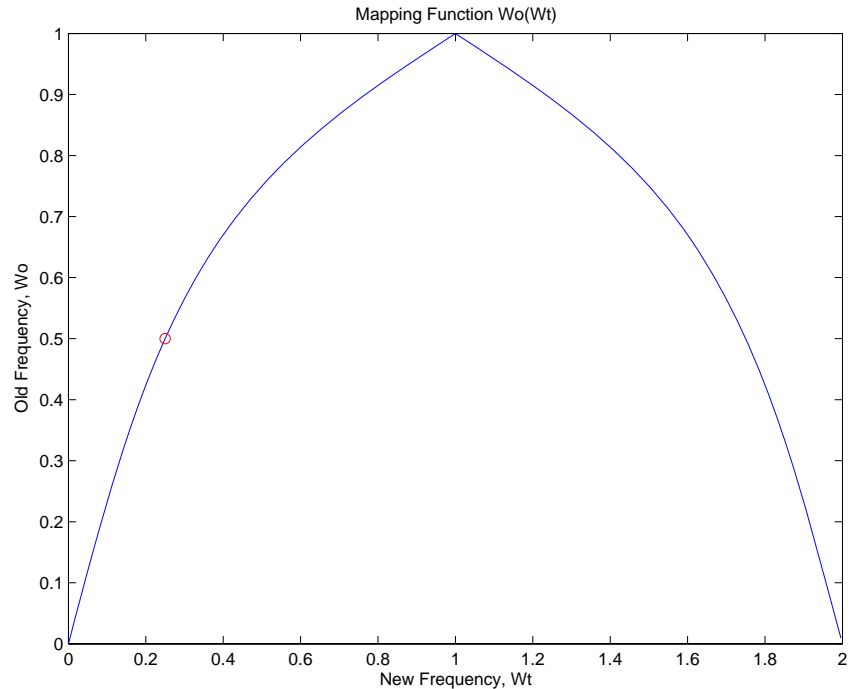
Calculate the frequency response of the mapping filter in the full range:

```
[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

`Wo`  
Frequency value to be transformed from the prototype filter

`Wt`  
Desired frequency location in the transformed target filter

`AllpassNum`  
Numerator of the mapping filter

AllpassDen  
Denominator of the mapping filter

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

## See Also

iirlp2lp, zpklp2lp

## References

- [1] Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.
- [2] 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.
- [3] 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.
- [4] Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.



<b>Purpose</b>	Allpass filter for lowpass to M-band transformation
<b>Syntax</b>	<pre>[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt) [AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)</pre>
<b>Description</b>	<p>[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 <math>W_0</math>, at the required target frequency locations, <math>W_{t1}, \dots, W_{tM}</math>. By default the DC feature is kept at its original location.</p> <p>[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.</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 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.</p>
<b>Examples</b>	Design the allpass filter changing the real lowpass filter with the cutoff frequency of $W_0=0.5$ into a real multiband filter with band edges of $W_t=[1:2:9]/10$ precisely defined:

## allpasslp2mb

---

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

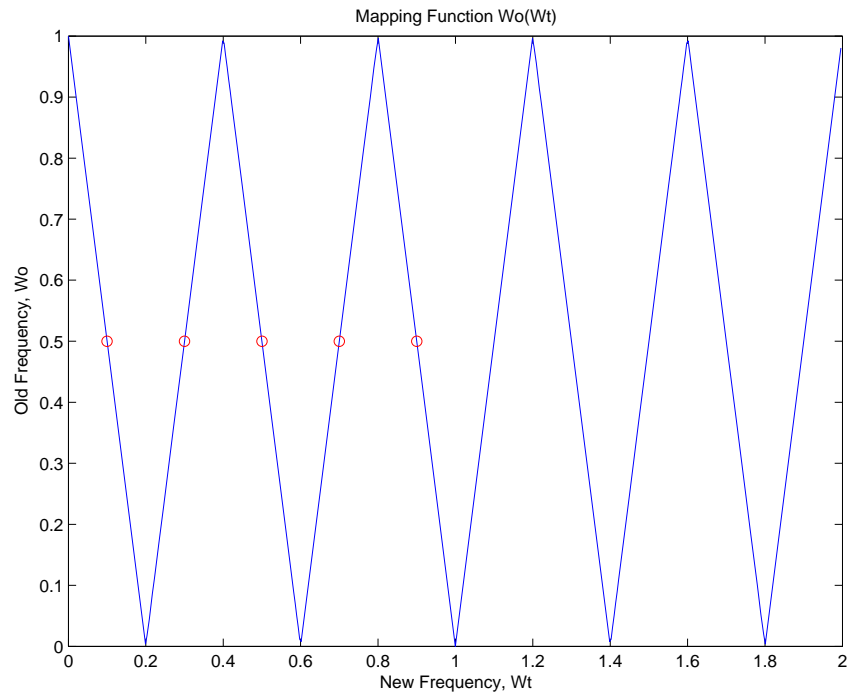
Calculate the frequency response of the mapping filter in the full range:

```
[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

$W_o$

Frequency value to be transformed from the prototype filter

$W_t$

Desired frequency locations in the transformed target filter

Pass

Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

# allpasslp2mb

---

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

## See Also

iirlp2mb, zpklp2mb

## References

- [1] Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," MSc Thesis, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.
- [2] 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.
- [3] Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.
- [4] 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** 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  $W_0$ , 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  $W_0=0.5$  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');

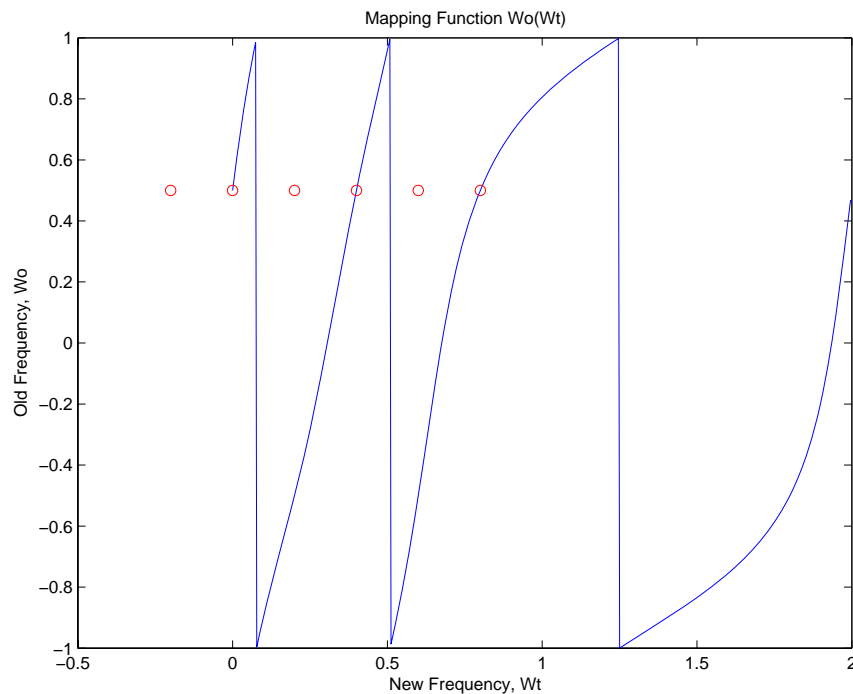
```

# allpasslp2mbc

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.



## Arguments

$W_o$

Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Wt

Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

## See Also

iirlp2mbc, zpklp2mbc

# allpasslp2xc

---

**Purpose** Allpass filter for lowpass to complex N-point transformation

**Syntax** [AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)

**Description** [AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt) returns the numerator, AllpassNum, and the denominator, AllpassDen, of the 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  $W_{o1}, \dots, W_{oN}$ , 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 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];
```



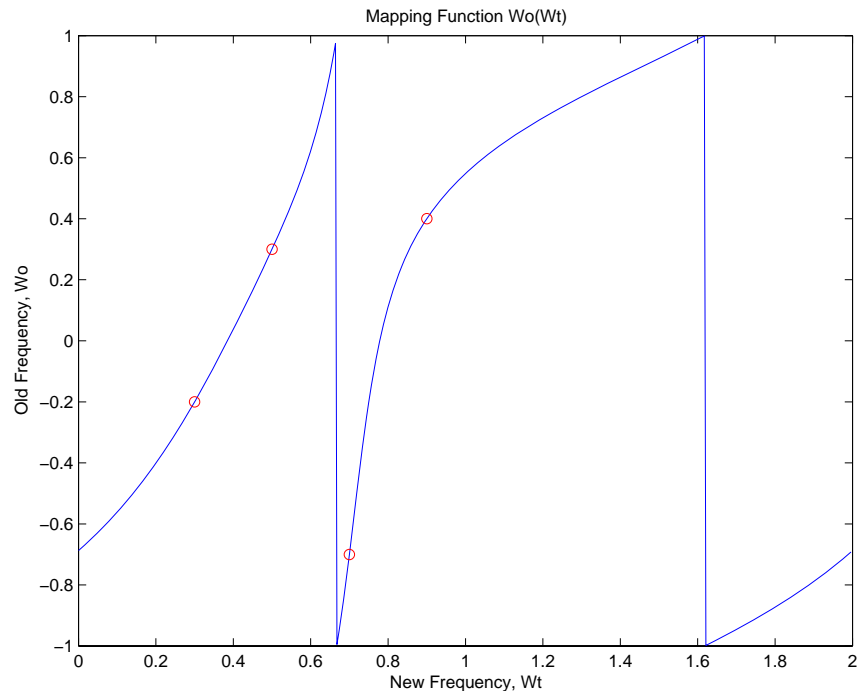
```
[AllpassNum, AllpassDen] = allpass1p2xc(Wo, Wt);
```

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.

# allpasslp2xc

---

## Arguments

Wo

Frequency values to be transformed from the prototype filter

Wt

Desired frequency locations in the transformed target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

iir1p2xc, zpk1p2xc

<b>Purpose</b>	Allpass filter for lowpass to N-point transformation
<b>Syntax</b>	<pre>[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt) [AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass)</pre>
<b>Description</b>	<p>[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 <math>W_{o1}, \dots, W_{oN}</math>, at the required target frequency locations, <math>W_{t1}, \dots, W_{tM}</math>. By default the DC feature is kept at its original location.</p> <p>[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.</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 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.</p>

# allpasslp2xn

---

## Arguments

Wo

Frequency values to be transformed from the prototype filter

Wt

Desired frequency locations in the transformed target filter

Pass

Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

iirlp2xn, zpk1p2xn

## References

[1] 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.

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

**Purpose** 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  $W_0=0.2$ :

```
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:

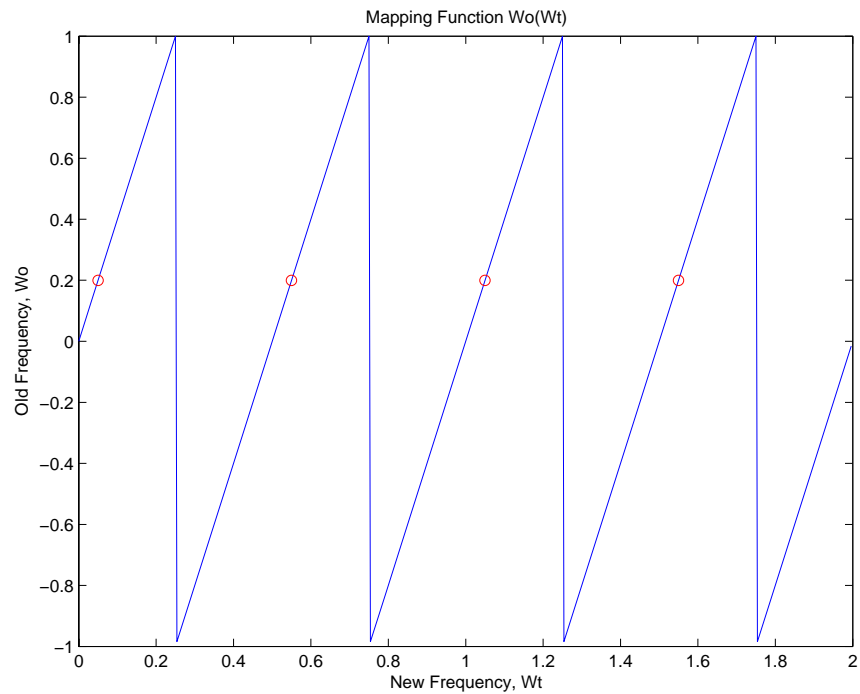
```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to  $\pi$ , which is in effect the mapping function  $W_0(W_t)$ :

```
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.

# allpassrateup



## Arguments

$N$   
Frequency replication ratio (upsampling ratio)

AllpassNum  
Numerator of the mapping filter

AllpassDen  
Denominator of the mapping filter

## See Also

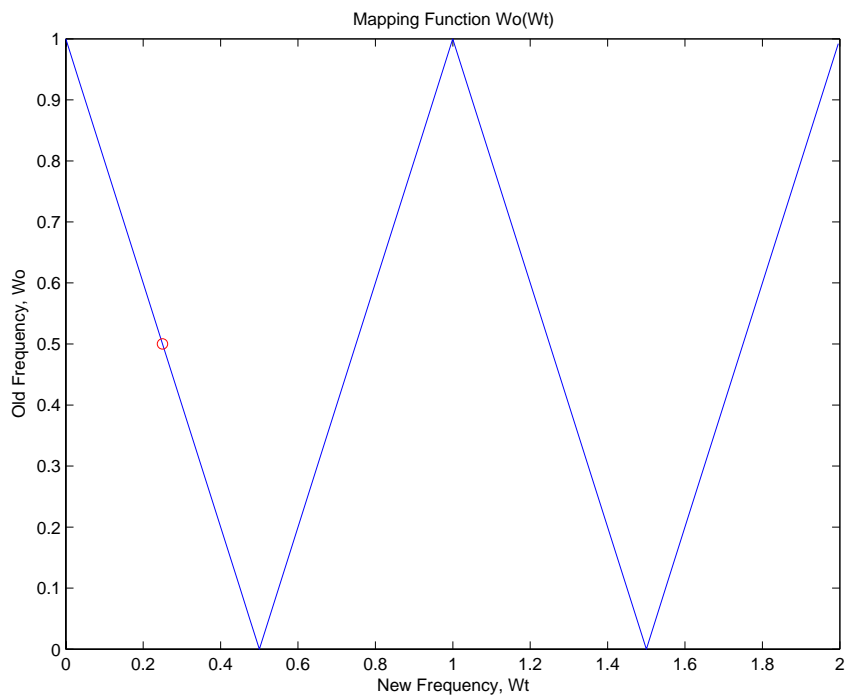
iirrateup, zpkrateup

<b>Purpose</b>	Allpass filter for real shift transformation
<b>Syntax</b>	<code>[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)</code>
<b>Description</b>	<p><code>[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)</code> 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 <math>W_o</math>, at the required target frequency location, <math>W_t</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_o</math> and <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 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.</p> <p>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.</p>
<b>Examples</b>	<p>Design the allpass filter precisely shifting one feature of the lowpass filter originally at <math>W_o=0.5</math> to the new frequencies of <math>W_t=0.25</math>:</p> <pre>Wo = 0.5; Wt = 0.25; [AllpassNum, AllpassDen] = allpassshift(Wo, Wt);</pre> <p>Calculate the frequency response of the mapping filter in the full range:</p> <pre>[h, f] = freqz(AllpassNum, AllpassDen, 'whole');</pre>

# allpassshift

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');
```



## Arguments

$W_o$   
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 0 and 1, with 1 corresponding to half the sample rate.

**See Also**      iirshift, zpkshift

# allpassshiftc

---

**Purpose** Allpass filter for complex shift transformation

**Syntax** [AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)

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

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

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

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

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

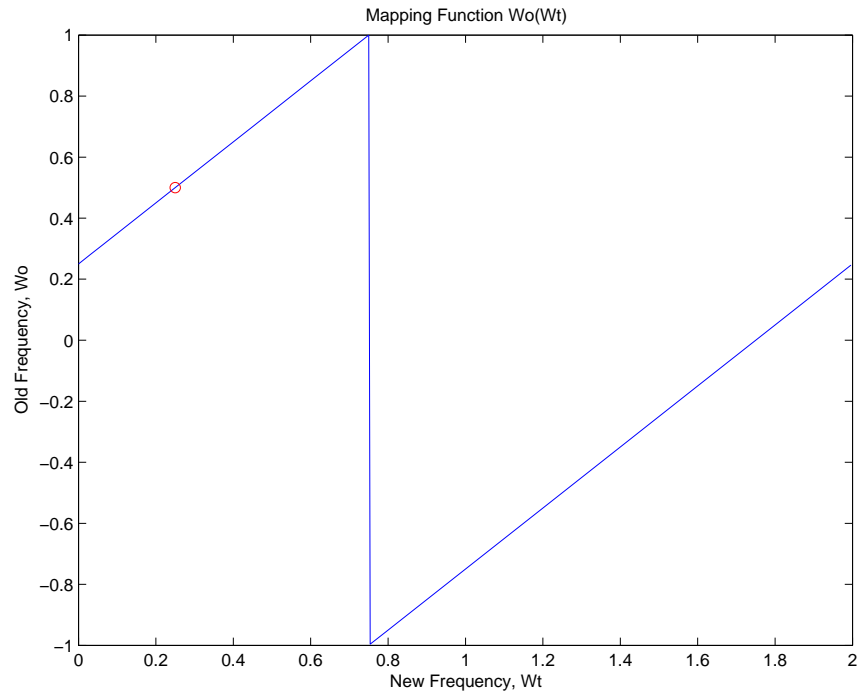
Calculate the frequency response of the mapping filter in the full range:

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

Plot the phase response normalized to  $\pi$ , 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

$W_o$

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

- [1] Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.
- [2] 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** Generate Signal Processing Blockset block from floating-point or fixed-point multirate (`mfilt`) filter objects

**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. Choose either <code>current</code> or <code>new</code> . Specifying <code>new</code> opens a new Simulink model and adds the block. <code>Current</code> adds the block to your current Simulink model. <code>Current</code> is the default setting.
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. Off is the default setting— <code>block</code> does not overwrite existing blocks with matching names. Switching from off to on 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 off means the states are not transferred to the block. Choosing on preserves the current filter states in the block.

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

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

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

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

## Examples

Two examples of using `block` demonstrate the syntax capabilities. Both examples start from an `mfilt` object with interpolation factor of three. In the first example, use `block` with the default syntax, letting the function determine the block name and configuration.

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

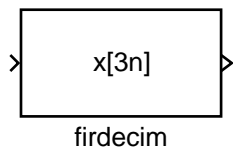
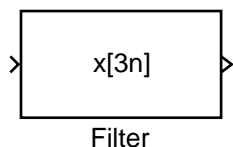
Now use the default syntax to create a block.

```
block(hm);
```

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

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

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



**See Also**

Refer to the Realize Model option in FDATool, and `realizemdl`

# butter

---

**Purpose** Design Butterworth IIR digital filter using the specifications in filter specification object

**Syntax**

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

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

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

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

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

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

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

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

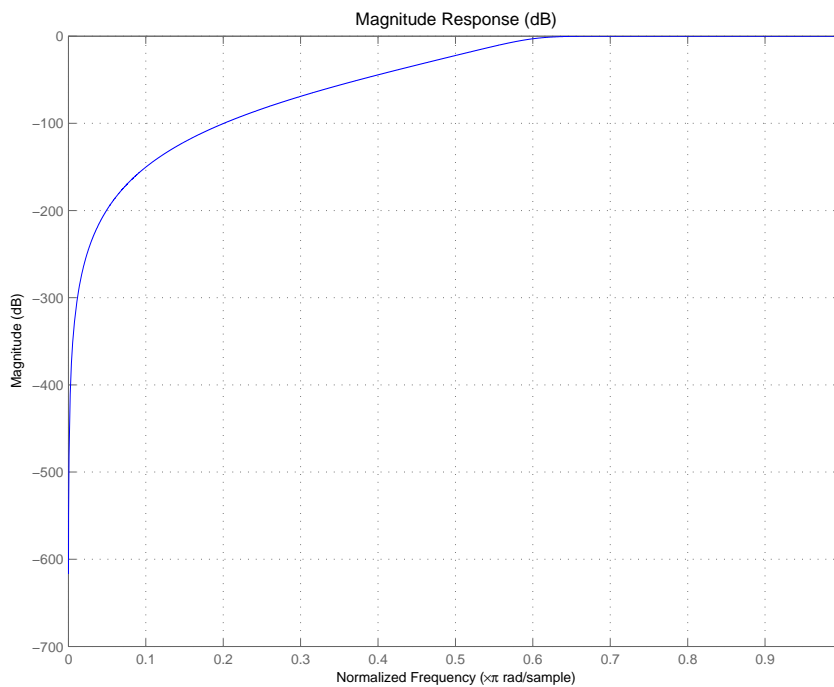
ans =

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

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

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





**See Also** cheby1, cheby2, ellip

**Purpose**

Convert coupled allpass filter form to transfer function forms

**Syntax**

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

**Description**

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

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

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

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

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

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

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

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

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

**Examples**

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

```
[b,a]=cheby1(10,.5,.4);
[d1,d2,beta]=tf2ca(b,a);           % tf2ca returns the
                                   % denominators of the
                                   % allpasses.

[num,den,numpc]=ca2tf(d1,d2,beta); % Reconstruct the original
                                   % 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

# cheby1

---

**Purpose** Design Chebyshev Type I digital filter using filter specification object

**Syntax**

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

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

`hd = design(d,'cheby1',designoption,value,designoption,value,...)` returns a Chebyshev I IIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `cheby1`, refer to the command line help system. For example, to get specific information about using `cheby1` with `d`, the specification object, enter the following at the MATLAB prompt.

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

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

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

    FilterStructure: 'df2sos'
    MatchExactly: 'passband'

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

d =

    Response: 'Lowpass'
    Specification: 'Fp,Fst,Ap,Ast'
    Description: {4x1 cell}
```

```
NormalizedFrequency: true
                    Fpass: 0.45
                    Fstop: 0.55
                    Apass: 1
                    Astop: 60
```

```
hd =
```

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

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

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

```
d =
```

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

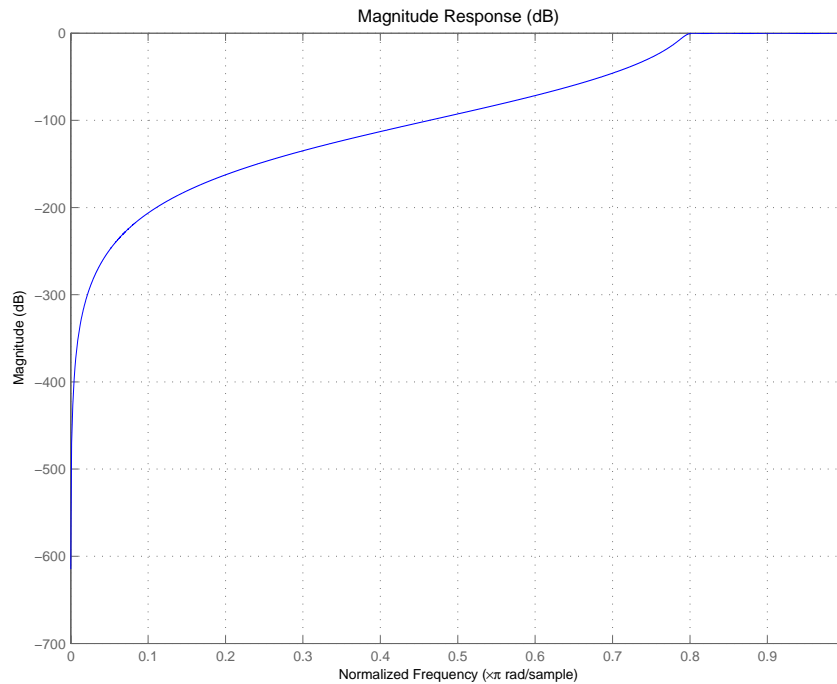
```
hd =
```

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

# cheby1

Use `fvtool` to view the resulting filter.

```
fvtool(hd)
```



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

## See Also

`butter`, `cheby2`, `ellip`

<b>Purpose</b>	Design Chebyshev Type II digital filter using filter specification object
<b>Syntax</b>	<pre>hd = design(d,'cheby2') hd = design(d,'cheby2',designoption,value,designoption,value,...)</pre>
<b>Description</b>	<p><code>hd = design(d,'cheby2')</code> designs a Chebyshev II IIR digital filter using the specifications supplied in the object <code>d</code>.</p> <p><code>hd = design(d,'cheby2',designoption,value,designoption,value,...)</code> returns a Chebyshev II IIR 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>cheby1</code>, refer to the command line help system. For example, to get specific information about using <code>cheby2</code> with <code>d</code>, the specification object, enter the following at the MATLAB prompt.</p> <pre>help(d,'cheby2')</pre>
<b>Examples</b>	<p>These examples use filter specification objects to construct Chebyshev type I filters. In the first example, you use the <code>matchexactly</code> option to ensure the performance of the filter in the passband.</p> <pre>d = fdesign.lowpass; hd = design(d,'cheby2','matchexactly','passband')</pre> <p><code>hd =</code></p> <pre>FilterStructure: 'Direct-Form II, Second-Order Sections' Arithmetic: 'double' sosMatrix: [5x6 double] ScaleValues: [6x1 double] PersistentMemory: false</pre>

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

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

```
d =
```

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

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

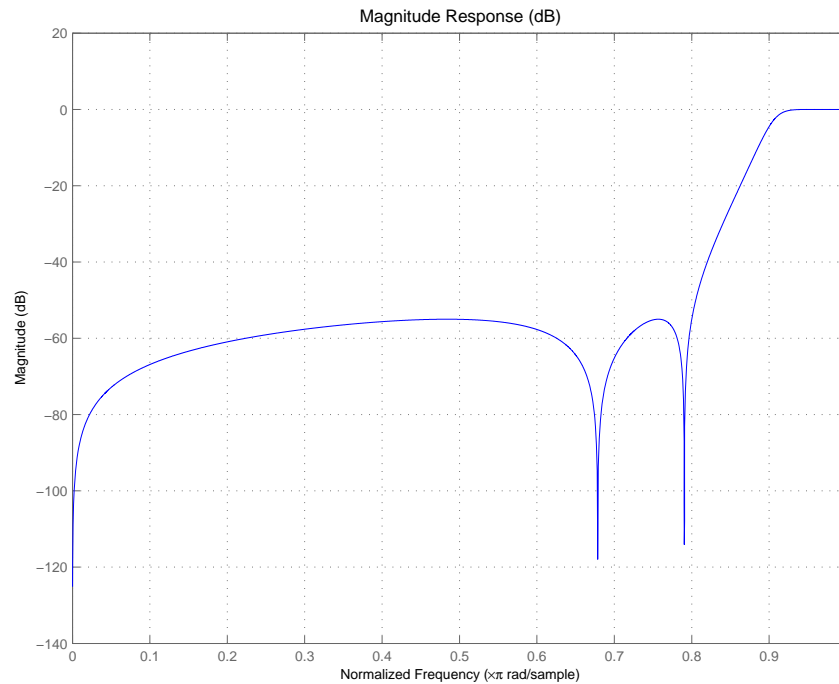
```
hd =
```

```
FilterStructure: 'Direct-Form II, Second-Order Sections'  
   Arithmetic: 'double'  
       sosMatrix: [3x6 double]  
   ScaleValues: [0.199517233712056;0.0879972176933622;0.145046319812257;1]  
 PersistentMemory: false
```

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

```
fvtool(hd)
```





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

**See Also**

`butter`, `cheby1`, `ellip`

# cl2tf

---

## Purpose

Convert coupled allpass lattice to transfer function form

## Syntax

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

## Description

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

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

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

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

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

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

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

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

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

## Examples

```
[b,a]=cheby1(10,.5,.4);
[k1,k2,beta]=tf2cl(b,a); %TF2CL returns the reflection coeffs
```

```
% Reconstruct the original filter
% plus the power complementary one.
[num,den,numpc]=cl2tf(k1,k2,beta);
[h,w,s1]=freqz(num,den);
hpc = freqz(numpc,den);
s.plot = 'mag';
s.yunits = 'sq';
% Plot the mag response of the original filter and the power
% complementary one.
freqzplot([h hpc],w,s1);
```

**See Also**

tf2c1, tf2ca, ca2tf, tf2latc, latc2tf, iirpowcomp

# coefficients

---

**Purpose** Filter coefficients for adaptive filters, discrete-time filters, and multirate filters

**Syntax**

```
c = coefficients(ha)
coefficients(ha)
c = coefficients(hd)
coefficients(hd)
c = coefficients(hm)
coefficients(hm)
```

**Description** The next sections describe common coefficients operation with adaptive, discrete-time, and multirate filters.

## Adaptive Filters

`c = coefficients(ha)` returns a cell array `c` containing the coefficients of adaptive filter `ha`. These are the instantaneous filter coefficients available at the time you use the function.

`coefficients(ha)` without an output argument opens FVTool in the coefficients analysis mode displaying the filter coefficients.

## Discrete-Time Filters

`c = coefficients(hd)` returns a cell array `c` that contains the coefficients of discrete-time filter `hd`.

`coefficients(hd)` without an output argument opens FVTool in the coefficients analysis mode displaying the filter coefficients.

## Multirate Filters

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

`coefficients(hm)` with no output argument opens FVTool in the coefficients analysis mode displaying the filter coefficients.

## Examples

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

```

hm=mfilt.firdecim(3)

hm =

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

c=coefficients(hm)

c =

    [1x72 double]

c{1}

ans =

    Columns 1 through 8

         0   -0.0000   -0.0001         0   0.0002   0.0003         0   -0.0005

    Columns 9 through 16

   -0.0007         0   0.0011   0.0014         0   -0.0022   -0.0028         0

    Columns 17 through 24

    0.0040   0.0048         0   -0.0068   -0.0080         0   0.0111   0.0129

    Columns 25 through 32

         0   -0.0177   -0.0207         0   0.0287   0.0342         0   -0.0513

    Columns 33 through 40

   -0.0659         0   0.1363   0.2749   0.3333   0.2749   0.1363         0

    Columns 41 through 48

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

    Columns 49 through 56

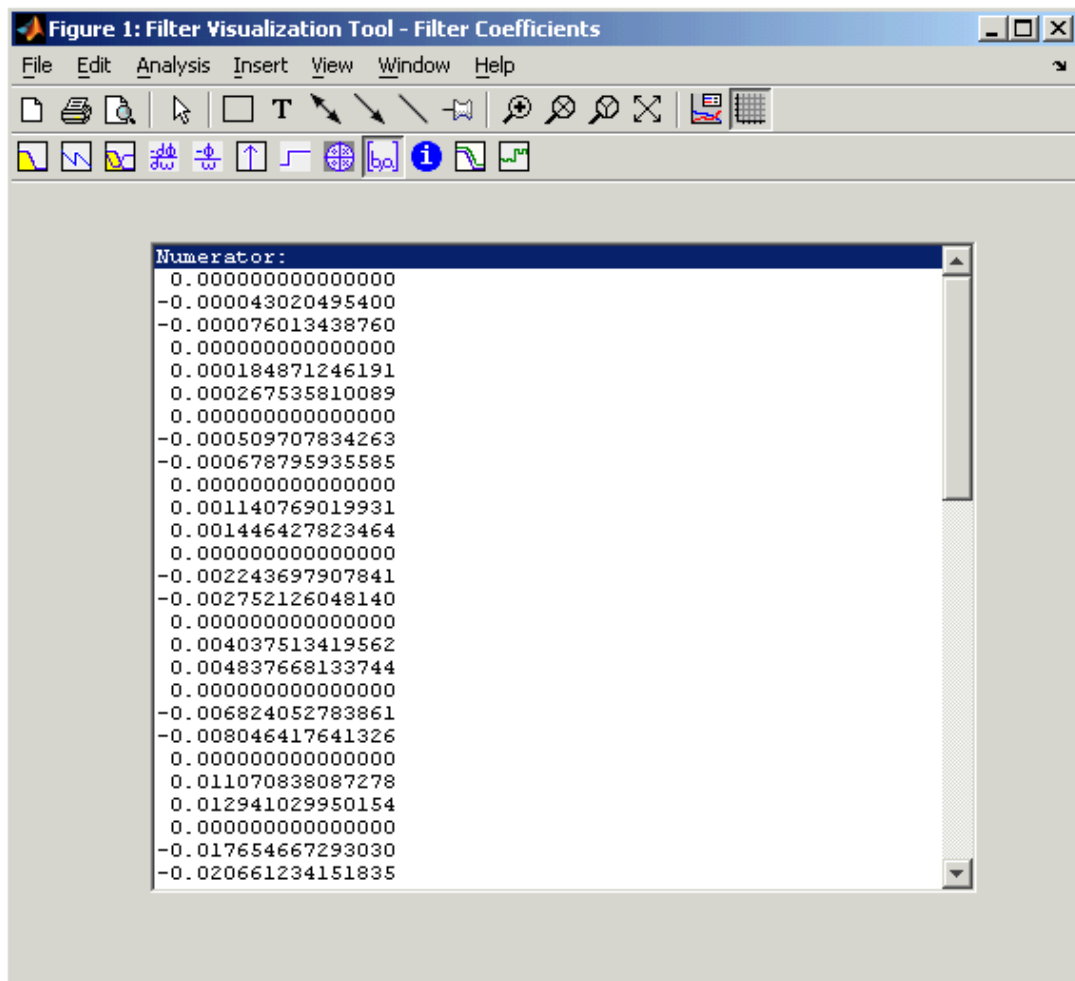
         0   0.0129   0.0111         0   -0.0080   -0.0068         0   0.0048

    Columns 57 through 64

```

# coefficients

```
0.0040      0 -0.0028 -0.0022      0 0.0014 0.0011      0
Columns 65 through 72
-0.0007 -0.0005      0 0.0003 0.0002      0 -0.0001 -0.0000
coefficients(hm)
```



**See Also**

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

# coeread

---

**Purpose** Read Xilinx CORE Generator™ coefficient (.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 CORE Generator™ coefficient (.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 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(tm) and the Filter Design Toolbox.
;
```

# coewrite

---

```
; Generated on: 4-Dec-2003 13:47:15
;
Radix = 10;
Coefficient_Width = 16;
CoefData =   -41,
            -851,
            -366,
             308,
             651,
              22,
            -873,
            -658,
             749,
            1504,
              21,
           -2367,
           -2012,
            3014,
            9900,
            ....
```

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

## See Also

coeread, dfilt, dfilt.dffir

**Purpose** Convert filter structures of discrete-time and multirate filters

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

**Description** **Discrete-Time Filters**

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

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

All filters can be converted to the following structures:

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

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

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

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

## Multirate Filters

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

### Cascaded Integrator-Comb Structures

- `cicdecim`—CIC-based decimator
- `cicdecimzerolat`—CIC-based decimator that exhibits no latency
- `cicinterp`—CIC-based interpolator
- `cicinterpzerolat`—CIC-based interpolater that does not induce latency

### FIR Structures

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

You cannot convert between the FIR and CIC structures.

**Examples**

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

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

For an example of changing the structure of a multirate filter, try the following conversion from a CIC interpolator to a CIC interpolator with zero latency.

```
hm = mfilt.cicinterp(2,2,3,8,8)
hm =
```

```
FilterStructure: 'Cascaded Integrator-Comb Interpolator'
Arithmetic: 'int'
DifferentialDelay: 2
NumberOfSections: 3
InterpolationFactor: 2
RoundMode: 'floor'
PersistentMemory: false
States: Integrator: [3x1 States]
Comb: [3x1 States]
```

```
InputWordLength: 8
```

```
SectionWordLengthMode: 'MinWordLengths'
```

```
OutputWordLength: 8
```

```
hm2=convert(hm,'cicinterpzerolat')
```

```
hm2 =
```

```
FilterStructure: 'Zero-Latency Cascaded Integrator-Comb Interpolator'
Arithmetic: 'int'
DifferentialDelay: 2
NumberOfSections: 3
InterpolationFactor: 2
RoundMode: 'floor'
PersistentMemory: false
States: Integrator: [3x1 States]
Comb: [3x1 States]
```

# convert

---

InputWordLength: 8

SectionWordLengthMode: 'MinWordLengths'

OutputWordLength: 8

## See Also

`mfilt`

`dfilt` in the Signal Processing Toolbox documentation

**Purpose** Estimate 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'
```

```
c=cost(hm)
```

```
c =
```

```
Number of Multipliers : 0
Number of Adders      : 4
Number of States      : 4
```



MultPerInputSample : 0  
AddPerInputSample : 2.5

**See Also** report

# cumsec

---

**Purpose** Vector of filters for cumulative sections

**Syntax**

```
h = cumsec(hd)
h = cumsec(hd,indices)
h = cumsec(hd,indices,secondary)
cumsec(hd)
```

**Description** `h = cumsec(hd)` returns a vector `h` of SOS filter objects with the cumulative sections. Each element in `h` is a filter with the structure of the original filter. The first element is the first filter section of `hd`. The second element of `h` is a filter that represents the combination of the first and second sections of `hd`. The third element of `h` is a filter which combines sections 1, 2, and 3 of `hd`. This pattern continues until the final element of `h` contains all the sections of `hd` and should be identical to `hd`.

`h = cumsec(hd,indices)` returns a vector `h` of SOS filter objects whose indices into the original filter are in the vector `indices`. Now you can specify the filter sections `cumsec` uses to compute the cumulative responses.

`h = cumsec(hd,indices,secondary)` when `secondary` is true, `cumsec` uses the secondary scaling points in the sections to determine where the sections should be split. This option applies only when `hd` is a `df2sos` and `df1tsos` filter. For these second-order section structures, the secondary scaling points refer to the scaling locations between the recursive and the nonrecursive parts of the section (the “middle” of the section). Argument `secondary` accepts either true or false. By default, `secondary` is false.

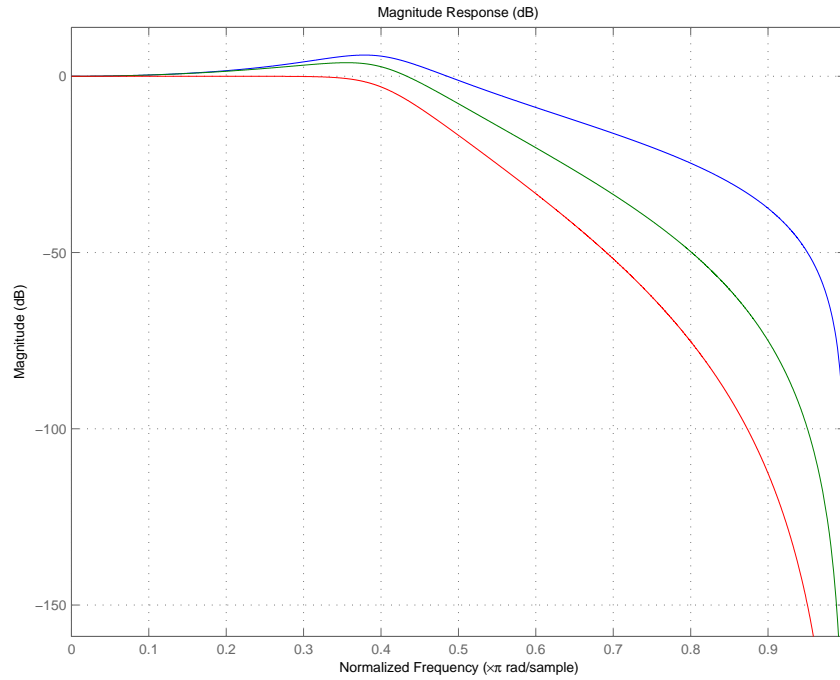
`cumsec(hd,...)` without an output arguments uses `FVTool` to plot the magnitude response of the cumulative sections.

## Examples

To demonstrate how `cumsec` works, this example plots the relative responses of the sections of a sixth-order filter SOS filter with three sections. Each curve adds one more section to form the filter response.

```
hs = fdesign.lowpass('n,fc',6,.4);
hd = butter(hs);
h = cumsec(hd);
hfvt = fvtool(h);
```

```
legend(hfvt, 'First Section', 'First Two Sections', 'Overall  
Filter');
```



**See Also** `scale`, `scalecheck`

# 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,fc',14,0.45)

d =

    Response: 'Highpass'
  Specification: 'N,Fc'
  Description: {'Filter Order';'Cutoff Frequency'}
NormalizedFrequency: true
  FilterOrder: 14
    Fcutoff: 0.45

hd = butter(d)

hd =

    FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
      sosMatrix: [7x6 double]
      ScaleValues: [8x1 double]
  PersistentMemory: false

hd.arithmetic='fixed'

hd =

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

```

    Arithmetic: 'fixed'
      sosMatrix: [7x6 double]
      ScaleValues: [8x1 double]
    PersistentMemory: false

    CoeffWordLength: 16
      CoeffAutoScale: true
      Signed: true

    InputWordLength: 16
    InputFracLength: 15

    StageInputWordLength: 16
    StageInputAutoScale: true

    StageOutputWordLength: 16
    StageOutputAutoScale: true

    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'

    StateWordLength: 16
    StateFracLength: 15

    ProductMode: 'FullPrecision'

    AccumMode: 'KeepMSB'
    AccumWordLength: 40
    CastBeforeSum: true

    RoundMode: 'convergent'
    OverflowMode: 'wrap'

```

```

hq=hd;
g=normalize(hq) '

```

g =

2 2 2 2 2 2 2

# denormalize

---

```
hq.SosMatrix
```

```
ans =
```

```
0.5000 -1.0000 0.5000 1.0000 -0.2817 0.8008
0.5000 -1.0000 0.5000 1.0000 -0.2359 0.5081
0.5000 -1.0000 0.5000 1.0000 -0.2051 0.3110
0.5000 -1.0000 0.5000 1.0000 -0.1842 0.1776
0.5000 -1.0000 0.5000 1.0000 -0.1704 0.0892
0.5000 -1.0000 0.5000 1.0000 -0.1619 0.0350
0.5000 -1.0000 0.5000 1.0000 -0.1579 0.0093
```

```
denormalize(hq)
```

```
hq.SosMatrix
```

```
ans =
```

```
1.0000 -2.0000 1.0000 1.0000 -0.2817 0.8008
1.0000 -2.0000 1.0000 1.0000 -0.2359 0.5081
1.0000 -2.0000 1.0000 1.0000 -0.2051 0.3110
1.0000 -2.0000 1.0000 1.0000 -0.1842 0.1776
1.0000 -2.0000 1.0000 1.0000 -0.1704 0.0892
1.0000 -2.0000 1.0000 1.0000 -0.1619 0.0350
1.0000 -2.0000 1.0000 1.0000 -0.1579 0.0093
```

## See Also

`normalize`

**Purpose** Implement FIR or IIR filter from discrete-time or multirate filter specification object

**Syntax**

```
h = design(d)
h = design(d,designmethod)
h = design(d,designmethod,specname,specvalue,...)
```

**Description** `h = design(d)` uses specifications object `d` to generate a filter `h`. When you do not provide a design method as an input argument, `design` chooses the design method to use by following these rules in the order listed.

- 1 Use `equiripple` if it applies to the object `d`.
- 2 When `equiripple` does not apply to `d`, use another FIR design method, such as `firls`.
- 3 If FIR design methods do not apply to `d`, use `ellip`.
- 4 When `ellip` does not apply to `d`, use another IIR design method, such as `butter` or `cheby2`, that applies to the object `d`.

More rules apply.

- `design` uses an FIR filter design method before using an IIR design method.
- `fdesign.nyquist` specifications objects use the `kaiserwin` design method as the first design choice, rather than `equiripple`, because `kaiserwin` produces better filters than `equiripple`.
- For decimators, interpolators, and rational sample rate changers that use `fdesign.nyquist` objects, the default design method is `kaiserwin`. Otherwise, those objects use the `equiripple` design method by default.

For more guidance about using `design` to design filters, refer to “Designing Fixed-Point Filters” on page 2-3 of the Filter Design Toolbox User’s Guide. In this section you find some examples that use `design` to design filters and use methods in the toolbox to analyze them.

`h = design(d,designmethod)` lets you specify a valid design method to design the filter as an input string. Note that the filter returned by `design` changes depending on the design method you choose. For more information about the filter that a design method returns, refer to the help for the design method.

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

```
designmethods(d)
```

for the specifications object `d`.

Valid entries depend on `d`. This is the complete set of design methods. The methods that apply to a specific specifications object usually represent a subset of this list.

- `butter`
- `cheby1`
- `cheby2`
- `ciccomp`
- `ellip`
- `equiripple`
- `firls`
- `ifir`
- `iirhilbert`
- `iirlinphase`
- `isinclp`
- `kaiserwin`
- `multistage`
- `window`

To help you design filters more quickly, the input argument `designmethod` accepts a variety of special keywords that force `design` to behave in different



ways. The following table presents the keywords you can use for `designmethod` and how `design` responds to the keyword.

<b>Designmethod Keyword</b>	<b>Description of the design Response</b>
<b>fir</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>iir</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>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 `designmethod` but also values for the filter specifications in the method. Provide the specifications in the order of the name of the specification, such as the `FilterOrder`, followed by the value to assign to the specification. Enter as many `specname/specvalue` pairs as you need to define your filter. Any specification you do not define uses the default specification value. To use the `specname/specvalue` syntax, you must provide the design method to use in `designmethod`.

## Examples

To demonstrate some of the design options, these examples use a few different input arguments and output arguments. For the first example, use `design` to return the default filter based on the default design method `equiripple`.

```
d = fdesign.lowpass(.2, .22);
hd = design(d) % Uses the default equiripple method.
```

```
hd =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'double'
      Numerator: [1x202 double]
 PersistentMemory: false
```

In this example, use the **allfir** keyword with `design` to return an FIR filter for each valid design method for the specifications in specifications object `d`.

```
designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):

    butter
    cheby1
    cheby2
    ellip
    equiripple
    ifir
    kaiserwin
```

```
multistage

hallfir=design(d,'allfir')

hallfir =

    dfilt.basefilter: 1-by-4
```

hallfir contains filters designed using the `ellip`, `equiripple`, `ifir`, and `multistage` design methods, in the order shown by `designmethods(d)`. The first filter in `hallfir` comes from the `ellip` design method; the second from the `equiripple` method; the third from using `ifir` to design the filter; and the fourth from using `multistage`.

To see an individual filter, use an index with the filter object. For example, to see the second filter in `hallfir`, enter `hallfir(2)`

```
hallfir(2)

ans =

    FilterStructure: Cascade
           Stage(1): Direct-Form FIR
           Stage(2): Direct-Form FIR
 PersistentMemory: false
```

Here is the multistage filter `hallfir(4)`

```
hallfir(4)

ans =

    FilterStructure: Cascade
           Stage(1): Direct-Form FIR Polyphase Decimator
           Stage(2): Direct-Form FIR Polyphase Decimator
           Stage(3): Direct-Form FIR Polyphase Decimator
           Stage(4): Direct-Form FIR Polyphase Interpolator
           Stage(5): Direct-Form FIR Polyphase Interpolator
           Stage(6): Direct-Form FIR Polyphase Interpolator
 PersistentMemory: false
```

This final example uses `equiripple` to design an FIR filter with the density factor set to 20 by using the `specname/specvalue` syntax.

```
[hd,res,err] = design(d,'equiripple','densityfactor',20);
hd

hd =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'double'
      Numerator: [1x202 double]
 PersistentMemory: false
res

res =

    0.9903

err

err =

    order: 201
   fgrid: [2060x1 double]
        H: [2060x1 double]
   error: [2060x1 double]
      des: [2060x1 double]
      wt: [2060x1 double]
   iextr: [102x1 double]
   fextr: [102x1 double]
 iterations: 12
    evals: 12905
 edgeCheck: [4x1 double]
returnCode: 0
```

`res` and `err` are optional output arguments that `design` returns when you specify the density factor with the `equiripple` design method.

## See Also

`designmethods`, `butter`, `cheby1`, `cheby2`, `ellip`, `equiripple`, `firls`, `fdesign.halfband`, `kaiserwin`, `fdesign.nyquist`, `fdesign.rsrc`

**Purpose** Design methods available for designing filter from filter specification object

**Syntax**

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

**Description** `m = designmethods(d)` returns a list of the design methods available for the filter specification object `d` with its Specification. When you change the Specification for a filter specification object, the methods available to design filters from the object change.

Here are all the design methods and the filters they produce.

Design Method	Filter Result
butter	IIR
cheby1	IIR
cheby2	IIR
ellip	IIR
equiripple	FIR
firls	FIR
ifir	Interpolated FIR
iirhilbert	IIR Hilbert filter
iirlinphase	IIR filter with linear phase
iirlpnorm	IIR filter from an arbitrary magnitude specifications object. Compare to <code>iirls</code> .
iirls	IIR filter from an arbitrary magnitude and phase specifications object. Compare to <code>iirlpnorm</code> .
kaiserwin	FIR with Kaiser window

# designmethods

---

Design Method	Filter Result
multistage	Multistage filter that cascades multiple filters
window	FIR with windowed impulse response

`m = designmethods(d, 'default')` returns the default design method for the filter specification object `d` and its current Specification.

`m = designmethods(d, type)` returns either the FIR or IIR design methods that apply to `d`, as specified by the type string, either `fir` or `iir`. By default, `designmethods` returns all the valid design methods when you omit the type string.

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

## Examples

Construct a lowpass filter specification object and determine the design methods available to design a filter from the object.

```
d=fdesign.lowpass('n,fc',10,12000,48000)
```

```
d =
```

```
      Response: 'Lowpass'  
Specification: 'N,Fc'  
Description: {'Filter Order';'Cutoff Frequency'}  
NormalizedFrequency: false  
           Fs: 48000  
FilterOrder: 10  
      Cutoff: 12000
```

```
designmethods(d)
```

```
Design Methods for class fdesign.lowpass (N,Fc):
```

```
window
hd=window(d)
hd =
    FilterStructure: 'Direct-Form FIR'
    Arithmetic: 'double'
    Numerator: [1x11 double]
    PersistentMemory: false
```

Now change the Specification string for d to 'fp,fst,ap,ast' and determine the design methods that apply to your modified specifications object.

```
set(d,'specification','fp,fst,ap,ast');
d
d =
    Response: 'Lowpass'
    Specification: 'Fp,Fst,Ap,Ast'
    Description: {4x1 cell}
    NormalizedFrequency: false
    Fs: 48000
    Fpass: 10800
    Fstop: 13200
    Apass: 1
    Astop: 60
```

```
m2 = designmethods(d)
m3 = designmethods(d, 'iir')
m4 = designmethods(d, 'iir', 'full')
```

```
m2 =
    'butter'
    'cheby1'
    'cheby2'
    'ellip'
    'equiripple'
    'ifir'
```

# designmethods

---

```
'kaiserwin'  
'multistage'
```

```
m3 =
```

```
'butter'  
'cheby1'  
'cheby2'  
'ellip'
```

```
m4 =
```

```
'Butterworth'  
'Chebyshev Type I'  
'Chebyshev Type II'  
'Elliptic'
```

Now you can get specific help on a particular design method for the specifications object. This example returns the help for the first design method for the m2 set of methods—butter.

```
help(d,m2{1})
```

This is the same as `help(d, 'butter')`.

## See Also

butter, cheby1, cheby2, designopts, ellip, equiripple, kaiserwin, multistage



**Purpose** Input arguments and default values applicable to filter 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`, as shown here.

```
help(d,'cheby2') % Get the help for design Chebyshev II filters.
```

```
DESIGN Design a Chebyshev Type II iir filter.
```

```
HD = DESIGN(D, 'cheby2') designs a Chebyshev Type II filter specified by the FDESIGN object H.
```

```
HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the structure STRUCTURE. STRUCTURE is 'df2sos' by default and can be any of the following.
```

```
'df1sos'
'df2sos'
'df1tsos'
'df2tsos'
```

```
HD = DESIGN(..., 'MatchExactly', MATCH) designs a Chebyshev Type II filter and matches the frequency and magnitude specification for the band MATCH exactly. The other band will exceed the specification. MATCH can be 'stopband' or 'passband' and is 'passband' by default.
```

## Examples

Design a minimum order, lowpass Butterworth filter. Use designmethods to determine the appropriate input arguments. Start by creating a lowpass filter specification object d.

```
d = fdesign.lowpass;
```

Because you want information about the input arguments for designing a filter using a design method, use designmethods(d) to get the list of valid methods.

```
designmethods(d)
```

```
Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```
butter  
cheby1  
cheby2  
ellip  
equiripple  
ifir  
kaiserwin  
multistage
```

Pick one method and determine the design options for that method.

```
options = designopts(d,'butter')
```

```
options =
```

```
FilterStructure: 'df2sos'  
MatchExactly: 'stopband'
```

In this example, the filter structure is Direct-Form II with second-order sections, and the design seeks to match the desired stopband performance exactly. As you see by reading the help, FilterStructure and MatchExactly are input arguments for designing the Butterworth filter.

Get help for designing a filter from d using the butter design method to see the arguments.

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

```
DESIGN Design a Butterworth IIR filter.
```

`HD = DESIGN(D, 'butter')` designs a Butterworth filter specified by the `FDESIGN` object `H`.

`HD = DESIGN(..., 'FilterStructure', STRUCTURE)` returns a filter with the structure `STRUCTURE`. `STRUCTURE` is `'df2sos'` by default and can be any of the following.

```
'df1sos'  
'df2sos'  
'df1tsos'  
'df2tsos'
```

`HD = DESIGN(..., 'MatchExactly', MATCH)` designs a Butterworth filter and matches the frequency and magnitude specification for the band `MATCH` exactly. The other band will exceed the specification. `MATCH` can be `'stopband'` or `'passband'` and is `'stopband'` by default.

## See Also

`design`, `designmethods`, `fdesign`

# dfilt

---

**Purpose** Discrete-time filters

**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>
<code>dfilt.allpass</code>	Allpass filter
<code>dfilt.cascadeallpass</code>	Cascade of allpass filter sections
<code>dfilt.cascadewdfallpass</code>	Cascade of allpass wave digital filters
<code>dfilt.delay</code>	Delay
<code>dfilt.df1</code>	Direct-form I
<code>dfilt.df1sos</code>	Direct-form I, second-order sections
<code>dfilt.df1t</code>	Direct-form I transposed
<code>dfilt.df1tsos</code>	Direct-form I transposed, second-order sections
<code>dfilt.df2</code>	Direct-form II

<b>dfilt.structure</b>	<b>Description</b>
dfilt.df2sos	Direct-form II, second-order sections
dfilt.df2t	Direct-form II transposed
dfilt.df2tsos	Direct-form II transposed, second-order sections
dfilt.dffir	Direct-form FIR
dfilt.dffirt	Direct-form FIR transposed
dfilt.dfsymfir	Direct-form symmetric FIR
dfilt.dfasymfir	Direct-form antisymmetric FIR
dfilt.fftfir	Overlap-add FIR
dfilt.latticeallpass	Lattice allpass
dfilt.latticear	Lattice autoregressive (AR)
dfilt.latticearma	Lattice autoregressive moving- average (ARMA)
dfilt.latticemamax	Lattice moving-average (MA) for maximum phase
dfilt.latticemamin	Lattice moving-average (MA) for minimum phase
dfilt.calattice	Coupled, allpass lattice
dfilt.calatticepc	Coupled, allpass lattice with power complementary output
dfilt.statespace	State-space
dfilt.scalar	Scalar gain object
dfilt.wdfallpass	Allpass wave digital filter object

# dfilt

<b>dfilt.structure</b>	<b>Description</b>
<code>dfilt.cascade</code>	Filters arranged in series
<code>dfilt.parallel</code>	Filters arranged in parallel

For more information on each structure, refer to its reference page.

`hd = design(d, 'designmethod')` returns the `dfilt` object `hd` resulting from the filter specification object `d` and the design method you specify in *designmethod*. When you omit the *designmethod* argument, `design` uses the default design method to construct a filter from the object `d`.

With this syntax, you design filters by

- 1 Specifying the filter specifications, such as the response shape (perhaps highpass) and details (passband edges and attenuation).
- 2 Selecting a method (such as `equiripple`) to design the filter.
- 3 Applying the method to the specifications object with `design(d, 'designmethod')`.

Using the specification-based technique can be more effective than the coefficient-based filter design techniques.

## Design Methods for design Syntax

When you use the `hd = design(d, 'designmethod')` syntax, you have a range of design methods available depending on `d`, the filter specification object. The table below lists all of the design methods in the toolbox.

<b>Design Method String</b>	<b>Filter Design Result</b>
<code>butter</code>	Butterworth IIR
<code>cheby1</code>	Chebyshev Type I IIR
<code>cheby2</code>	Chebyshev Type II IIR
<code>ellip</code>	Elliptic IIR

<b>Design Method String</b>	<b>Filter Design Result</b>
equiripple	Equiripple with the same ripple in the pass and stopbands
firls	Least-squares FIR
fregsamp	Frequency-Sampled FIR
ifir	Interpolated FIR
iirlpnorm	Least Pth norm IIR
iirls	Least-Squares IIR
kaiserwin	Kaiser-windowed FIR
multistage	Multistage FIR
window	Windowed FIR

As specifications object `d` changes, the methods that apply for designing filters from `d` change. For instance, if `d` is a lowpass filter, these are the applicable methods:

```
d=fdesign.lowpass % Create an object to design a lowpass filter.
```

```
d =
```

```

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

```

```
designmethods(d) % What design methods apply to object d?
```

```
Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

When `d` is a bandstop filter, the design methods change.

```
d=fdesign.bandstop % Create a default bandstop specifications
object.
```

```
d =
```

```
           Response: 'Bandstop'
Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
Description: {7x1 cell}
NormalizedFrequency: true
           Fpass1: 0.35
           Fstop1: 0.45
           Fstop2: 0.55
           Fpass2: 0.65
           Apass1: 1
           Astop: 60
           Apass2: 1
```

```
designmethods(d) % Find out which design methods apply to d.
```

```
Design Methods for class fdesign.bandstop
(Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2):
```

```
butter
cheby1
cheby2
ellip
```



---

equiripple  
kaiserwin

Notice that `ifir` and `multistage` design methods do not apply to this bandstop specifications object `d`.

### Analysis Methods

Methods provide ways of performing functions directly on your `dfilt` object without having to specify the filter parameters again. You can apply these methods directly on the variable you assigned to your `dfilt` object.

For example, if you create a `dfilt` object, `hd`, you can check whether it has linear phase with `islinphase(hd)`, view its frequency response plot with `fvtool(hd)`, or obtain its frequency response values with `h = freqz(hd)`. You can use all of the methods below 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 below have the same name as functions in the Signal Processing or Filter Design Toolboxes. They behave similarly.

<b>Method</b>	<b>Description</b>
addstage	Adds a stage to a cascade or parallel object, where a stage is a separate, modular filter. Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .
block	(Available only with the 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 the 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

<b>Method</b>	<b>Description</b>
<code>fcfwrite</code>	<p>Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. If the Filter Design Toolbox 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 directory. 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:  'hex' for hexadecimal  'dec' for decimal  'bin' for binary representation.</p>
<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.
<code>isallpass</code>	Returns a logical 1 (i.e., true) if the <code>dfilt</code> object in an allpass filter or a logical 0 (i.e., false) if it is not

<b>Method</b>	<b>Description</b>
<code>iscascade</code>	Returns a logical 1 if the <code>dfilt</code> object is cascaded or a logical 0 if it is not
<code>isfir</code>	Returns a logical 1 if the <code>dfilt</code> object has finite impulse response (FIR) or a logical 0 if it does not
<code>islinphase</code>	Returns a logical 1 if the <code>dfilt</code> object is linear phase or a logical 0 if it is not
<code>ismaxphase</code>	Returns a logical 1 if the <code>dfilt</code> object is maximum-phase or a logical 0 if it is not
<code>isminphase</code>	Returns a logical 1 if the <code>dfilt</code> object is minimum-phase or a logical 0 if it is not
<code>isparallel</code>	Returns a logical 1 if the <code>dfilt</code> object has parallel stages or a logical 0 if it does not
<code>isreal</code>	Returns a logical 1 if the <code>dfilt</code> object has real-valued coefficients or a logical 0 if it does not
<code>isscalar</code>	Returns a logical 1 if the <code>dfilt</code> object is a scalar or a logical 0 if it is not scalar
<code>issos</code>	Returns a logical 1 if the <code>dfilt</code> object has second-order sections or a logical 0 if it does not
<code>isstable</code>	Returns a logical 1 if the <code>dfilt</code> object is stable or a logical 0 if it are not
<code>nsections</code>	Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using <code>nsections</code> returns the total number of sections in all the stages (a stage with a single section returns 1).
<code>nstages</code>	Returns the number of stages of the filter, where a stage is a separate, modular filter
<code>nstates</code>	Returns the number of states for an object

<b>Method</b>	<b>Description</b>
order	Returns the filter order. If <code>hd</code> is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If <code>hd</code> 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 <code>dfilt</code> filters. Refer to <code>dfilt.parallel</code> .
phasez	Plots the phase response in <code>fvtool</code>

Method	Description
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 new model. Valid values are 'Current' and 'New'.</p> <p>'OverwriteBlock' specifies whether to overwrite an existing block that was created by realizemdl or create a new block. Valid values are 'on' and 'off'. Note that only blocks created by realizemdl are overwritten.</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>

Method	Description
removestage	Removes a stage from a cascade or parallel dfilt. Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .
setstage	Overwrites a stage of a cascade or parallel dfilt. Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .
sos	<p>Converts the dfilt to a second-order sections dfilt. 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	Converts the dfilt to state-space. To see the separate A,B,C,D matrices for the state-space model, use <code>[A,B,C,D]=ss(hd)</code> .

Method	Description
stepz	Plots the step response in fvtool  stepz(hd,n) computes the first n samples of the step response.  stepz(hd,n,Fs) separates the time samples by $T = 1/Fs$ , where Fs is assumed to be in Hz.
tf	Converts the dfilt to a transfer function
zerophase	Plots the zero-phase response in fvtool
zpk	Converts the dfilt to zeros-pole-gain form
zplane	Plots a pole-zero plot in fvtool

## Viewing Properties

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

```
get(hd, 'property')
```

To see all properties for an object, use

```
get(hd)
```

---

**Note** If you have the Filter Design Toolbox, `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 the Filter Design Toolbox and the Fixed-Point Toolbox, you can change the `arithmetic` property to fixed-point using:

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

---



## Changing Properties

To set specific properties, use

```
set(hd, 'property1', value, 'property2', value, ...)
```

Note that you must use single quotation marks around the property name. Use single quotation marks around the value argument when the value is a string, such as `specifyall` or `fixed`.

## Copying an Object

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

```
h2 = copy(hd)
```

---

**Note** Using the syntax `H2 = hd` copies only the object handle and does not create a new, independent object.

---

## Converting Between Filter Structures

To change the filter structure of a `dfilt` object `hd`, use

```
hd2 = convert(hd, 'structure_string');
```

where `structure_string` is any valid structure name in single quotation marks. If `hd` is a `cascade` or `parallel` structure, each stage is converted to the new structure.

## Using Filter States

Two properties control the filter states:

- `states`—stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions. For `df1`, `df1t`, `df1sos` and `df1tsos` structures, `states` returns a `filtstates` object.
- `PersistentMemory`—controls whether filter states are saved. The default value is `'false'`, which causes the initial conditions to be reset to zero before filtering and turns off the display of states information. Setting `PersistentMemory` to `'true'` allows the filter to use your initial conditions

or to reuse the final conditions from a previous filtering operation as the initial conditions of the next filtering operation. The `true` setting also displays information about the filter states.

---

**Note** If you set the states and want to use them for filtering, you must set `PersistentMemory` to `'true'` before you use the filter.

---

## Examples

Create a direct-form I filter and use a method to see if it is stable.

```
[b,a] = butter(8,0.25);
hd = dfilt.df1(b,a)

hd =
    FilterStructure: 'Direct-Form I'
      Numerator: [1x9 double]
      Denominator: [1x9 double]
 PersistentMemory: false

isstable(hd)
ans =
     1
```

If a `dfilt`'s numerator values do not fit on a single line, a description of the vector is displayed. To see the specific numerator values for this example, use

```
get(hd, 'numerator')

ans =
Columns 1 through 6
    0.0001    0.0009    0.0030    0.0060    0.0076    0.0060
Columns 7 through 9
    0.0030    0.0009    0.0001
```

Create an array containing two `dfilt` objects, apply a method and verify that the method acts on both objects, and use a method to test whether the objects are FIR objects.

```
b = fir1(5,.5);
hd = dfilt.dffir(b);           % create an FIR object
```

```

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

[h,w] = freqz(hd);
size(h)                            % Verify that resulting h is
ans =                               % 2 columns.
      8192                2
size(w)                            % Verify that resulting w is
ans =                               % 1 column.
      8192                1

test_fir = isfir(hd)
test_fir =
      1      0                    % hd(1) is FIR and hd(2) is not.

```

Refer to the reference pages for each structure for more examples.

## See Also

dfilt, design, fdesign, realizemdl, sos, stepz  
dfilt.cascade, dfilt.df1, dfilt.df1t, dfilt.df2, dfilt.df2t,  
dfilt.dfasymfir, dfilt.dffir, dfilt.dffirt, dfilt.dfsymfir,  
dfilt.latticeallpass, dfilt.latticear, dfilt.latticearma,  
dfilt.latticemamax, dfilt.latticemamin, dfilt.parallel,  
dfilt.statespace, filter, freqz, grpdelay, impz, zplane in the Signal  
Processing Toolbox documentation

# dfilt.allpass

---

**Purpose** Construct allpass filter object

**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. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to <code>filtstates</code> in your 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];
hd = dfilt.allpass(c) % Creates a second-order dfilt object.
hd =
```

```
    FilterStructure: 'Minimum-Multiplier Allpass'
  AllpassCoefficients: [1.5 0.7]
    PersistentMemory: false
           States: [0;0;0;0]
```

```
info(hd) % Gets information about the filter.
Discrete-Time IIR Filter (real)
```

```
-----
Filter Structure      : Minimum-Multiplier Allpass
Number of Multipliers : 2
Stable               : Yes
Linear Phase         : No
```

```
Implementation Cost
Number of Multipliers : 2
Number of Adders      : 4
```

# dfilt.allpass

---

Number of States : 4  
MultPerInputSample : 2  
AddPerInputSample : 4

## See Also

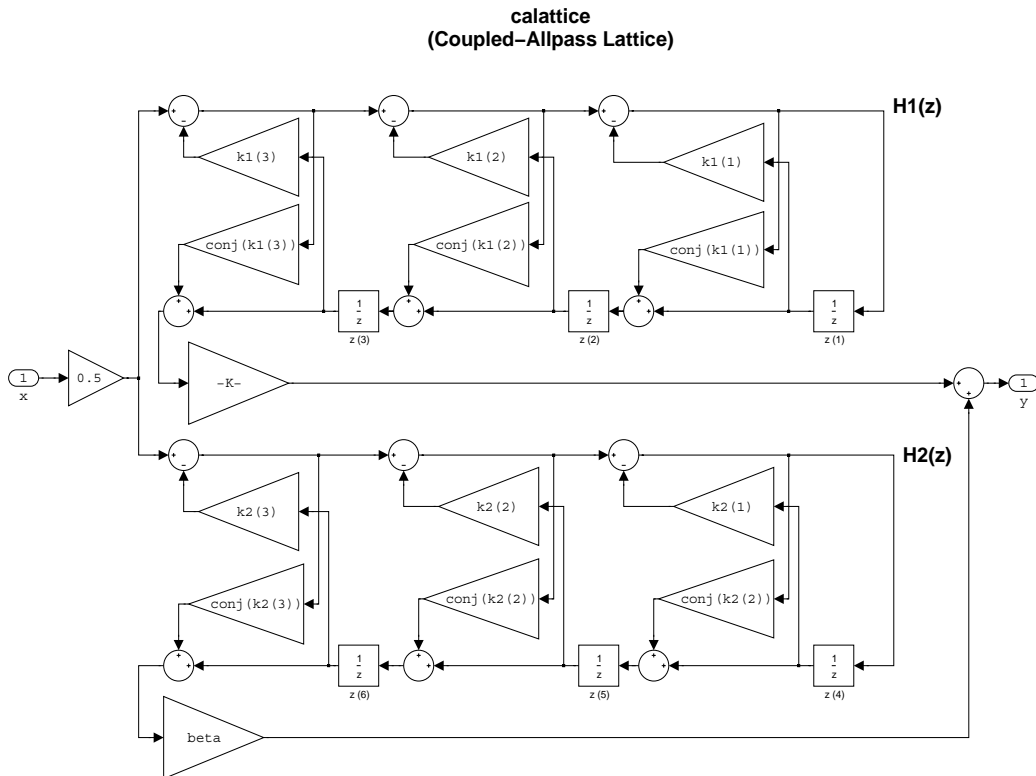
dfilt, dfilt.cascadeallpass, dfilt.cascadewdfallpass,  
dfilt.latticeallpass, mfilt.iirdecim, mfilt.iirinterp

**Purpose** Construct discrete-time, coupled-allpass, lattice filter object

**Syntax** `hd = dfilt.calattice(k1,k2,beta)`  
`hd = dfilt.calattice`

**Description** `hd = dfilt.calattice(k1,k2,beta)` returns a discrete-time, coupled-allpass, lattice filter object `hd`, which is two allpass, lattice filter structures coupled together. The lattice coefficients for each structure are vectors `k1` and `k2`. Input argument `beta` is shown in the diagram below.

`hd = dfilt.calattice` returns a default, discrete-time coupled-allpass, lattice filter object, `hd`. The default values are `k1 = k2 = []`, which is the default value for `dfilt.latticeallpass`, and `beta = 1`. This filter passes the input through to the output unchanged.



## Example

Specify a third-order lattice coupled-allpass filter structure for a `dfilt` filter, `hd` with the following code.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i]
k2 = 0.7502 - 0.1218i
beta = 0.1385 + 0.9904i
hd = dfilt.calattice(k1,k2,beta)
```

```
k1 =
    0.9511 + 0.3088i
    0.7511 + 0.1158i
```



```
k2 =  
    0.7502 - 0.1218i
```

```
beta =  
    0.1385 + 0.9904i
```

```
hd =  
    FilterStructure: 'Coupled-Allpass Lattice'  
    Allpass1: [2x1 double]  
    Allpass2: 0.7502- 0.1218i  
    Beta: 0.1385+ 0.9904i  
    PersistentMemory: false  
    States: [3x1 double]
```

Notice that the Allpass1 and Allpass2 properties store vectors of coefficients.

```
hd.Allpass1  
  
ans =  
    0.9511 + 0.3088i  
    0.7511 + 0.1158i
```

## See Also

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

# dfilt.calatticepc

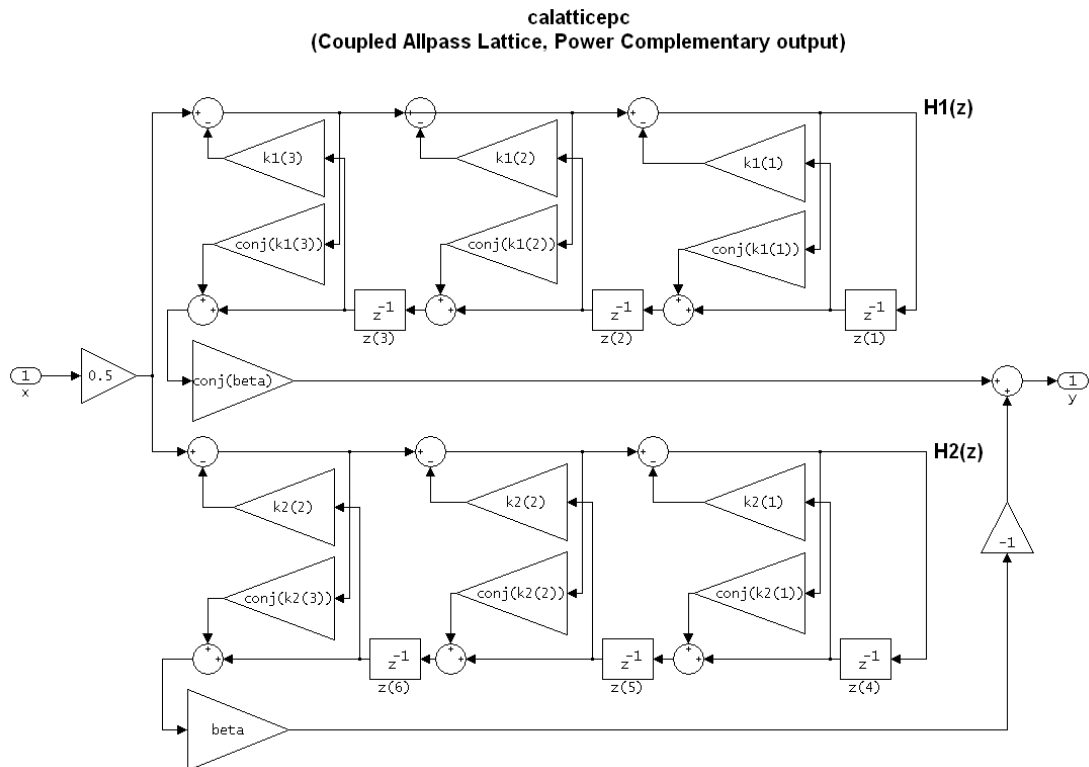
---

**Purpose** Construct discrete-time, coupled-allpass, power-complementary lattice filter object

**Syntax** `hd = dfilt.calatticepc(k1,k2,beta)`  
`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 diagram below

`hd = dfilt.calatticepc` returns a default, discrete-time, coupled-allpass, lattice filter object `hd`, with power-complementary output. The default values are `k1=k2=[]`, which is the default value for the `dfilt.latticeallpass`. The default for `beta=1`. This filter passes the input through to the output unchanged.

**Example**

Specify a third-order lattice coupled-allpass power complementary filter structure for a filter  $hd$  with the following code. You see from the returned properties that Allpass1 and Allpass2 contain vectors of coefficients for the constituent filters.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i]
k2 = 0.7502 - 0.1218i
beta = 0.1385 + 0.9904i
hd = dfilt.calatticepc(k1,k2,beta)
k1 =
```

```
0.9511 + 0.3088i
0.7511 + 0.1158i
```

```
k2 =  
  
    0.7502 - 0.1218i  
  
beta =  
  
    0.1385 + 0.9904i  
  
hd =  
  
    FilterStructure: 'Coupled-Allpass Lattice, Power  
Complementary Output'  
    Allpass1: [2x1 double]  
    Allpass2: 0.7502- 0.1218i  
    Beta: 0.1385+ 0.9904i  
    PersistentMemory: false  
    States: [3x1 double]
```

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

```
get(hd, 'Allpass1')  
  
ans =  
  
    0.9511 + 0.3088i  
    0.7511 + 0.1158i
```

## See Also

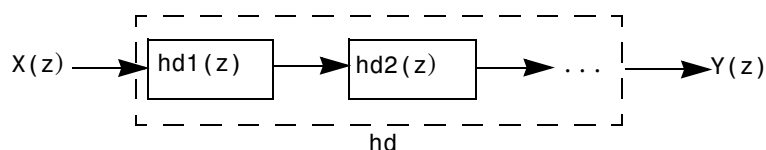
`dfilt.calattice`  
`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticearma`,  
`dfilt.latticemax`, `dfilt.latticemamin` in your Signal Processing Toolbox  
documentation

- Purpose** Construct cascade of discrete-time filter objects
- Syntax** Refer to `dfilt.cascade` in the Signal Processing Toolbox 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.

You can use the standard notation to cascade one or more filters:

```
cascade(hd1,hd2,...)
```

where `hd1`, `hd2`, and so on can be mixed types, such as `dfilt` objects and `mfilt` objects.



`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the cascade must be the same arithmetic format—double, single, or fixed. `hd`, the filter object returned, inherits the format of the cascaded filters.

- Examples** Cascade a lowpass filter and a highpass filter to produce a bandpass filter.

```
[b1,a1]=butter(8,0.6);           % Lowpass
[b2,a2]=butter(8,0.4,'high');    % Highpass
h1=dfilt.df2t(b1,a1);
h2=dfilt.df2t(b2,a2);
hcas=dfilt.cascade(h1,h2)       % Bandpass with passband 0.4-0.6

hcas =
    Filterstructure: Cascade
      Section(1): Direct Form II Transposed
      Section(2): Direct Form II Transposed
 PersistentMemory: false
```

# dfilt.cascade

---

To view the details of one filter section, use

```
hcas.section(1)
ans =
    FilterStructure: 'Direct Form II Transposed'
    Arithmetic: 'double'
    Numerator: [1x9 double]
    Denominator: [1x9 double]
    PersistentMemory: false
    States: [8x1 double]
```

## See Also

dfilt, dfilt.parallel, dfilt.scalar

**Purpose** Construct cascade of allpass discrete-time filter objects

**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.

# dfilt.cascadeallpass

Property Name	Brief Description
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to <code>filtstates</code> in your 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 `cascadeallpas` filter object.

```
tw = 100; % Transition width of filter to be designed, 100 Hz.
ast = 80; % Stopband attenuation of filter to be designed, 80dB.
fs = 2000; % Sampling frequency of signal to be filtered.
% Store halfband design specs in the specifications object d.
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

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

```
hd = design(d,'ellip','filterstructure','cascadeallpass');
% Get summary information about one dfilt.cascadeallpass stage.
hd.Stage(2).Stage(1)
ans =

FilterStructure: 'CascadeMinimum-MultiplierAllpass'
AllpassCoefficients: Section1: [0 0.0602973909571244]
                    Section2: [0 0.412590720361056]
                    Section3: [0 0.772715653742923]
```



```
PersistentMemory: false
States: [0;0;0;0;0;0;0;0]
NumSamplesProcessed: 0

hd

hd =

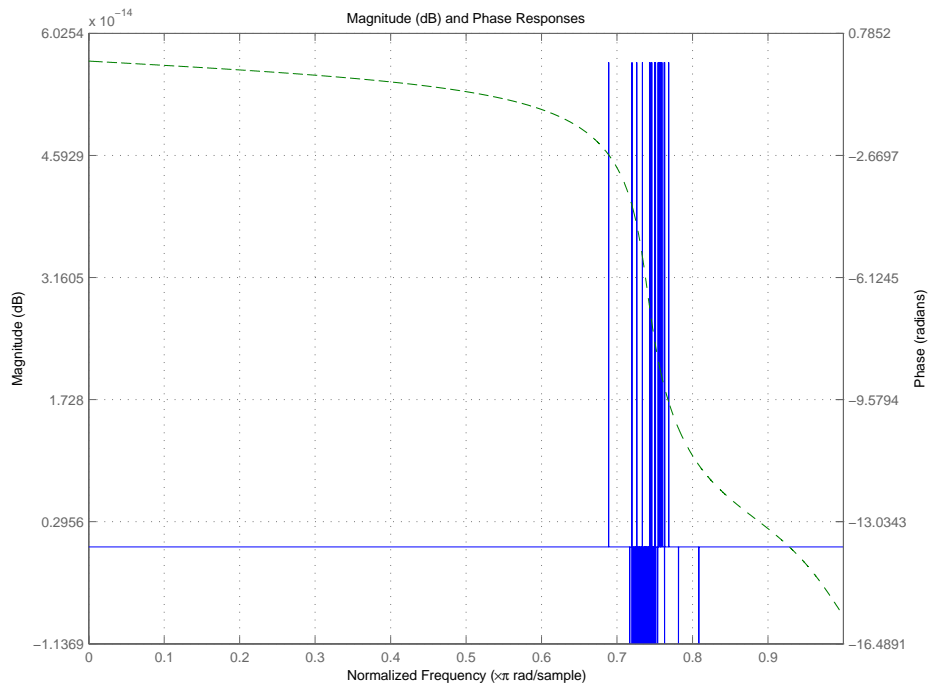
    FilterStructure: Cascade
      Stage(1): Scalar
      Stage(2): Parallel
        Stage(1): Cascade Minimum-Multiplier Allpass
        Stage(2): Cascade
          Stage(1): Delay
          Stage(2): Cascade Minimum-Multiplier Allpass
    PersistentMemory: false
```

This second example constructs a `dfilt.cascadeallpass` filter object directly given allpass coefficients for the input vectors.

```
section1 = 0.8;
section2 = [1.2,0.7];
section3 = [1.3,0.9];
hd = dfilt.cascadeallpass(section1,section2,section3);
info(hd) % Get information about the filter.
fvtool(hd) % Visualize the filter.
```

`hd` looks like this, showing both the magnitude and phase responses in FVTool. Note the units for the magnitude response on the left  $y$ -axis. Clearly this is an allpass filter.

# dfilt.cascadeallpass



## See Also

`dfilt`, `dfilt.allpass`, `dfilt.cascadewdfallpass`, `mfilt.iirdecim`,  
`mfilt.iirinterp`

**Purpose** Construct allpass wave digital filter (WDF) object by cascading allpass WDF filter objects

**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`.

# dfilt.cascadewdfallpass

## Properties

In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
AllpassCoefficients	Contains the coefficients for the allpass wave digital filter object
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to <code>filtstates</code> in your 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 to design, 100 Hz.  
ast = 80; % Stopband attenuation of filter to design, 80 dB.  
fs = 2000; % Sampling frequency of signal to filter.  
d = fdesign.halfband('tw,ast',tw,ast,fs); % Store halfband specs.
```

Now perform the actual halfband design process. `hd` contains two `dfilt.cascadewdfallpass` filters.

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

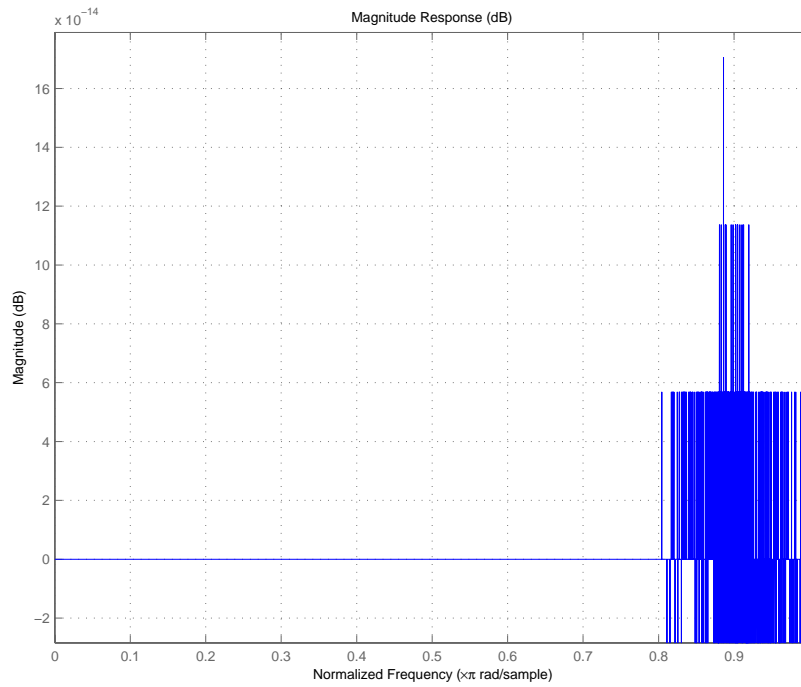
This example demonstrates direct construction of a `dfilt.cascadewdfallpass` filter with allpass coefficients.

```
section1 = 0.8;
section2 = [1.5,0.7];
section3 = [1.8,0.9];
hd = dfilt.cascadewdfallpass(section1,section2,section3);
info(hd) % Show information about the filter.
fvtool(hd) % Visualize the filter.
```

Using `FVTool` lets you view the filter response.

# dfilt.cascadewdfallpass

---

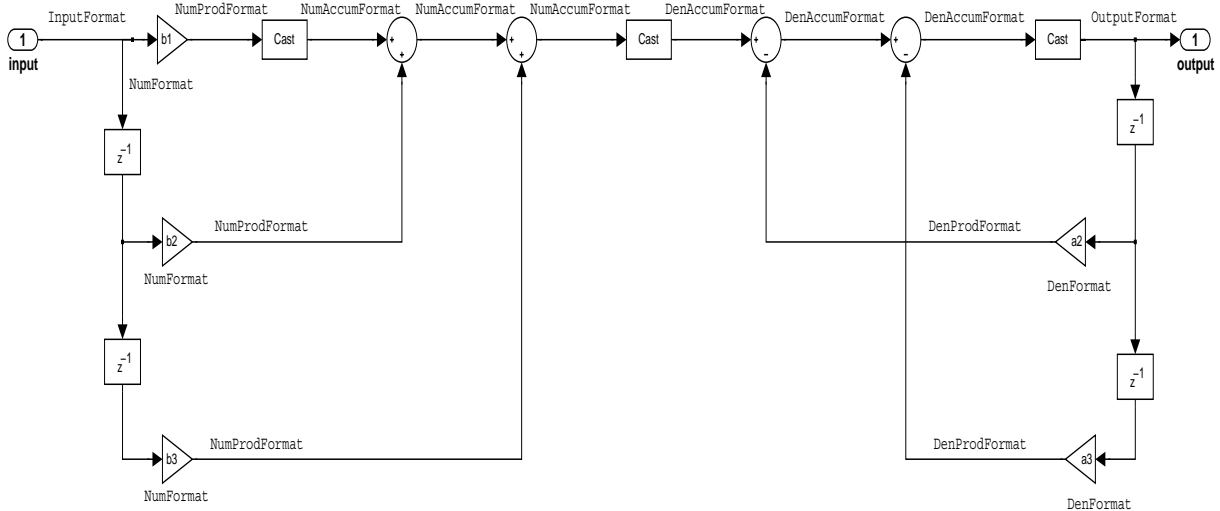


## See Also

dfilt, dfilt.wdfallpass

---

<b>Purpose</b>	Construct discrete-time, direct-form I filter object
<b>Syntax</b>	Refer to <code>dfilt.df1</code> in the Signal Processing Toolbox.
<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 <code>set(hd,'arithmetic','single');</code></li><li>• To change to fixed-point filtering, enter <code>set(hd,'arithmetic','fixed');</code></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “<code>Arithmetic</code>” on page 7-20.</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 figure below 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 flow 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.

You see that 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	Word Length Property	Fraction Length Property	Related Properties
DenAccumFormat	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFormat	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
InputFormat	InputWordLength	InputFracLength	
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 7-3.

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.
<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>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.
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.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputWordLength	Determines the word length used for the output data.

<b>Property Name</b>	<b>Brief Description</b>
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	<p data-bbox="691 305 1267 430">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 data-bbox="691 461 1285 1055" style="list-style-type: none"><li data-bbox="691 461 1285 522">• <code>convergent</code>—Round up to the next allowable quantized value.</li><li data-bbox="691 534 1285 730">• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li data-bbox="691 743 1285 835">• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li data-bbox="691 847 1285 907">• <code>floor</code>—Round down to the next allowable quantized value.</li><li data-bbox="691 920 1285 1055">• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</li></ul> <p data-bbox="691 1086 1267 1242">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. Notice that the states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in your Signal Processing Toolbox documentation or in the Help system.

## Examples

Specify a second-order direct-form I structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1(b,a)
hd =
```

```

    FilterStructure: 'Direct-Form I'
      Arithmetic: 'double'
      Numerator: [0.3000 0.6000 0.3000]
      Denominator: [1 0 0.2000]
 PersistentMemory: false
      States: Numerator: [2x1 double]
            Denominator:[2x1 double]
```

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

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

hd =
```

```

    FilterStructure: 'Direct-Form I'
      Arithmetic: 'fixed'
```

# dfilt.df1

---

```
    Numerator: [0.3000 0.6000 0.3000]
    Denominator: [1 0 0.2000]
PersistentMemory: false
    States: Numerator: [2x1 fi]
           Denominator:[2x1 fi]
```

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

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

```
    OutputWordLength: 16
    OutputFracLength: 15
```

```
    ProductMode: 'FullPrecision'
```

```
    AccumMode: 'KeepMSB'
    AccumWordLength: 40
    CastBeforeSum: true
```

```
    RoundMode: 'convergent'
    OverflowMode: 'wrap'
```

## See Also

dfilt, dfilt.df1t, dfilt.df2, dfilt.df2t



**Purpose** Construct discrete-time, direct-form I filter object that uses second-order sections

**Syntax** Refer to `dfilt.df1sos` in the Signal Processing Toolbox.

**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 7-20.

`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.

---



the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

<b>Signal Flow Label</b>	<b>Word Length Property</b>	<b>Fraction Length Property</b>	<b>Related Properties</b>
DenAccumFormat	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFormat	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
DenStateFormat	DenStateWordLength	DenStateFracLength	CastBeforeSum, States
InputFormat	InputWordLength	InputFracLength	
NumAccumFormat	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFormat	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
NumStateFormat	NumStateWordLength	NumStateFracLength	States
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFormat	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 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 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 7-3.

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> .
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>
DenStateFracLength	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
DenStateWordLength	Specifies the word length used to represent the states associated with denominator coefficients in the filter.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumStateFracLength	Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
NumWordFracLength	Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.

Property Name	Brief Description
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 data-bbox="808 578 1334 708">▪ AvoidOverflow—directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li data-bbox="808 716 1334 847">▪ BestPrecision—directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li data-bbox="808 855 1334 956">▪ SpecifyPrecision—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.
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.

<b>Property Name</b>	<b>Brief Description</b>
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>convergent</code>—Round up to the next allowable quantized value.</li> <li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li> <li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li> <li>• <code>floor</code>—Round down to the next allowable quantized value.</li> <li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

Property Name	Brief Description
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Notice that the states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in your Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Specify a fixed-point, second-order section, direct-form I `dfilt` object with the following code:

```
b=[0.3 0.6 0.3];  
a=[1 0 0.2];  
hd=dfilt.df1sos(b,a)
```

```
hd =
```

```
FilterStructure: 'Direct-Form I, Second-Order Sections'  
Arithmetic: 'double'  
sosMatrix: [0.3000 0.6000 0.3000 1 0 0.2000]  
ScaleValues: [2x1 double]
```

```
PersistentMemory: false
    States: Numerator: [2x1 double]
           Denominator:[2x1 double]

hd.arithmetic='fixed'

hd =

    FilterStructure: 'Direct-Form I, Second-Order Sections'
    ScaleValues: [2x1 double]
    Arithmetic: 'fixed'
    sosMatrix: [0.3000 0.6000 0.3000 1 0 0.2000]
    PersistentMemory: false
        States: Numerator: [2x1 fi]
               Denominator:[2x1 fi]

    CoeffWordLength: 16
    CoeffAutoScale: true
    Signed: true

    InputWordLength: 16
    InputFracLength: 15

    OutputWordLength: 16
    OutputMode: 'AvoidOverflow'

    NumStateWordLength: 16
    NumStateFracLength: 15

    DenStateWordLength: 16
    DenStateFracLength: 15

    ProductMode: 'FullPrecision'

    AccumMode: 'KeepMSB'
    AccumWordLength: 40
    CastBeforeSum: true

    RoundMode: 'convergent'
    OverflowMode: 'wrap'
```

# dfilt.df1sos

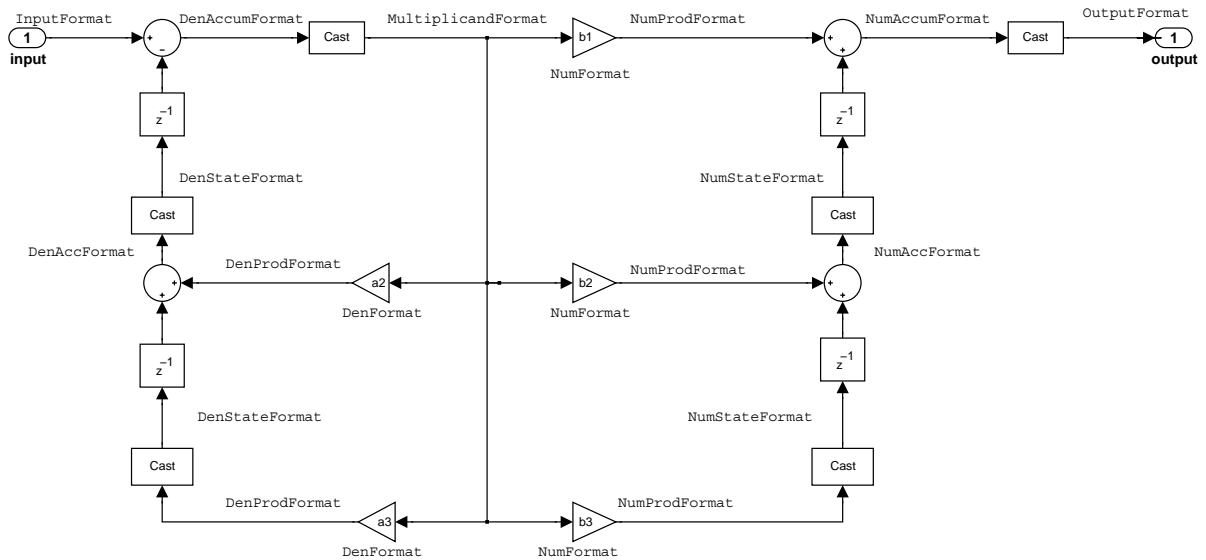
---

## See Also

dfilt, dfilt.df2tsos

---

<b>Purpose</b>	Construct discrete-time, direct-form I transposed filter object
<b>Syntax</b>	Refer to <code>dfilt.df1t</code> in the Signal Processing Toolbox.
<b>Description</b>	<p><code>hd = dfilt.df1t(b,a)</code> returns a discrete-time, direct-form I 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> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <code>set(hd,'arithmetic','single');</code></li><li>• To change to fixed-point filtering, enter <code>set(hd,'arithmetic','fixed');</code></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “Arithmetic” on page 7-20.</p> <p><code>hd = dfilt.df1t</code> returns a default, discrete-time, direct-form I transposed filter object <code>hd</code>, with <code>b=1</code> and <code>a=1</code>. This filter passes the input through to the output unchanged.</p>
	<hr/> <p><b>Note</b> The leading coefficient of the denominator <code>a(1)</code> 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>	The figure below shows the signal flow for the transposed direct-form I filter implemented by <code>dfilt.df1t</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.



### 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 flow 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.

You see that 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>Word Length Property</b>	<b>Fraction Length Property</b>	<b>Related Properties</b>
DenAccumFormat	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFormat	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
DenStateFormat	DenStateWordLength	DenStateFracLength	CastBeforeSum, States
InputFormat	InputWordLength	InputFracLength	
MultiplicandFormat	MultiplicandWordLength	MultiplicandFracLength	CastBeforeSum
NumAccumFormat	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFormat	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
NumStateFormat	NumStateWordLength	NumStateFracLength	States
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 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 7-3.

---

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

---



<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 false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Holds the denominator coefficients for the filter.
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.

<b>Property Name</b>	<b>Brief Description</b>
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.
Numerator	Holds the numerator coefficient values for the filter.
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.
NumStateFracLength	For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
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>▪ AvoidOverflow—directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>▪ BestPrecision—directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>▪ SpecifyPrecision—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.

<b>Property Name</b>	<b>Brief Description</b>
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	<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>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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	<p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>

Property Name	Brief Description
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Specify a second-order direct-form I transposed filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1t(b,a)

hd =

    FilterStructure: 'Direct-Form I Transposed'
    Arithmetic: 'double'
    Numerator: [0.3000 0.6000 0.3000]
    Denominator: [1 0 0.2000]
    PersistentMemory: false
    States: Numerator: [2x1 double]
           Denominator:[2x1 double]
```

Now convert the filter to single-precision filtering arithmetic.

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

    FilterStructure: 'Direct-Form I Transposed'
```

```
    Arithmetic: 'fixed'
      Numerator: [0.3000 0.6000 0.3000]
      Denominator: [1 0 0.2000]
    PersistentMemory: false
      States: Numerator: [2x1 fi]
             Denominator: [2x1 fi]

    CoeffWordLength: 16
      CoeffAutoScale: true
      Signed: true

    InputWordLength: 16
    InputFracLength: 15

    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'

    MultiplicandWordLength: 16
    MultiplicandFracLength: 15

    StateWordLength: 16
      StateAutoScale: true

      ProductMode: 'FullPrecision'

      AccumMode: 'KeepMSB'
    AccumWordLength: 40
    CastBeforeSum: true

      RoundMode: 'convergent'
    OverflowMode: 'wrap'
```

**See Also**

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

# dfilt.df1tsos

---

<b>Purpose</b>	Construct discrete-time, second-order section, direct-form I transposed filter object
<b>Syntax</b>	Refer to <code>dfilt.df1tsos</code> in the Signal Processing Toolbox.
<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 <code>set(hd, 'arithmetic', 'single');</code></li><li>• To change to fixed-point filtering, enter <code>set(hd, 'arithmetic', 'fixed');</code></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “<code>Arithmetic</code>” on page 7-20.</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>

---

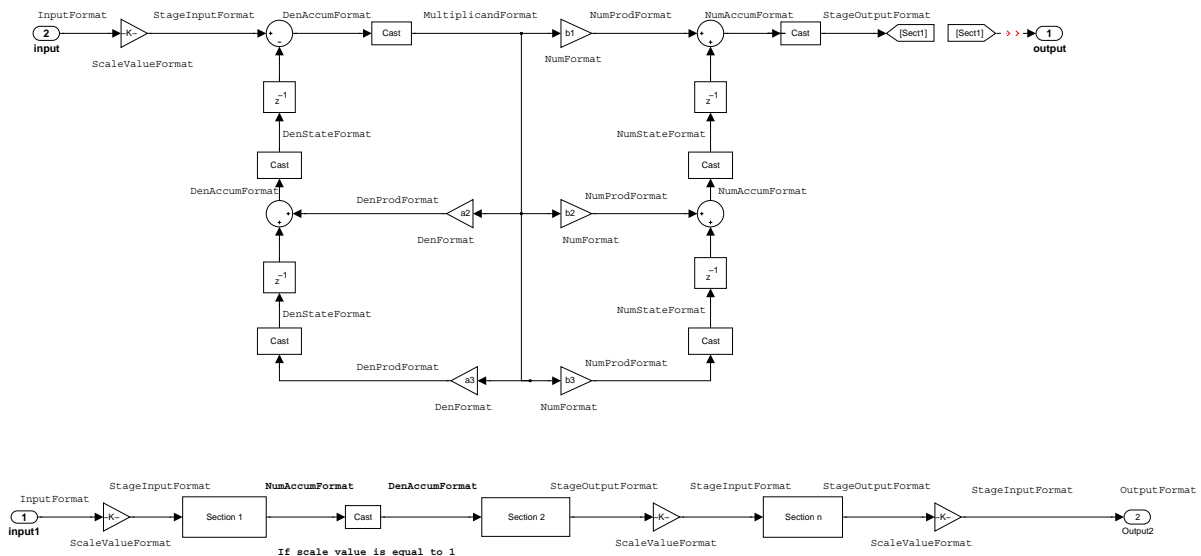
**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---



## Fixed-Point Filter Structure

The figure below 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 flow 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.

You see that 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>Word Length Property</b>	<b>Fraction Length Property</b>	<b>Related Properties</b>
DenAccumFormat	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFormat	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
DenStateFormat	DenStateWordLength	DenStateFracLength	CastBeforeSum, States
InputFormat	InputWordLength	InputFracLength	
MultiplicandFormat	MultiplicandWordLength	MultiplicandFracLength	CastBeforeSum
NumAccumFormat	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFormat	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
NumStateFormat	NumStateWordLength	NumStateFracLength	States
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFormat	CoeffWordLength	ScaleValueFracLength	CoeffAutoScale, ScaleValues
StageInputFormat	StageInputWordLength	StageInputFracLength	StageInputAutoScale
StageOutputFormat	StageOutputWordLength	StageOutputFracLength	StageOutputAutoScale

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 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 7-3.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.

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

<b>Property Name</b>	<b>Brief Description</b>
NumStateFracLength	For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
NumStateWordLength	For IIR filters, this defines the word length applied to the numerator states of the filter. Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
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>▪ AvoidOverflow—directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>▪ BestPrecision—directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>▪ SpecifyPrecision—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.

<b>Property Name</b>	<b>Brief Description</b>
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	<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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>



<b>Property Name</b>	<b>Brief Description</b>
ScaleValues	Scaling for the filter objects in SOS filters.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
StageInputAutoScale	Tells the filter whether to set the stage input data format to minimize the occurrence of overflow conditions.
StageInputFracLength	Lets you set the fraction length for stage inputs in SOS filters, if you set StageInputAutoScale to false.
StageInputWordLength	Lets you set the word length for stage inputs in SOS filters, if you set StageInputAutoScale to false.
StageOutputAutoScale	Tells the filter whether to set the stage output data format to minimize the occurrence of overflow conditions.
StageOutputFracLength	Lets you set the fraction length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.
StageOutputWordLength	Lets you set the word length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.

Property Name	Brief Description
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
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 we convert the filter to fixed-point operation.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1t(b,a)

hd =

    FilterStructure: 'Direct-Form I Transposed'
    Arithmetic: 'double'
    Numerator: [0.3000 0.6000 0.3000]
    Denominator: [1 0 0.2000]
    PersistentMemory: false
    States: Numerator: [2x1 double]
           Denominator:[2x1 double]

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

hd =
```

```
FilterStructure: 'Direct-Form I Transposed'  
  Arithmetic: 'fixed'  
    Numerator: [0.3000 0.6000 0.3000]  
    Denominator: [1 0 0.2000]  
PersistentMemory: false  
  States: Numerator: [2x1 fi]  
          Denominator: [2x1 fi]
```

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

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

```
OutputWordLength: 16  
  OutputMode: 'AvoidOverflow'
```

```
MultiplicandWordLength: 16  
MultiplicandFracLength: 15
```

```
StateWordLength: 16  
  StateAutoScale: true
```

```
  ProductMode: 'FullPrecision'
```

```
    AccumMode: 'KeepMSB'  
AccumWordLength: 40  
  CastBeforeSum: true
```

```
    RoundMode: 'convergent'  
  OverflowMode: 'wrap'
```

**See Also**

dfilt, dfilt.df1sos, dfilt.df2sos, dfilt.df2tsos

# dfilt.df2

---

**Purpose** Construct discrete-time, direct-form II filter object

**Syntax** Refer to `dfilt.df2` in the Signal Processing Toolbox.

**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 7-20.

`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 figure below 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.



# dfilt.df2

Signal Flow Label	Word Length Property	Fraction Length Property	Related Properties
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
InputFormat	InputWordLength	InputFracLength	
NumAccumFormat	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFormat	CoeffWordLength	NumProdFracLength	ProductWordLength, 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 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 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 7-3.

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.

<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 false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Holds the denominator coefficients for IIR filters.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
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.
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.

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

---

<b>Property Name</b>	<b>Brief Description</b>
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> .
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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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	<p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>

Property Name	Brief Description
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Specify a second-order direct-form II filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2(b,a)

hd =
    FilterStructure: 'Direct Form II'
        Numerator: [0.3000 0.6000 0.3000]
        Denominator: [1 0 0.2000]
    NumberOfSamplesProcessed: 0
        ResetStates: 'on'
            States: [2x1 double]
```

To convert the filter to fixed-point arithmetic, change the value of the `Arithmetic` property

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

to specify the fixed-point option.

## See Also

`dfilt`, `dfilt.df1`, `dfilt.df1t`, `dfilt.df2t`

# dfilt.df2sos

---

**Purpose** Construct discrete-time, second-order section, direct-form II filter object

**Syntax** Refer to `dfilt.df2sos` in the Signal Processing Toolbox.

**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 7-20.

`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.

---

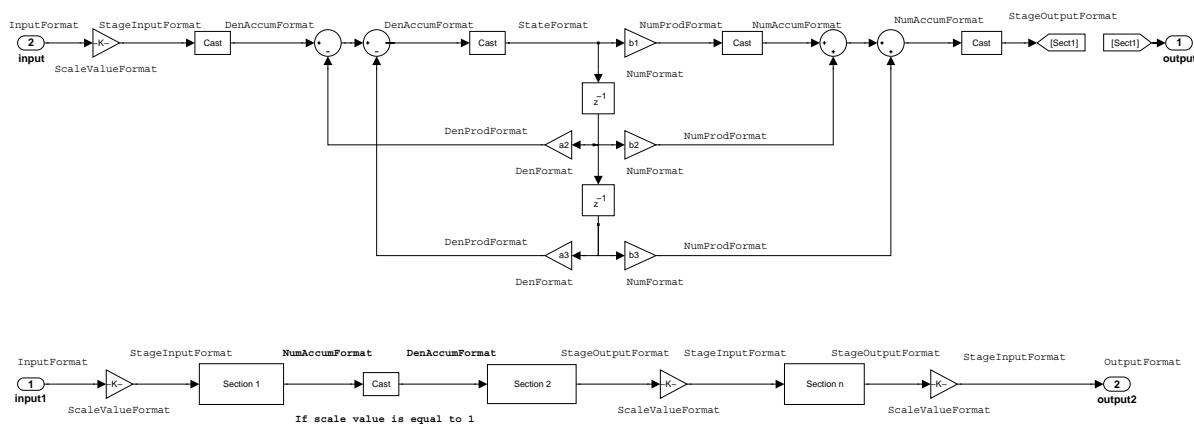
**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

## 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 flow 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.

You see that 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>Word Length Property</b>	<b>Fraction Length Property</b>	<b>Related Properties</b>
DenAccumFormat	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFormat	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, sosMatrix
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength, sosMatrix
InputFormat	InputWordLength	InputFracLength	
NumAccumFormat	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, sosMatrix
NumProdFormat	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFormat	CoeffWordLength	ScaleValueFracLength	CoeffAutoScale, ScaleValues
StageInputFormat	StageInputWordLength	StageInputFracLength	StageInputAutoScale
StageOutputFormat	StageOutputWordLength	StageOutputFracLength	StageOutputAutoScale
StateFormat	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 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 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 7-3.

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 false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.

<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 AccumMode to SpecifyPrecision.
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.

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

<b>Property Name</b>	<b>Brief Description</b>
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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

<b>Property Name</b>	<b>Brief Description</b>
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
StageInputAutoScale	Tells the filter whether to set the stage input data format to minimize the occurrence of overflow conditions.
StageInputFracLength	Lets you set the fraction length for stage inputs in SOS filters, if you set StageInputAutoScale to false.
StageInputWordLength	Lets you set the word length for stage inputs in SOS filters, if you set StageInputAutoScale to false.
StageOutputAutoScale	Tells the filter whether to set the stage output data format to minimize the occurrence of overflow conditions.
StageOutputFracLength	Lets you set the fraction length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.
StageOutputWordLength	Lets you set the word length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.

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)

hd =

    FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
      sosMatrix: [15x6 double]
    ScaleValues: [16x1 double]
 PersistentMemory: false
       States: [2x15 double]
```

With the SOS filter constructed, now change the filter operation to single-precision filtering, and then to fixed-point filtering.

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

hd =

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



```
        sosMatrix: [15x6 double]
        ScaleValues: [16x1 double]
PersistentMemory: false
        States: [2x15 single]

hd.arithmetic='fixed'

hd =

    FilterStructure: 'Direct-Form II, Second-Order Sections'
        Arithmetic: 'fixed'
        sosMatrix: [15x6 double]
        ScaleValues: [16x1 double]
PersistentMemory: false
        States: [1x1 embedded.fi]

    CoeffWordLength: 16
    CoeffAutoScale: true
    Signed: true

    InputWordLength: 16
    InputFracLength: 15

    StageInputWordLength: 16
    StageInputAutoScale: true

    StageOutputWordLength: 16
    StageOutputAutoScale: true

    OutputWordLength: 16
    OutputMode: 'AvoidOverflow'

    StateWordLength: 16
    StateFracLength: 15

    ProductMode: 'FullPrecision'

    AccumMode: 'KeepMSB'
    AccumWordLength: 40
    CastBeforeSum: true
```

## dfilt.df2sos

---

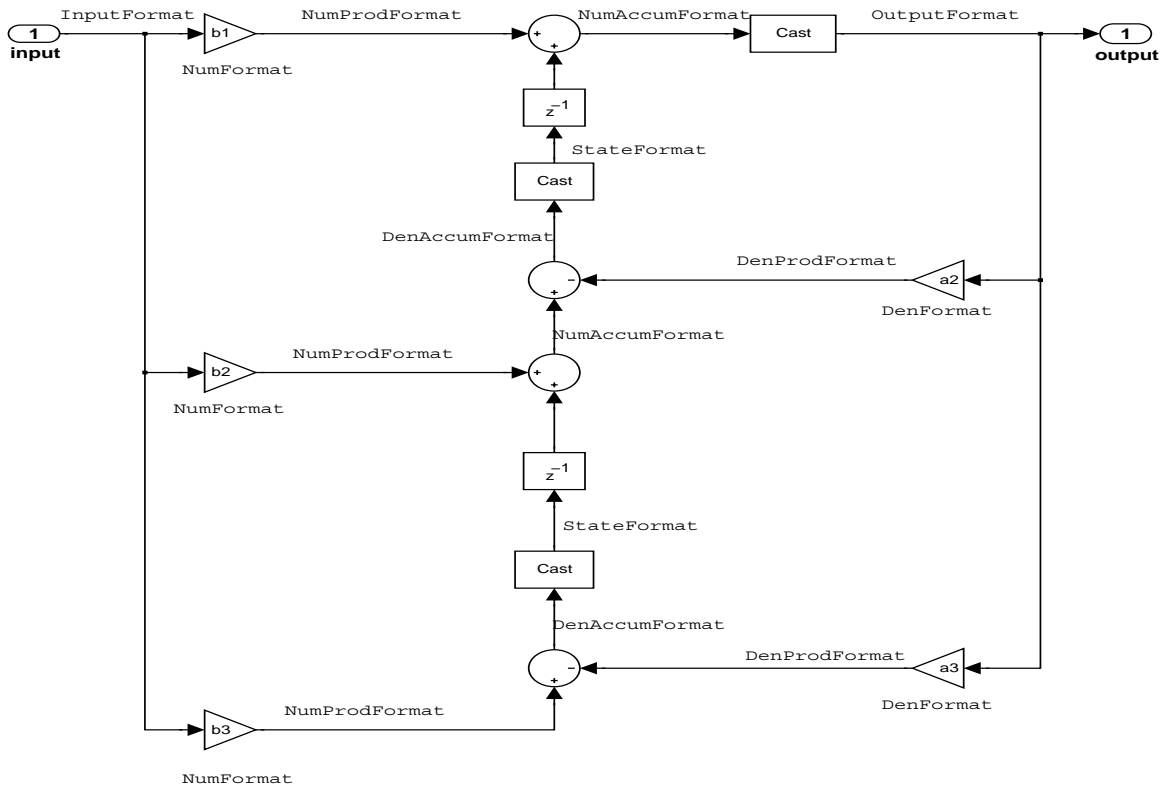
RoundMode: 'convergent'  
OverflowMode: 'wrap'

### See Also

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

---

<b>Purpose</b>	Construct discrete-time, direct-form II transposed filter object
<b>Syntax</b>	Refer to <code>dfilt.df2t</code> in the Signal Processing Toolbox.
<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> <ul style="list-style-type: none"><li>• To change to single-precision filtering, enter <code>set(hd,'arithmetic','single');</code></li><li>• To change to fixed-point filtering, enter <code>set(hd,'arithmetic','fixed');</code></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “Arithmetic” on page 7-20.</p> <p><code>hd = dfilt.df2t</code> returns a default, discrete-time, direct-form II transposed filter object <code>hd</code>, with <code>b=1</code> and <code>a=1</code>. This filter passes the input through to the output unchanged.</p>
	<hr/> <p><b>Note</b> The leading coefficient of the denominator <code>a(1)</code> 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>	The figure below shows the signal flow for the direct-form II transposed filter implemented by <code>dfilt.df2t</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.



### 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 flow 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.

You see that 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>Word Length Property</b>	<b>Fraction Length Property</b>	<b>Related Properties</b>
DenAccumFormat	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFormat	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
InputFormat	InputWordLength	InputFracLength	
NumAccumFormat	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFormat	CoeffWordLength	NumProdFracLength	ProductWordLength, 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 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 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 7-3.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.

<b>Property Name</b>	<b>Brief Description</b>
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 ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.



Property Name	Brief Description
OutputMode	<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>▪ AvoidOverflow—directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>▪ BestPrecision—directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>▪ SpecifyPrecision—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 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.</p>

<b>Property Name</b>	<b>Brief Description</b>
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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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	<p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>

Property Name	Brief Description
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Create a fixed-point filter by specifying a second-order direct-form II transposed filter structure for a `dfilt` object, and then converting the double-precision arithmetic setting to fixed-point.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2t(b,a)

hd =

    FilterStructure: 'Direct-Form II Transposed'
    Arithmetic: 'double'
    Numerator: [0.3000 0.6000 0.3000]
    Denominator: [1 0 0.2000]
    PersistentMemory: false
    States: [2x1 double]

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

```
hd
```

```
hd =
```

```
    FilterStructure: 'Direct-Form II Transposed'  
        Arithmetic: 'fixed'  
            Numerator: [0.3000 0.6000 0.3000]  
            Denominator: [1 0 0.2000]  
    PersistentMemory: false  
        States: [1x1 embedded.fi]  
  
    CoeffWordLength: 16  
        CoeffAutoScale: true  
            Signed: true  
  
    InputWordLength: 16  
    InputFracLength: 15  
  
    OutputWordLength: 16  
    OutputFracLength: 15  
  
    StateWordLength: 16  
        StateAutoScale: true  
  
        ProductMode: 'FullPrecision'  
  
            AccumMode: 'KeepMSB'  
    AccumWordLength: 40  
        CastBeforeSum: true  
  
            RoundMode: 'convergent'  
            OverflowMode: 'wrap'
```

**See Also**

dfilt, dfilt.df1, dfilt.df1t, dfilt.df2

# dfilt.df2tsos

---

**Purpose** Construct discrete-time, second-order section direct-form II transposed filter object

**Syntax** Refer to `dfilt.df2tsos` in the Signal Processing Toolbox.

**Description** `hd = dfilt.df2tsos(s)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients given in the matrix `s`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter  
`set(hd,'arithmetic','single');`
- To change to fixed-point filtering, enter  
`set(hd,'arithmetic','fixed');`

For more information about the property `Arithmetic`, refer to “Arithmetic” on page 7-20.

`hd = dfilt.df2tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`hd = dfilt.df2tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df2tsos` returns a default, discrete-time, second-order section, direct-form II, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

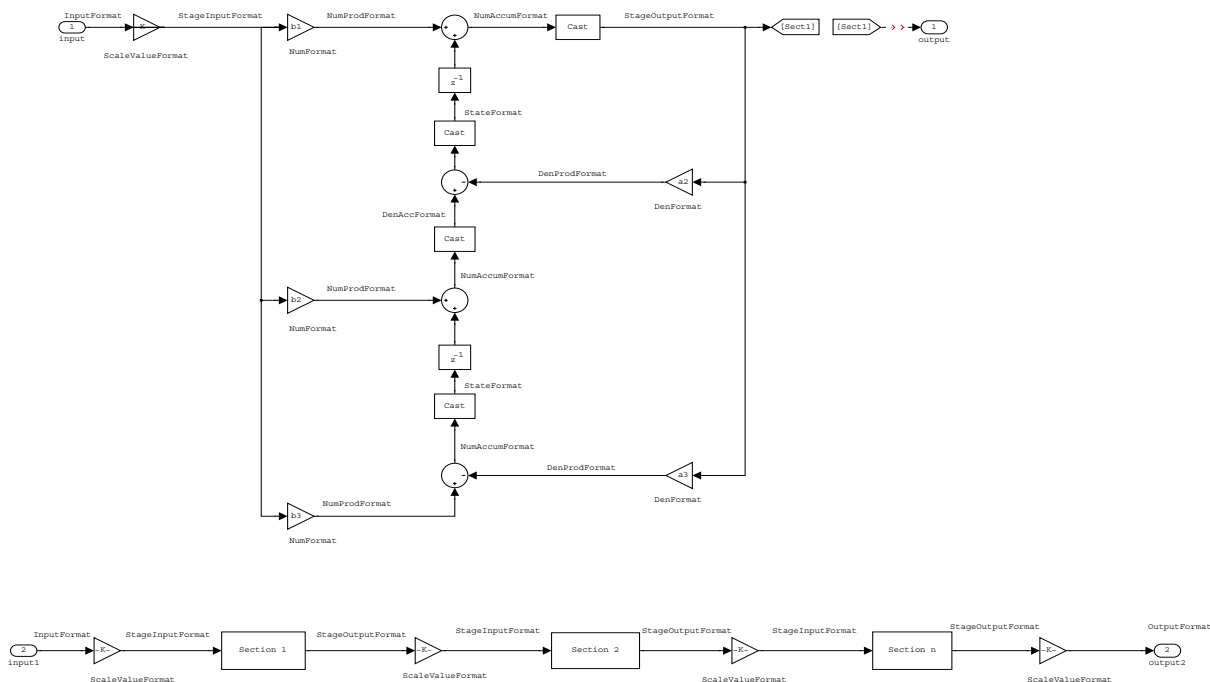
---

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

---

## Fixed-Point Filter Structure

The figure below shows the signal flow for the second-order section transposed direct-form II filter implemented by `dfilt.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 flow 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.

You see that 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	Word Length Property	Fraction Length Property	Related Properties
DenAccumFormat	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFormat	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFormat	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
InputFormat	InputWordLength	InputFracLength	
NumAccumFormat	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFormat	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFormat	CoeffWordLength	ScaleValueFracLength	CoeffAutoScale, ScaleValues
StageInputFormat	StageInputWordLength	StageInputFracLength	StageInputAutoScale
StageOutputFormat	StageOutputWordLength	StageOutputFracLength	StageOutputAutoScale
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 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 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 7-3.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.

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

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

<b>Property Name</b>	<b>Brief Description</b>
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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

<b>Property Name</b>	<b>Brief Description</b>
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—you use set and get to modify them. Displays the matrix in the format [sections x coefficients/section data type]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
StageInputFracLength	Lets you set the fraction length for stage inputs in SOS filters, if you set StageInputAutoScale to false.
StageInputWordLength	Lets you set the word length for stage inputs in SOS filters, if you set StageInputAutoScale to false.
StageOutputAutoScale	Tells the filter whether to set the stage output data format to minimize the occurrence of overflow conditions.
StageOutputFracLength	Lets you set the fraction length for stage outputs in SOS filters, if you set StageOutputAutoScale to off.
StageOutputWordLength	Lets you set the word length for stage outputs in SOS filters, if you set StageOutputAutoScale to false.

Property Name	Brief Description
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Construct a second-order section Butterworth filter for fixed-point filtering. Start by specifying a Butterworth filter, and then convert the filter to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);
[s,g] = zp2sos(z,p,k);
hd = dfilt.df2tsos(s,g)

hd =

    FilterStructure: [1x48 char]
      Arithmetic: 'double'
      sosMatrix: [15x6 double]
    ScaleValues: [16x1 double]
 PersistentMemory: false
      States: [2x15 double]
```



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

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

```
hd =
```

```
    FilterStructure: [1x48 char]
      Arithmetic: 'fixed'
      sosMatrix: [15x6 double]
      ScaleValues: [16x1 double]
    PersistentMemory: false
      States: [1x1 embedded.fi]

    CoeffWordLength: 16
      CoeffAutoScale: true
      Signed: true

    InputWordLength: 16
    InputFracLength: 15

    StageInputWordLength: 16
    StageInputFracLength: 15

    StageOutputWordLength: 16
    StageOutputFracLength: 15

    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'

    StateWordLength: 16
      StateAutoScale: true

      ProductMode: 'FullPrecision'

      AccumMode: 'KeepMSB'
    AccumWordLength: 40
      CastBeforeSum: true

      RoundMode: 'convergent'
```

## dfilt.df2tsos

---

OverflowMode: 'wrap'

### See Also

dfilt, dfilt.df1sos, dfilt.df1tsos, dfilt.df2sos

**Purpose** Construct discrete-time, direct-form antisymmetric FIR filter object

**Syntax** Refer to `dfilt.dfasymfir` in the Signal Processing Toolbox.

**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 7-20.

`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.

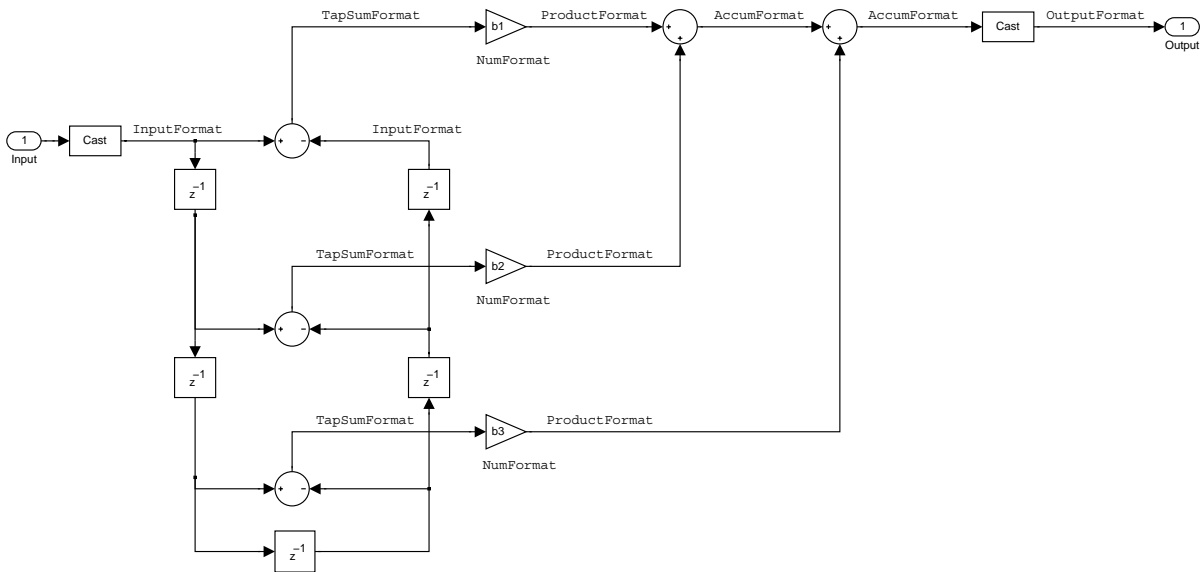
---

**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 figure below 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 flow 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.

You see that 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	Word Length Property	Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	
InputFormat	InputWordLength	InputFracLength	
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
OutputFormat	OutputWordLength	OutputFracLength	
ProductFormat	ProductWordLength	ProductFracLength	
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)`

# dfilt.dfasymfir

---

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 7-3.

<b>Name</b>	<b>Values</b>	<b>Description</b>
<code>AccumFracLength</code>	Any positive or negative integer number of bits [27]	Specifies the fraction length used to interpret data output by the accumulator.
<code>AccumWordLength</code>	Any integer number of bits [33]	Sets the word length used to store data in the accumulator.
<code>Arithmetic</code>	<code>fixed</code> for fixed-point filters	Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.
<code>CoeffAutoScale</code>	<code>[true]</code> , <code>false</code>	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.
<code>CoeffWordLength</code>	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.

# dfilt.dfasymfir

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [27]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [33]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.



Name	Values	Description
RoundMode	[convergent], ceil,fix,floor, round	<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>convergent</b>—Round up to the next allowable quantized value.</li> <li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li> <li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li> <li>• <b>floor</b>—Round down to the next allowable quantized value.</li> <li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

# dfilt.dfasymfir

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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system.

## Examples

### Odd Order

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

```
b = [-0.008 0.06 -0.44 0.44 -0.06 0.008];
hd = dfilt.dfasymfir(b)

hd =

    FilterStructure: 'Direct-Form Antisymmetric FIR'
      Arithmetic: 'double'
      Numerator: [-0.0080 0.0600 -0.4400 0.4400 -0.0600 0.0080]
 PersistentMemory: false

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

    FilterStructure: 'Direct-Form Antisymmetric FIR'
      Arithmetic: 'fixed'
      Numerator: [-0.0080 0.0600 -0.4400 0.4400 -0.0600 0.0080]
 PersistentMemory: false

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

```
InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

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

```
get(hd, 'numerator')

ans =

    -0.0080    0.0600   -0.4400    0.4400   -0.0600    0.0080
```

## Even Order

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

```
b = [-0.01 0.1 0.0 -0.1 0.01];
hd = dfilt.dfasymfir(b)

hd =

    FilterStructure: 'Direct-Form Antisymmetric FIR'
      Arithmetic: 'double'
      Numerator: [-0.0100 0.1000 0 -0.1000 0.0100]
 PersistentMemory: false
```

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

```
hd =

    FilterStructure: 'Direct-Form Antisymmetric FIR'
      Arithmetic: 'fixed'
      Numerator: [-0.0100 0.1000 0 -0.1000 0.0100]
 PersistentMemory: false

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

# dfilt.dfasymfir

---

```
InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'

get(hd, 'numerator')

ans =

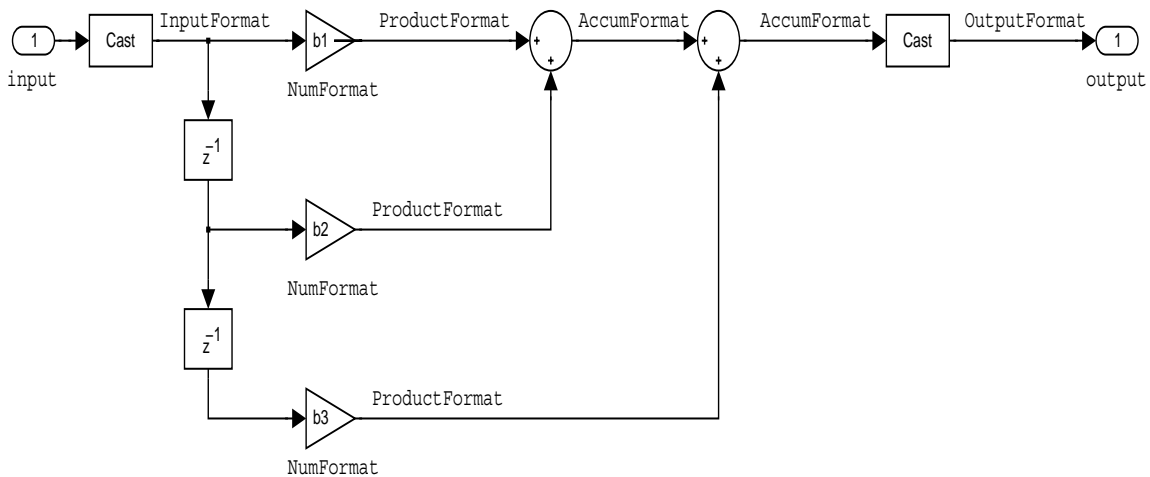
    -0.0100    0.1000         0   -0.1000    0.0100
```

## See Also

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

---

<b>Purpose</b>	Construct discrete-time direct-form FIR filter object
<b>Syntax</b>	Refer to <code>dfilt.dffir</code> in the Signal Processing Toolbox.
<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 <code>set(hd,'arithmetic','single');</code></li><li>• To change to fixed-point filtering, enter <code>set(hd,'arithmetic','fixed');</code></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “<code>Arithmetic</code>” on page 7-20.</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 figure below 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 flow 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.

You see that 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	Word Length Property	Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	
InputFormat	InputWordLength	InputFracLength	
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
OutputFormat	OutputWordLength	OutputFracLength	
ProductFormat	ProductWordLength	ProductFracLength	

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 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)`

## dfilt.dffir

---

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 7-3.

Name	Values	Description
<code>AccumFracLength</code>	Any positive or negative integer number of bits [30]	Specifies the fraction length used to interpret data output by the accumulator.
<code>AccumWordLength</code>	Any integer number of bits [34]	Sets the word length used to store data in the accumulator.
<code>Arithmetic</code>	<code>fixed</code> for fixed-point filters	Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.
<code>CoeffAutoScale</code>	<code>[true]</code> , <code>false</code>	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.
<code>CoeffWordLength</code>	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.



Name	Values	Description
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.

## dfilt.dffir

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [30]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [32]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Name	Values	Description
RoundMode	[convergent], ceil,fix,floor, round	<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>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

# dfilt.dffir

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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system.

## Examples

Specify a second-order direct-form FIR filter structure for a `dfilt` object `hd`, with the following code that constructs the filter in double-precision format and then converts the filter to fixed-point operation:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffir(b)

hd =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'double'
      Numerator: [0.0500 0.9000 0.0500]
 PersistentMemory: false

hd.arithmetic='fixed'

hd =

    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'fixed'
      Numerator: [0.0500 0.9000 0.0500]
 PersistentMemory: false

    CoeffWordLength: 16
      CoeffAutoScale: true
```

```
        Signed: true

        InputWordLength: 16
        InputFracLength: 15

        FilterInternals: 'FullPrecision'
hd.filterInternals='specifyPrecision'

hd =

        FilterStructure: 'Direct-Form FIR'
            Arithmetic: 'fixed'
            Numerator: [0.0500 0.9000 0.0500]
        PersistentMemory: false

        CoeffWordLength: 16
        CoeffAutoScale: true
        Signed: true

        InputWordLength: 16
        InputFracLength: 15

        FilterInternals: 'SpecifyPrecision'

        OutputWordLength: 34
        OutputFracLength: 30

        ProductWordLength: 32
        ProductFracLength: 30

        AccumWordLength: 34
        AccumFracLength: 30

        RoundMode: 'convergent'
        OverflowMode: 'wrap'
```

**See Also**

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

# dfilt.dffirt

---

**Purpose** Construct discrete-time, direct-form FIR transposed filter object

**Syntax** Refer to `dfilt.dffirt` in the Signal Processing Toolbox.

**Description** `hd = dfilt.dffirt(b)` returns a discrete-time, direct-form FIR transposed filter object `hd`, with numerator coefficients `b`.

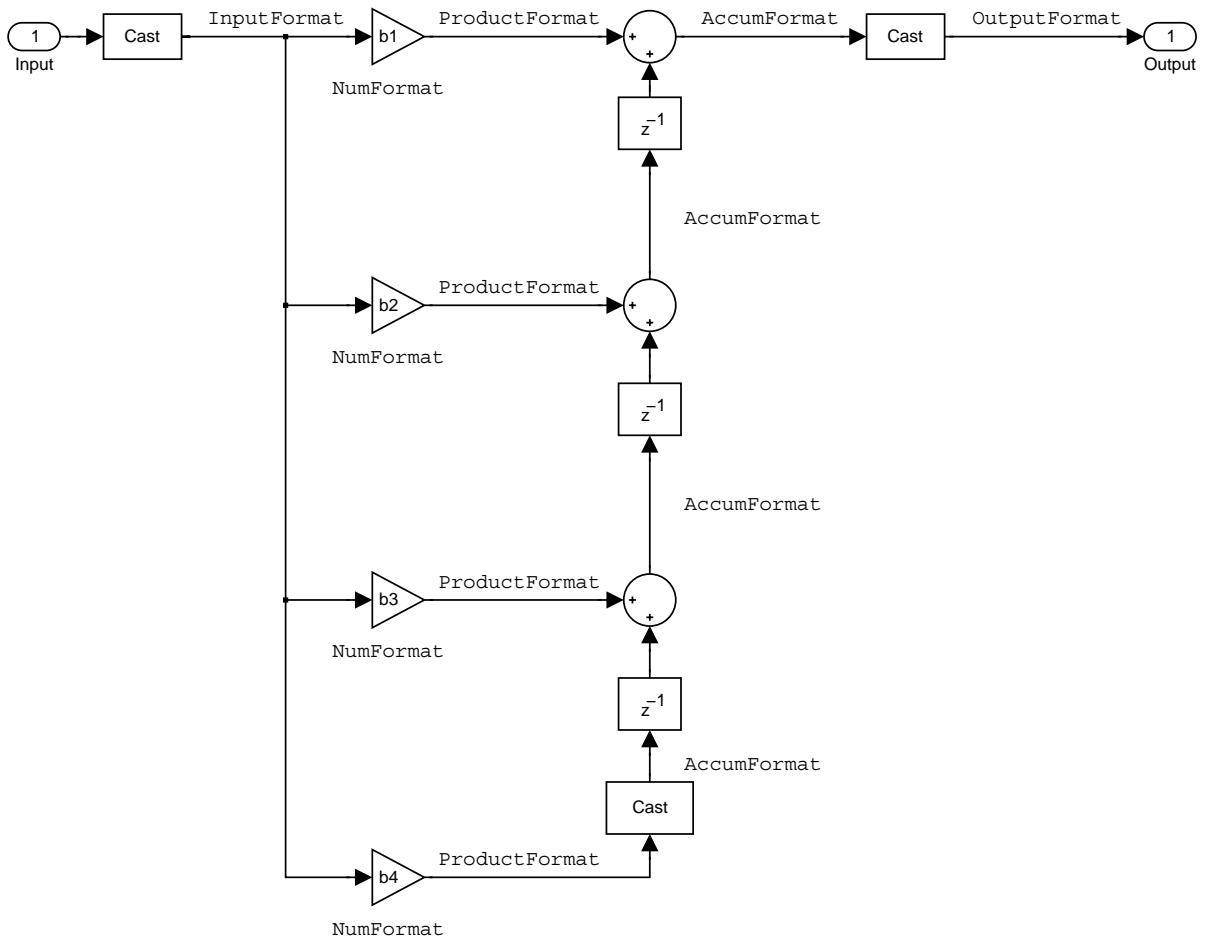
Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter  
`set(hd, 'arithmetic', 'single');`
- To change to fixed-point filtering, enter  
`set(hd, 'arithmetic', 'fixed');`

For more information about the property `Arithmetic`, refer to “`Arithmetic`” on page 7-20.

`hd = dfilt.dffirt` returns a default, discrete-time, direct-form FIR transposed filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

**Fixed-Point Filter Structure** The figure below shows the signal flow for the transposed direct-form FIR filter implemented by `dfilt.dffirt`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data flow 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.

You see that 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>Word Length Property</b>	<b>Fraction Length Property</b>	<b>Related Properties</b>
AccumFormat	AccumWordLength	AccumFracLength	
InputFormat	InputWordLength	InputFracLength	
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
OutputFormat	OutputWordLength	OutputFracLength	
ProductFormat	ProductWordLength	ProductFracLength	

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 7-3.

<b>Name</b>	<b>Values</b>	<b>Description</b>
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	[ <code>true</code> ], <code>false</code>	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.

## dfilt.dffirt

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.

<b>Name</b>	<b>Values</b>	<b>Description</b>
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.

## dfilt.dffirt

Name	Values	Description
RoundMode	[convergent], ceil,fix,floor, 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>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system.

## Examples

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

```
b = [0.05 0.9 0.05];
hd = dfilt.dffirt(b)

hd =

    FilterStructure: 'Direct-Form FIR Transposed'
    Arithmetic: 'double'
    Numerator: [0.0500 0.9000 0.0500]
    PersistentMemory: false
```

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

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

hd =

    FilterStructure: 'Direct-Form FIR Transposed'
    Arithmetic: 'fixed'
    Numerator: [0.0500 0.9000 0.0500]
    PersistentMemory: false
```

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

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'FullPrecision'

hd.filterInternals='specifyPrecision'

hd =

    FilterStructure: 'Direct-Form FIR Transposed'
      Arithmetic: 'fixed'
      Numerator: [0.0500 0.9000 0.0500]
    PersistentMemory: false

    CoeffWordLength: 16
      CoeffAutoScale: true
        Signed: true

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'SpecifyPrecision'

    OutputWordLength: 34
    OutputFracLength: 30

    ProductWordLength: 32
    ProductFracLength: 30

    AccumWordLength: 34
    AccumFracLength: 30

      RoundMode: 'convergent'
      OverflowMode: 'wrap'
```

**See Also**

`dfilt`, `dfilt.dffir`, `dfilt.dfasymfir`, `dfilt.dfsymfir`

# dfilt.dfsymfir

---

**Purpose** Construct discrete-time, direct-form symmetric FIR filter object

**Syntax** Refer to `dfilt.dfsymfir` in the Signal Processing Toolbox.

**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 7-20.

`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.

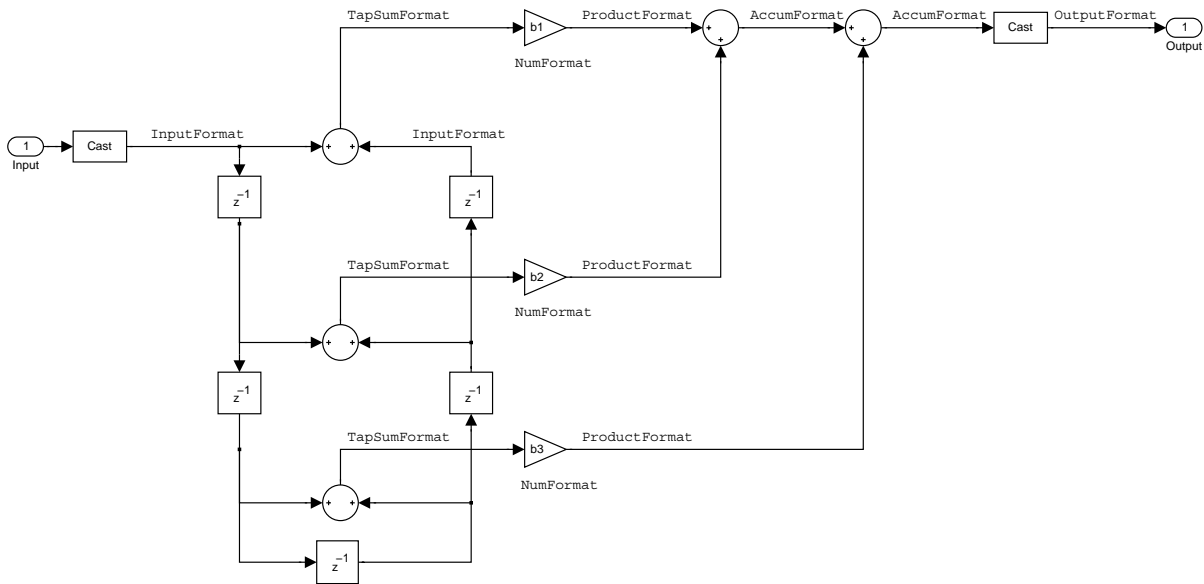
---

**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 figure below, 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 flow 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.

You see that 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

# dfilt.dfsymfir

the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Word Length Property	Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	
InputFormat	InputWordLength	InputFracLength	
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
OutputFormat	OutputWordLength	OutputFracLength	
ProductFormat	ProductWordLength	ProductFracLength	
TapSumFormat	InputWordLength	InputFracLength	

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 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 7-3..

<b>Name</b>	<b>Values</b>	<b>Description</b>
<code>AccumFracLength</code>	Any positive or negative integer number of bits [ 27 ]	Specifies the fraction length used to interpret data output by the accumulator.
<code>AccumWordLength</code>	Any integer number of bits [ 33 ]	Sets the word length used to store data in the accumulator.
<code>Arithmetic</code>	<code>fixed</code> for fixed-point filters	Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.
<code>CoeffAutoScale</code>	[ <code>true</code> ], <code>false</code>	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.
<code>CoeffWordLength</code>	Any integer number of bits [ 16 ]	Specifies the word length to apply to filter coefficients.

# dfilt.dfsymfir

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 saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [29]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [33]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

# dfilt.dfsymfir

Name	Values	Description
RoundMode	[convergent], ceil,fix,floor, 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>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system.

## Examples

### Odd Order

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

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
hd = dfilt.dfsymfir(b)

hd =

    FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'double'
    Numerator: [-0.0080 0.0600 0.4400 0.4400 0.0600 -0.0080]
 PersistentMemory: false

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

hd =

    FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'fixed'
    Numerator: [-0.0080 0.0600 0.4400 0.4400 0.0600 -0.0080]
 PersistentMemory: false
```

# dfilt.dfsymfir

---

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

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'FullPrecision'

hd.filterinternals='specifyPrecision'

hd =

    FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'fixed'
      Numerator: [-0.0080 0.0600 0.4400 0.4400 0.0600 -0.0080]
    PersistentMemory: false

    CoeffWordLength: 16
      CoeffAutoScale: true
        Signed: true

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'SpecifyPrecision'

    OutputWordLength: 36
    OutputFracLength: 31

    ProductWordLength: 33
    ProductFracLength: 31

    AccumWordLength: 36
    AccumFracLength: 31

      RoundMode: 'convergent'
      OverflowMode: 'wrap'
```



To use `hd` for fixed-point filtering, change the value of the property `Arithmetic` to `fixed` with the following command:

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

## Even Order

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

```
b = [-0.01 0.1 0.8 0.1 -0.01];
hd = dfilt.dfsymfir(b)

hd =

    FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'double'
      Numerator: [-0.0100 0.1000 0.8000 0.1000 -0.0100]
 PersistentMemory: false

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

hd =

    FilterStructure: 'Direct-Form Symmetric FIR'
      Arithmetic: 'fixed'
      Numerator: [-0.0100 0.1000 0.8000 0.1000 -0.0100]
 PersistentMemory: false

    CoeffWordLength: 16
      CoeffAutoScale: true
           Signed: true

    InputWordLength: 16
    InputFracLength: 15

    FilterInternals: 'FullPrecision'

hd.filterinternals='specifyPrecision'
```

# dfilt.dfsymfir

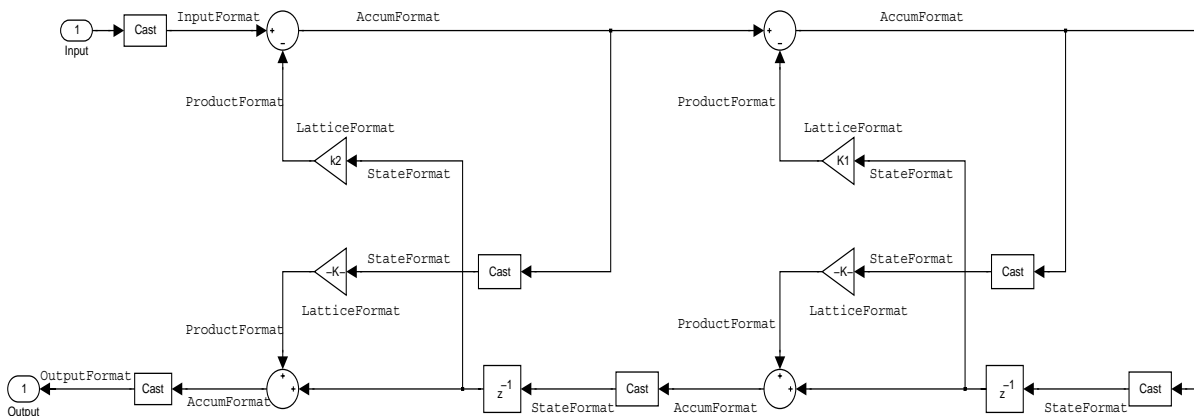
---

```
hd =  
  
    FilterStructure: 'Direct-Form Symmetric FIR'  
        Arithmetic: 'fixed'  
        Numerator: [-0.0100 0.1000 0.8000 0.1000 -0.0100]  
PersistentMemory: false  
  
    CoeffWordLength: 16  
        CoeffAutoScale: true  
            Signed: true  
  
    InputWordLength: 16  
    InputFracLength: 15  
  
    FilterInternals: 'SpecifyPrecision'  
  
    OutputWordLength: 36  
    OutputFracLength: 30  
  
    ProductWordLength: 33  
    ProductFracLength: 30  
  
    AccumWordLength: 36  
    AccumFracLength: 30  
  
        RoundMode: 'convergent'  
        OverflowMode: 'wrap'
```

## See Also

[dfilt](#), [dfilt.dfasymfir](#), [dfilt.dffir](#), [dfilt.dffirt](#)

<b>Purpose</b>	Construct discrete-time, lattice allpass filter object
<b>Syntax</b>	Refer to <code>dfilt.latticeallpass</code> in the Signal Processing Toolbox.
<b>Description</b>	<p><code>hd = dfilt.latticeallpass(k)</code> returns a discrete-time, lattice allpass 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 <code>set(hd, 'arithmetic', 'single');</code></li><li>• To change to fixed-point filtering, enter <code>set(hd, 'arithmetic', 'fixed');</code></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “Arithmetic” on page 7-20.</p> <p><code>hd = dfilt.latticeallpass</code> returns a default, discrete-time, lattice allpass 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>	The figure below shows the signal flow for the allpass lattice filter implemented by <code>dfilt.latticeallpass</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.



## 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 flow 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.

You see that 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	Word Length Property	Fraction Length Property	Related Properties
<code>AccumFormat</code>	<code>AccumWordLength</code>	<code>AccumFracLength</code>	<code>AccumMode</code>
<code>InputFormat</code>	<code>InputWordLength</code>	<code>InputFracLength</code>	

Signal Flow Label	Word Length Property	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 allpass lattice implementation of `dfilt` objects.

---

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time.

To view all the properties for a filter at any time, use  
`get(hd)`

where `hd` is a filter.

---

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties” on page 7-3.

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 false enables you to change the LatticeFracLength property value to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients. No default value.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

<b>Property Name</b>	<b>Brief Description</b>
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 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.</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 ProductMode to SpecifyPrecision.</p>



<b>Property Name</b>	<b>Brief Description</b>
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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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	<p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>

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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to filtstates in your Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Specify a third-order lattice allpass filter structure for a dfilt object hd, with the following code:

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

Now convert hd to fixed-point arithmetic form.

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

```
hd =
```

```

    FilterStructure: 'Lattice Allpass'
      Arithmetic: 'fixed'
        Lattice: [0.6600 0.7000 0.4400]
PersistentMemory: false
          States: [1x1 embedded.fi]

    CoeffWordLength: 16
      CoeffAutoScale: true
          Signed: true

```

# dfilt.latticeallpass

---

```
InputWordLength: 16
InputFracLength: 15

OutputWordLength: 16
    OutputMode: 'AvoidOverflow'

StateWordLength: 16
StateFracLength: 15

    ProductMode: 'FullPrecision'

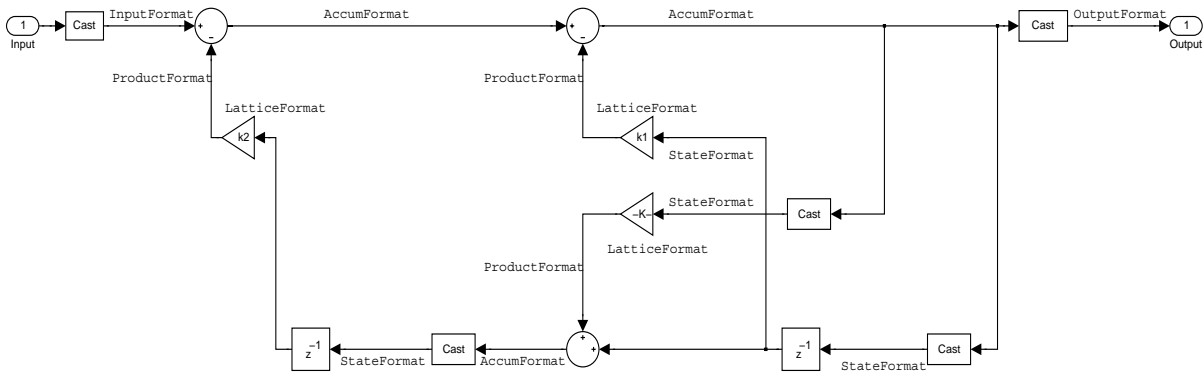
        AccumMode: 'KeepMSB'
AccumWordLength: 40
CastBeforeSum: true

        RoundMode: 'convergent'
OverflowMode: 'wrap'
```

## See Also

dfilt, dfilt.latticear, dfilt.latticearma, dfilt.latticemax,  
dfilt.latticemin

<b>Purpose</b>	Construct discrete-time, lattice, autoregressive filter object
<b>Syntax</b>	Refer to <code>dfilt.latticear</code> in the Signal Processing Toolbox.
<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 <code>set(hd, 'arithmetic', 'single');</code></li><li>• To change to fixed-point filtering, enter <code>set(hd, 'arithmetic', 'fixed');</code></li></ul> <p>For more information about the property <code>Arithmetic</code>, refer to “<code>Arithmetic</code>” on page 7-20.</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>	The figure below 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.



## 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 flow 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.

You see that 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	Word Length Property	Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	AccumMode
InputFormat	InputWordLength	InputFracLength	

Signal Flow Label	Word Length Property	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 7-3.

<b>Property Name</b>	<b>Brief Description</b>
<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 false enables you to change the LatticeFracLength to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

<b>Property Name</b>	<b>Brief Description</b>
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 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.</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 ProductMode to SpecifyPrecision.</p>

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

<b>Property Name</b>	<b>Brief Description</b>
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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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	<p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>

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. Notice that the states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in your Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Specify a third-order lattice autoregressive filter structure for a `dfilt` object, `hd`, with the following code that creates a fixed-point filter.

```
k = [.66 .7 .44];
hd1=dfilt.latticear(k)

hd1 =

    FilterStructure: 'Lattice Autoregressive (AR)'
    Arithmetic: 'double'
    Lattice: [0.6600 0.7000 0.4400]
    PersistentMemory: false
    States: [3x1 double]

hd1.arithmetic='fixed'

hd1 =

    FilterStructure: 'Lattice Autoregressive (AR)'
    Arithmetic: 'fixed'
```

```
Lattice: [0.6600 0.7000 0.4400]
PersistentMemory: false
States: [1x1 embedded.fi]

CoeffWordLength: 16
CoeffAutoScale: true
Signed: true

InputWordLength: 16
InputFracLength: 15

OutputWordLength: 16
OutputMode: 'AvoidOverflow'

StateWordLength: 16
StateFracLength: 15

ProductMode: 'FullPrecision'

AccumMode: 'KeepMSB'
AccumWordLength: 40
CastBeforeSum: true

RoundMode: 'convergent'
OverflowMode: 'wrap'
```

```
specifyall(hd1)
```

```
hd1
```

```
hd1 =
```

```
FilterStructure: 'Lattice Autoregressive (AR)'  
Arithmetic: 'fixed'  
Lattice: [0.6600 0.7000 0.4400]  
PersistentMemory: false  
States: [1x1 embedded.fi]

CoeffWordLength: 16  
CoeffAutoScale: false  
LatticeFracLength: 15
```

```
Signed: true

InputWordLength: 16
InputFracLength: 15

OutputWordLength: 16
OutputMode: 'SpecifyPrecision'
OutputFracLength: 12

StateWordLength: 16
StateFracLength: 15

ProductMode: 'SpecifyPrecision'
ProductWordLength: 32
ProductFracLength: 30

AccumMode: 'SpecifyPrecision'
AccumWordLength: 40
AccumFracLength: 30
CastBeforeSum: true

RoundMode: 'convergent'
OverflowMode: 'wrap'
```

**See Also**

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

# dfilt.latticearma

---

**Purpose** Construct discrete-time, lattice, autoregressive, moving-average filter object

**Syntax** Refer to `dfilt.latticearma` in the Signal Processing Toolbox.

**Description** `hd = dfilt.latticearma(k)` returns a discrete-time, lattice moving-average autoregressive filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

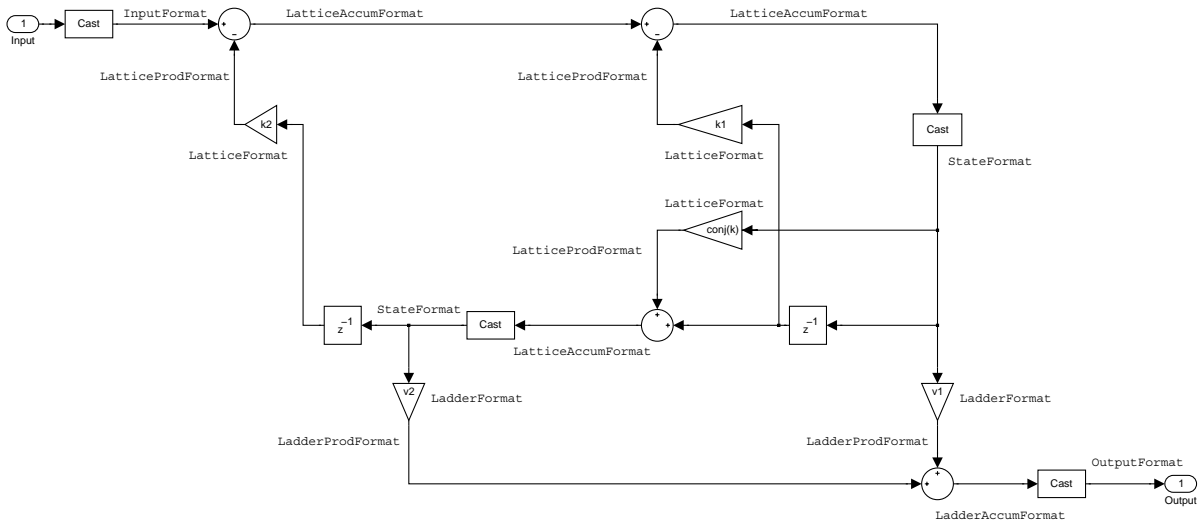
- To change to single-precision filtering, enter  
`set(hd, 'arithmetic', 'single');`
- To change to fixed-point filtering, enter  
`set(hd, 'arithmetic', 'fixed');`

For more information about the property `Arithmetic`, refer to “`Arithmetic`” on page 7-20.

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

**Fixed-Point Filter Structure** The figure below shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticearma`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.





## Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data flow 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.

You see that 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

# dfilt.latticearma

the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Word Length Property	Fraction Length Property	Related Properties
InputFormat	InputWordLength	InputFracLength	
LadderAccumFormat	AccumWordLength	LadderAccumFracLength	AccumMode
LadderFormat	CoeffWordLength	LadderFracLength	CoeffAutoScale
LadderProdFormat	ProductWordLength	LadderProdFracLength	ProductMode
LatticeAccumFormat	AccumWordLength	LatticeAccumFracLength	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 7-3.

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> .
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.

<b>Property Name</b>	<b>Brief Description</b>
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	<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>▪ AvoidOverflow—directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>▪ BestPrecision—directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>▪ SpecifyPrecision—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.

<b>Property Name</b>	<b>Brief Description</b>
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
PersistentMemory	<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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

# dfilt.latticearma

---

Property Name	Brief Description
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. Notice that the states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in your Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

## See Also

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticemamin`, `dfilt.latticemamin`



**Purpose** Construct discrete-time, lattice, moving-average filter object with maximum phase

**Syntax** Refer to `dfilt.latticemamax` in the Signal Processing Toolbox.

**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 7-20.

---

**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 figure below 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.



Signal Flow Label	Word Length Property	Fraction Length Property	Related Properties
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 7-3.

<b>Property Name</b>	<b>Brief Description</b>
<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 false enables you to change the LatticeFracLength property to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

<b>Property Name</b>	<b>Brief Description</b>
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 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.</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 ProductMode to SpecifyPrecision.</p>

<b>Property Name</b>	<b>Brief Description</b>
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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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	<p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>



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. Notice that the states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in your Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Specify a fourth-order lattice, moving-average, maximum phase filter structure for a `dfilt` object, `hd`, with the following code:

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

hd =
    FilterStructure: 'Lattice maximum phase'
      Lattice: [1x4 double]
  NumberOfSamplesProcessed: 0
      ResetStates: 'on'
        States: [4x1 double]
```

## See Also

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

# dfilt.latticemamin

---

**Purpose** Construct discrete-time, lattice, moving-average filter object with minimum phase

**Syntax** Refer to `dfilt.latticemamin` in the Signal Processing Toolbox.

**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 7-20.

---

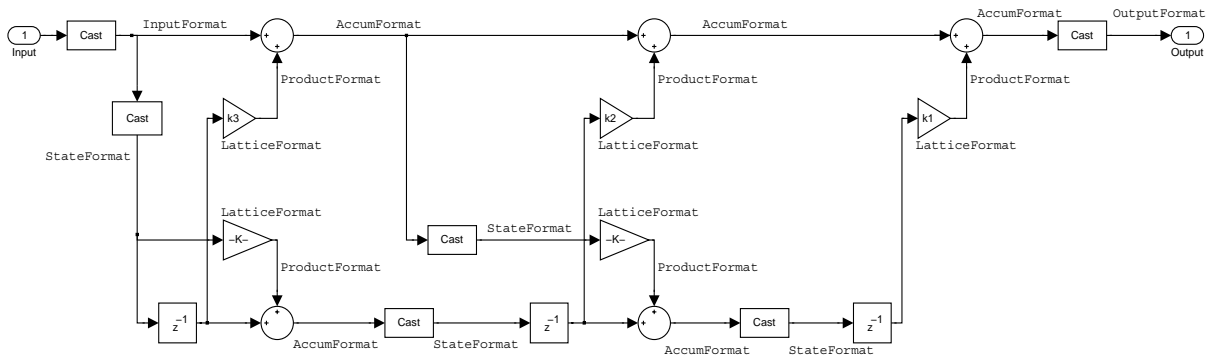
**Note** When the `k` coefficients define a minimum phase filter, the resulting filter in this structure is minimum phase. When your coefficients do not define a minimum phase filter, placing them in this structure does not produce a minimum phase filter.

---

`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 figure below 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 flow 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.

You see that 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	Word Length Property	Fraction Length Property	Related Properties
<code>AccumFormat</code>	<code>AccumWordLength</code>	<code>AccumFracLength</code>	<code>AccumMode</code>
<code>InputFormat</code>	<code>InputWordLength</code>	<code>InputFracLength</code>	
<code>LatticeFormat</code>			

# dfilt.latticemamin

---

Signal Flow Label	Word Length Property	Fraction Length Property	Related Properties
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ProductFormat	ProductWordLength	ProductFracLength	ProductMode
StateFormat	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the minimum phase, moving average lattice implementation of `dfilt` objects.

---

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time.

To view all the properties for a filter at any time, use  
`get(hd)`

where `hd` is a filter.

---

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties” on page 7-3.

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

## dfilt.latticemamin

---

<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 false enables you to change the LatticeFracLength property to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

Property Name	Brief Description
OutputMode	<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>▪ AvoidOverflow—directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>▪ BestPrecision—directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>▪ SpecifyPrecision—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 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.</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 ProductMode to SpecifyPrecision.</p>

<b>Property Name</b>	<b>Brief Description</b>
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>convergent</code>—Round up to the next allowable quantized value.</li><li>• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <code>floor</code>—Round down to the next allowable quantized value.</li><li>• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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	<p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>

# dfilt.latticemamin

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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to filtstates in your Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

## Examples

Specify a third-order lattice, moving-average, minimum phase, filter structure for a dfilt object, hd, with the following code:

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

hd =

    FilterStructure: 'Lattice Moving-Average (MA) For Minimum
Phase'
    Arithmetic: 'double'
    Lattice: [0.6600 0.7000 0.4400]
    PersistentMemory: false
    States: [3x1 double]

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

hd =
```

```
FilterStructure: 'Lattice Moving-Average (MA) For Minimum
Phase'
    Arithmetic: 'fixed'
        Lattice: [0.6600 0.7000 0.4400]
    PersistentMemory: false
        States: [1x1 embedded.fi]

    CoeffWordLength: 16
        CoeffAutoScale: false
    LatticeFracLength: 15
        Signed: true

    InputWordLength: 16
        InputFracLength: 15

    OutputWordLength: 16
        OutputMode: 'SpecifyPrecision'
    OutputFracLength: 12

    StateWordLength: 16
        StateFracLength: 15

        ProductMode: 'SpecifyPrecision'
    ProductWordLength: 32
    ProductFracLength: 30

        AccumMode: 'SpecifyPrecision'
    AccumWordLength: 40
    AccumFracLength: 30
        CastBeforeSum: true

        RoundMode: 'convergent'
        OverflowMode: 'wrap'
```

## See Also

dfilt, dfilt.latticeallpass, dfilt.latticear, dfilt.latticearma,  
dfilt.latticemamax

# dfilt.parallel

**Purpose** Construct discrete-time, parallel structure filter object

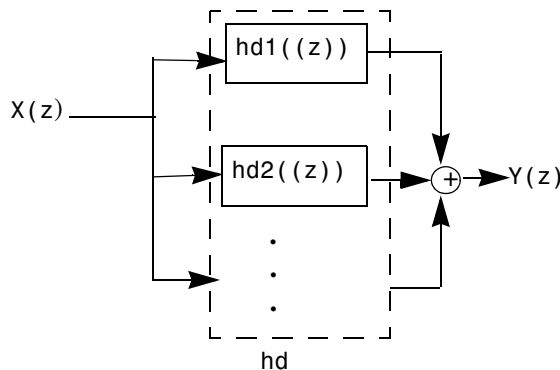
**Syntax** Refer to `dfilt.parallel` in the Signal Processing Toolbox.

**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`

`dfilt.cascade`, `dfilt.parallel` in your Signal Processing Toolbox documentation

**Purpose** Construct discrete-time, scalar filter object

**Syntax** Refer to `dfilt.scalar` in the Signal Processing Toolbox.

**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 7-20.

`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 7-3.

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

<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.
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>▪ AvoidOverflow—directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li><li>▪ BestPrecision—directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li><li>▪ SpecifyPrecision—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.

<b>Property Name</b>	<b>Brief Description</b>
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.



Property Name	Brief Description
RoundMode	<p data-bbox="783 300 1329 430">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 data-bbox="783 456 1329 1055" style="list-style-type: none"><li data-bbox="783 456 1329 517">• <code>convergent</code>—Round up to the next allowable quantized value.</li><li data-bbox="783 534 1329 725">• <code>ceil</code>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li data-bbox="783 743 1329 838">• <code>fix</code>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li data-bbox="783 855 1329 916">• <code>floor</code>—Round down to the next allowable quantized value.</li><li data-bbox="783 933 1329 1055">• <code>round</code>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</li></ul> <p data-bbox="783 1081 1329 1237">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>

# dfilt.scalar

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. Notice that the states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in your Signal Processing Toolbox documentation or in the Help system.

## Example

Create a direct-form I filter object `hd_filt` and a scalar object with a gain of 3 `hd_gain` and cascade them together.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd_filt = dfilt.df1(b,a)
hd_gain = dfilt.scalar(3)
hd=cascade(hd_gain,hd_filt)
fvtool(hd_filt,hd_gain,hd)

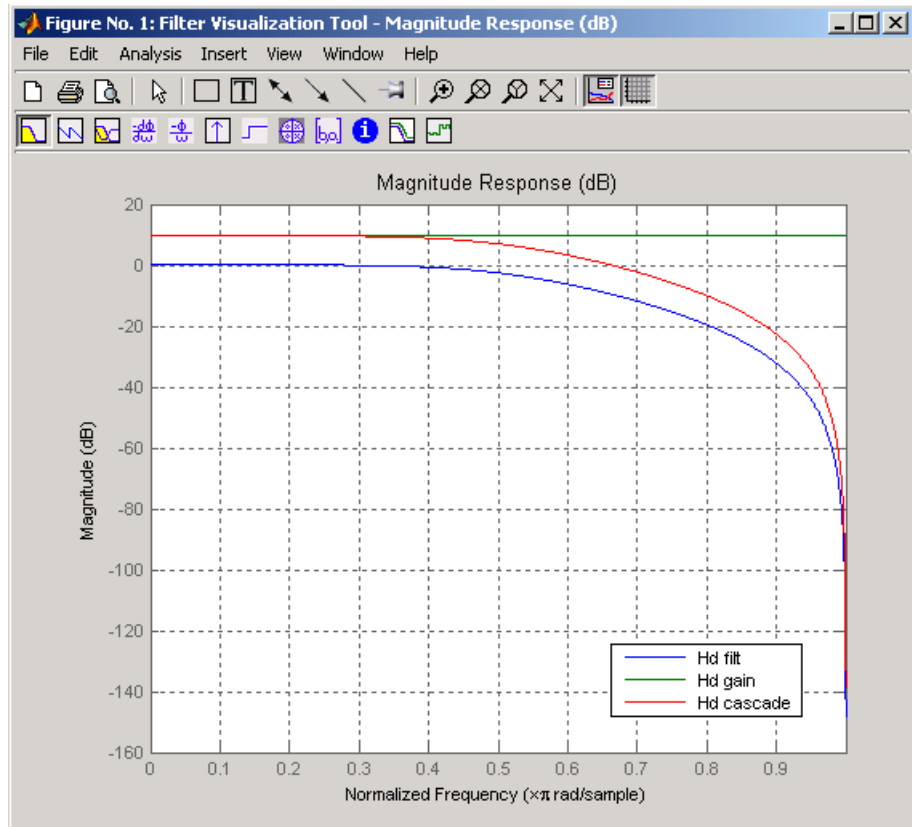
hd_filt =
    FilterStructure: 'direct-form I'
    Arithmetic: 'double'
    Numerator: [0.3000 0.6000 0.3000]
    Denominator: [1 0 0.2000]
    PersistentMemory: false
    States: [4x1 double]

hd_gain =
    FilterStructure: 'Scalar'
    Arithmetic: 'double'
    Gain: 3
    PersistentMemory: false
    States: []
```

```

hd =
    FilterStructure: Cascade
           Section(1): Scalar
           Section(2): Direct Form I
 PersistentMemory: false

```



To view the sections of the cascaded filter, use

```

hd.section(1)

ans =
    FilterStructure: 'Scalar'
    Arithmetic: 'double'

```

## dfilt.scalar

---

```
Gain: 3  
PersistentMemory: false  
States: []
```

and

```
hd.section(2)
```

```
ans =
```

```
FilterStructure: 'Direct Form I'  
Arithmetic: 'double'  
Numerator: [0.3000 0.6000 0.3000]  
Denominator: [1 0 0.2000]  
PersistentMemory: false  
States: [4x1 double]
```

### See Also

dfilt, dfilt.cascade

**Purpose** Construct wave digital allpass filter object

**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. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to <code>filtstates</code> in your Signal Processing Toolbox documentation or in the Help system.

## Filter Structure

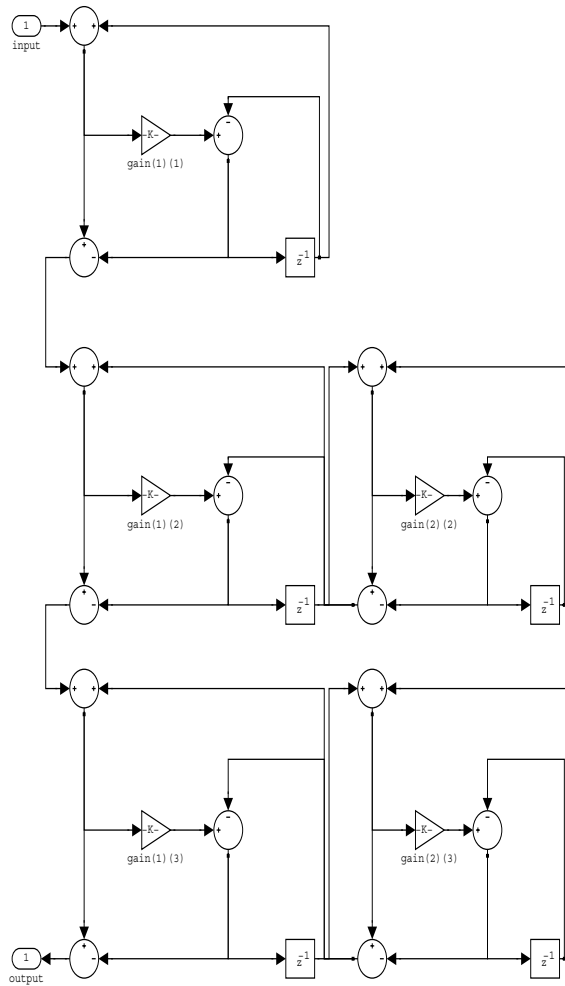
When you change the order of the wave digital filters in the cascade, the filter structure changes as well.

As shown in this example, `realizemdl` lets you see the filter structure used for your filter, if you have Simulink installed.

```
section11=0.8;  
section12=[1.5,0.7];  
section13=[1.8,0.9];  
hd1=dfilt.cascadewdfallpass(section11,section12,section13);  
realizemdl(hd1)
```

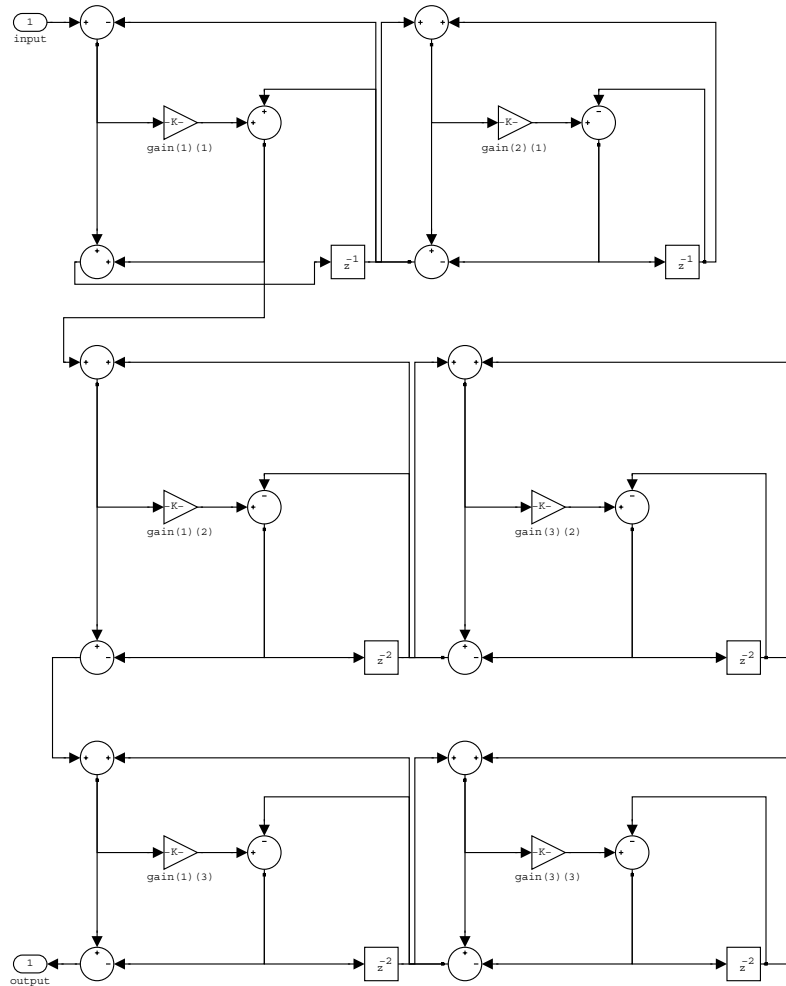
```
section21=[0.8,0.4];  
section22=[0,1.5,0,0.7];  
section23=[0,1.8,0,0.9];  
hd2=dfilt.cascadewdfallpass(section21,section22,section23);  
realizemdl(hd2)
```

hd1 has this filter structure with three sections.



# dfilt.wdfallpass

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



## Examples

Construct a second-order wave digital allpass filter with two coefficients.

```
c = [1.5,0.7];  
hd = dfilt.wdfallpass(c);  
info(hd)
```





# disp

---

**Purpose** Filter object with 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

# double

---

**Purpose** Cast fixed-point filter to filter that uses 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 `refilter(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 `refilter` 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.
h.arithmetic = 'fixed'; % Set h to use fixed-point arithmetic
% to filter. Quantize the coeffs.
hd = double(h); % Cast h to double-precision
% floating-point coefficients.
n = 0:99; x = sin(0.7*pi*n(:)); % Set up an input signal.
y = filter(h,x); % Fixed-point output.
yd = filter(hd,x); % Floating-point output.
norm(yd-double(y),inf)
ans =
```

```
9.2014e-004
```

`norm` shows that the largest difference (maximum error) between the output values from the fixed versus floating filtering comparison is about 0.0009—either good or less good depending on your application.

**See Also**

refilter

# ellip

---

**Purpose** Design elliptical or Causer digital filter using filter 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')
```

```
ans =
```

```
FilterStructure: 'df2sos'  
MatchExactly: 'both'
```

```
hd = design(d,'ellip','matchexactly','both');
```

```
hd =
```

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

```

Arithmetic: 'double'
  sosMatrix: [4x6 double]
  ScaleValues: [5x1 double]
PersistentMemory: false

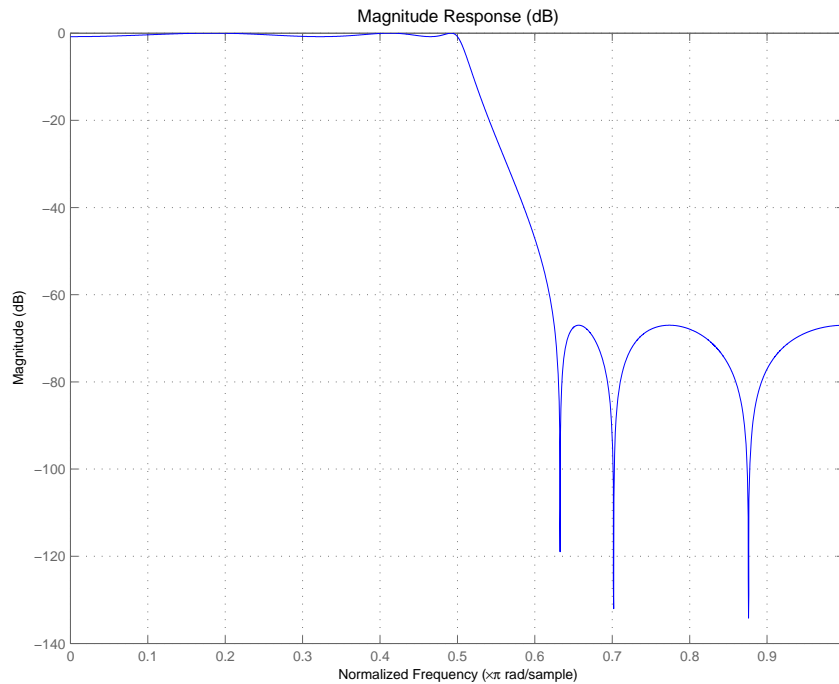
```

Example 2—construct a lowpass object with order, passband-edge frequency, stopband-edge frequency, and passband ripple specifications, and then design an elliptic filter.

```

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

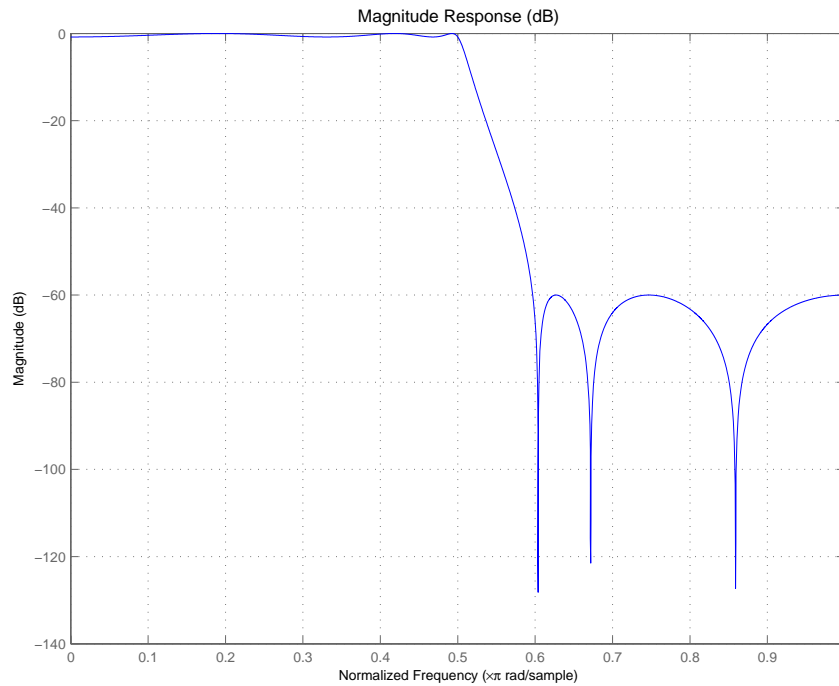
```



Example 3—construct a lowpass object with filter order, passband edge frequency, passband ripple, and stopband attenuation specifications, and then design an elliptic filter.

# ellip

```
d = fdesign.lowpass('n,fp,ap,ast',6,20,.8,60,80);  
design(d,'ellip'); % Starts FVTool to display the filter.
```



**See Also** butter, cheby1, cheby2



**Purpose** Use Euclid's theorem to return integer 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 the factors for plain 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

---

**Purpose** Design equiripple single-rate or multirate FIR filter from filter specification object

**Syntax**

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

**Description** `hd = design(d,'equiripple')` designs an equiripple FIR digital filter or multirate filter using the specifications supplied in the object `d`. Equiripple filter designs minimize the maximum ripple in the pass- and stopbands.

`hd` is either a `dfilt` object (a single-rate digital filter) or an `mfilt` object (a multirate digital filter) depending on the `Specification` property of the filter specification object `d` and the specifications object type—halfband or interpolator.

When you use `equiripple` with Nyquist filter specification objects, you might encounter design cases where the filter design does not converge. Convergence errors occur mostly at large filter orders, or small transition widths, or large stopband attenuations. These specifications, alone or combined, can cause design failures. For more information, refer to `fdesign.nyquist` in the online Help system.

`hd = design(d,'equiripple',designoption,value,designoption,... value,...)` returns an equiripple FIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

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

```
help(d,'equiripple')
```

**Examples** Here is an example of designing a single-rate equiripple filter from a halfband filter specification object.

```
d = fdesign.halfband
```

```
designopts(d,'equiripple')
```

```
ans =
```

```
    FilterStructure: 'dffir'
```

```
        MinPhase: 0
```

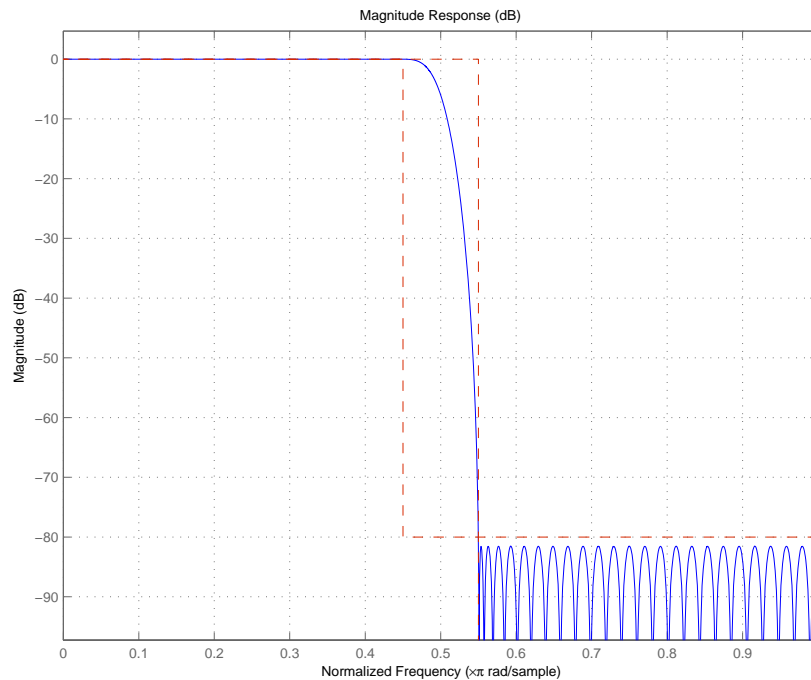
```
        StopbandShape: 'flat'
```

```
        StopbandDecay: 0
```

```
hd = design(d,'equiripple','stopbandshape','flat');
```

```
fvtool(hd);
```

Displaying the filter in FVTool shows the equiripple nature of the filter.



equiripple also designs multirate filters. This example generates a halfband interpolator filter.

# equiripple

---

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

hd

hd =

    FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
      Arithmetic: 'double'
      Numerator: [1x95 double]
InterpolationFactor: 2
  PersistentMemory: false
```

This final example designs an equiripple filter with a direct-form structure by specifying the **filterstructure** argument.

```
d = fdesign.lowpass('fp,fst,ap,ast');
designopts(d,'equiripple')

ans =

    FilterStructure: 'dffir'
      DensityFactor: 16
      MinPhase: 0
      MinOrder: 'any'
      StopbandShape: 'flat'
      StopbandDecay: 0

hd = design(d,'equiripple','filterstructure','dffir');
hd

hd =

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

## See Also

fdesign.nyquist, firls, kaiserwin

**Purpose** Implement Farrow filter

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

**Description** `hd = farrow.structure(delay, ...)` returns a Farrow filter `hd` that associates `delay`, the fractional delay, with a filter structure specified by `structure`.

More information about Farrow filters is available in [References](#).

In contrast to most single-rate filters, Farrow filters use two inputs—the input data and the fractional delay. 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.

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.

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:

<b>structure String</b>	<b>Description</b>
<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. Start with basic coefficients from Lagrange polynomials (also called interpolation polynomials). 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 that you use most often with digital filters are

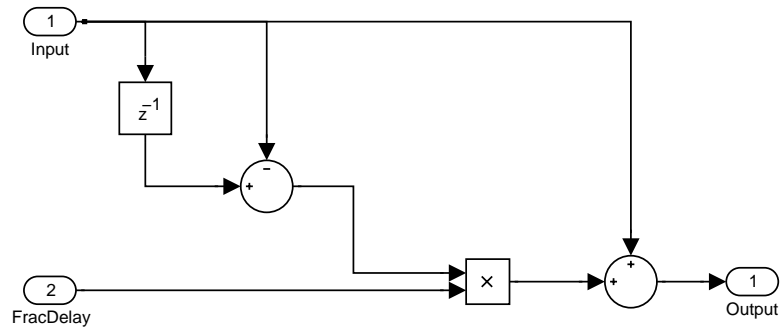
Function	Description
cost	Estimate the hardware implementation cost in terms of mathematical operations like add and multiply
filter	Execute the filter by using it to filter data
fvtool	Display and analyze the filter
freqz	Compute the instantaneous frequency response of the filter
realizemdl	Generate a Simulink subsystem model of the filter as a block (Requires Simulink)

## 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

- [1] 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.
- [2] Marvasti, F., *Nonuniform Sampling—Theory and Practice*, Kluwer Academic/Plenum Publishers, New York, 2001.

## See Also

adaptfilt, dfilt, fdesign, mfil

# fcfwrite

---

**Purpose** Write file containing filter coefficients for multirate, adaptive, or discrete-time filter objects

**Syntax**

```
fcfwrite(h)
fcfwrite(h,filename)
fcfwrite(...,'fmt')
```

**Description** `fcfwrite(h)` writes a filter coefficient ASCII file to a directory you choose, or your current MATLAB working directory. `h` can be a single filter object or a vector of filter objects. On execution, `fcfwrite` opens the **Export Filter Coefficients to .FCF File** dialog to let you assign a file name for the output file. You can choose the destination directory within this dialog as well.

The default file name is `untitled.fcf`. When you have the Filter Design Toolbox, you can use `fcfwrite(h)` to write filter coefficient files for multirate filters, adaptive filters, and discrete-time filters.

`fcfwrite(h,filename)` writes the filter coefficients and general information to a text file called `filename` in your present MATLAB working directory and opens the file in the MATLAB editor for you to review or modify.

If you do not include a file extension in `filename`, `fcfwrite` adds the extension `fcf` to `filename`.

`fcfwrite(...,'fmt')` writes the filter coefficients in the format specified by the input argument `fmt`. Valid `fmt` values are `hex` for hexadecimal, `dec` for decimal, or `bin` for binary representation of the filter coefficients.

**Examples** To demonstrate `fcfwrite`, create a fixed-point IIR filter at the command line, and then write the filter coefficients to a file named `iirfilter.fcf`.

```
d=fdesign.lowpass

d =

      Response: 'Lowpass'
  Specification: 'Fp,Fst,Ap,Ast'
      Description: {4x1 cell}
NormalizedFrequency: true
           Fpass: 0.45
```



```

                                Fstop: 0.55
                                Apass: 1
                                Astop: 60

hd=butter(d)

hd =

    FilterStructure: 'Direct-Form II, Second-Order Sections'
    Arithmetic: 'double'
    sosMatrix: [13x6 double]
    ScaleValues: [14x1 double]
    PersistentMemory: false

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

fcbwrite(hd,'iirfilter.fcb');

```

Here is the output from `fcbwrite` as it appears in the MATLAB editor. Not shown here is the filename—`iirfilter.fcb` as specified and some comments at the top of the file.

```

%
%
% Coefficient Format: Decimal
%
% Discrete-Time IIR Filter (real)
% -----
% Filter Structure      : Direct-Form II, Second-Order Sections
% Number of Sections   : 13
% Stable                : Yes
% Linear Phase         : No
% Arithmetic           : fixed
% Numerator            : s16,13 -> [-4 4)
% Denominator          : s16,14 -> [-2 2)
% Scale Values         : s16,14 -> [-2 2)
% Input                : s16,15 -> [-1 1)
% Section Input        : s16,8 -> [-128 128)
% Section Output       : s16,10 -> [-32 32)
% Output               : s16,10 -> [-32 32)

```

```
% State           : s16,15 -> [-1 1)
% Numerator Prod  : s32,28 -> [-8 8)
% Denominator Prod : s32,29 -> [-4 4)
% Numerator Accum  : s40,28 -> [-2048 2048)
% Denominator Accum : s40,29 -> [-1024 1024)
% Round Mode      : convergent
% Overflow Mode   : wrap
% Cast Before Sum : true
```

SOS matrix:

```
1 2 1 1 -0.22222900390625 0.88262939453125
1 2 1 1 -0.19903564453125 0.68621826171875
1 2 1 1 -0.18060302734375 0.5303955078125
1 2 1 1 -0.1658935546875 0.40570068359375
1 2 1 1 -0.154052734375 0.305419921875
1 2 1 1 -0.14453125 0.22479248046875
1 2 1 1 -0.136962890625 0.16015625
1 2 1 1 -0.13092041015625 0.10906982421875
1 2 1 1 -0.126220703125 0.06939697265625
1 2 1 1 -0.12274169921875 0.0399169921875
1 2 1 1 -0.12030029296875 0.01947021484375
1 2 1 1 -0.118896484375 0.0074462890625
1 1 0 1 -0.0592041015625 0
```

Scale Values:

```
0.41510009765625
0.371826171875
0.33746337890625
0.3099365234375
0.287841796875
0.27008056640625
0.25579833984375
0.2445068359375
0.23577880859375
0.22930908203125
0.22479248046875
0.22216796875
0.47039794921875
1
```

To write two or more filters out to one file, provide the filters as a vector to `fcfwrite`:

```
fcfwrite([hd hd1 hd2])
```

## See Also

`adaptfilt`, `mfilt`

`dfilt` in the Signal Processing Toolbox documentation

# fdatool

---

**Purpose** Open Filter Design and Analysis Tool

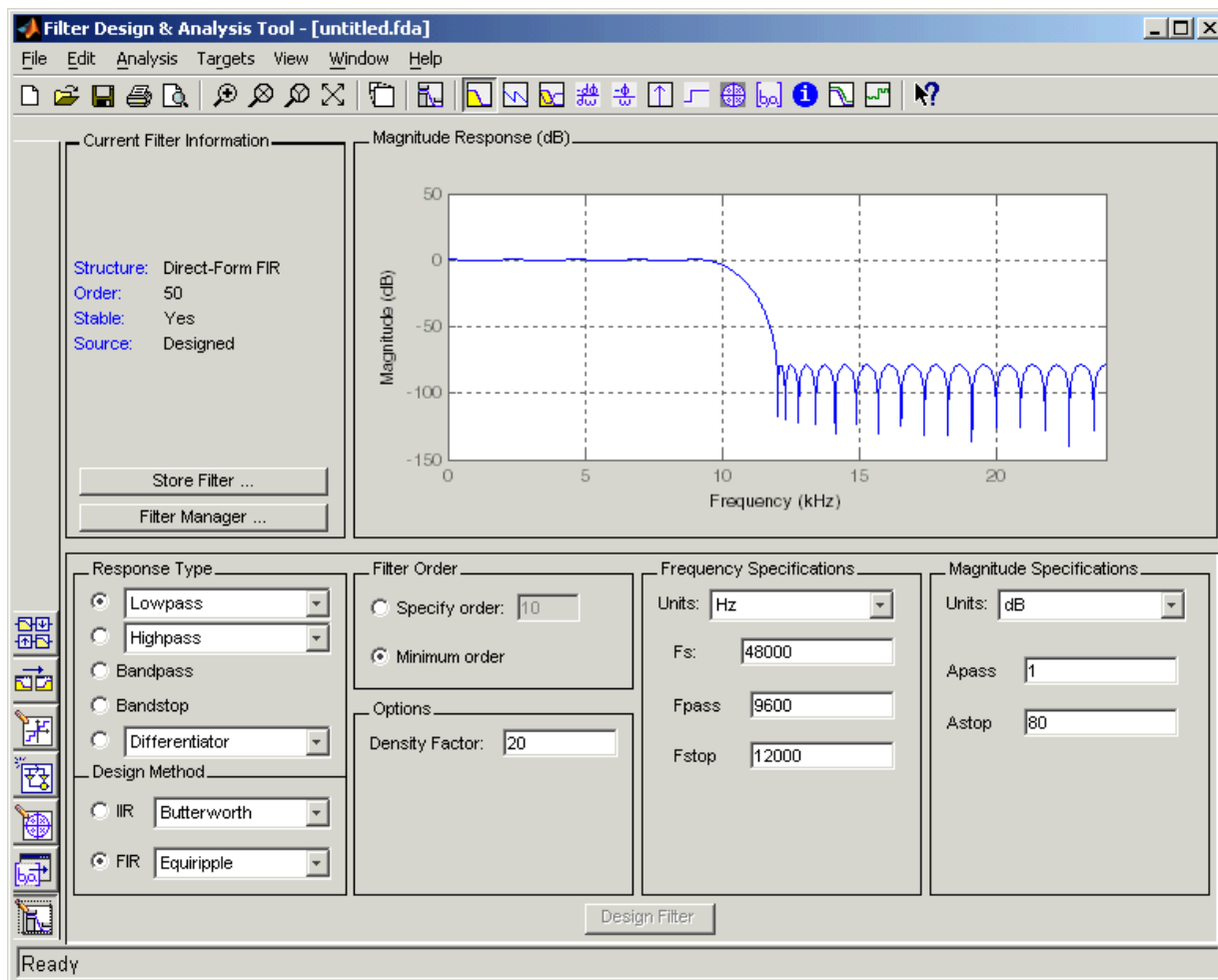
**Syntax** `fdatool`

**Description** `fdatool` opens the Filter Design and Analysis Tool (FDATool). Use this tool to:

- Design filters
- Quantize filters (with Filter Design Toolbox installed)
- Analyze filters
- Modify existing filter designs
- Create multirate filters (with Filter Design Toolbox installed)
- Realize Simulink models of quantized, direct-form, FIR filters (with Filter Design Toolbox installed)
- Import filters into FDATool
- Perform digital frequency transformations of filters (with Filter Design Toolbox installed)

Refer to “Using FDATool with the Filter Design Toolbox” for more information about using the analysis, design, and quantization features of FDATool. For general information about using FDATool, refer to “Filter Design and Analysis Tool” in your Signal Processing Toolbox documentation.

When you open FDATool and you have Filter Design Toolbox installed, FDATool incorporates features that are added by Filter Design Toolbox. With Filter Design Toolbox 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.



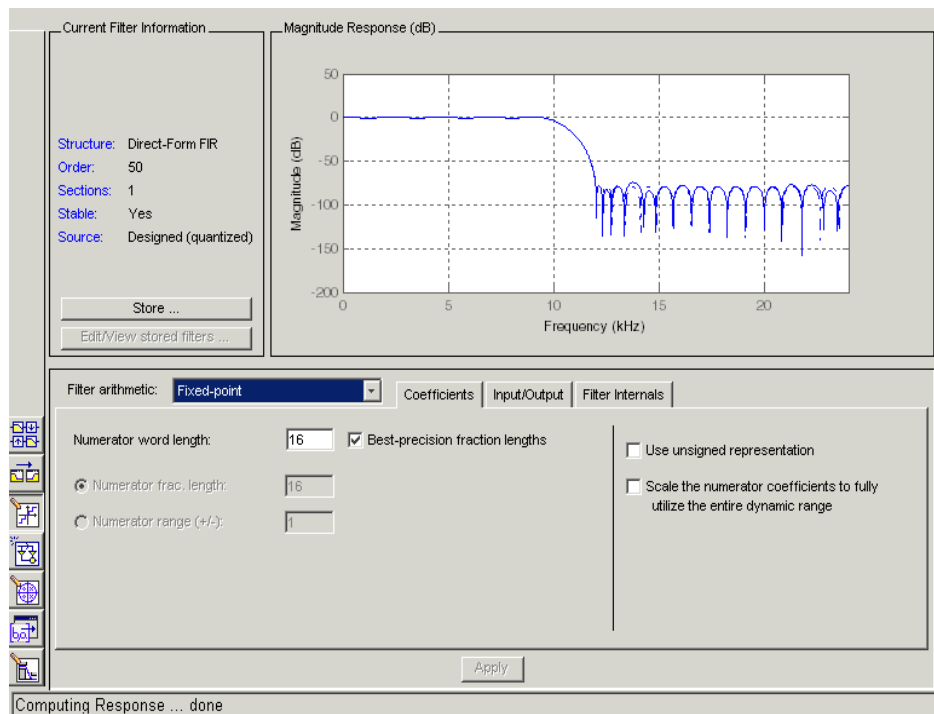
Use the buttons on the sidebar to configure the design area to use various tools in FDATool.

**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 (the design area), as shown in the figure.

**Transform Filter**—clicking this button opens the **Frequency Transformations** pane so you can use digital frequency transformations to change the magnitude response of your filter.

**Create a multirate filter**—clicking this button switches FDATool to multirate filter design mode so you can design interpolators, decimators, and fractional rate change filters.

**Realize Model**—starting from your quantized, direct-form, FIR filter, clicking this button creates a Simulink model of your filter structure in new model window.



Other options in the menu bar let you convert the filter structure to a new structure, change the order of second-order sections in a filter, or change the scaling applied to the filter, among many possibilities.

**Remarks**

By incorporating many advanced filter design methods from Filter Design Toolbox, FDATool provides more design methods than the SPTool Filter Designer.

**See Also**

fdatool, fvtool, sptool in your Signal Processing Toolbox documentation

**Purpose** Create filter specification object

**Syntax**

```
d = fdesign.response
d = fdesign.response(spec)
d = fdesign.response(spec,specvalue1,specvalue2,...)
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 8-544.

Here is how you design filters using `fdesign`.

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

---

**Note** `fdesign` does not create filters. `fdesign` returns a filter specification object that contains the specifications for a filter, such as the passband cutoff or attenuation in the stopband.

To design a filter `hd` from a filter specification object `d`, use `d` with a filter design method such as `butter`—`hd = design(d, 'butter')`.

---

For more guidance about using `fdesign` to design filters, refer to “Designing Fixed-Point Filters” on page 2-3 of the Filter Design Toolbox User’s Guide. This section provides examples that use `fdesign` to design filters and that use methods in the toolbox to analyze them.



*reponse* 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>
arbmag	<code>fdesign.arbmag</code> creates an object to design IIR filters that have arbitrary magnitude responses defined by the input arguments.
arbmagnphase	<code>fdesign.arbmagnphase</code> creates an object to design IIR filters that have arbitrary magnitude and phase responses defined by the input arguments.
bandpass	<code>fdesign.bandpass</code> creates an object to design bandpass filters.
bandstop	<code>fdesign.bandstop</code> creates an object to design bandstop filters.
ciccomp	<code>fdesign.ciccomp</code> creates an object to design filters that compensate for the CIC decimator or interpolator response curves.
decimator	<code>fdesign.decimator</code> creates an object to design decimators.
differentiator	<code>fdesign.differentiator</code> creates an object to design differentiators.
halfband	<code>fdesign.halfband</code> creates an object to design halfband filters.
highpass	<code>fdesign.highpass</code> creates an object to design highpass filters.
hilbert	<code>fdesign.hilbert</code> creates an object to design Hilbert filters.

<b>fdesign Response String</b>	<b>Description</b>
interpolator	<code>fdesign.interpolator</code> creates an object to design interpolators.
isincpl	<code>fdesign.isincpl</code> creates an object to design lowpass filters that use inverse-sinc form.
lowpass	<code>fdesign.lowpass</code> creates an object to design lowpass filters.
nyquist	<code>fdesign.nyquist</code> creates an object to design nyquist filters.
rsrc	<code>fdesign.rsrc</code> creates an object to design rational-factor sample-rate convertors.

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

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

gets more information about the `lowpass` or `bandstop` structure objects.

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

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

**Properties**

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

<b>Property Name</b>	<b>Default Value</b>	<b>Description</b>
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 $F_{stop}$ or the filter order $N$ .
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.

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
<code>DecimationFactor</code>	Specifies the amount to decrease the sampling rate. Always a positive integer.
<code>InterpolationFactor</code>	Specifies the amount to increase the sampling rate. Always a positive integer.
<code>PolyphaseLength</code>	Polyphase length is the length of each polyphase subfilter that composes the decimator or interpolator or rate-change factor filters. Total filter length is the product of <code>p1</code> and the rate change factors. <code>p1</code> must be an even integer.

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

For example, when you create a default lowpass filter specification object `d`, `fdesign` sets the passband frequency `Fpass`, the stopband frequency `Fstop`, the stopband attenuation `Astop`, and the passband attenuation `Apass` (ripple in the passband) for `d`:

```
d = fdesign.lowpass
```

```
d =
```

```
    Response: 'Lowpass'  
Specification: 'Fp,Fst,Ap,Ast'  
Description: {4x1 cell}
```

```
NormalizedFrequency: true
                Fpass: 0.45
                Fstop: 0.55
                Apass: 1
                Astop: 60
```

However, lowpass design syntax accepts any one of the following Spec strings (among others) to define the filter response:

<b>Spec String</b>	<b>Description</b>
Fp,Fst,Ap,Ast	Define the filter by specifying the passband cutoff, the stopband cutoff, the ripple in the passband, and the attenuation in the stopband. This is the default string for a lowpass filter.
N,Fc	Set the filter order and the cutoff frequency to define the filter.
N,Fp,Ap	Set the filter order, passband cutoff frequency, and passband ripple.
N,Fst,Ast	Define the filter by setting the order, stopband frequency, and stopband attenuation.
N,Fp,Ap,Ast	Set the order, passband cutoff frequency, passband ripple, and stopband attenuation.
N,Fp,Fst,Ap	Set the filter order, passband cutoff frequency, stopband frequency, and passband ripple.

Other filter object types, such as Nyquist or highpass, accept a different set of strings for Spec. Refer to the Help system for details about the strings for each filter type.

One important note is that the Spec string you choose controls which design method works for the specifications object.

For the lowpass filter specification object `d` from earlier, you can use `butter`, `cheby1`, `cheby2`, or `ellip` (to name a few) to design a filter. However, if the Spec

string had been 'n,fp,fst,ap', you could only use the `ellip` design method to design your filter.

When you implement this lowpass filter `hd` using a filter design method such as Butterworth (the `butter` design function), the constraints in `fp`, `fst`, `ap`, and `ast` (the default string and filter specification) define the response of the final minimum-order lowpass filter:

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

hd =

    FilterStructure: 'Direct-Form II, Second-Order Sections'
    Arithmetic: 'double'
    sosMatrix: [13x6 double]
    ScaleValues: [14x1 double]
    PersistentMemory: false
```

FVTool shows that `hd` is a lowpass filter that meets the design specification.

`d = fdesign.type(...,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.type(...,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 dB. Note that `fdesign` stores all magnitude specifications in dB (converting to dB 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>
<code>butter</code>	Implement a Butterworth filter resulting in an SOS filter with direct-form II structure.
<code>cheby1</code>	Implement a Chebyshev Type I filter, resulting in a direct-form II second-order filter.
<code>cheby2</code>	Implement a Chebyshev Type II filter, resulting in an SOS filter with direct-form II structure
<code>ellip</code>	Implement an elliptic filter resulting in an SOS filter with direct-form II structure
<code>equiripple</code>	Implement an equiripple filter.
<code>firls</code>	Implement a least-squares filter.
<code>kaiserwin</code>	Implement a filter that uses a Kaiser window.
<code>multistage</code>	Implement a multistage filter

When you use any of the design methods without providing an output argument, the resulting filter design appears in FVTool by default.

Along with filter design methods, `fdesign` works with supporting methods that help you create filter specification objects or determine which design methods work for a given specifications object.

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

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

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

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

## Examples

These examples show a few default filter objects constructed from the MATLAB command prompt, and how to design a Butterworth filter.

Example 1—Halfband filter specification object with filter order and stopband attenuation provided as input arguments. Add the `linear` magunits option so you specify the attenuation in decimal—0.0001.

```
n = 80;
ast = 1e-4;
fs = 48000
d=fdesign.halfband('n,ast',n,ast,fs,'linear') % specifications
object.

d =
```



```

        Response: [1x51 char]
    Specification: 'N,Ast'
        Description: {2x1 cell}
NormalizedFrequency: false
                Fs: 48000
    FilterOrder: 80
        Astop: 80

```

```
d.description
```

```
ans =
```

```

    'Filter Order'
    'Stopband Attenuation (dB)'

```

#### Example 2—Interpolator filter specification object

```
d = fdesign.interpolator % A specifications object.
```

```
d =
```

```

        Response: 'Minimum-order halfband'
    Specification: 'TW,Ast'
        Description: {2x1 cell}
InterpolationFactor: 2
NormalizedFrequency: true
                Fs: 'Normalized'
    TransitionWidth: 0.1000
        Astop: 80

```

```
d.Description
```

```
ans =
```

```

    'Transition Width'
    'Stopband Attenuation (dB)'

```

#### Example 3—Highpass filter specification object

```
d=fdesign.highpass % Creates a specifications object.
```

```
d =
```

```
           Response: 'Minimum-order highpass'  
Specification   : 'Fst,Fp,Ast,Ap'  
Description: {4x1 cell}  
NormalizedFrequency: true  
           Fs: 'Normalized'  
           Fstop: 0.4500  
           Fpass: 0.5500  
           Astop: 60  
           Apass: 1
```

```
d.Description
```

```
ans =
```

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

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

#### Example 4—Lowpass Butterworth filter specification object

Use a filter specification object to construct a lowpass Butterworth filter with default `Specification` `fp`, `fst`, `ap`, `ast`—the edge frequencies of the passband and stopband, the attenuation in the passband, and the attenuation in the stopband. Start by creating the specifications object `d` and providing the filter order and cutoff frequency values.

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

```
d
```

```
d =
```

```
           Response: 'Minimum-order lowpass'
```

```
Specification: 'Fp,Fst,Ap,Ast'  
Description: {4x1 cell}  
NormalizedFrequency: true  
Fs: 'Normalized'  
Fpass: 0.4000  
Fstop: 0.5000  
Apass: 1  
Astop: 80
```

Determine which design methods apply to d.

```
designmethods(d)
```

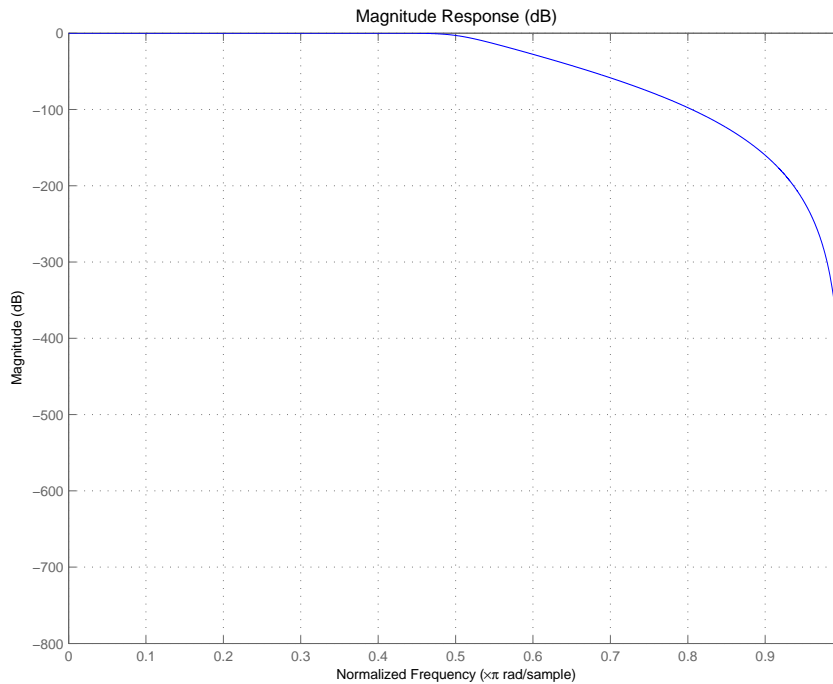
```
Design Methods for class fdesign.lowpass:
```

```
butter  
cheby1  
cheby2  
ellip
```

Now use d and the butter design method to design a Butterworth filter.

```
hd = design(d,'butter','matchexactly','passband'); % A filter.  
fvtool(hd);
```

The resulting filter magnitude response shown by FVTool appears below.



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
```

Notice that only the Kaiser window-based design method applies to default Nyquist filter objects.

## See Also

butter, cheby1, cheby2, designmethods, ellip, equiripple, fdatool, fdesign.bandpass, fdesign.bandstop, fdesign.decimator, fdesign.halfband, fdesign.highpass, fdesign.interpolator, fdesign.lowpass, fdesign.nyquist, fdesign.rsrc, fir1s, fvtool, kaiserwin, setspecs

# fdesign.arbmag

---

**Purpose** Construct filter specification object for designing arbitrary response magnitude filters

**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 following strings. Specification strings are not case sensitive and must be entered as shown.

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. FIR and IIR ( <code>n</code> is the order for both numerator and denominator).
<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 arguments in the strings are

<b>Argument</b>	<b>Description</b>
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 <code>fdesign.arbmagnphase</code> .
b	Number of bands in the multiband filter.
f	Frequency vector. Frequency values in f specify locations where you provide specific filter response amplitudes. When you provide f you must also provide a.
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. Here is an example that 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 a real filter.

Define the frequency vector f as [0 0.1 0.2 0.4 0.5 0.6 0.9 1.0]

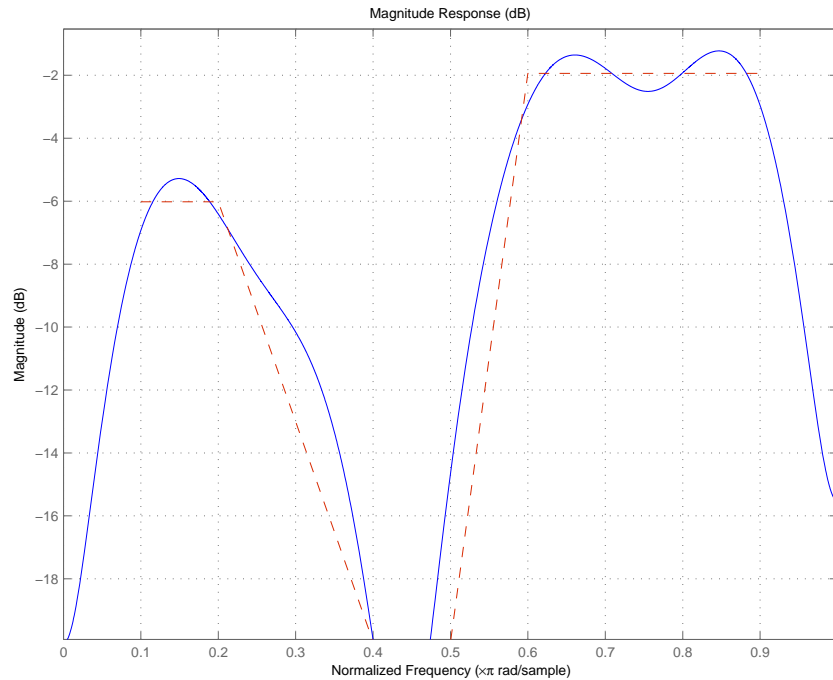
Define the response vector a as [0 0.5 0.5 0.1 0.1 0.8 0.8 0]

With those specifications,  $f$  and  $a$  are connected as follows:

<b><math>f</math> (normalized frequency)</b>	<b><math>a</math> (response desired at <math>f</math>)</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 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  $a$ . A filter that used  $f$  and  $a$  might look like this





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. Notice that the design process uses the default design method for the `n,f,a` specification.

```
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]

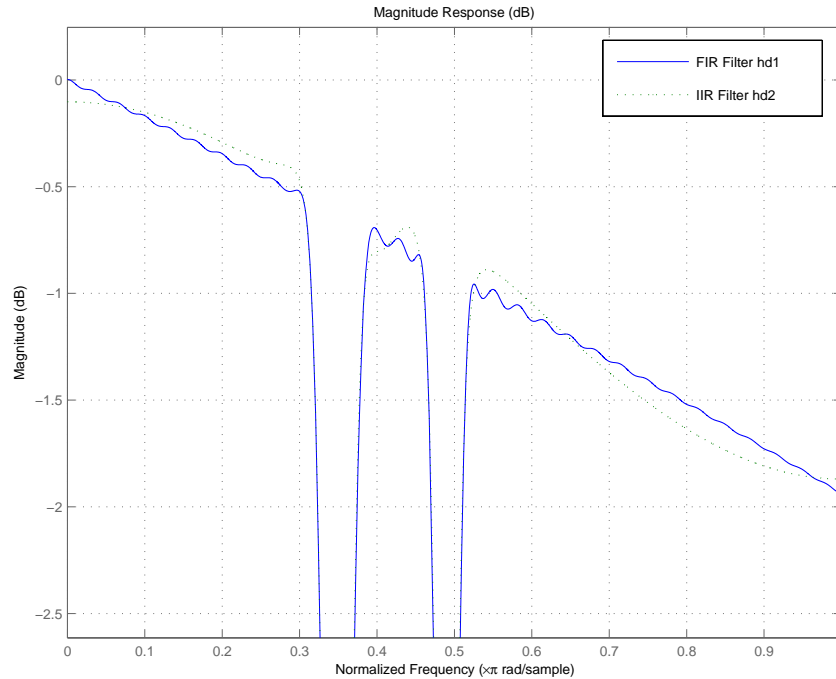
hd2=design(d,'iirlpnorm') % Design an IIR filter from the same object.

hd2 =

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

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

FVTool returns the following plot for the filters.

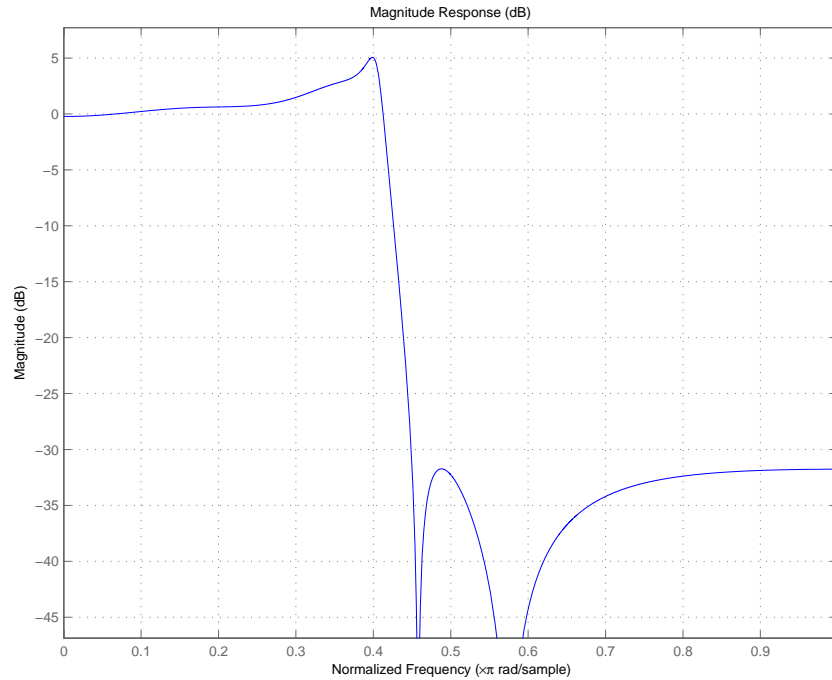


For the third example, design a multiband filter for noise shaping when you are simulating the Rayleigh fading phenomenon in a wireless communications channel. This example uses the default design method for `fdesign.arbmag` specifications objects with the `nb,na,nbands` specification—`iirlpnorm`.

```
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];
```

# fdesign.arbmag

```
d = fdesign.arbmag('nb,na,b,f,a',nb,na,nbands,f1,a1,f2,a2);  
design(d); % Starts FVTool to display the filter response.
```



The filter response shows the characteristic shape for noise shaping—increasing gain with increasing frequency in the passband, and a narrow transition region.

## See Also

`design`, `designopts`, `fdesign`, `setspecs`

**Purpose** Design discrete-time filter specification object for arbitrary magnitude and phase response

**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(specification)` 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 following strings. Specification strings are not case sensitive and must be entered as shown.

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

The arguments in the strings are

Argument	Description
b	Number of bands in the multiband filter.
f	Frequency vector. Frequency values in f specify locations where you provide specific filter response amplitudes. When you provide f you must also provide h which contains the response values.
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. Here is an example that 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 a real 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]

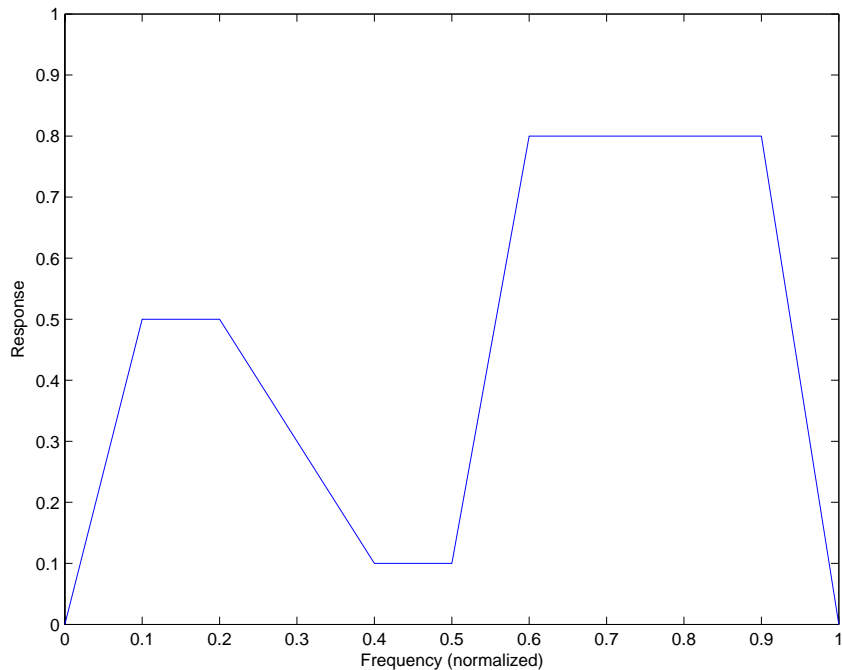
With those specifications,  $f$  and  $h$  are connected as follows:

<b><math>f</math> (normalized frequency)</b>	<b><math>h</math> (response desired at <math>f</math>)</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 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 this figure that resembles a filter with two passbands.

# fdesign.arbmagnphase

---



The second example in Examples shows this in more detail with a complex filter response for  $h$ . In the example,  $h$  uses complex values for the response.

Different specification types often have different design methods available. Use `designmethods(d)` to get a list of design methods available for a given specification string and specifications object.

`d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,...)` initializes the filter specification object with `specvalue1`, `specvalue2`, and so on. Use `get(d,'description')` for descriptions of the various specifications `specvalue1`, `specvalue2`,...`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.

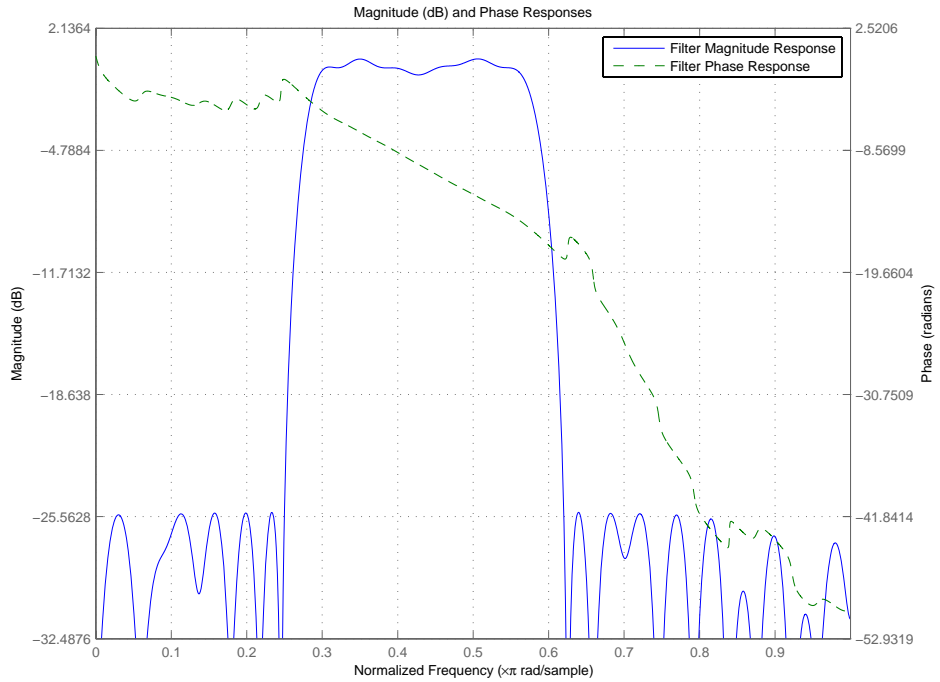
```
d=fdesign.arbmagnphase('n,f,h',100); % N=100, f and h set to defaults.
design(d,'freqsamp');
```

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')
```

Displaying the filter in FVTool shows both the magnitude response and the nearly linear phase.

# fdesign.arbmagnphase



## See Also

fdesign, design, designmethods, setspecs

**Purpose** Construct bandpass filter specification object

**Syntax**

```
d = fdesign.bandpass
d = fdesign.bandpass(spec)
d = fdesign.bandpass(spec,specvalue1,specvalue2,...)
d = fdesign.bandpass(specvalue1,specvalue2,specvalue3,specvalue4,
    specvalue4,specvalue5,specvalue6,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 design method generates a `dfilt` object.

`d = fdesign.bandpass(spec)` constructs object `d` and sets its `Specification` property to `spec`. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown below and used to define the bandpass filter. The strings are not case sensitive.

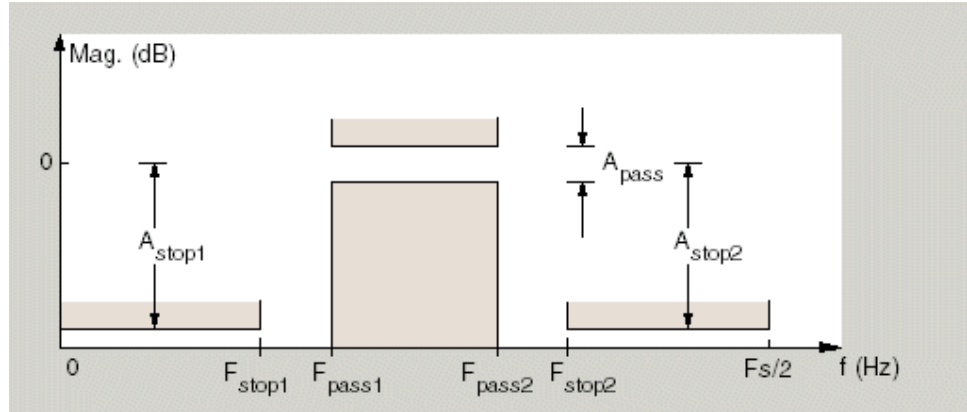
- `fst1,fp1,fp2,fst2,ast1,ap,ast2` (default `spec`)
- `n,f3dB1,f3dB2`
- `n,f3dB1,f3dB2,ap`
- `n,f3dB1,f3dB2,ast`
- `n,f3dB1,f3dB2,ast1,ap,ast2`
- `n,f3dB1,f3dB2,bwp`
- `n,f3dB1,f3dB2,bwst`
- `n,fc1,fc2`
- `n,fp1,fp2,ap`
- `n,fp1,fp2,ast1,ap,ast2`
- `n,fst1,fp1,fp2,fst2`
- `n,fst1,fp1,fp2,fst2,ap`

- `n, fst1, fst2, ast`
- `nb, na, fst1, fp1, fp2, fst2`

The string entries are defined as follows:

- `ap`—amount of ripple allowed in the pass band. Also called `Apass`.
- `ast1`—attenuation in the first stop band in dB (the default units). Also called `Astop1`.
- `ast2`—attenuation in the second stop band in dB (the default units). Also called `Astop2`.
- `bwp`—bandwidth of the filter passband. Specified in normalized frequency units.
- `bwst`—bandwidth of the filter stopband. Specified in normalized frequency units.
- `f3dB1`—cutoff frequency for the point 3dB point below the passband value for the first cutoff. Specified in normalized frequency units. (IIR filters)
- `f3dB2`—cutoff frequency for the point 3dB point below the passband value for the second cutoff. Specified in normalized frequency units. (IIR filters)
- `fc1`—cutoff frequency for the point 3dB point below the passband value for the first cutoff. Specified in normalized frequency units. (FIR filters)
- `fc2`—cutoff frequency for the point 3dB point below the passband value for the second cutoff. Specified in normalized frequency units. (FIR filters)
- `fp1`—frequency at the edge of the start of the pass band. Specified in normalized frequency units. Also called `Fpass1`.
- `fp2`—frequency at the edge of the end of the pass band. Specified in normalized frequency units. Also called `Fpass2`.
- `fst1`—frequency at the edge of the start of the first stop band. Specified in normalized frequency units. Also called `Fstop1`.
- `fst2`—frequency at the edge of the start of the second stop band. Specified in normalized frequency units. Also called `Fstop2`.
- `n`—filter order for FIR filters. Or both the numerator and denominator orders for IIR filters when `na` and `nb` are not provided.
- `na`—denominator order for IIR filters
- `nb`—numerator order for IIR filters

Graphically, the filter specifications look like this.



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 design methods to determine which design method applies to an object and its specification string.

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

`d = fdesign.bandpass(specvalue1, specvalue2, specvalue3, specvalue4, specvalue4, specvalue5, specvalue6)` constructs `d`, an object with the default Specification property string, using the values you provide as input arguments for `specvalue1`, `specvalue2`, `specvalue3`, `specvalue4`, `specvalue4`, `specvalue5`, `specvalue6` and `specvalue7`.

`d = fdesign.bandpass(..., fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

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

- `linear`—specify the magnitude in linear units

# fdesign.bandpass

---

- 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 dB. Note that fdesign stores all magnitude specifications in dB (converting to dB when necessary) regardless of how you specify the magnitudes.

## Examples

These examples show how to construct a bandpass filter specification object. First, create a default specifications object without using input arguments.

```
d = fdesign.bandpass
d =

    Response: 'Minimum-order bandpass'
    Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
    Description: {7x1 cell}
    NormalizedFrequency: true
    Fstop1: 0.3500
    Fpass1: 0.4500
    Fpass2: 0.5500
    Fstop2: 0.6500
    Astop1: 60
    Apass: 1
    Astop2: 60
```

Now, pass the filter specifications that correspond to the default Specification—fst1,fp1,fp2,fst2,ast1,ap,ast2—without specifying the Specification string. Notice that we add fs as the final input argument to specify the sampling frequency of 48 Hz.

```
d = fdesign.bandpass(10, 12, 14, 16, 80, .5, 60, 48)
d =

    Response: 'Minimum-order bandpass'
    Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
    Description: {7x1 cell}
    NormalizedFrequency: false
    Fs: 48
    Fstop1: 10
    Fpass1: 12
```

```
Fpass2: 14
Fstop2: 16
Astop1: 80
Apass: 0.5000
Astop2: 60
```

Next create a specifications object by passing a specification type string 'n,fc1,fc2'—the resulting object uses default values for n, fc1, and fc2.

```
d = fdesign.bandpass('n,fc1,fc2')
d =
```

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

Create the same filter, passing the specification values to the object rather than accepting the default values for n, fc1, and fc2. Notice that you can include the sampling frequency fs as the final input argument, and that you specify the cutoff frequencies in Hz since fs is in Hz.

```
d = fdesign.bandpass('n,fc1,fc2', 10, 9600, 14400, 48000)
d =
```

```
Response: 'Bandpass with cutoff'
Specification: 'N,Fc1,Fc2'
Description: {3x1 cell}
NormalizedFrequency: false
Fs: 48000
FilterOrder: 10
Fcutoff1: 9600
Fcutoff2: 14400
```

## See Also

fdesign, fdesign.bandstop, fdesign.highpass, fdesign.lowpass

# fdesign.bandstop

---

**Purpose** Construct bandstop filter specification object

**Syntax**

```
d = fdesign.bandstop
d = fdesign.bandstop(spec)
d = fdesign.bandstop(spec,specvalue1,specvalue2,...)
d = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,
    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` (default `spec`)
- `n,f3dB1,f3dB2`
- `n,f3dB1,f3dB2,ap`
- `n,f3dB1,f3dB2,ap,ast`
- `n,f3dB1,f3dB2,ast`
- `n,f3dB1,f3dB2,bwp`
- `n,f3dB1,f3dB2,bwst`
- `n,fc1,fc2`
- `n,fp1,fp2,ap`
- `n,fp1,fp2,ap,ast`
- `n,fp1,fst1,fst2,fp2`
- `n,fp1,fst1,fst2,fp2,ap`
- `n,fst1,fst2,ast`
- `nb,na,fp1,fst1,fst2,fp2`

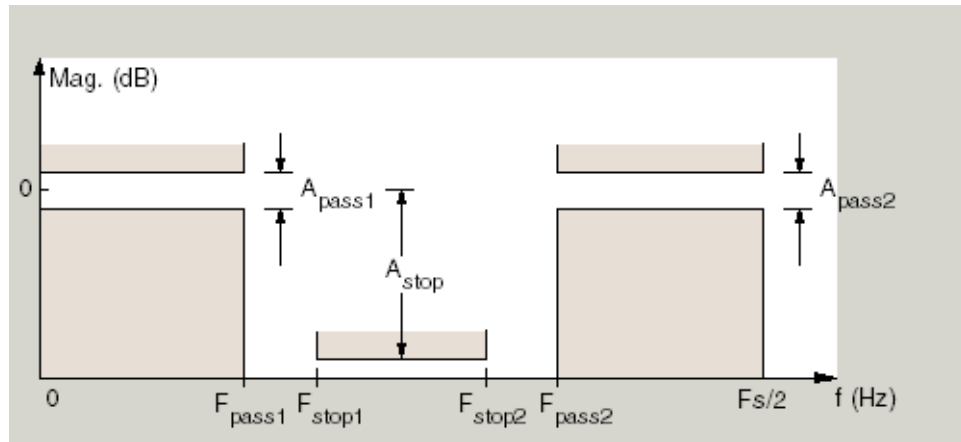
The string entries are defined as follows:



- **ap**—amount of ripple allowed in the pass band in dB (the default units). Also called **Apass**.
- **ast**—attenuation in the first stop band in dB (the default units). Also called **Astop1**.
- **bwp**—bandwidth of the filter passband. Specified in normalized frequency units.
- **bwst**—bandwidth of the filter stopband. Specified in normalized frequency units.
- **f3dB1**—cutoff frequency for the point 3dB point below the passband value for the first cutoff. Specified in normalized frequency units.
- **f3dB2**—cutoff frequency for the point 3dB point below the passband value for the second cutoff. Specified in normalized frequency units.
- **fp1**—frequency at the start of the pass band. Specified in normalized frequency units. Also called **Fpass1**.
- **fp2**—frequency at the end of the pass band. Specified in normalized frequency units. Also called **Fpass2**.
- **fst1**—frequency at the end of the first stop band. Specified in normalized frequency units. Also called **Fstop1**.
- **fst2**—frequency at the start of the second stop band. Specified in normalized frequency units. Also called **Fstop2**.
- **n**—filter order.
- **na**—denominator order for IIR filters
- **nb**—numerator order for IIR filters.

Graphically, the filter specifications look like this:

# fdesign.bandstop



Regions between specification values like  $f_{p1}$  and  $f_{s1}$  are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandstop filter specification object change depending on the Specification string. Use designmethods to determine which design method applies to an object and its specification string.

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

`d = fdesign.bandstop(fpass1,fstop1,fstop2,fpass2,apass1,...,astop,apass2)` constructs an object `d` with the default Specification property string, using the values you provide for `specvalue1`, `specvalue2`, `specvalue3`, `specvalue4`, `specvalue4`, `specvalue5`, `specvalue6` and `specvalue7`.

`d = fdesign.bandstop(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

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

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

- `squared`—specify the magnitude in power units

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

## Examples

These examples show how to construct a bandpass filter specification object. First, create a default specifications object without using input arguments.

```
d = fdesign.bandstop
d =

    Response: 'Minimum-order bandstop'
    Description: {7x1 cell}
    Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
    NormalizedFrequency: true
    Fpass1: 0.3500
    Fstop1: 0.4500
    Fstop2: 0.5500
    Fpass2: 0.6500
    Apass1: 1
    Astop: 60
    Apass2: 1
```

Now create an object by passing a specification type string `'n,fc1,fc2'`—the resulting object uses default values for `n`, `fc1`, and `fc2`.

```
d=fdesign.bandstop('n,f3dB1,f3dB2')
d =

    Response: 'Bandstop with cutoff'
    Specification: 'N,F3dB1,F3dB2'
    Description: {3x1 cell}
    NormalizedFrequency: true
    FilterOrder: 10
    Fcutoff1: 0.4000
    Fcutoff2: 0.6000
```

```
designmethods(d)
```

# fdesign.bandstop

---

Design Methods for class fdesign.bandstop:

```
butter
cheby1
cheby2
ellip
```

Create another bandstop filter, passing the specification values to the object rather than accepting the default values for `n`, `f3db1`, and `fc2`. Notice that you can add `fs` as the final input argument to specify the sampling frequency of 48 kHz.

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

```
d =
```

```
           Response: 'Bandstop with cutoff'
Specification: 'N,F3dB1,F3dB2'
Description: {3x1 cell}
NormalizedFrequency: false
           Fs: 48000
FilterOrder: 10
      Fcutoff1: 9600
      Fcutoff2: 14400
```

For this bandstop filter, pass the filter specifications that correspond to the default Specification—`fp1`, `fst1`, `fst2`, `fp2`, `ap1`, `ast`, `ap2`.

```
d = fdesign.bandstop(0.3,0.4,0.6,0.7,0.5,60,1)
```

```
d =
```

```
           Response: 'Minimum-order bandstop'
Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
Description: {7x1 cell}
NormalizedFrequency: true
      Fpass1: 0.3000
      Fstop1: 0.4000
      Fstop2: 0.6000
      Fpass2: 0.7000
```

```
    Apass1: 0.5000
    Astop: 60
    Apass2: 1
```

And for the final example, pass the magnitude specifications in squared units, using the `magunits` option squared.

```
d = fdesign.bandstop(0.4,0.5,0.6,0.7,0.98,0.01,0.99,'squared')
d =
```

```
    Response: 'Minimum-order bandstop'
    Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
    Description: {7x1 cell}
    NormalizedFrequency: true
    Fpass1: 0.4000
    Fstop1: 0.5000
    Fstop2: 0.6000
    Fpass2: 0.7000
    Apass1: 0.0877
    Astop: 20
    Apass2: 0.0436
```

## See Also

`fdesign`, `fdesign.bandpass`, `fdesign.highpass`, `fdesign.lowpass`

# fdesign.ciccomp

---

**Purpose** Construct filter cascaded-integrator comb (CIC) compensator filter specification object

**Syntax**

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

**Description** `h = fdesign.ciccomp` constructs a CIC compensator specifications object `d`, applying default values for the properties `Fpass`, `Fstop`, `Apass`, and `Astop`. In this syntax, the filter has two sections and the differential delay is 1.

Using `fdesign.ciccomp` with a design method creates a `dfilt` object, a single-rate discrete-time filter.

`h = fdesign.ciccomp(d,nsections)` constructs a CIC compensator specifications object with the filter differential delay set to `d` and the number of sections in the filter set to `nsections`. By default, `d` and `nsections` are 1 and 2 if you omit them as input arguments.

`h = fdesign.ciccomp(...,spec)` constructs a CIC Compensator specifications object and sets its `Specification` property to `spec`. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown in the list below. The strings are not case sensitive.

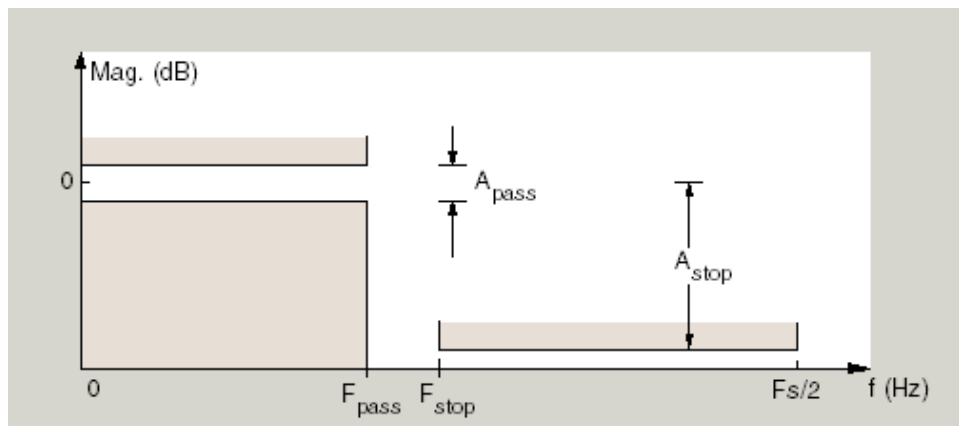
- `fp,fst,ap,ast` (default `spec`)
- `n,fc,ap,ast`
- `n,fp,ap,ast`
- `n,fp,fst`
- `n,fst,ap,ast`

The string entries are defined as follows:

- `ap`—amount of ripple allowed in the pass band in dB (the default units). Also called `Apass`.
- `ast`—attenuation in the stop band in dB (the default units). Also called `Astop`.

- $f_c$ —cutoff frequency for the point 3dB point below the passband value. Specified in normalized frequency units.
- $f_p$ —frequency at the end of the pass band. Specified in normalized frequency units. Also called  $F_{pass}$ .
- $f_{st}$ —frequency at the start of the stop band. Specified in normalized frequency units. Also called  $F_{stop}$ .
- $n$ —filter order.

In graphic form, the filter specifications look like this:



Regions between specification values like  $f_p$  and  $f_{st}$  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.

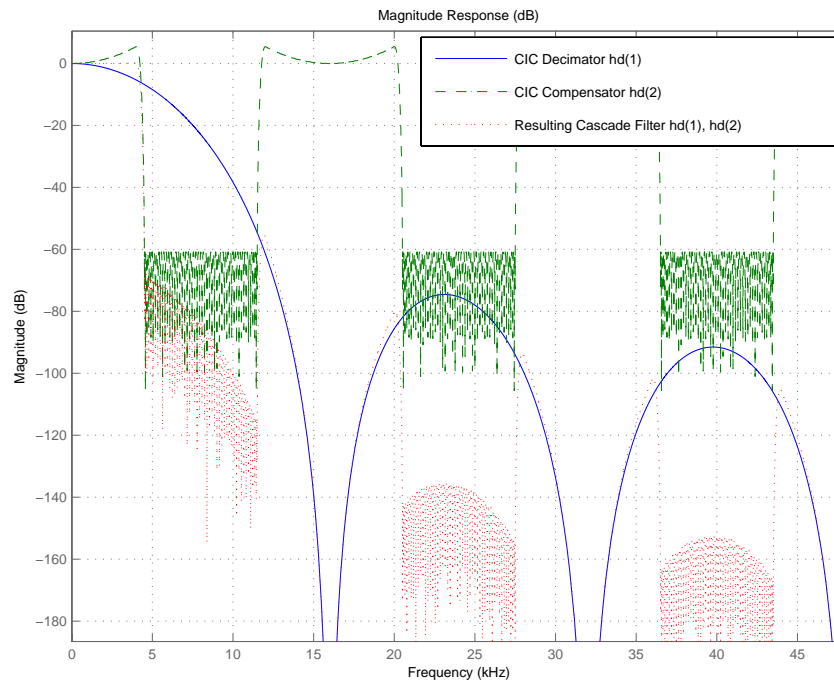
`h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)` constructs an object and sets the specifications in the order they are specified in the spec input when you construct the object.

### 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,  $F_{pass}$ .

By taking the number of sections, passband, and differential delay from your CIC filter and using them in the definition of the CIC compensator, the



resulting compensator filter effectively corrects for the passband droop of the CIC filter, and narrows the transition region.

As a demonstration of this concept, this example creates a CIC decimator and its compensator.

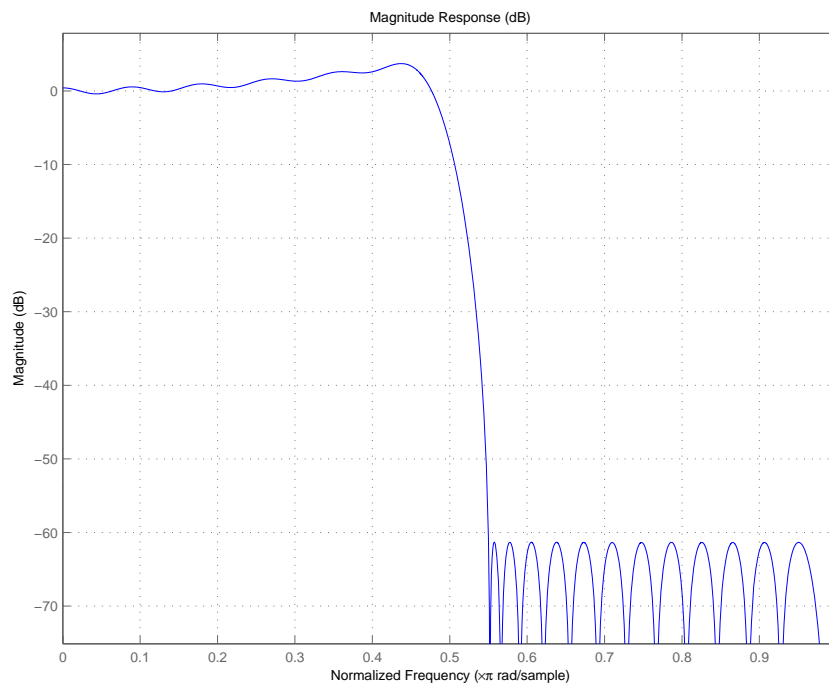
```
fs = 96e3; % Input sampling frequency.
fpass = 4e3; % Frequency band of interest.
m = 6; % Decimation factor.
hcic = design(fdesign.decimator(m, 'cic', 1, fpass, 60, fs));
hd = cascade(dfilt.scalar(1/gain(hcic)), hcic);
hd(2) = design(fdesign.ciccomp(hcic.differentialdelay, ...
    hcic.numberofsections, fpass, 4.5e3, .1, 60, fs/m));
fvtool(hd(1), hd(2), cascade(hd(1), hd(2)), 'Fs', [96e3 96e3/m 96e3])
```

You see the results in the preceding plot.

## Examples

Designed to compensate for the roll-off inherent in CIC filters, CIC compensators can improve the performance of your CIC design. This example designs a compensator *d* with five sections and a differential delay equal to one. The plot displayed after the code shows the increasing gain in the passband that is characteristic of CIC compensators, to overcome the droop in the CIC filter passband. Ideally, cascading the CIC compensator with the CIC filter results in a lowpass filter with flat passband response and narrow transition region.

```
h = fdesign.ciccomp;
set(h, 'NumberOfSections', 5, 'DifferentialDelay', 1);
hd = equiripple(h);
fvtool(hd);
```



This compensator would work for a decimator or interpolator that had differential delay of 1 and 5 sections.

**See Also**

`fdesign.decimator`, `fdesign.interpolator`

**Purpose** Construct 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
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.
ciccomp	Sets the design for the decimator specifications object to CIC compensator.
halfband	Sets the design for the decimator specifications object to halfband.

# fdesign.decimator

<b>design String (Continued)</b>	<b>Description</b>
highpass	Sets the design for the decimator specifications object to highpass.
isinclp	Sets the design for the decimator specifications object to inverse-sinc lowpass.
lowpass	Sets the design for the decimator specifications object to lowpass.
nyquist	Sets the design for the decimator specifications object to Nyquist.

Notice the entries in the first column. They match the design method names. However, when you create your specifications object, the Response property contains the full name of the response, such as CIC Compensator or Inverse-Sinc Lowpass, rather than the shorter method names `isinclp` or `ciccomp`. So, when you seek to design a new filter object, use the design method name shown in 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.

Notice that the decimation factor  $m$  is not in the specification strings. Various design types provide different specifications, as shown in this table. .

<b>Design Type</b>	<b>Valid Specification Strings</b>
Bandpass	<ul style="list-style-type: none"> <li>• fst1, fp1, fp2, fst2, ast1, ap, ast2 (default string)</li> <li>• n, fc1, fc2</li> <li>• n, fst1, fp1, fp2, fst2</li> </ul>
Bandstop	<ul style="list-style-type: none"> <li>• n, fc1, fc2</li> <li>• n, fp1, fst1, fst2, fp2</li> <li>• fp1, fst1, fst2, fp2, ap1, ast, ap2 (default string)</li> </ul>
CIC	<ul style="list-style-type: none"> <li>• fp, ast (default and only string)</li> </ul>
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>
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>

# fdesign.decimator

Design Type	Valid Specification Strings
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,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>

The string entries are defined as follows:

- ap—amount of ripple allowed in the pass band in dB (the default units). Also called Apass.

- `ap1`—amount of ripple allowed in the pass band in dB (the default units). Also called `Apass1`. Bandpass and bandstop filters use this option.
- `ap2`—amount of ripple allowed in the pass band in dB (the default units). Also called `Apass2`. Bandpass and bandstop filters use this option.
- `ast`—attenuation in the first stop band in dB (the default units). Also called `Astop`.
- `ast1`—attenuation in the first stop band in dB (the default units). Also called `Astop1`. Bandpass and bandstop filters use this option.
- `ast2`—attenuation in the first stop band in dB (the default units). Also called `Astop2`. Bandpass and bandstop filters use this option.
- `fc1`—cutoff frequency for the point 3dB 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 3dB 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.
- `n`—filter order.
- `tw`—width of the transition region between the pass and stop bands. Both halfband and Nyquist filters use this option.

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

# fdesign.decimator

---

`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 dB. Note that `fdesign` stores all magnitude specifications in dB (converting to dB 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(2,0.1,80) % Set tw=0.1, and ast=80.
```

```
d =
```

```
      MultirateType: 'Decimator'  
      Response: 'Nyquist'  
      DecimationFactor: 2  
      Specification: 'TW,Ast'  
      Description: {'Transition Width';'Stopband Attenuation (dB)'}  
      NormalizedFrequency: true  
      TransitionWidth: 0.1  
      Astop: 80
```

Now create an object by passing a specification type string `'fst1,fp1,fp2,fst2,ast1,ap,ast2'` and a design—the resulting object uses default values for the filter specifications. You must provide the design input argument, `bandpass` in this example, when you include a specification.

```
d=fdesign.decimator(8,'bandpass', 'fst1,fp1,fp2,fst2,...  
ast1,ap,ast2')
```

```
d =
```



```
MultirateType: 'Decimator'  
    Response: 'Bandpass'  
DecimationFactor: 8  
    Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'  
    Description: {7x1 cell}  
NormalizedFrequency: true  
    Fstop1: 0.35  
    Fpass1: 0.45  
    Fpass2: 0.55  
    Fstop2: 0.65  
    Astop1: 60  
    Apass: 1  
    Astop2: 60
```

Create another decimating filter specification object, passing the specification values to the object rather than accepting the default values for fp,fst,ap,ast.

```
d=fdesign.decimator(3,'lowpass',.45,0.55,.1,60)
```

```
d =
```

```
MultirateType: 'Decimator'  
    Response: 'Lowpass'  
DecimationFactor: 3  
    Specification: 'Fp,Fst,Ap,Ast'  
    Description: {4x1 cell}  
NormalizedFrequency: true  
    Fpass: 0.45  
    Fstop: 0.55  
    Apass: 0.1  
    Astop: 60
```

Now pass the filter specifications that correspond to the specifications—  
n,fc,ap,ast.

```
d=fdesign.decimator(3,'cic compensator','n,fc,ap,ast',...  
20,0.45,.05,50)
```

```
d =
```

# fdesign.decimator

---

```
    MultirateType: 'Decimator'  
        Response: 'CIC Compensator'  
DecimationFactor: 3  
    Specification: 'N,Fc,Ap,Ast'  
        Description: {4x1 cell}  
    NumberOfSections: 2  
    DifferentialDelay: 1  
    NormalizedFrequency: true  
        FilterOrder: 20  
            Fcutoff: 0.45  
                Apass: 0.05  
                Astop: 50
```

Now design a decimator using the kaiserwin design method.

```
hm = kaiserwin(d)
```

Pass a new specification type for the filter, specifying the filter order. Note that the inputs must include the differential delay `dd` with the CIC input argument to design a CIC specification object.

```
m = 5;  
dd = 2;  
d = fdesign.decimator(m,'cic',dd,'fp,ast',0.55,55)
```

```
d =
```

```
    MultirateType: 'Decimator'  
        Response: 'CIC'  
DecimationFactor: 5  
    Specification: 'Fp,Ast'  
        Description: {'Passband Frequency'; 'Stopband Attenuation(dB)'}  
    DifferentialDelay: 2  
    NormalizedFrequency: true  
        Fpass: 0.55
```

In this example, you specify a sampling frequency as the last input argument.

```
d=fdesign.decimator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,...  
ap,ast2',0.25,0.35,.55,.65,50,.05,50,1e3) % Fs = 1000 Hz.
```

```
d =
```

```
    MultirateType: 'Decimator'
```

```
Response: 'Bandpass'  
DecimationFactor: 8  
Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'  
Description: {7x1 cell}  
NormalizedFrequency: false  
Fs: 1000  
Fstop1: 0.25  
Fpass1: 0.35  
Fpass2: 0.55  
Fstop2: 0.65  
Astop1: 50  
Apass: 0.05  
Astop2: 50
```

In this, the last example, use the linear option for the filter specification object and specify the stopband ripple attenuation in linear format.

```
hs = fdesign.decimator(4, 'lowpass', 'n,fst,ap,ast',15,0.55,.05,50,...  
1e-3,'linear') % 1e-3 = 60dB.
```

```
hs =
```

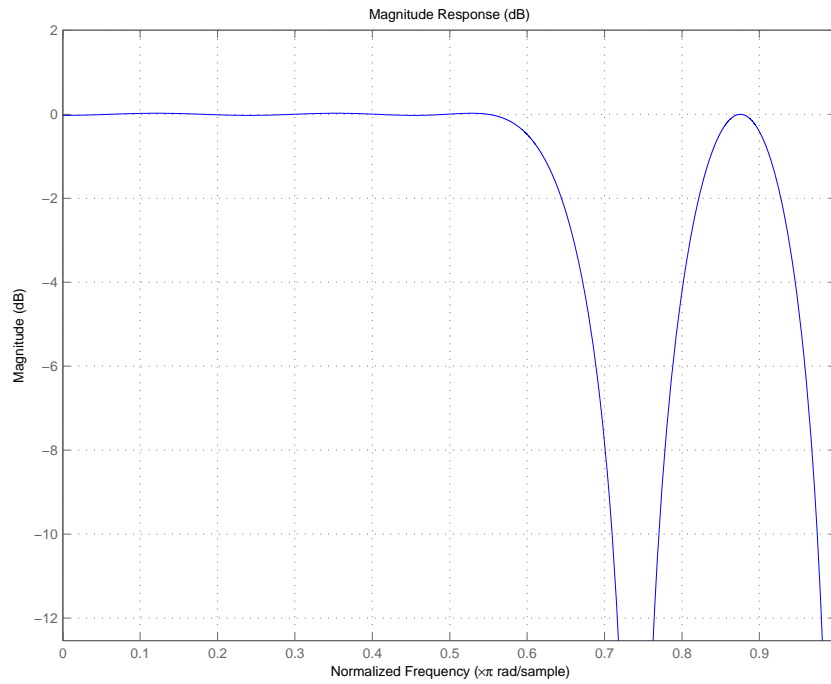
```
Response: 'Lowpass decimator'  
Specification: 'TW,Ast'  
Description: {'Transition Width';'Stopband Attenuation (dB)'}  
DecimationFactor: 4  
NormalizedFrequency: false  
Fs: 500  
TransitionWidth: 0.1  
Astop: 60
```

Design the filter and display the magnitude response in FVTool.

```
designmethods(hs);  
equiripple(hs); % Starts FVTool to display the response.
```

# fdesign.decimator

---



## See Also

fdesign, fdesign.interpolator, fdesign.rsrc

**Purpose** Construct 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` 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 dB (the default units). Also called `Apass`.
- `ast`—attenuation in the stop band in dB (the default units). Also called `Astop`.
- `fp`—frequency at the start of the pass band. Specified in normalized frequency units. Also called `Fpass`.
- `fst`—frequency at the end of the stop band. Specified in normalized frequency units. Also called `Fstop`.
- `n`—filter order.

By default, `fdesign.differentiator` assumes that all frequency specifications are provided in normalized frequency units. Also, dB is the default for all magnitude specifications.

# fdesign.differentiator

---

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 dB. Note that `fdesign` stores all magnitude specifications in dB (converting to dB when necessary) regardless of how you specify the magnitudes.

## Examples

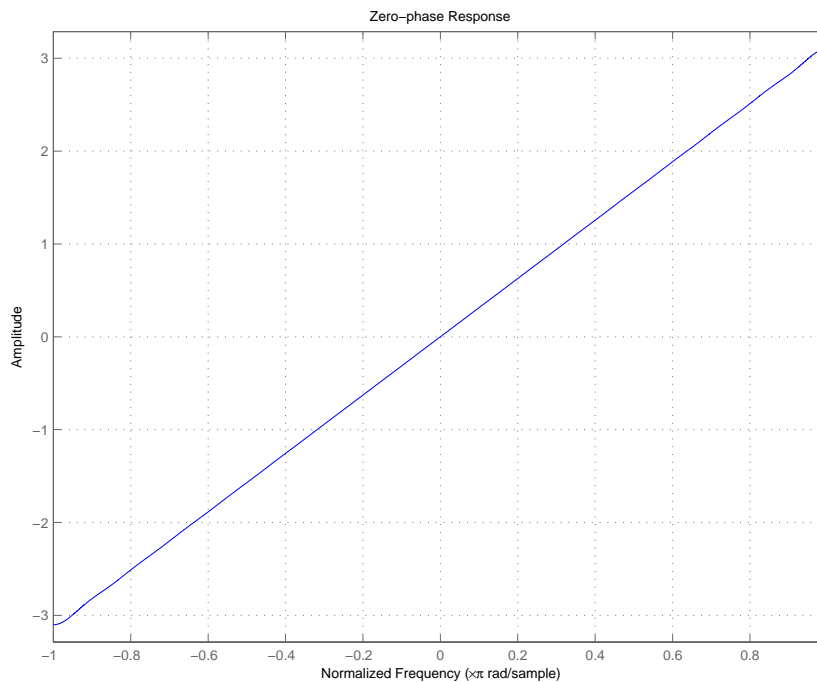
The toolbox lets you design a range of differentiators. These examples present a few possible designs. The first example designs a 33rd-order full band differentiator. The FVTool plot following the code shows the resulting 33rd-order filter.

```
d = fdesign.differentiator(33); % N is the filter order of 33.  
designmethods(d);
```

```
hd = design(d,'firls');  
fvtool(hd,'magnitudedisplay','zero-phase','frequencyrange',...  
    '[-pi, pi]')
```

Design Methods for class `fdesign.differentiator` (N):

`equiripple`  
`firls`

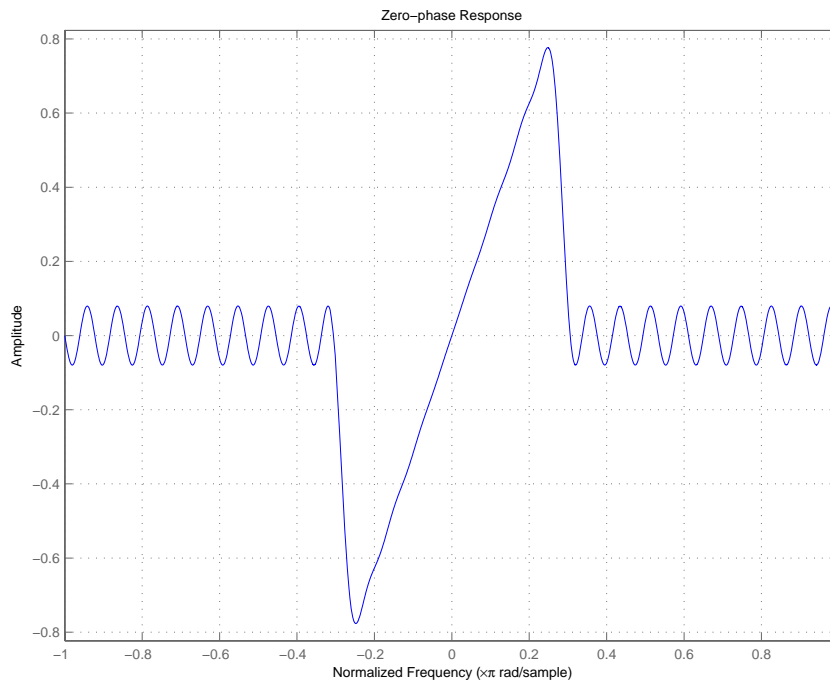


For the second example, design a narrow band differentiator. Differentiate the first 25 percent of the frequencies in the Nyquist range and filter the higher frequencies.

# fdesign.differentiator

```
d = fdesign.differentiator('n,fp,fst',54,.25,.3);  
designmethods(d);  
hd = design(d,'equiripple');  
fvtool(hd,'magnitudedisplay','zero-phase');  
set(hf,'frequencyrange','[-fs/2, fs/2]')
```

Here is the view from FVTool.

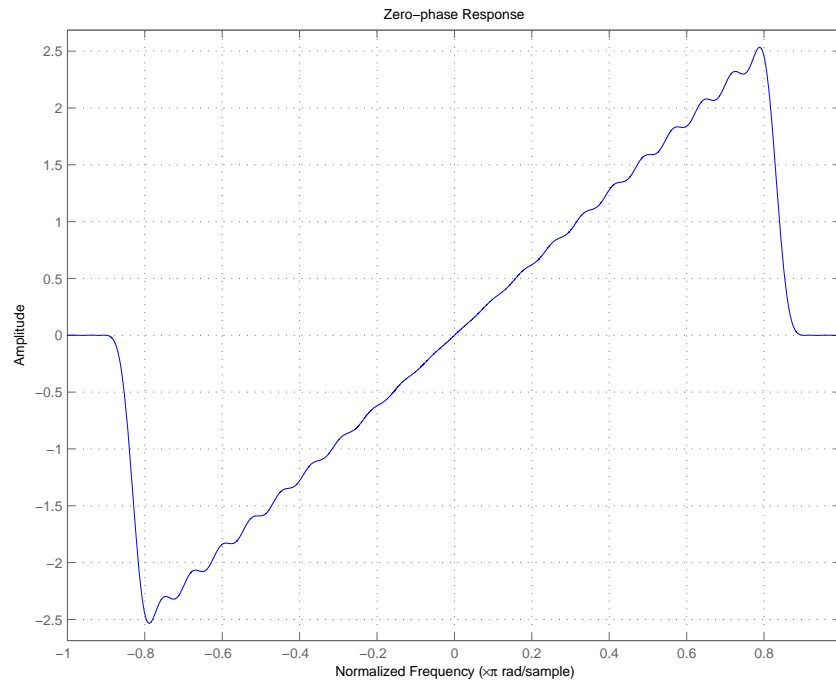


Finally, design a minimum-order, wide-band differentiator.

```
d = fdesign.differentiator('fp,fst,ap,ast',.8,.9,1,80);  
designmethods(d);  
hd = design(d,'equiripple');  
fvtool(hd,'magnitudedisplay','zero-phase','frequencyrange')
```

FVTool returns this plot.





**See Also** [design](#), [fdesign](#), [setspecs](#)

# fdesign.halfband

---

**Purpose** Construct halfband filter specification object

**Syntax**

```
d = fdesign.halfband
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(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 dB (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 halfband filter specification object change depending on the Specification string. Use `designmethods` to determine which design method applies to an object and its specification string.

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

`d = fdesign.halfband(specvalue1,specvalue2)` constructs an object `d` assuming the default Specification property string `tw,ast`, using the values

you provide for the input arguments `specvalue1` and `specvalue2` for `tw` and `ast`.

`d = fdesign.halfband(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

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

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

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

## Examples

These examples show how to construct a halfband filter specification object. First, create a default specifications object without using input arguments.

```
d=fdesign.halfband

d =

    Response: 'Minimum-order halfband'
  Specification: 'TW,Ast'
    Description: {2x1 cell}
  NormalizedFrequency: true
    TransitionWidth: 0.1000
           Astop: 80
```

Now create an object by passing a specification type string `'n,ast'`—the resulting object uses default values for `n` and `ast`.

```
d=fdesign.halfband('n,ast')

d =

    Response: 'Halfband with filter order and stopband attenuation'
  Specification: 'N,Ast'
    Description: {2x1 cell}
```

# fdesign.halfband

---

```
NormalizedFrequency: true
FilterOrder: 10
Astop: 80
```

Create another halfband filter object, passing the specification values to the object rather than accepting the default values for `n` and `ast`.

```
d = fdesign.halfband('n,ast', 42, 80)

d =

    Response: 'Halfband with filter order and stopband attenuation'
Specification: 'N,Ast'
Description: {2x1 cell}
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: 'Minimum-order halfband'
Specification: 'TW,Ast'
Description: {2x1 cell}
NormalizedFrequency: true
TransitionWidth: 0.0100
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

hs =

    Response: [1x51 char]
Specification: 'N,Ast'
Description: {2x1 cell}
NormalizedFrequency: true
FilterOrder: 80
Astop: 70
```

```
equiripple(hs); % Opens FVTool automatically.
```

In the final example, pass the 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: [1x47 char]
    Specification: 'N,TW'
      Description: {2x1 cell}
NormalizedFrequency: true
      FilterOrder: 42
  TransitionWidth: 0.0400
```

```
designmethods(hs)
```

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

```
equiripple
kaiserwin
firls
```

```
hd=firls(hs)
```

```
hd =
```

```
FilterStructure: 'Direct-Form FIR'
  Arithmetic: 'double'
    Numerator: [1x43 double]
PersistentMemory: false
      States: [42x1 double]
```

## See Also

`fdesign`, `fdesign.decimator`, `fdesign.interpolator`, `fdesign.nyquist`

# fdesign.highpass

---

**Purpose** Construct highpass filter specification object

**Syntax**

```
d = fdesign.highpass
d = fdesign.highpass(spec)
d = fdesign.highpass(spec,specvalue1,specvalue2,...)
d = fdesign.highpass(specvalue1,specvalue2,specvalue3,specvalue4)
d = fdesign.highpass(...,fs)
d = fdesign.highpass(...,magunits)
```

**Description** `d = fdesign.highpass` constructs a highpass filter specification object `d`, applying default values for the properties `fst`, `fp`, `ast` and `ap`.

Using `fdesign.highpass` with a design method generates a `dfilt` object.

`d = fdesign.highpass(spec)` constructs object `d` and sets its 'Specification' to `spec`. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown below. The strings are not case sensitive.

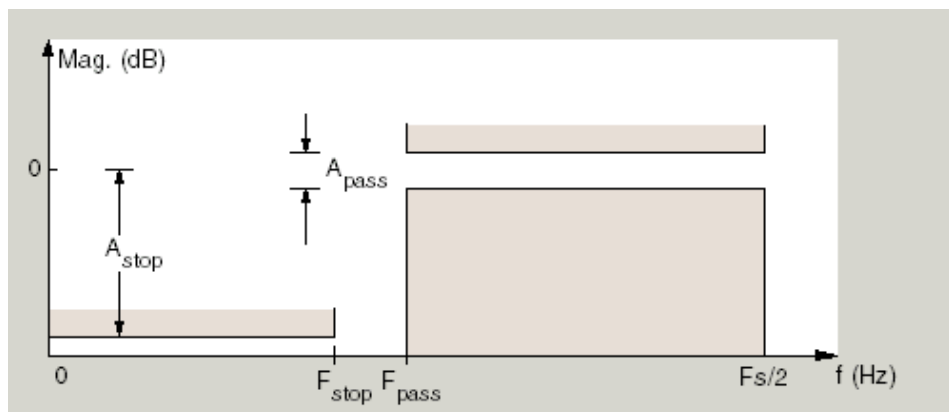
- `fst,fp,ast,ap` (default `spec`)
- `n,f3db`
- `n,f3db,ap`
- `n,f3db,ast`
- `n,f3db,ast,ap`
- `n,f3db,fp`
- `n,fc`
- `n,fc,ast,ap`
- `n,fp,ap`
- `n,fp,ast,ap`
- `n,fst,ast`
- `n,fst,ast,ap`
- `n,fst,f3db`
- `n,fst,fp`
- `n,fst,fp,ap`
- `n,fst,fp,ast`

- nb, na, fst, fp

The string entries are defined as follows:

- ap—amount of ripple allowed in the pass band in dB (the default units). Also called  $A_{\text{pass}}$ .
- ast—attenuation in the stop band in dB (the default units). Also called  $A_{\text{stop}}$ .
- f3db—cutoff frequency for the point 3dB point below the passband value. Specified in normalized frequency units.
- fc—cutoff frequency for the point 3dB point below the passband value. Specified in normalized frequency units.
- fp—frequency at the start of the pass band. Specified in normalized frequency units. Also called  $F_{\text{pass}}$ .
- fst—frequency at the end of the stop band. Specified in normalized frequency units. Also called  $F_{\text{stop}}$ .
- n—filter order.
- na and nb are the order of the denominator and numerator.

Graphically, the filter specifications look like this:



Regions between specification values like  $f_{\text{st}1}$  and  $f_p$  are transition regions where the filter response is not explicitly defined.

# fdesign.highpass

---

The filter design methods that apply to a highpass filter specification object change depending on the Specification string. Use designmethods to determine which design method applies to an object and its specification string.

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

`d = fdesign.highpass(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `d` with the values for the default Specification property string, using the specifications you provide as input arguments `specvalue1,specvalue2,specvalue3,specvalue4`.

`d = fdesign.highpass(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

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

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

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in dB. Note that `fdesign` stores all magnitude specifications in dB (converting to dB 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
```

This time, pass the specifications that correspond to the default Specification string.

```
hs = fdesign.highpass(.4,.5,80,1);
```

```
hs =
```

```
Response: 'Minimum-order highpass'  
Specification: 'Fst,Fp,Ast,Ap'  
Description: {4x1 cell}  
NormalizedFrequency: true  
Fstop: 0.4000  
Fpass: 0.5000  
Astop: 80  
Apass: 1
```

Now create an object by passing a specification type string 'n,fc'—the resulting object uses default values for n and fc.

```
d=fdesign.highpass('n,fc')
```

```
d =
```

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

Create the same filter, passing the values for n and fc rather than accepting the default values. Notice that you can add include the sampling frequency fs as the final input argument. Adding fs puts all the frequency specifications into linear frequency format, rather than normalized frequency.

```
d=fdesign.highpass('n,fc',10,9600,48000)
```

```
d =
```

## fdesign.highpass

---

```
Response: 'Highpass with cutoff'  
Specification: 'N,Fc'  
Description: {2x1 cell}  
NormalizedFrequency: false  
Fs: 48000  
FilterOrder: 10  
Fcutoff: 9600
```

Finally, pass values for the filter specifications that match the default Specification string—`fp = 10`, `fst = 12`, `ast = 80` and `ap = 0.5`. Add the sampling frequency on the end.

```
d=fdesign.highpass(10,12,80,0.5,48000)
```

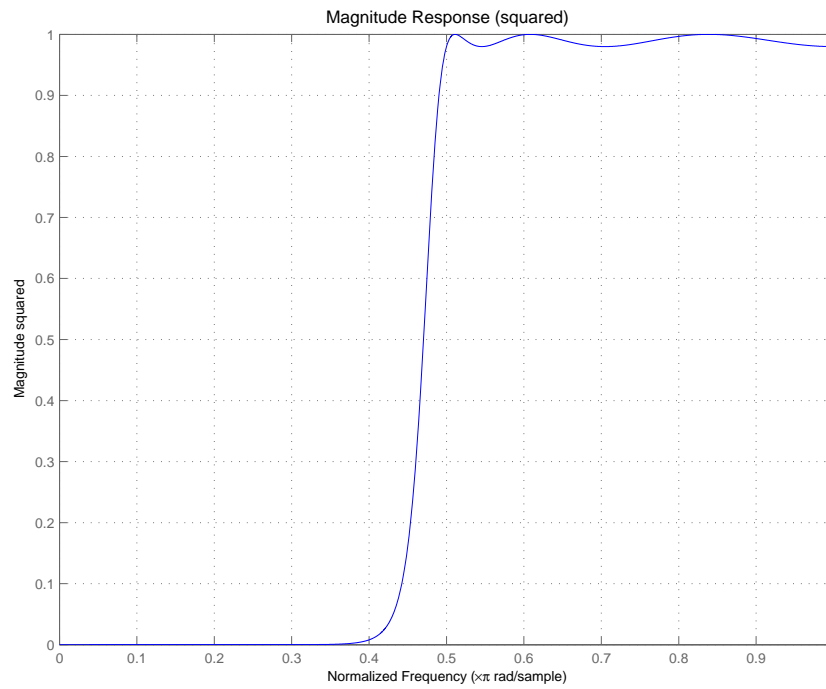
```
d =
```

```
Response: 'Minimum-order highpass'  
Specification: 'Fst,Fp,Ast,Ap'  
Description: {4x1 cell}  
NormalizedFrequency: false  
Fs: 48000  
Fstop: 10  
Fpass: 12  
Astop: 80
```

To demonstrate the `magunits` input option, pass the magnitude specifications in squared units and include the squared input argument for `magunits`.

```
hs = fdesign.highpass(.4, .5, .02, .98, 'squared');  
hd = cheby1(hs);  
fvtool(hd,'MagnitudeDisplay','Magnitude Squared');
```

The figure below shows the filter response.



## See Also

`fdesign`, `fdesign.bandpass`, `fdesign.bandstop`, `fdesign.lowpass`

# fdesign.hilbert

---

**Purpose** Construct 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 dB (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, dB 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 dB. Note that `fdesign` stores all magnitude specifications in dB (converting to dB when necessary) regardless of how you specify the magnitudes.

## Examples

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

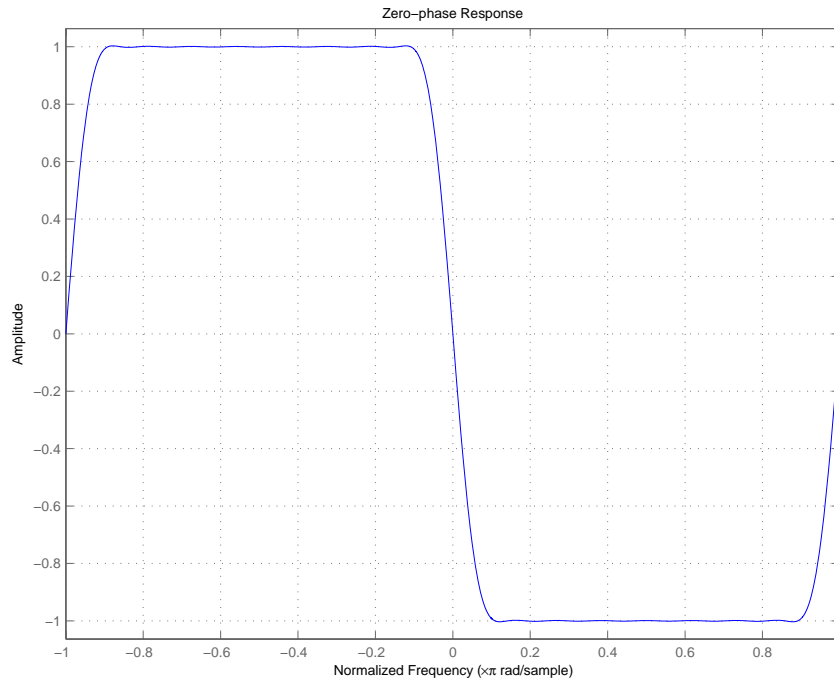
```
d = fdesign.hilbert(30,0.2); % n,tw specification string.
designmethods(d);

hd = design(d,'firls');
fvtool(hd,'magnitudedisplay','zero-phase','frequencyrange',...
'[-pi, pi]')
```

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

```
ellip
iirlinphase
```

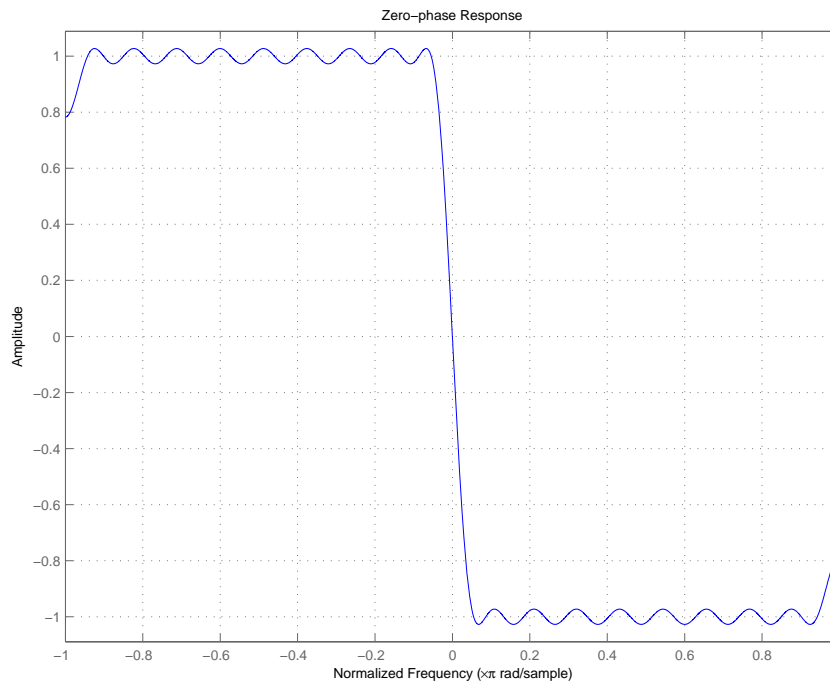
equiripple  
firls



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

```
d = fdesign.hilbert('n,tw',35,0.1);  
designmethods(d);  
hd = design(d,'equiripple');  
hf = fvtool(hd,'magnitudedisplay','zero-phase','frequencyrange')  
set(hf,'frequencyrange',[-fs/2, fs/2])
```

Here is the view from FVTool.

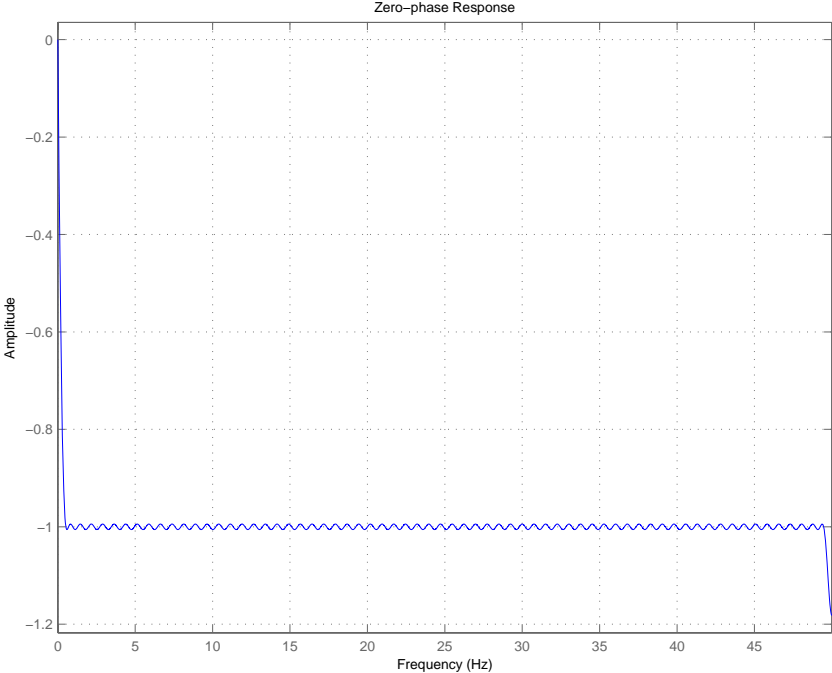


Finally, design a minimum-order transformer that has a sampling frequency of 100 Hz—add  $F_s$  as an input argument in Hz.

```
d = fdesign.hilbert('tw,ap',1,0.1,100); % Fs = 100 Hz.
designmethods(d);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase');
set(hf,'frequencyrange','[-fs/2, fs/2]')
```

FVTool returns this plot.

# fdesign.hilbert



**See Also** design, fdesign, setspecs



**Purpose** Design quasi-linear phase IIR filter from halfband filter specification object

**Syntax**

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

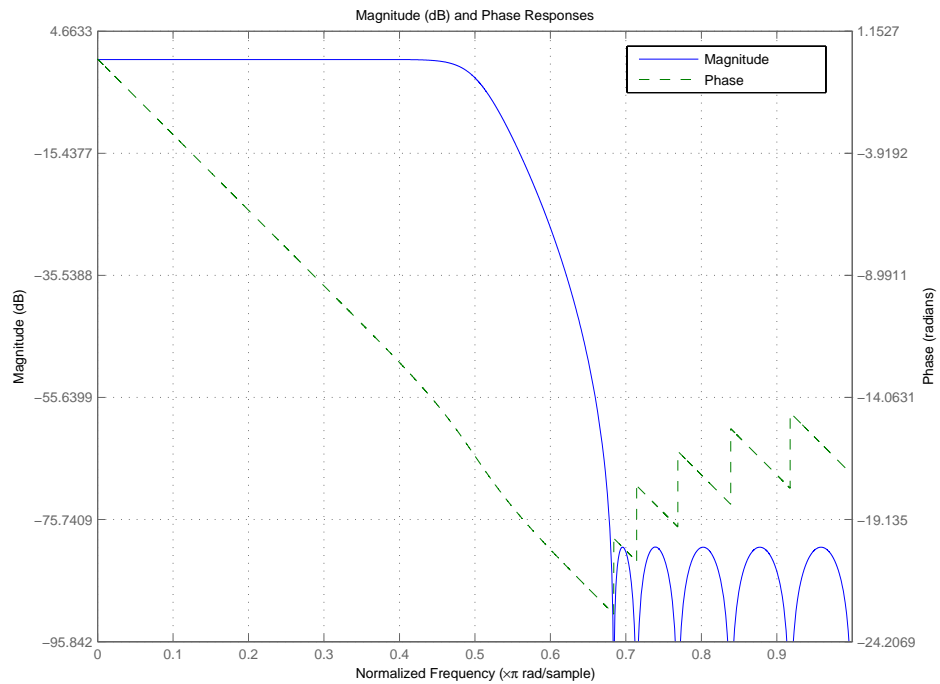
Notice the characteristic halfband nature of the ripple in the stopband. If you measure the resulting filter, you see it meets the specifications.

# iirlinphase

```
measure(hd)
```

```
ans =
```

```
Sampling Frequency : N/A (normalized frequency)  
Passband Edge      : 0.32  
3-dB Point         : 0.5  
6-dB Point         : 0.51911  
Stopband Edge      : 0.68  
Passband Ripple    : 4.0866e-008 dB  
Stopband Atten.    : 80.2642 dB  
Transition Width    : 0.36
```



**See Also**

`fdesign.nyquist`

**Purpose** Construct interpolator filter specification object

**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
Bandpass	Sets the response for the interpolator specifications object to bandpass.
Bandstop	Sets the response for the interpolator specifications object to bandstop.
CIC	Sets the response for the interpolator specifications object to CIC filter.
CIC Compensator	Sets the response for the interpolator specifications object to CIC compensator.
Halfband	Sets the response for the interpolator specifications object to halfband.

# fdesign.interpolator

<b>design String (Continued)</b>	<b>Description</b>
Highpass	Sets the response for the interpolator specifications object to highpass.
Inverse-Sinc Lowpass	Sets the response for the interpolator specifications object to inverse-sinc lowpass.
Lowpass	Sets the response for the interpolator specifications object to lowpass.
Nyquist	Sets the response for the interpolator specifications object to Nyquist.

`d = fdesign.interpolator(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.

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.

Notice that the interpolation factor `1` is not in the specification strings. Various design types provide different specifications, as shown in this table.

<b>Design Type</b>	<b>Valid Specification Strings</b>
Arbitrary Magnitude	<ul style="list-style-type: none"><li>• <code>n,b,f,a</code></li><li>• <code>n,f,a</code></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>

<b>Design Type</b>	<b>Valid Specification Strings</b>
Bandstop	<ul style="list-style-type: none"> <li>• n, fc1, fc2</li> <li>• n, fp1, fst1, fst2, fp2</li> <li>• fp1, fst1, fst2, fp2, ap1, ast, ap2 (default string)</li> </ul>
CIC	<ul style="list-style-type: none"> <li>• fp, ast (default and only string)</li> </ul>
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>
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>
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>

Design Type	Valid Specification Strings
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>

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 dB (the default units). Also called Apass.
- ap1—amount of ripple allowed in the pass band in dB (the default units). Also called Apass1. Bandpass and bandstop filters use this option.
- ap2—amount of ripple allowed in the pass band in dB (the default units). Also called Apass2. Bandpass and bandstop filters use this option.
- ast—attenuation in the first stop band in dB (the default units). Also called Astop.
- ast1—attenuation in the first stop band in dB (the default units). Also called Astop1. Bandpass and bandstop filters use this option.
- ast2—attenuation in the first stop band in dB (the default units). Also called Astop2. Bandpass and bandstop filters use this option.
- b—number of filter bands.

- `f`—vector of specific frequency points in the filter response. In combination with `a`, this specifies the desired filter response.
- `fc1`—cutoff frequency for the point 3dB 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 3dB 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.
- `n`—filter order.
- `tw`—width of the transition region between the pass and stop bands. Halfband, Hilbert, and Nyquist filters use this option.

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

`d = fdesign.interpolator(...,fs)` adds the argument `fs`, specified in Hz, to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

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

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

# fdesign.interpolator

---

- squared—specify the magnitude in power units.

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

## Examples

These examples show how to construct interpolating filter specification objects. First, create a default specifications object without using input arguments except for the interpolation factor `l`.

```
l = 2;
d = fdesign.interpolator(2)

d =

    MultirateType: 'Interpolator'
      Response: 'Nyquist'
DecimationFactor: 2
  Specification: 'TW,Ast'
  Description: {'Transition Width';'Stopband Attenuation (dB)'}
NormalizedFrequency: true
  TransitionWidth: 0.1
           Astop: 80
```

Now create an object by passing a specification string `'fst1,fp1,fp2,fst2,ast1,ap,ast2'` and a design—the resulting object uses default values for all of the filter specifications. You must provide the design input argument when you include a specification.

```
d=fdesign.interpolator(8,'bandpass','fst1,fp1,fp2,fst2,...
ast1,ap,ast2')

d =

    MultirateType: 'Interpolator'
      Response: 'Bandpass'
DecimationFactor: 8
  Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
  Description: {7x1 cell}
NormalizedFrequency: true
      Fstop1: 0.35
      Fpass1: 0.45
```



```
Fpass2: 0.55  
Fstop2: 0.65  
Astop1: 60  
Apass: 1  
Astop2: 60
```

Create another interpolating filter object, passing the specification values to the object rather than accepting the default values for, in this case, fp,fst,ap,ast.

```
d=fdesign.interpolator(3,'lowpass',.45,0.55,.1,60)
```

```
d =
```

```
    MultirateType: 'Interpolator'  
        Response: 'Lowpass'  
DecimationFactor: 3  
    Specification: 'Fp,Fst,Ap,Ast'  
        Description: {4x1 cell}  
NormalizedFrequency: true  
        Fpass: 0.45  
        Fstop: 0.55  
        Apass: 0.1  
        Astop: 60
```

Now pass the filter specifications that correspond to the specifications—n,fc,ap,ast.

```
d=fdesign.interpolator(3,'cic compensator','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
```

# fdesign.interpolator

---

```
FilterOrder: 20
Fcutoff: 0.45
Apass: 0.05
Astop: 50
```

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

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

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

```
d = fdesign.interpolator(5,'CIC','fp,ast',0.55,55)
```

```
d =
```

```
MultirateType: 'Interpolator'
Response: 'CIC'
DecimationFactor: 5
Specification: 'Fp,Aa'
Description: {'Passband Frequency';'Stopband Attenuation(dB)'}
DifferentialDelay: 1
NormalizedFrequency: true
Fpass: 0.55
```

In this example, you specify a sampling frequency as the rightmost input argument.

```
d=fdesign.interpolator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,...
ap,ast2',0.25,0.35,.55,.65,50,.05,50,1e3) % Fs = 1000 Hz.
```

```
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,...
    50,1e3,'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 to display the response.
```

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.

Notice that 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 =
```

```
MultirateType: 'Interpolator'
InterpolationFactor: 32
Response: 'CIC'
Specification: 'Fp,Ast'
Description: {'Passband Frequency';'Imaging Attenuation(dB)'}
DifferentialDelay: 2
NormalizedFrequency: false
```

# fdesign.interpolator

---

```
        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'
```

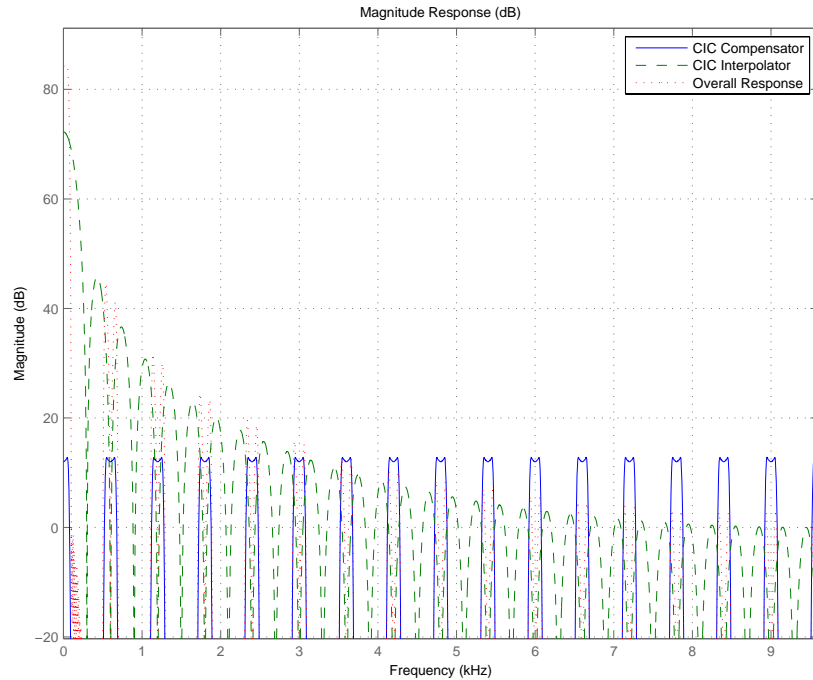
This next example results in a minimum-order CIC compensator that interpolates by 4 and compensates for the droop in the passband for the CIC filter `hm` from the previous example.

```
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.

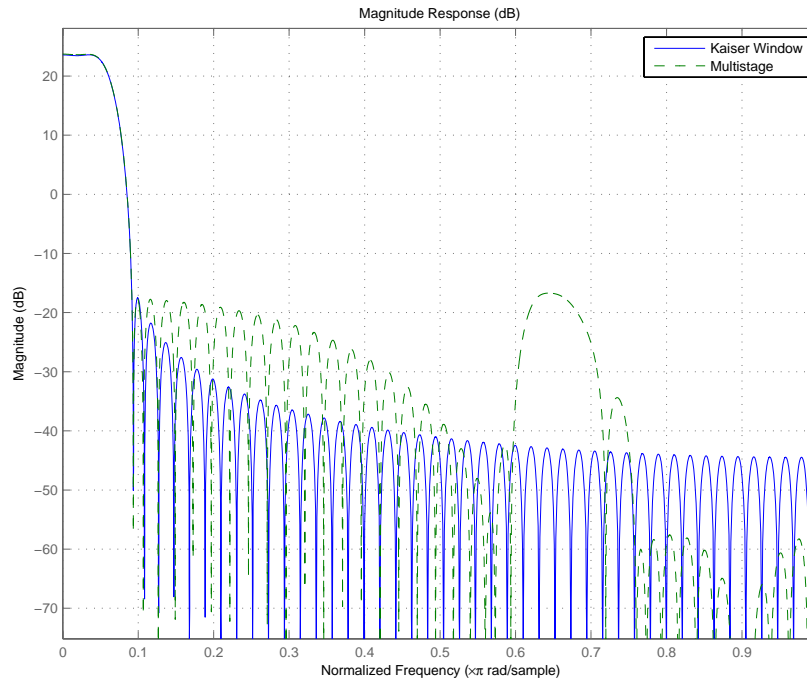


For the third example, use `fdesign.interpolator` to design a minimum-order Nyquist interpolator that uses a Kaiser window. For comparison, design a multistage interpolator as well and compare the responses.

```

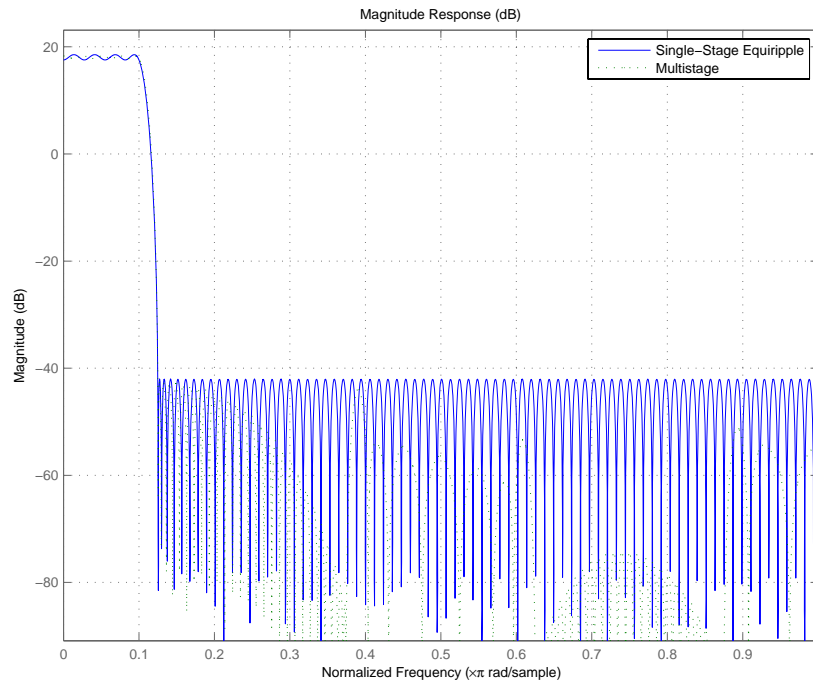
l = 15; % Set the interpolation factor and the Nyquist band.
tw = 0.05; % Specify the normalized transition width.
ast = 40; % Set the minimum stopband attenuation in dB.
d = fdesign.interpolator(1,'nyquist',1,tw,ast);
hm = design(d,'kaiserwin');
hm2 = design(d,'multistage'); % Design the multistage 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');
hm(2) = design(d,'multistage','usehalfbands',true); % Use...
                                                % halfband filters whenever possible.
fvtool(hm);
legend('Single-Stage Equiripple','Multistage')
```



## See Also

`fdesign`, `fdesign.decimator`, `fdesign.rsrc`, `setspecs`

# fdesign.isinclp

---

**Purpose** Construct inverse-sinc filter specification object

**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 dB (the default units). Also called `Astop`.
- `ap`—amount of ripple allowed in the pass band in dB (the default units). Also called `Apass`.
- `fp`—frequency at the start of the pass band. Specified in normalized frequency units. Also called `Fpass`.
- `fst`—frequency at the end of the first stop band. Specified in normalized frequency units. Also called `Fstop`.
- `n`—filter order.

The filter design methods that apply to an inverse-sinc lowpass filter specification object change depending on the Specification string. Use `designmethods` to determine which design method applies to an object and its specification string.



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

`d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `d` assuming the default Specification property string `fp,fst,ap,ast`, using the values you provide in `specvalue1,specvalue2,specvalue3`, and `specvalue4`.

`d = fdesign.isinclp(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

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

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

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in dB. Note that `fdesign` stores all magnitude specifications in dB (converting to dB 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** Construct lowpass filter specification object

**Syntax**

```
d = fdesign.lowpass
d = fdesign.lowpass(spec)
d = fdesign.lowpass(spec,specvalue1,specvalue2, )
d = fdesign.lowpass(specvalue1,specvalue2,specvalue3,specvalue4)
d = fdesign.lowpass(...,fs)
d = fdesign.lowpass(...,magunits)
```

**Description** `d = fdesign.lowpass` constructs a lowpass filter specification object `d`, applying default values for the properties `fp`, `fst`, `ap`, and `ast`.

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

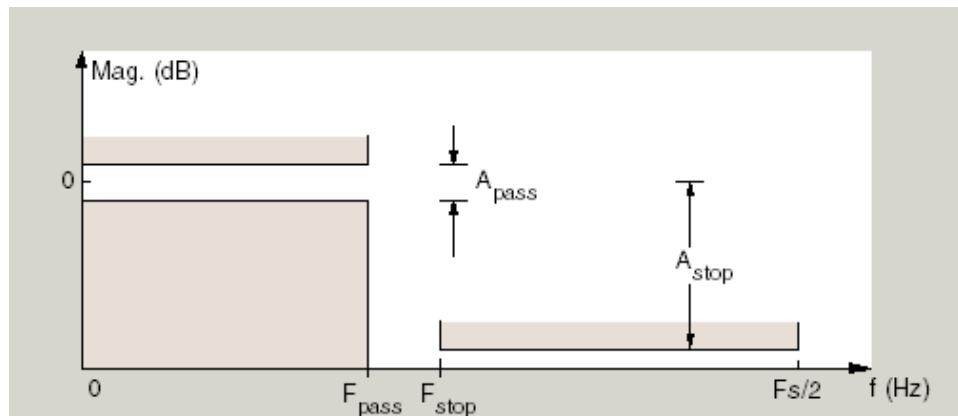
`d = fdesign.lowpass(spec)` constructs object `d` and sets its 'Specification' to `spec`. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown below. The strings are not case sensitive.

- `fp,fst,ap,ast` (default `spec`)
- `n,f3db`
- `n,f3db,ap`
- `n,f3db,ap,ast`
- `n,f3db,ast`
- `n,f3db,fst`
- `n,fc`
- `n,fc,ap,ast`
- `n,fp,ap`
- `n,fp,ap,ast`
- `n,fp,fst,ap`
- `n,fp,f3db`
- `n,fp,fst`
- `n,fp,fst,ap`
- `n,fp,fst,ast`
- `n,fst,ap,ast`
- `n,fst,ast`
- `nb,na,fp,fst`

The string entries are defined as follows:

- $a_p$ —amount of ripple allowed in the pass band in dB (the default units). Also called  $A_{pass}$ .
- $a_{st}$ —attenuation in the stop band in dB (the default units). Also called  $A_{stop}$ .
- $f_{3db}$ —cutoff frequency for the point 3dB point below the passband value. Specified in normalized frequency units.
- $f_c$ —cutoff frequency for the point 3dB point below the passband value. Specified in normalized frequency units.
- $f_p$ —frequency at the start of the pass band. Specified in normalized frequency units. Also called  $F_{pass}$ .
- $f_{st}$ —frequency at the end of the stop band. Specified in normalized frequency units. Also called  $F_{stop}$ .
- $n$ —filter order.
- $n_a$  and  $n_b$  are the order of the denominator and numerator.

Graphically, the filter specifications look like this:



Regions between specification values like  $f_p$  and  $f_{st}$  are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a lowpass filter specification object change depending on the Specification string. Here are all the valid strings for lowpass filter specification objects.

- $f_p, f_{st}, a_p, a_{st}$

# fdesign.lowpass

---

- `n, f3dB`
- `n, f3dB, Ap`
- `n, f3dB, Ap, Ast`
- `n, f3dB, Ast`
- `n, f3dB, Fst`
- `n, fc`
- `n, fc, Ap, Ast`
- `n, fp, ap`
- `n, fp, ap, ast`
- `n, fp, f3db`
- `n, fp, fst`
- `n, fp, fst, ap`
- `n, fp, fst, ast`
- `n, fst, ap, ast`
- `n, fst, ast`
- `n, fp, ap, ast`
- `nb, na, fp, fst`

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

`d = fdesign.lowpass(fp, fst, ap, ast)` constructs an object `d` with values for the default Specification property string `fp, fst, ap, ast` using the specifications you provide as input arguments `specvalue1, specvalue2, specvalue3, specvalue4`.

`d = fdesign.lowpass(..., fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

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

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

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in dB. Note that `fdesign` stores all magnitude specifications in dB

(converting to dB when necessary) regardless of how you specify the magnitudes.

## Examples

These examples show how to construct a lowpass filter specification object. First, create a default lowpass filter object without using input arguments.

```
d=fdesign.lowpass

d =

    Response: 'Minimum-order lowpass'
  Specification: 'Fp,Fst,Ap,Ast'
   Description: {4x1 cell}
NormalizedFrequency: true
           Fpass: 0.4500
           Fstop: 0.5500
           Apass: 1
           Astop: 60
```

Now create an object by passing specifications for the passband and stopband edge frequencies and the passband and stopband attenuations—the resulting object uses the input values for `fp`, `fst`, `ap`, and `ast`.

```
hs = fdesign.lowpass(.4,.5,1,80);
hs

hs =

    Response: 'Minimum-order lowpass'
  Specification: 'Fp,Fst,Ap,Ast'
   Description: {4x1 cell}
NormalizedFrequency: true
           Fpass: 0.4000
           Fstop: 0.5000
           Apass: 1
           Astop: 80
```

Create another filter object, passing the values for `n` and `fc` rather than accepting the default values. Notice that you can also include the sampling frequency `fs` as the final input argument.

# fdesign.lowpass

---

```
d=fdesign.lowpass('n,fc',10, 9600,48000)
```

```
d =
```

```
           Response: 'Lowpass with cutoff'  
Specification: 'N,Fc'  
Description: {2x1 cell}  
NormalizedFrequency: false  
                Fs: 48000  
FilterOrder: 10  
          Fcutoff: 9600
```

Finally, pass values for the filter specifications that match the default Specification string entries— $f_p = 0.4$ ,  $f_{st} = 0.5$ ,  $a_{st} = 80$  and  $a_p = 1.0$ . Add the sampling frequency on the end.

```
hs = fdesign.lowpass(.4,.5,1,80)
```

```
hs =
```

```
           Response: 'Minimum-order lowpass'  
Specification: 'Fp,Fst,Ap,Ast'  
Description: {4x1 cell}  
NormalizedFrequency: true  
                Fpass: 0.4000  
                Fstop: 0.5000  
                Apass: 1  
                Astop: 80
```

Finally, the next examples add the sampling frequency specification in Hz, and then the magunits option.

```
hs = fdesign.lowpass('N,Fp,Ap', 10, 9600, .5, 48000);
```

and

```
hsmag = fdesign.lowpass(.4, .5, .98, .02, 'squared');
```

Using the last example filter object, create a highpass filter.

```
hd = design(hsmag,'cheby1');
```

## See Also

fdesign, fdesign.bandpass, fdesign.bandstop, fdesign.highpass

**Purpose** Construct Nyquist filter specification object

**Syntax**

```
d = fdesign.nyquist
d = fdesign.nyquist(l,spec)
d = fdesign.nyquist(l,spec,specvalue1,specvalue2, )
d = fdesign.nyquist(l,specvalue1,specvalue2)
d = fdesign.nyquist(...,fs)
d = fdesign.nyquist(...,magunits)
```

**Description** `d = fdesign.nyquist` constructs a Nyquist or L-band filter specification object `d`, applying default values for the properties `tw` and `ast`. By default, the filter object designs a minimum-order half-band ( $L=2$ ) Nyquist filter.

Using `fdesign.nyquist` with a design method generates a `dfilt` object.

`d = fdesign.nyquist(l,spec)` constructs object `d` and sets its `Specification` property to `spec`. Use `l` to specify the desired value for `L`.  $L=2$  design a half-band FIR filter,  $L=3$  a third-band FIR filter, and so on. When you use a Nyquist filter as an interpolator, `l` or `L` is the interpolation factor. The first input argument must be `l` when you are not using the default syntax `d = fdesign.nyquist`.

Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown below. The strings are not case sensitive.

- `tw,ast` (default `spec`)
- `n,tw`
- `n`
- `n,ast`

The string entries are defined as follows:

- `ast`—attenuation in the stop band in dB (the default units).
- `n`—filter order.
- `tw`—width of the transition region between the pass and stop bands. Specified in normalized frequency units.

The filter design methods that apply to an interpolating filter specification object change depending on the `Specification` string. Paired with each string

in the following table are the design methods for interpolating filter specification objects that use that string.

<b>Specification String</b>	<b>Applicable Design Method</b>
tw,ast	kaiserwin
n,tw	kaiserwin
n	window
n,ast	kaiserwin

`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 `1`, `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 dB. Note that `fdesign` stores all magnitude specifications in dB (converting to dB when necessary) regardless of how you specify the magnitudes.



## Limitations of the Nyquist fdesign Object

Using Nyquist filter specification objects with the `equiripple` design method imposes a few limitations on the resulting filter, caused by the `equiripple` design algorithm.

- When you request a minimum-order design from `equiripple` with your Nyquist object, the design algorithm might not converge and can fail with a filter convergence error.
- When you specify the order of your desired filter, and use the `equiripple` design method, the design might not converge.
- Generally, the following specifications, alone or in combination with one another, can cause filter convergence problems with Nyquist objects and the `equiripple` design method.
  - very high order
  - small transition width
  - very large stopband attenuation

Note that halfband filters (filters where `band = 2`) do not exhibit convergence problems.

When convergence issues arise, either in the cases mentioned or in others, you might be able to design your filter with the `kaiserwin` method.

In addition, if you use Nyquist objects to design decimators or interpolators (where the interpolation or decimation factor is not a prime number), using multistage filter designs might be your best approach.

## Examples

These examples show how to construct a Nyquist filter specification object. First, create a default specifications object without using input arguments.

```
d=fdesign.nyquist

d =

    Response: 'Nyquist'
  Specification: 'TW,Ast'
  Description: {'Transition Width';'Stopband Attenuation (dB)'}
         Band: 2
  NormalizedFrequency: true
    TransitionWidth: 0.1
         Astop: 80
```

Now create an object by passing a specification type string 'n,ast'—the resulting object uses default values for n and ast.

```
d=fdesign.nyquist(2,'n,ast')  
  
d =  
  
    Response: 'Nyquist'  
    Specification: 'N,Ast'  
    Description: {'Filter Order';'Stopband Attenuation (dB)'}  
    Band: 2  
    NormalizedFrequency: true  
    FilterOrder: 10  
    Astop: 80
```

Create another Nyquist filter object, passing the specification values to the object rather than accepting the default values for n and ast.

```
d=fdesign.nyquist(3,'n,ast',42,80)  
  
d =  
  
    Response: 'Nyquist'  
    Specification: 'N,Ast'  
    Description: {'Filter Order';'Stopband Attenuation (dB)'}  
    Band: 3  
    NormalizedFrequency: true  
    FilterOrder: 42  
    Astop: 80
```

Finally, pass the filter specifications that correspond to the default Specification—tw,ast. When you pass only the values, fdesign.nyquist assumes the default Specification string.

```
d = fdesign.nyquist(4,.01,80)  
  
d =  
  
    Response: 'Nyquist'  
    Specification: 'TW,Ast'  
    Description: {'Transition Width';'Stopband Attenuation (dB)'}  
    Band: 4  
    NormalizedFrequency: true  
    TransitionWidth: 0.01  
    Astop: 80
```

Now design a Nyquist filter using the kaiserwin design method.

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

```
hd =  
  
    FilterStructure: 'Direct-Form FIR'  
      Arithmetic: 'double'  
      Numerator: [1x1005 double]  
 PersistentMemory: false
```

**See Also**

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

# fdesign.rsrc

---

**Purpose** Construct rational-factor sample-rate converter specifications object

**Syntax**

```
d = fdesign.rsrc(l,m)
d = fdesign.rsrc(...,design)
d = fdesign.rsrc(...,design,spec)
d = fdesign.rsrc(l,m,design,spec,specvalue1,specvalue2)
d = fdesign.rsrc(...,fs)
d = fdesign.rsrc(...,magunits)
```

**Description** `d = fdesign.rsrc(l,m)` constructs a rational-factor sample-rate converter 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>
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.

<b>design String (Continued)</b>	<b>Description</b>
CIC	Sets the design for the rational-factor sample-rate converter specifications object to CIC filter.
CIC Compensator	Sets the design for the rational-factor sample-rate converter specifications object to CIC compensator.
Halfband	Sets the design for the rational-factor sample-rate converter specifications object to halfband.
Highpass	Sets the design for the rational-factor sample-rate converter specifications object to highpass.
Inverse-Sinc Lowpass	Sets the design for the rational-factor sample-rate converter specifications object to inverse-sinc lowpass.
Lowpass	Sets the design for the rational-factor sample-rate converter specifications object to lowpass.
Nyquist	Sets the design for the rational-factor sample-rate converter specifications object to Nyquist.

`d = fdesign.rsrc(...,design,spec)` constructs object `d` and sets its `Specification` property to `spec`. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` depend on the design type of the specifications object.

When you add the `spec` input argument, you must also add the `design` input argument.

Because you are designing multirate filters, the specification strings available are not the same as the specifications for designing single-rate filters with such design methods as `fdesign.lowpass`. The strings are not case sensitive.

Notice that the interpolation factor 1 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>
Bandpass	<ul style="list-style-type: none"><li>• fst1,fp1,fp2,fst2,ast1,ap,ast2 (default string)</li><li>• n,fc1,fc2</li><li>• n,fst1,fp1,fp2,fst2</li></ul>
Bandstop	<ul style="list-style-type: none"><li>• n,fc1,fc2</li><li>• n,fp1,fst1,fst2,fp2</li><li>• fp1,fst1,fst2,fp2,ap1,ast,ap2 (default string)</li></ul>
CIC	<ul style="list-style-type: none"><li>• fp,ast (default and only string)</li></ul>
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>
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>

Design Type	Valid Specification Strings
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>

The string entries are defined as follows:

- ap—amount of ripple allowed in the pass band in dB (the default units). Also called Apass.

- `ap1`—amount of ripple allowed in the pass band in dB (the default units). Also called `Apass1`. Bandpass and bandstop filters use this option.
- `ap2`—amount of ripple allowed in the pass band in dB (the default units). Also called `Apass2`. Bandpass and bandstop filters use this option.
- `ast`—attenuation in the first stop band in dB (the default units). Also called `Astop`.
- `ast1`—attenuation in the first stop band in dB (the default units). Also called `Astop1`. Bandpass and bandstop filters use this option.
- `ast2`—attenuation in the first stop band in dB (the default units). Also called `Astop2`. Bandpass and bandstop filters use this option.
- `fc1`—cutoff frequency for the point 3dB 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 3dB 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.
- `n`—filter order.
- `tw`—width of the transition region between the pass and stop bands. Both halfband and Nyquist filters use this option.

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



`d = fdesign.rsrc(...,fs)` adds the argument `fs`, specified in Hz, to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

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

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

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in dB. Note that `fdesign` stores all magnitude specifications in dB (converting to dB 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',.05,40);
designmethods(d)
hm = design(d,'kaiserwin'); % Use Kaiser window to design rate
changer.
```

For this example, specify the filter order (12) when you create the specifications object `d`.

```
d = fdesign.rsrc(5,3,'nyquist','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','n,tw',12,0.1,5)
designmethods(d);
design(d,'equiripple'); % Opens FVTool to display the response.
```

Specify a stopband ripple in linear units.

## fdesign.rsrc

---

```
d = fdesign.rsrc(4,7,'nyquist','tw,ast',.1,1e-3,5,...  
'linear') % 1e-3 = 60dB attenuation in the stopband.
```

### See Also

design, designmethods, fdesign.decimator, fdesign.interpolator,  
setspecs

**Purpose** Frequency-domain coefficients used when filtering with discrete-time and adaptive filter objects

**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.pbfda`
- `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
```

# fftcoeffs

---

```
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

```

# fftcoeffs

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Columns 14 through 16

```
0 0 0
0 0 0
0 0 0
0 0 0
```

Filtering a signal `y` produces complex nonzero coefficients that you use `fftcoeffs` to see.

```
[y,e] = filter(ha,x,d);
c=fftcoeffs(ha)
```

`c =`

Columns 1 through 4

```
0.1425 - 0.0957i 0.0487 - 0.0503i -0.0479 + 0.0315i 0.0769 - 0.0435i
0.7264 - 0.7605i -0.7423 - 0.6382i 0.1758 + 0.6679i 0.2018 - 0.6544i
-0.1604 + 0.0747i -0.0709 + 0.2610i -0.1634 + 0.2929i -0.1488 + 0.3610i
-0.0396 + 0.0416i 0.0985 + 0.0095i 0.0733 + 0.0011i 0.0700 + 0.0348i
```

Columns 5 through 8

```
-0.0604 + 0.1767i 0.0732 - 0.0648i -0.0870 + 0.0383i 0.0298 - 0.0852i
-0.1665 + 0.3741i 0.3174 - 0.5234i -0.1990 + 0.4150i 0.3657 - 0.4760i
-0.2198 + 0.4273i -0.2690 + 0.3981i -0.2820 + 0.3095i -0.3633 + 0.3517i
-0.0537 - 0.0855i -0.0190 + 0.0336i 0.0091 - 0.0061i -0.0299 + 0.0001i
```

Columns 9 through 12

```
-0.0437 + 0.0676i 0.0499 - 0.0164i -0.0397 + 0.0165i 0.0455 - 0.0085i
-0.3293 + 0.3076i 0.4986 - 0.3949i -0.3300 + 0.3448i 0.5492 - 0.2633i
-0.2671 + 0.3238i -0.3813 + 0.2999i -0.4130 + 0.2333i -0.2910 + 0.2823i
-0.0300 + 0.0236i -0.0103 + 0.0438i 0.0244 + 0.0476i 0.1043 + 0.0359i
```

Columns 13 through 16

```
-0.0602 + 0.1189i -0.0227 - 0.1076i -0.0282 + 0.0634i 0.0170 - 0.0464i
-0.4385 + 0.0549i 0.5232 - 0.1904i -0.6414 - 0.1717i 0.5580 + 0.6477i
-0.4511 + 0.3217i -0.4301 + 0.1765i -0.2805 + 0.1270i -0.2531 + 0.0299i
0.1076 - 0.0383i -0.0166 + 0.0020i 0.0004 - 0.0376i 0.0071 - 0.0714i
```

## See Also

`adaptfilt.fdaf`, `adaptfilt.pbfdaf`, `adaptfilt.pbufdaf`, `adaptfilt.ufdaf`

<b>Purpose</b>	Apply filter objects to data and access states and filtering information
<b>Syntax</b>	<b>Fixed-Point Filter Syntaxes</b> <code>y = filter(hd,x)</code> <code>y = filter(hd,x,dim)</code>  <b>Adaptive Filter Syntax</b> <code>y = filter(ha,x,d)</code> <code>[y,e] = filter(ha,x,d)</code>  <b>Multirate Filter Syntax</b> <code>y = filter(hm,x)</code> <code>y = filter(hm,x,dim)</code>

**Description** This reference page contains three sections that describe the syntaxes for the filter objects:

- Fixed-Point Filter Syntaxes
- “Adaptive Filter Syntaxes” on page 8-650
- “Multirate Filter Syntaxes” on page 8-651

### 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,  $y = \text{filter}(hd, x)$  filters along each column of  $x$  to produce a matrix  $y$  of independent channels. If  $x$  is a multidimensional array,  $y = \text{filter}(hd, x)$  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.

$y = \text{filter}(hd, x, dim)$  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

$y = \text{filter}(ha, x, d)$  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.

$[y, e] = \text{filter}(ha, x, d)$  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`.

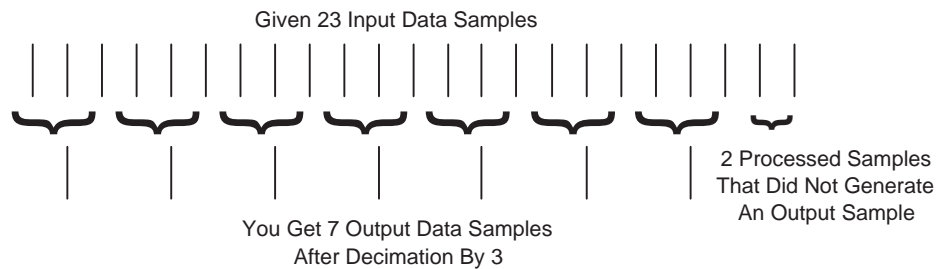
When you are filtering multichannel data, `dim` lets you specify which dimension of the input matrix to filter along—whether a row represents a channel or a column represents a channel. When you provide the `dim` input argument, the filter operates along the dimension specified by `dim`. When your input 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.

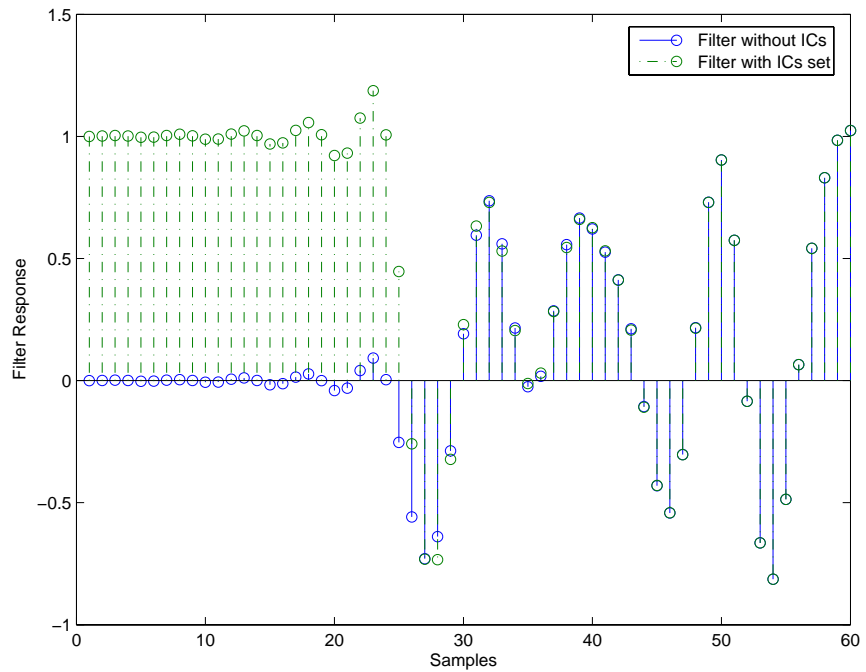
% Do not set specific initial conditions.

y1 = filter(hd,x);         % 'PersistentMemory' is '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 the 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.

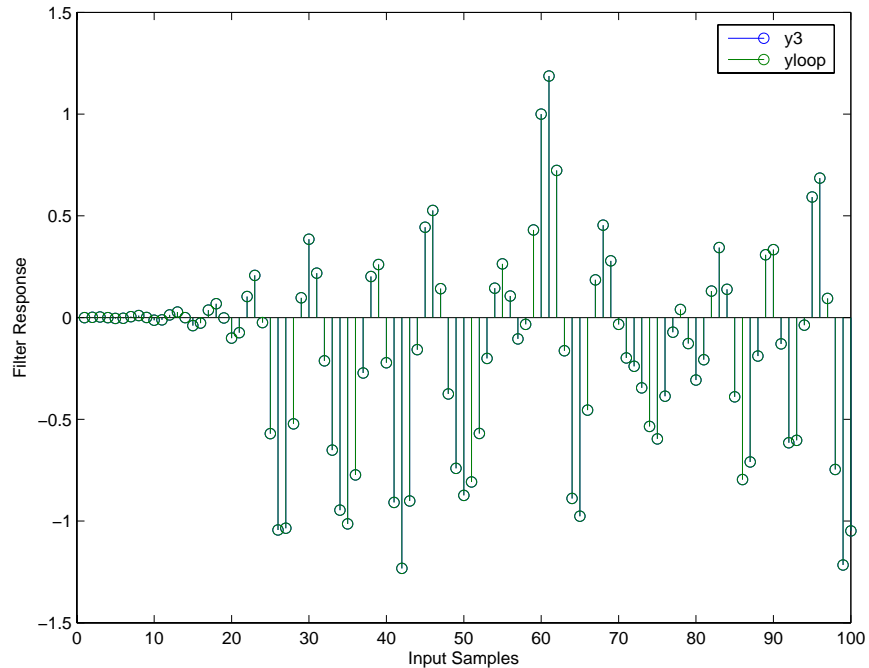
```
reset(hd);           % Clear filter history.
y3 = filter(hd,x);   % Filter the entire signal in one block.
```

As an experiment, repeat the process, filtering the data as sections, rather than in streaming form.

```
reset(hd);           % Clear filter history.
yloop = zeros(100,1) % Preallocate output array.
xblock = reshape(x,[20 5]);
for i=1:5,
    yloop = [yloop; filter(hd,xblock(:,i))];
end
```

Use a stem plot to see the comparison between streaming and block-by-block filtering.

```
stem([y3 yloop]);
```



Filtering the signal section-by-section is equivalent to filtering the entire signal at once.

To show the similarity between filtering with discrete-time and with multirate filters, this example demonstrates multirate filtering.

```
Fs = 44.1e3; % Original sampling frequency: 44.1kHz.  
n = [0:10239].'; % 10240 samples, 0.232 second long signal.  
x = sin(2*pi*1e3/Fs*n); % Original signal, sinusoid at 1kHz.  
m = 2; % Decimation factor.  
hm = mfilt.firdecim(m); % Use the default filter.
```

First, filter without setting initial conditions.

```
y1 = filter(hm,x);      % PersistentMemory is false (default).
zf = hm.states;        % Final conditions.
```

This time, set nonzero initial conditions before filtering the data.

```
hm.persistentmemory = true;
hm.states = 1;         % Uses scalar expansion to set ICs.
y2 = filter(Hm,x);
stem([y1(1:60) y2(1:60)]) % Show the filtering results.
```

Note the different sequences at the start of filtering.

Finally, try filtering streaming data.

```
reset(hm);             % Clear the filter history.
y3 = filter(hm,x);     % Filter the 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 the Signal Processing Toolbox

## References

[1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

- Purpose**  $B_{\max}$ , most significant bit, of cascaded integrator-comb (CIC) filter
- Syntax** `filtmsb(hm)`
- Description** `filtmsb(hm)` returns the most significant bit (MSB) of the filter output and is a function of the following parameters of filter `hm`.
- R—the interpolation factor or decimation factor depending on the filter form
  - M—the differential delay
  - N—the number of sections in the filter

Because the output of the integrators can grow without bound, the MSB returned represents the maximum number of bits that can propagate through the filter without losing data. This MSB is both the MSB at the filter output and the MSB for all stages.

- Examples** Using the `mfilt.cicdecim` filter constructor, create a multirate filter and determine the most significant bit. For the decimator specifications used here, refer to the CIC decimator design example D on pp. 159 in [1].

```
hm = mfilt.cicdecim(25,1,4,16,16)
```

```
hm =
```

```

    FilterStructure: 'Cascaded Integrator-Comb Decimator'
      Arithmetic: 'fixed'
DifferentialDelay: 1
  NumberOfSections: 4
    DecimationFactor: 25
    PersistentMemory: false

```

```
    InputWordLength: 16
```

```
    InputFracLength: 15
```

```
    SectionWordLengthMode: 'MinWordLengths'
```

```
    OutputWordLength: 16
```

```
bmax=filtmsb(hm)
```

```
bmax =
```

Reviewing the referenced Hogenauer paper [1] shows that 34 is the correct result.

## Algorithm

`filtmsb` calculates the most significant bit for interpolators and decimators using the following algorithms and filter property values. In each case, `hm` is a multirate filter of the appropriate form, either decimator or interpolator. Both equations derive from [1].

### Decimators

From equation 11 in [1], calculate `Bmax` as follows for decimators:

```
bmax = ceil(hm.NumberOfSections*log2(hm.DecimationFactor*  
hm.DifferentialDelay) + hm.InputWordLength - 1)
```

### Interpolators

Interpolators use a slightly different formulation, equation 23 in [1].

```
bmax = ceil(hm.InputWordLength + log2(Gmax))
```

where

```
gmax = (((hm.InterpolationFactor*hm.DifferentialDelay)^hm.NumberOfSections)/...  
hm.InterpolationFactor)
```

## See Also

`gain`, `mfilt`



**Purpose**

Object for storing states of cascaded-integrator comb (CIC) filters

**Description**

`filtstates.cic` objects hold the states information for CIC filters. Once you create a CIC filter, the states for the filter are stored in `filtstates.cic` objects, and you can access them and change them as you would any property of the filter. This arrangement parallels that of the `filtstates` object that IIR direct-form I filters use (refer to `filtstates` for more information).

Each `States` property in the CIC filter comprises two properties—`Numerator` and `Comb`—that hold `filtstates.cic` objects. Within the `filtstates.cic` objects are the numerator-related and comb-related filter states. The states are column vectors, where each column represents the states for one section of the filter. For example, a CIC filter with four decimator sections and four interpolator sections has `filtstates.cic` objects that contain four columns of states each.

**Examples**

To show you the `filtstates.cic` object, create a CIC decimator and filter a signal.

```
hm=mfilt.cicdecim(5,2,4)
```

```
hm =
```

```

    FilterStructure: 'Cascaded Integrator-Comb Decimator'
      Arithmetic: 'fixed'
DifferentialDelay: 2
  NumberOfSections: 4
    DecimationFactor: 5
    PersistentMemory: false
```

```
    InputWordLength: 16
```

```
    InputFracLength: 15
```

```
    SectionWordLengthMode: 'MinWordLengths'
```

```
hm.persistentMemory=true
```

```
hm =
```

```

    FilterStructure: 'Cascaded Integrator-Comb Decimator'
      Arithmetic: 'fixed'
```

```
DifferentialDelay: 2
NumberOfSections: 4
DecimationFactor: 5
PersistentMemory: true
    States: Integrator: [4x1 States]
           Comb: [4x1 States]
InputOffset: 0

InputWordLength: 16
InputFracLength: 15

SectionWordLengthMode: 'MinWordLengths'
```

Use hm to filter some input data.

```
fs = 44.1e3;           % Original sampling frequency: 44.1kHz.
n = 0:10239;          % 10240 samples, 0.232 second long signal.
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1kHz.
y=filter(hm,x)
```

hm has nonzero states now.

```
s=hm.states

s =

    Integrator: [4x1 States]
    Comb: [4x1 States]
```

```
s.Integrator
```

```
ans =
```

```
1.0e+003 *

    0.0043
   -2.0347
   -0.4175
    0.8206
```

```
s.Comb
```

```
ans =
    1.0e+003 *
    -3.1301
    -0.8493
    -2.5474
     1.7888
    -1.6253
     3.1981
     0.4729
     3.4559
```

You can use `int` to see the states as 32-bit integers.

```
int(s.Integrator)

ans =
    142435
   -8334019
   -427469
    210081
```

`whos` shows you the `filtstates.cic` object.

```
whos
  Name          Size          Bytes  Class

Fs              1x1              8  double array
ans             4x1             16  int32 array
hm              1x1
n              1x10240         81920  double array
s              1x1
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 the Signal Processing Toolbox documentation

**Purpose** Perform constrained-band equiripple FIR filter design

**Syntax**

```
b = fircband(n,f,a,w,c)
b = fircband(n,f,a,s)
b = fircband(...,'1')
b = fircband(...,'minphase')
b = fircband(...,'check')
b = fircband(...,{lgrid})
[b,err] = fircband(...)
[b,err,res] = fircband(...)
```

**Description** `fircband` is a minimax filter design algorithm that you use to design the following types of real FIR filters:

- Types 1-4 linear phase
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase,
- Minimum order (even or odd), extra ripple
- Maximal ripple
- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain
- Arbitrary shape frequency response curve filters

`b = fircband(n,f,a,w,c)` designs filters having constrained error magnitudes (ripples). `c` is a cell array of strings of the same length as `w`. The entries of `c` must be either 'c' to indicate that the corresponding element in `w` is a constraint (the ripple for that band cannot exceed that value) or 'w' indicating that the corresponding entry in `w` is a weight. There must be at least one unconstrained band—`c` must contain at least one `w` entry. For instance,

Example 1 below uses a weight of one in the passband, and constrains the stopband ripple not to exceed 0.2 (about 14 dB).

A hint about using constrained values: if your constrained filter does not touch the constraints, increase the error weighting you apply to the unconstrained bands.

Notice that, when you work with constrained stopbands, you enter the stopband constraint according to the standard conversion formula for power—the resulting filter attenuation or constraint equals  $20 \cdot \log(\text{constraint})$  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 = firband(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 = firband(...,'1')` designs a type 1 filter (even-order symmetric). You could also specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), or type 4 (odd-order antisymmetric) filters. Note there are restrictions on `a` at `f = 0` or `f=1` for types 2, 3, and 4.

`b = 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] = fircband(...)` returns the unweighted approximation error magnitudes. `err` has one element for each independent approximation error.

`[b,err,res] = fircband(...)` returns a structure `res` of optional results computed by `fircband`, and contains the following fields:

Structure Field	Contents
<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 extremal frequencies
<code>res.fextr</code>	Vector of extremal frequencies
<code>res.order</code>	Filter order
<code>res.edgecheck</code>	<p>Transition-region anomaly check. One element per band edge. Element values have the following meanings:</p> <p>1 = OK  0 = probable transition-region anomaly  -1 = edge not checked</p> <p>Computed when you specify the 'check' input option in the function syntax.</p>

# firband

---

<b>Structure Field</b>	<b>Contents</b>
res.iterations	Number of Remez iterations for the optimization
res.ivals	Number of function evaluations for the optimization

## Examples

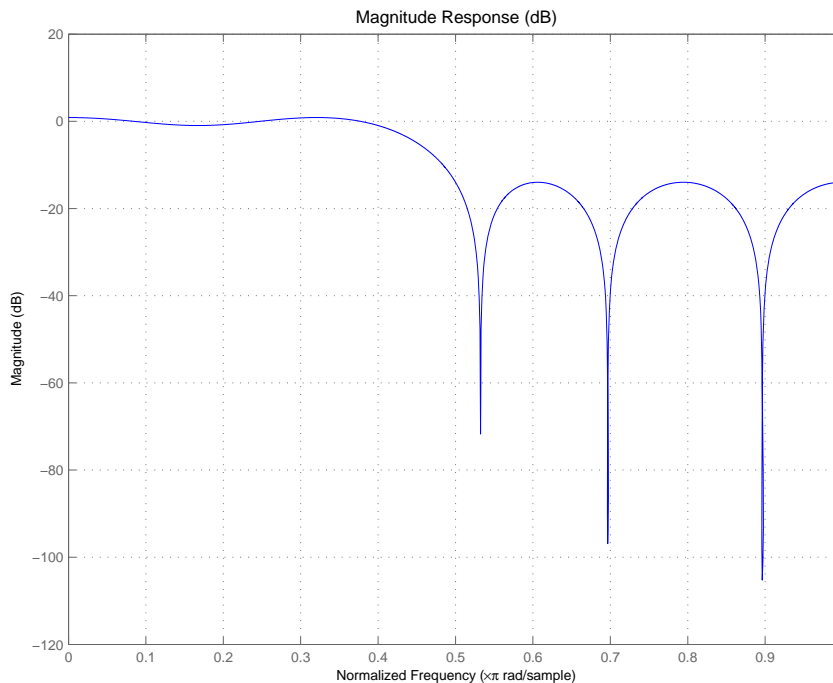
Two examples of designing filters with constrained bands.

Example 1—design a 12th-order lowpass filter with a constraint on the filter response.

```
b = firband(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2], {'w' 'c'});
```

Using `fvtool` to display the result `b` shows you the response of the filter you designed.



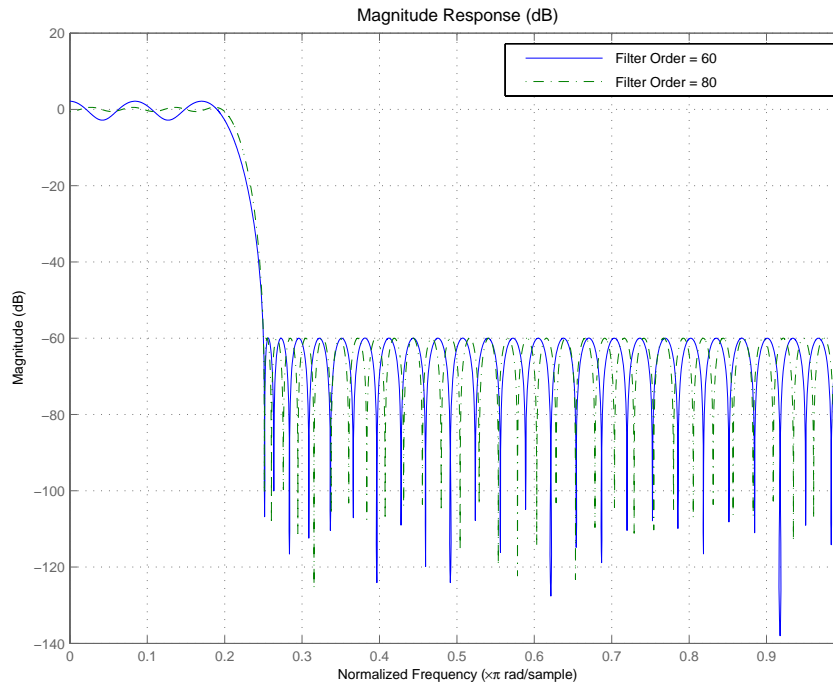


Example 2—design two filters of different order with the stopband constrained to 60 dB. Use excess order (80) in the second filter to improve the passband ripple.

```
b1=fircband(60,[0 .2 .25 1],[1 1 0 0],[1 .001],{'w','c'});
b2=fircband(80,[0 .2 .25 1],[1 1 0 0],[1 .001],{'w','c'});
fvtool(b1,1,b2,1)
```

To set the stopband constraint to 60 dB, enter 0.001, since  $20 \cdot \log(0.001) = -60$ , or 60 dB of signal attenuation.

# firband



## See Also

`firceqrip`, `firgr`, `firls`

`firpm` in the Signal Processing Toolbox

Also refer to “Constrained Band Equiripple FIR Filter Design” in Demos

<b>Purpose</b>	Design equiripple FIR interpolators
<b>Syntax</b>	<pre>b = fireqint(n,1,alpha) b = fireqint(n,1,alpha,w) b = fireqint('minorder',1,alpha,r) b = fireqint({'minorder',initord},1,alpha,r)</pre>
<b>Description</b>	<p><code>b = fireqint(n,1,alpha)</code> designs an FIR equiripple filter useful for interpolating input signals. <code>n</code> is the filter order and it must be an integer. <code>1</code>, also an integer, is the interpolation factor. <code>alpha</code> is the bandlimitedness factor, identical to the same feature in <code>intfilt</code>.</p> <p><code>alpha</code> 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 <code>alpha</code> allows you to control how much of the Nyquist interval your input signal occupies. This can be beneficial for signals to be interpolated because it allows you to increase the transition band width without affecting the interpolation, resulting in better stopband attenuation for a given <code>1</code>. If you set <code>alpha</code> to <code>1</code>, <code>fireqint</code> assumes that your signal occupies the entire Nyquist interval. Setting <code>alpha</code> 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 <code>alpha</code> to <code>0.5</code>.</p> <p>The signal to be interpolated is assumed to have zero (or negligible) power in the frequency band between <math>(\alpha*\pi)</math> and <math>\pi</math>. <code>alpha</code> must therefore be a positive scalar between <code>0</code> and <code>1</code>. <code>fireqint</code> treat such bands as don't-care regions for assessing filter design.</p> <p><code>b = fireqint(n,1,alpha,w)</code> allows you to specify a vector of weights in <code>w</code>. The number of weights required in <code>w</code> is given by <code>1 + floor(1/2)</code>. The weights in <code>w</code> are applied to the passband ripple and stopband attenuations. Using weights (values between <code>0</code> and <code>1</code>) 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.</p> <p><code>b = fireqint('minorder',1,alpha,r)</code> allows you to design a minimum-order filter that meets the design specifications. <code>r</code> is a vector of maximum deviations or ripples from the ideal filter magnitude response. When</p>

# fireqint

you use the input argument **minorder**, you must provide the vector **r**. The number of elements required in **r** is given by  $1 + \text{floor}(1/2)$ .

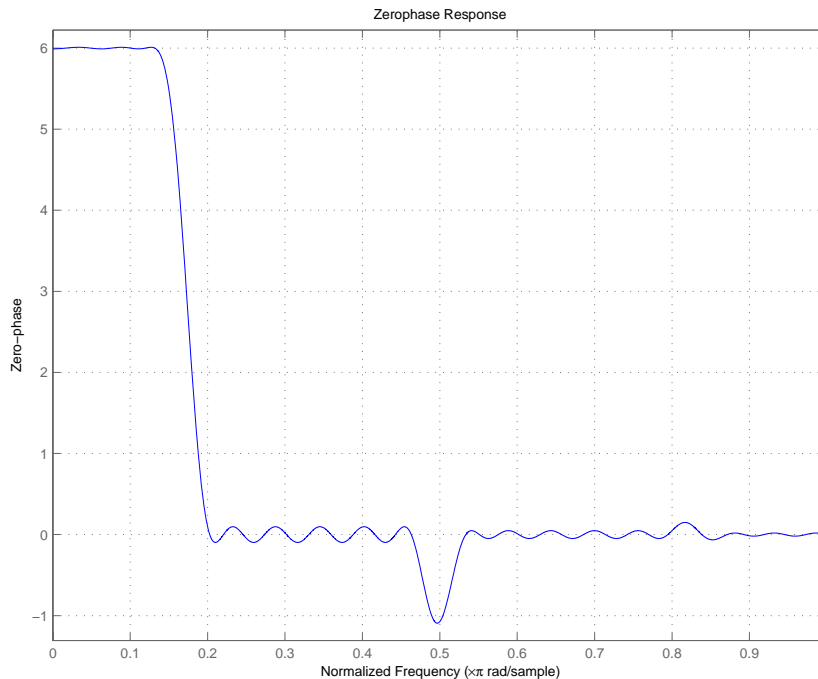
`b = fireqint({'minorder',initord},1,alpha,r)` adds the argument `initord` so you can provide an initial estimate of the filter order. Any positive integer is valid here. Again, you must provide **r**, the vector of maximum deviations or ripples, from the ideal filter magnitude response.

## 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 = mfilter.firinterp(6,b); % Create a polyphase interpolator filter  
zerophase(hm);
```

Here is the resulting plot of the zero-phase 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 your Signal Processing Toolbox documentation

# firceqrip

---

**Purpose** Design constrained, equiripple FIR filter

**Syntax**

```
hd = firceqrip(n,wo,del)
hd = firceqrip(...,'slope',r)
hd = firceqrip(...,'passedge')
hd = firceqrip(...,'stopedge')
hd = firceqrip(...,'high')
hd = firceqrip(...,'min')
hd = firceqrip(...,'invsinc',c)
```

**Description** `hd = firceqrip(n,wo,del)` design an order  $n$  filter (filter length equal  $n+1$ ) lowpass FIR filter with linear phase.

`firceqrip` produces the same equiripple lowpass filters that `firpm` produces using the Parks-McClellan algorithm. The difference is how you specify the filter characteristics for the function.

Input argument `wo` specifies the cutoff frequency. The two-element vector `del` specifies the peak or maximum error allowed in the passband and stopbands. Enter `[d1 d2]` for `del` where `d1` sets the passband error and `d2` sets the stopband error. Since `firceqrip` works in the normalized frequency domain, you must set `wo` to be between 0 and 1 ( $0 < wo < 1$ ).

`hd = firceqrip(...,'slope',r)` uses the input keyword `'slope'` and input argument `r` to design a filter with a stopband that does not demonstrate equiripple characteristics. `r` determines the slope of the stopband in dB when  $r > 0$ .

In this constrained equiripple design approach, you can specify a stopband slope (increasing attenuation with increasing frequency). Enter your desired slope in dB as a positive value. Larger slope values create increasing attenuation of the stopband as frequency increases.

Slope is defined in the following ways:

- For filters specified in linear frequency, the slope is defined over every  $F_s/2$  frequency bands.
- For filters specified in normalized frequency, the slope is defined over  $\pi$  rad/sample.

Here is a description of how slope works. The algorithm defines slope in dB per half of the Nyquist interval. If you are working in normalized frequency and you set the slope to 40 dB, the stopband attenuation increases by 40 dB every rad/sample.

Try setting  $r$  to 10 to see the effect on the filter frequency response. In the Examples section, example 3 designs a filter with  $r$  equal to 20.

`hd = firceqrip(..., 'passedge')` designs a filter where  $w_0$  specifies the frequency at which the passband starts to roll off.

`hd = firceqrip(..., 'stopedge')` designs a filter where  $w_0$  specifies the frequency at which the stopband begins.

`hd = firceqrip(..., 'high')` designs a high pass FIR filter instead of a lowpass filter.

`hd = firceqrip(..., 'min')` designs an FIR filter with minimum phase.

`hd = firceqrip(..., 'invsinc', c)` designs a lowpass filter whose passband has the shape of the inverse sinc function. For this syntax, keyword **invsinc** applies the inverse sinc function as defined by whether  $c$  is a scalar or a two-element vector:

- When you use  $c$  as a scalar with the **invsinc** keyword, `firceqrip` applies the function  $1/\text{sinc}(c*w)$ , where  $w$  is the normalized frequency, to the passband.
- When you use  $c$  as a two-element vector entered as  $[c \ p]$ , with the **invsinc** keyword, `firceqrip` applies the function  $1/\text{sinc}(c*w)^p$  to the passband, where  $w$  is the normalized frequency.

In both cases,  $c$  must meet the condition  $c < 1/w_0$ .

When you use a cascaded-integrated comb (CIC) filter in series with this FIR filter, argument  $p$  lets you compensate for the droop in the passband of the CIC filter. Setting  $p$  equal to the number of stages in your CIC generally produces an FIR filter whose passband neatly compensates for the CIC passband shape.

To let you specify precisely the FIR filter to design, use any or all of the optional input arguments together. Any ordering of the optional arguments works—order is not important in the function call. Refer to Examples 3 and 4 to see multiple optional input arguments being used.

---

**Note** If the `wo` you specify is too small or too large, or if either `c` or `p` is too large, your filter specifications may be unfeasible, causing the design algorithm to fail to generate your filter.

---

## Examples

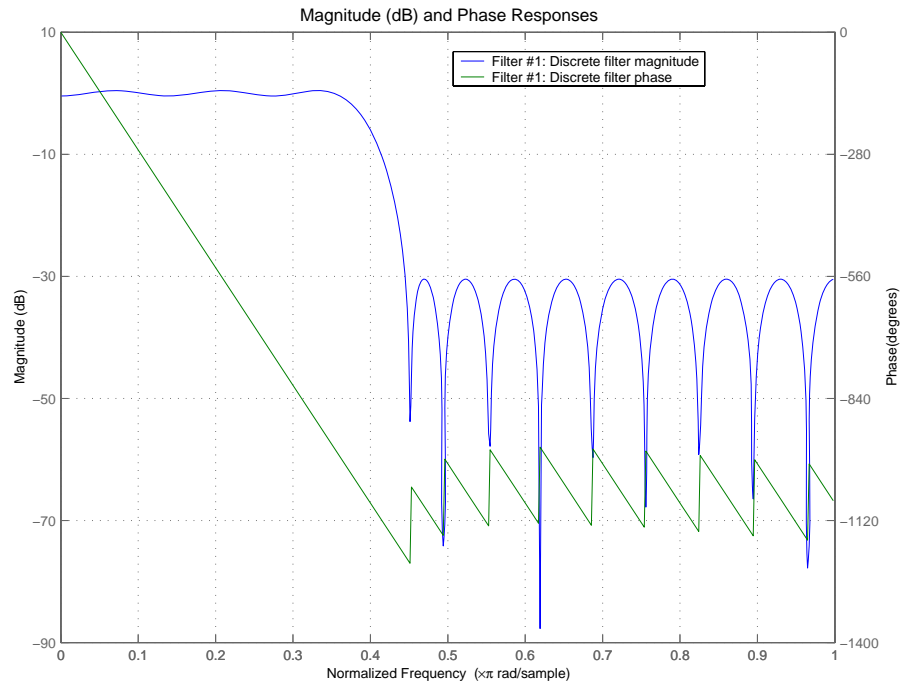
To introduce a few of the variations on FIR filters that you design with `firceqrip`, these five examples cover both the default syntax `hd = firceqrip(n,wo,del)` and some of the optional input arguments. For each example, the input arguments `n`, `wo`, and `del` remain the same.

**Example 1**—Design an order = 30 FIR filter without using optional input arguments or keywords.

```
hd = firceqrip(n,wo,del); fvtool(hd)
```

Both the phase and magnitude response for the resulting lowpass filter appear in the plot shown here.





Example 2—Design an order = 30 FIR filter with the **stopedge** keyword to define the response at the edge of the filter stopband.

```
hd = firceqrip(n,wo,del,'stopedge'); fvtool(hd,1)
```

Example 3—Design an order = 30 FIR filter with the **slope** keyword and  $r = 20$ .

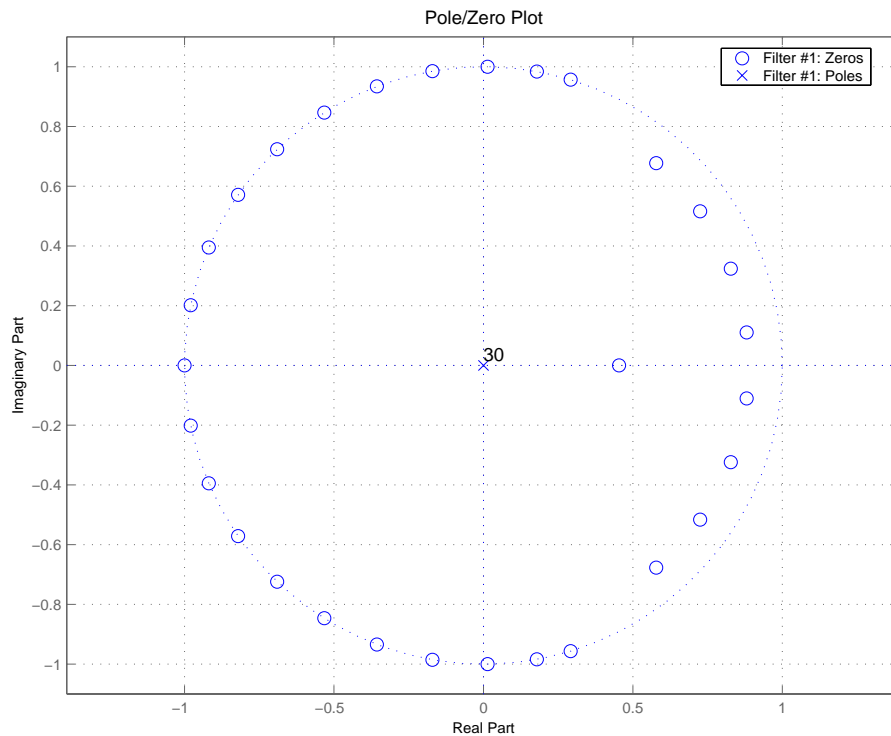
```
hd = firceqrip(n,wo,del,'slope',20,'stopedge'); fvtool(hd)
```

Example 4—Design an order = 30 FIR filter defining the stopband and specifying that the resulting filter is minimum phase with the **min** keyword.

```
hd = firceqrip(n,wo,del,'stopedge','min'); fvtool(hd)
```

Comparing this filter to the filter in Example 1, notice that 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.

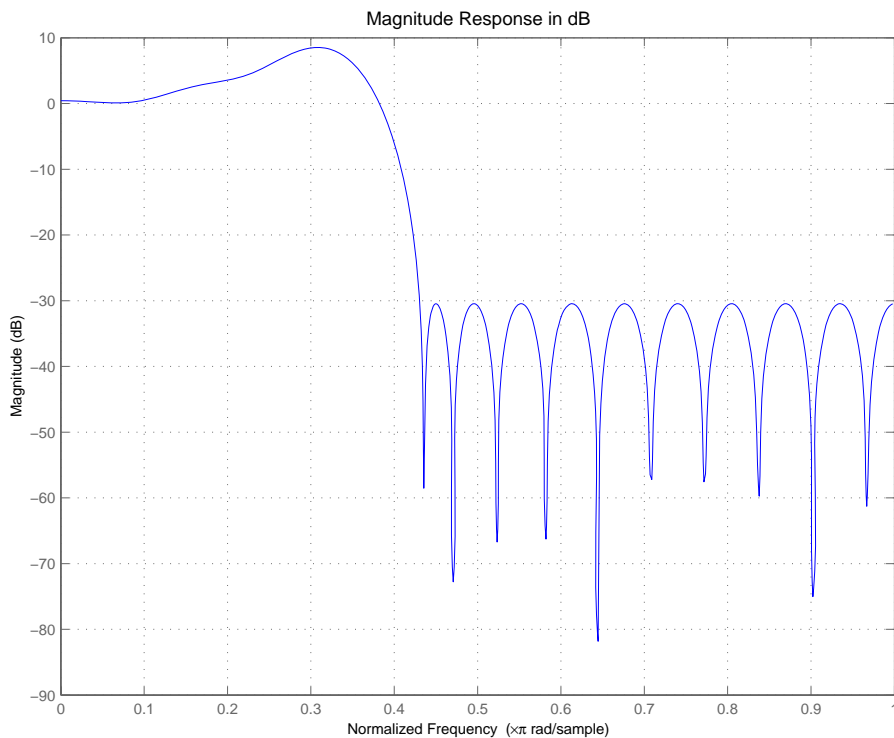
Viewing the zero-pole plot shown here reveals this is a minimum phase FIR filter—the zeros lie on or inside the unit circle,  $z = 1$ .



Example 5—Design an order = 30 FIR filter with the **invsinc** keyword to shape the filter passband with an inverse sinc function.

```
hd = firceqrip(n,wo,del,'invsinc',[2 1.5]); fvtool(hd,1)
```

With the inverse sinc function being applied defined as  $1/\text{sinc}(2*w)^{1.5}$ , the figure shows the reshaping of the passband that results from using the `invsinc` keyword option, and entering `c` as the two-element vector `[2 1.5]`.



### See Also

`firhalfband`, `firnyquist`, `firgr`, `ifir`, `iirgrpdelay`, `iirlpnorm`, `iirlpnormc`, `fircls`, `firls`, `firpm` in your Signal Processing Toolbox documentation

# firgr

---

## Purpose

Use Parks-McClellan technique to design digital FIR filter

## Syntax

```
b = firgr(n,f,a,w)
b = firgr(n,f,a,'hilbert')
b = firgr(n,f,a,'differentiator')
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)
```

## Description

`firgr` is a minimax filter design algorithm you use to design the following types of real FIR filters:

- Types 1-4 linear phase:
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase
- Minimum order (even or odd)
- Extra ripple
- Maximal ripple
- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain
- Arbitrary shape frequency response curve filters

`b = firgr(n,f,a,w)` returns a length  $n+1$  linear phase FIR filter which has the best approximation to the desired frequency response described by `f` and `a` in the minimax sense. `w` is a vector of weights, one per band. When you omit `w`, all bands are weighted equally. For more information on the input arguments, refer to `firpm` in *Signal Processing Toolbox User's Guide*.

`b = firgr(n,f,a,'hilbert')` and `b = firgr(n,f,a,'differentiator')` design FIR Hilbert transformers and differentiators. For more information on designing these filters, refer to `firpm` in *Signal Processing Toolbox User's Guide*.

`b = firgr(m,f,a,r)`, where `m` is one of `'minorder'`, `'mineven'` or `'minodd'`, designs filters repeatedly until the minimum order filter, as specified in `m`, that meets the specifications is found. `r` is a vector containing the peak ripple per frequency band. You must specify `r`. When you specify `'mineven'` or `'minodd'`, the minimum even or odd order filter is found.

`b = firgr({m,ni},f,a,r)` where `m` is one of `'minorder'`, `'mineven'` or `'minodd'`, uses `ni` as the initial estimate of the filter order. `ni` is optional for common filter designs, but it must be specified for designs in which `firpmord` cannot be used, such as while designing differentiators or Hilbert transformers.

`b = firgr(n,f,a,w,e)` specifies independent approximation errors for different bands. Use this syntax to design extra ripple or maximal ripple filters. These filters have interesting properties such as having the minimum transition width. `e` is a cell array of strings specifying the approximation errors to use. Its length must equal the number of bands. Entries of `e` must be in the form `'e#'` where `#` indicates which approximation error to use for the corresponding band. For example, when `e = {'e1','e2','e1'}`, the first and third bands use the same approximation error `'e1'` and the second band uses a different one `'e2'`. Note that when all bands use the same approximation error, such as `{'e1','e1','e1',...}`, it is equivalent to omitting `e`, as in `b = firgr(n,f,a,w)`.

`b = firgr(n,f,a,s)` is used to design filters with special properties at certain frequency points. `s` is a cell array of strings and must be the same length as `f` and `a`. Entries of `s` must be one of:

- `'n'` - normal frequency point.
- `'s'` - single-point band. The frequency “band” is given by a single point. The corresponding gain at this frequency point must be specified in `a`.
- `'f'` - forced frequency point. Forces the gain at the specified frequency band to be the value specified.
- `'i'` - indeterminate frequency point. Use this argument when adjacent bands abut one another (no transition region).

For example, the following command designs a bandstop filter with zero-valued single-point stop bands (notches) at 0.25 and 0.55.

```
b = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],[1 1 0 1 1 0 1 1],...  
{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'})
```

```
b = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...  
{'n' 'i' 'f' 'n' 'n' 'n'})
```

designs a highpass filter with the gain at 0.06 forced to be zero. The band edge at 0.055 is indeterminate since the first two bands actually touch. The other band edges are normal.

`b = firgr(n,f,a,s,w,e)` specifies weights and independent approximation errors for filters with special properties. The weights and properties are included in vectors `w` and `e`. Sometimes, you may need to use independent approximation errors to get designs with forced values to converge. For example,

```
b = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1],...  
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

`b = firgr(...,'1')` designs a type 1 filter (even-order symmetric). You can specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), and type 4 (odd-order antisymmetric) filters as well. Note that restrictions apply to `a` at `f=0` or `f=1` for FIR filter types 2, 3, and 4.

`b = firgr(...,'minphase')` designs a minimum-phase FIR filter. You can use the argument `'maxphase'` to design a maximum phase FIR filter.

`b = firgr(..., 'check')` returns a warning when there are potential transition-region anomalies.

`b = firgr(...,{lgrid})`, where `{lgrid}` is a scalar cell array. The value of the scalar controls the density of the frequency grid by setting the number of samples used along the frequency axis.

`[b,err] = firgr(...)` returns the unweighted approximation error magnitudes. `err` contains one element for each independent approximation error returned by the function.

[b,err,res] = firgr(...) returns the structure res comprising optional results computed by firgr. res contains the following fields.

Structure Field	Contents
res.fgrid	Vector containing the frequency grid used in the filter design optimization
res.des	Desired response on fgrid
res.wt	Weights on fgrid
res.h	Actual frequency response on the frequency grid
res.error	Error at each point (desired response - actual response) on the frequency grid
res.iextr	Vector of indices into fgrid of extremal frequencies
res.fextr	Vector of extremal frequencies
res.order	Filter order
res.edgecheck	<p>Transition-region anomaly check. One element per band edge. Element values have the following meanings:</p> <p>1 = OK  0 = probable transition-region anomaly  -1 = edge not checked</p> <p>Computed when you specify the 'check' input option in the function syntax.</p>
res.iterations	Number of s iterations for the optimization
res.evals	Number of function evaluations for the optimization

`firgr` is also a “function function”, allowing you to write a function that defines the desired frequency response.

`b = firgr(n,f,fresp,w)` returns a length  $N+1$  FIR filter which has the best approximation to the desired frequency response as returned by the user-defined function `fresp`. Use the following `firgr` syntax to call `fresp`:

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

where:

- `fresp` is the string variable that identifies the function that you use to define your desired filter frequency response.
- `n` is the filter order.
- `f` is the vector of frequency band edges which must appear monotonically between 0 and 1, where 1 is one-half of the sampling frequency. The frequency bands span  $f(k)$  to  $f(k+1)$  for  $k$  odd. The intervals  $f(k+1)$  to  $f(k+2)$  for  $k$  odd are “transition bands” or “don't care” regions during optimization.
- `gf` is a vector of grid points that have been chosen over each specified frequency band by `firgr`, and determines the frequencies at which `firgr` evaluates the response function.
- `w` is a vector of real, positive weights, one per band, for use during optimization. `w` is optional in the call to `firgr`. If you do not specify `w`, it is set to unity weighting before being passed to `fresp`.
- `dh` and `dw` are the desired frequency response and optimization weight vectors, evaluated at each frequency in grid `gf`.

`firgr` includes a predefined frequency response function named `'firpmfrf2'`. You can write your own based on the simpler `'firpmfrf'`. See the help for `private/firpmfrf` for more information.

`b = firgr(n,f,{fresp,p1,p2,...},w)` specifies optional arguments `p1`, `p2`,..., `pn` to be passed to the response function `fresp`.

`b = firgr(n,f,a,w)` is a synonym for `b = firgr(n,f,{'firpmfrf2',a},w)`, where `a` is a vector containing your specified response amplitudes at each band edge in `f`. By default, `firgr` designs symmetric (even) FIR filters. `'firpmfrf2'` is the predefined frequency response function. If you do not specify your own



frequency response function (the `fresp` string variable), `firgr` uses `'firpmfrf2'`.

`b = firgr(..., 'h')` and `b = firgr(..., 'd')` design antisymmetric (odd) filters. When you omit the `'h'` or `'d'` arguments from the `firgr` command syntax, each frequency response function `fresp` can tell `firgr` to design either an even or odd filter. Use the command syntax `sym = fresp('defaults', {n, f, [], w, p1, p2, ...})`.

`firgr` expects `fresp` to return `sym = 'even'` or `sym = 'odd'`. If `fresp` does not support this call, `firgr` assumes even symmetry.

For more information about the input arguments to `firgr`, refer to `firpm`.

## Examples

These examples demonstrate some filters you might design using `firgr`.

Example 1—design an FIR filter with two single-band notches at 0.25 and 0.55

```
b1 = firgr(42, [0 0.2 0.25 0.3 0.5 0.55 0.6 1], [1 1 0 1 1 0 1 1], ...
{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'});
```

Example 2—design a highpass filter whose gain at 0.06 is forced to be zero. The gain at 0.055 is indeterminate since it should 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'});
```

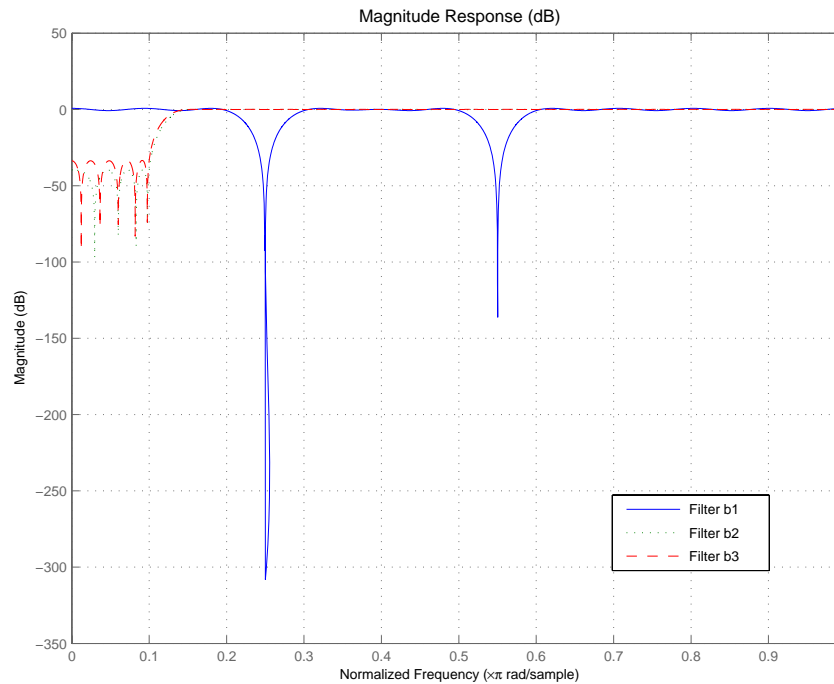
Example 3—design a second highpass filter with forced values and independent approximation errors.

```
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, firls, fircls, and firpm in your Signal Processing Toolbox documentation

## Reference

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** Design 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 banks. The **minphase** option for

# firhalfband

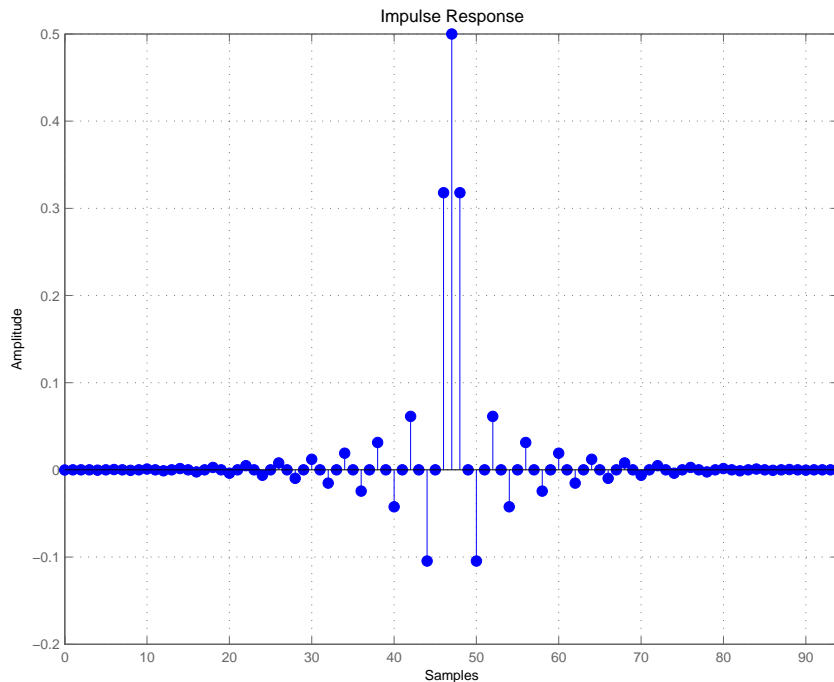
`firhalfband` is not available for the window-based halfband filter designs—  
`b = firhalfband(n,win)` and  
`b = firhalfband('minorder',fp,dev,'kaiser')`.

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 your 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 your Signal Processing Toolbox documentation

**References** [1] 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 = firlp2hp(b)
g = firlp2hp(b, 'narrow')
g = firlp2hp(b, 'wide')
```

**Description** `g = firlp2hp(b)` transforms the lowpass FIR filter `b` into a Type I highpass FIR filter `g` with zero-phase response  $H_r(\pi-w)$ . 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`.

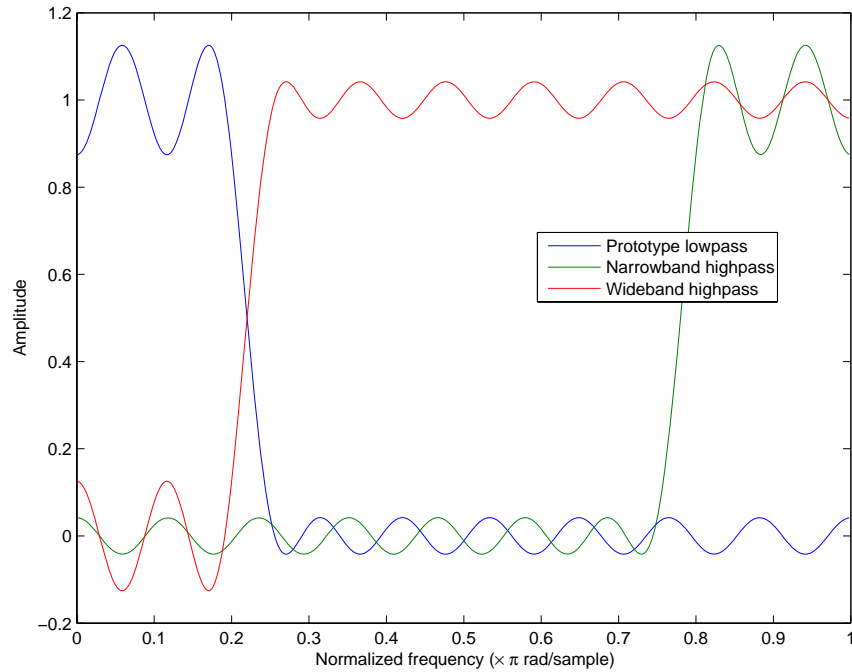
`g = firlp2hp(b, 'narrow')` transforms the lowpass FIR filter `b` into a Type I narrow band highpass FIR filter `g` with zero-phase response  $H_r(\pi-w)$ . `b` can be any FIR filter, including a nonlinear-phase filter.

`g = firlp2hp(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 plot below shows the results.

```
b = firgr(36,[0 .2 .25 1],[1 1 0 0],[1 3]);
zerophase(b); hold on;
h = firlp2hp(b);
zerophase(h);
g = firlp2hp(b, 'wide');
zerophase(g); hold off
```



## See Also

firlp2lp

zerophase in your Signal Processing Toolbox documentation

## References

[1] 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** Least P-norm optimal FIR filter design

**Syntax**

```
b = firlpnorm(n,f,edges,a)
b = firlpnorm(n,f,edges,a,w)
b = firlpnorm(n,f,edges,a,w,p)
b = firlpnorm(n,f,edges,a,w,p,dens)
b = firlpnorm(n,f,edges,a,w,p,dens,initnum)
b = firlpnorm(...,'minphase')
[b,err] = firlpnorm(...)
```

**Description** `b = firlpnorm(n,f,edges,a)` returns a filter of numerator order `n` which represents the best approximation to the frequency response described by `f` and `a` in the least-Pth norm sense. `P` is set to 128 by default, which essentially equivalent to the infinity norm. Vector `edges` specifies the band-edge frequencies for multiband designs. `firlpnorm` uses an unconstrained quasi-Newton algorithm to design the specified filter.

`f` and `a` must have the same number of elements, which can exceed the number of elements in `edges`. This lets you specify filters with any gain contour within each band. However, the frequencies in `edges` must also be in vector `f`. Always use `freqz` to check the resulting filter.

---

**Note** `firlpnorm` uses a nonlinear optimization routine that may not converge in some filter design cases. Furthermore the algorithm is not well-suited for certain large-order (order > 100) filter designs.

---

`b = firlpnorm(n,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `firlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. For example,

```
b = firlpnorm(20,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband, and with emphasis placed on minimizing the error in the stopband.

# firlpnorm

---

`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 the Signal Processing Toolbox 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.

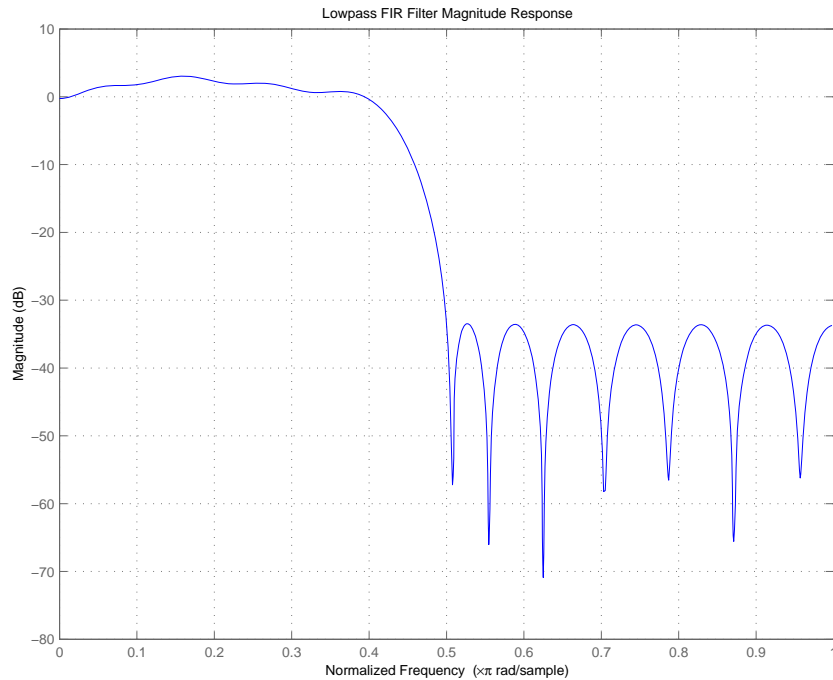
`[b,err] = firlpnorm(...)` returns the least- $p$ th approximation error `err`.

## 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)
```

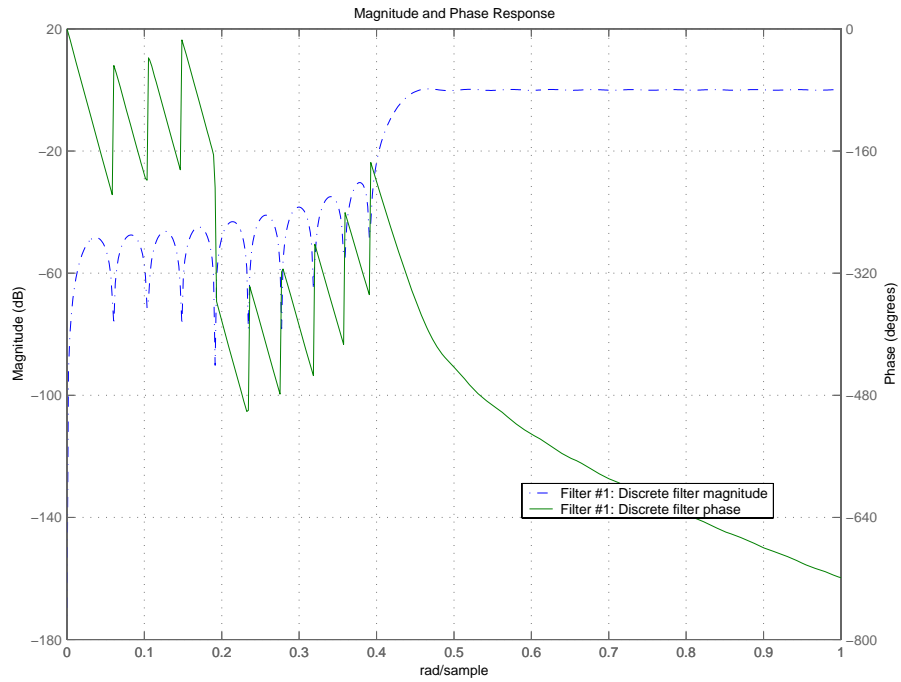
From the figure you see the resulting filter is lowpass, with the desired 1.4 peak in the passband (notice the 1.4 specified in vector a).



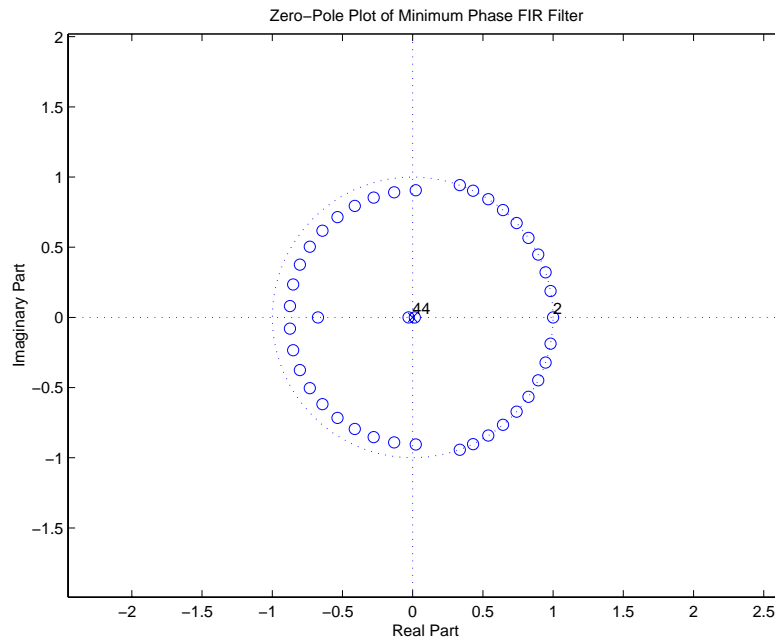
Now for the minimum-phase filter.

```
% 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 your Signal Processing Toolbox  
documentation

## References

[1] 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.

# firls

---

**Purpose** Design filter from filter specification object using least-square minimization technique

**Syntax** `hd = firls(d)`

**Description** `hd = firls(d)` designs a discrete-time FIR filter using a least-squares error minimization method. Only halfband and interpolation specifications objects with Specification of 'n,tw' or 'pl,tw' work as specifications objects for `firls`.

`hd` is either a `dfilt` object (a single-rate digital filter) or an `mfilt` object (a multirate digital filter) depending on the Specification property of the filter specification object `d` and the filter specification object type—halfband or interpolator.

**Examples** Here are two examples of using `firls` to design filters. The first example returns a single-rate halfband filter.

```
d = fdesign.halfband('n,tw',120,.04); % 120 is the filter order.
hd = firls(d);
```

Now use `firls` to design a multirate halfband interpolator filter.

```
d = fdesign.interpolator(2,'pl,tw',60,.04); % 60 is the polyphase
% length.
hm = firls(d);
```

**See Also** `equiripple`, `kaiserwin`

**Purpose** Compute 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  $g = \text{fliplr}(h)$ , and  $b = \text{conv}(h, g)$ .

---

**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 your Signal Processing Toolbox documentation

**References** [1] 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.

# firnyquist

---

**Purpose** Design lowpass Nyquist (Lth-band) FIR filter

**Syntax** `firnyquist(n,l,r,varargin)`

**Description** `b = firnyquist(n,l,r)` designs an Nth order, Lth band, Nyquist FIR filter with a roll-off factor  $r$  and an equiripple characteristic.

The rolloff factor  $r$  is related to the normalized transition width  $tw$  by  $tw = 2\pi(r/l)$  (rad/sample). The order,  $n$ , must be even.  $l$  must be an integer greater than one. If  $l$  is not specified, it defaults to 4.  $r$  must satisfy  $0 < r < 1$ . If  $r$  is not specified, it defaults to 0.5.

`b = firnyquist('minorder',l,r,dev)` designs a minimum-order, Lth band Nyquist FIR filter with a rolloff factor  $r$  using the Kaiser window. The peak ripple is constrained by the scalar  $dev$ .

`b = firnyquist(n,l,r,decay)` designs an Nth order, Lth band, Nyquist FIR filter where the scalar  $decay$ , specifies the rate of decay in the stopband.  $decay$  must be nonnegative. If omitted or left empty,  $decay$  defaults to 0 which yields an equiripple stopband. A nonequiripple stopband may be desirable for decimation purposes.

`b = firnyquist(n,l,r,'nonnegative')` returns an FIR filter with nonnegative zero-phase response. This filter can be spectrally factored into minimum-phase and maximum-phase “square-root” filters. This allows using the spectral factors in applications such as matched-filtering.

`b = firnyquist(n,l,r,'minphase')` returns the minimum-phase spectral factor  $b_{min}$  of order  $n$ .  $b_{min}$  meets the condition  $b = \text{conv}(b_{min}, b_{max})$  so that  $b$  is an Lth band FIR Nyquist filter of order  $2n$  with rolloff factor  $r$ . Obtain  $b_{max}$ , the maximum phase spectral factor by reversing the coefficients of  $b_{min}$ . For example,  $b_{max} = b_{min}(\text{end}:-1:1)$ .

**Example** 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, firfs, firminphase  
firrcos, firfs in your Signal Processing Toolbox documentation

**References**

[1] T. Saramaki, 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.

# firpr2chfb

---

**Purpose** Design 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`. The stopband-ripple of `h1` is also be given by `dev`, while the maximum stopband-ripple for both `g0` and `g1` is  $(2*\text{dev})$ .

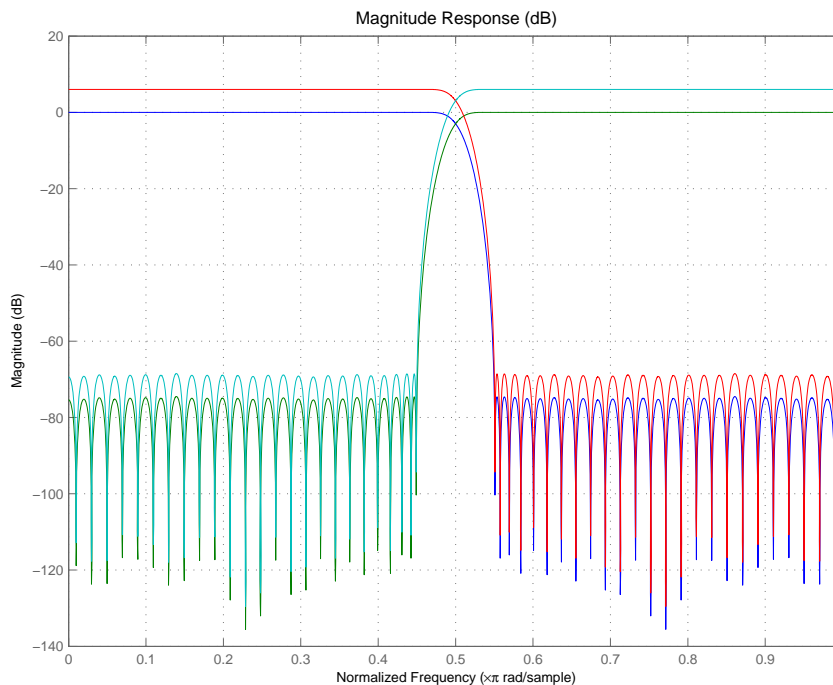
`[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)` designs the four filters such that `h0` meets the passband-edge specification `fp` and the stopband-ripple `dev` using minimum order filters to meet the specification.

**Algorithm** For perfect reconstruction, filters that compose the filter bank must fulfill these conditions.

**Examples** Design a filter bank with filters of order `n` equal to 99 and passband edges of 0.45 and 0.55.

```
n = 99;
[h0,h1,g0,g1] = firpr2chfb(n,.45);
fvtool(h0,1,h1,1,g0,1,g1,1);
```

Here are the filters, showing clearly the passband edges.



Use the following stem plots to verify perfect reconstruction using the filter bank created by `firpr2chfb`.

```
stem(1/2*conv(g0,h0)+1/2*conv(g1,h1))
n=0:n;
stem(1/2*conv((-1).^n.*h0,g0)+1/2*conv((-1).^n.*h1,g1))
stem(1/2*conv((-1).^n.*g0,h0)+1/2*conv((-1).^n.*g1,h1))
stem(1/2*conv((-1).^n.*g0,(-1).^n.*h0)+1/2*conv((-1).^n.*g1,...
(-1).^n.*h1))
stem(conv((-1).^n.*h1,h0)-conv((-1).^n.*h0,h1))
```

### See Also

`firceqrip`, `firgr`, `firhalfband`, `firnyquist`

# firtype

---

**Purpose** Determine type of linear phase FIR filter

**Syntax**  
`t = firtype(hd)`  
`t = firtype(hm)`

**Description** The next sections describe common `firtype` operation with discrete-time and multirate filters.

## Discrete-Time Filters

`t = firtype(hd)` determines the type (1 through 4) of a discrete-time FIR filter object `hd`, returning the type number in `t`. Filter `hd` must be both real and have linear phase.

Filter types 1 through 4 are defined as follows:

- Type 1—even order symmetric coefficients
- Type 2—odd order symmetric coefficients
- Type 3—even order antisymmetric coefficients
- Type 4—odd order antisymmetric coefficients

When `hd` is a cascade or parallel filter and therefore has multiple stages, each stage must be a real FIR filter with linear phase. In this case, `t` is a cell array containing the filter type of each stage.

## Multirate Filters

`t = firtype(hm)` determines the type (1 through 4) of the multirate filter object `hm`. The filter must be real and have linear phase.

Filter types 1 through 4 are defined as follows:

- Type 1—even order symmetric coefficients
- Type 2—odd order symmetric coefficients
- Type 3—even order antisymmetric coefficients
- Type 4—odd order antisymmetric coefficients

When `hm` has multiple sections, all sections must be real FIR filters with linear phase. In this case, `t` is a cell array containing the filter type of each section.

**Examples**

Determine the type of the default interpolator for L=4.

```
l = 4;  
hm = mfilt.firinterp(l);  
firtype(hm)  
ans =  
  
      1
```

**See Also**

islinphase

# freqsamp

**Purpose** Design real or complex frequency-sampled FIR filters from filter specification objects

**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 below shows a cell array input argument.
- The window vector itself.

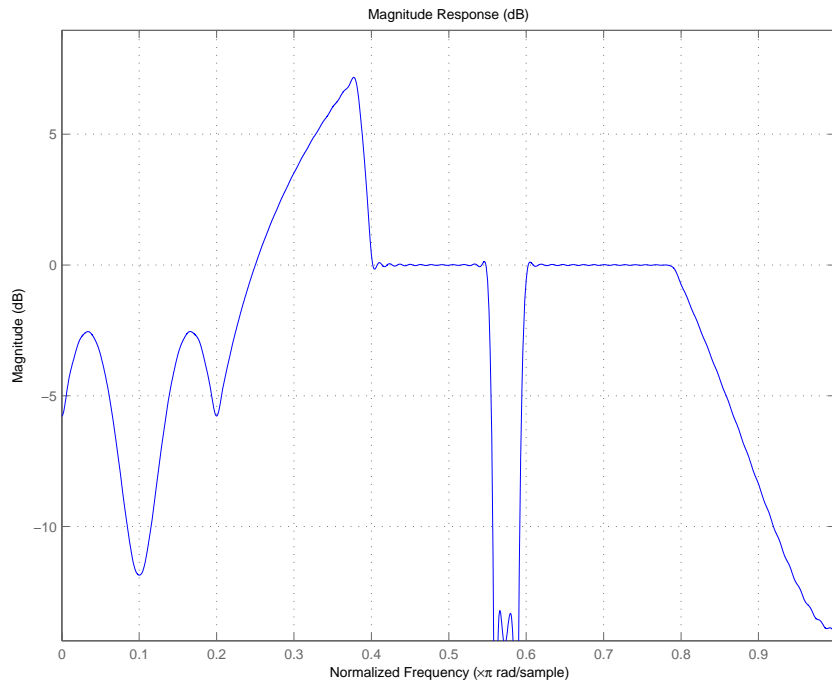
## Examples

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,'fregsamp','window',{@kaiser,.5}); % Filter.  
fvtool(hd)
```

The plot from FVTool shows the response for hd.



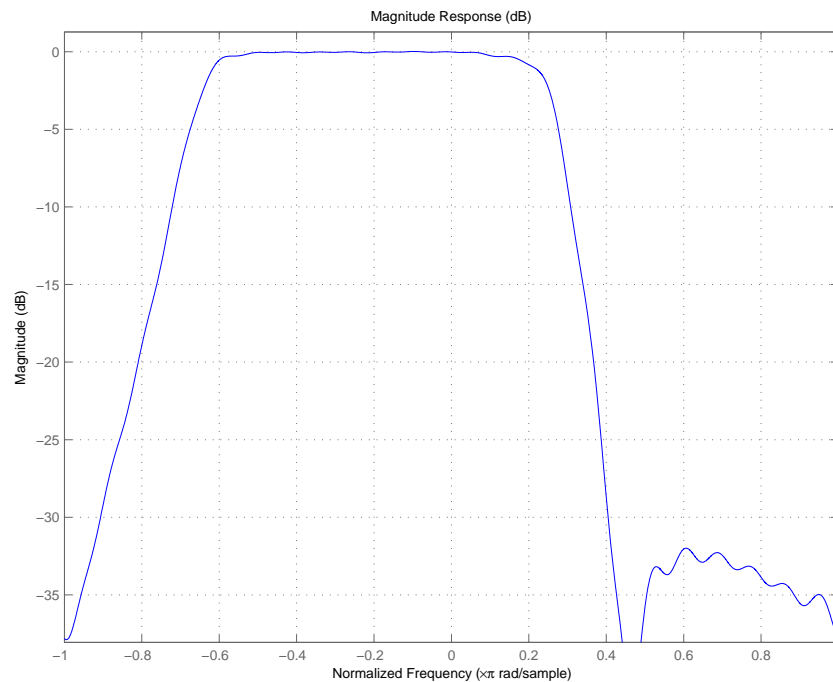
Now design the arbitrary-magnitude complex FIR filter. Recall that vector `f` contains frequency locations and vector `a` contains the desired filter response values at the locations specified in `f`.

```
f = [-1 -.93443 -.86885 -.80328 -.7377 -.67213 -.60656 -.54098 ...
    -.47541,-.40984 -.34426 -.27869 -.21311 -.14754 -.081967 ...
    -.016393 .04918 .11475,.18033 .2459 .31148 .37705 .44262 ...
    .5082 .57377 .63934 .70492 .77049,.83607 .90164 1];
a = [.0095848 .021972 .047249 .099869 .23119 .57569 .94032 ...
    .98084 .99707,.99565 .9958 .99899 .99402 .99978 .99995 .99733 ...
    .99731 .96979 .94936,.8196 .28502 .065469 .0044517 .018164 ...
    .023305 .02397 .023141 .021341,.019364 .017379 .016061];
n = 48;
d = fdesign.arbmag('n,f,a',n,f,a); % Second spec. object.
```



```
hdc = design(d,'fregsamp','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** Compute frequency response of discrete-time filters, adaptive filters, or multirate filters

**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 the Signal Processing Toolbox.

## 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 `hd`. When `hd` is a vector of multirate filters, `freqz` returns the matrix `h`. Each column of `h` corresponds to one filter in the vector `hd`.

`[h,w] = freqz(hd,n)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the multirate filter `hd`. `freqz` uses the transfer function associated with the multirate filter to calculate the frequency response of the filter with the current coefficient values. The vectors `h` and `w` are both of length `n`. The frequency vector `w` has values ranging from 0 to  $\pi$  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.

# freqz

---

## 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`.

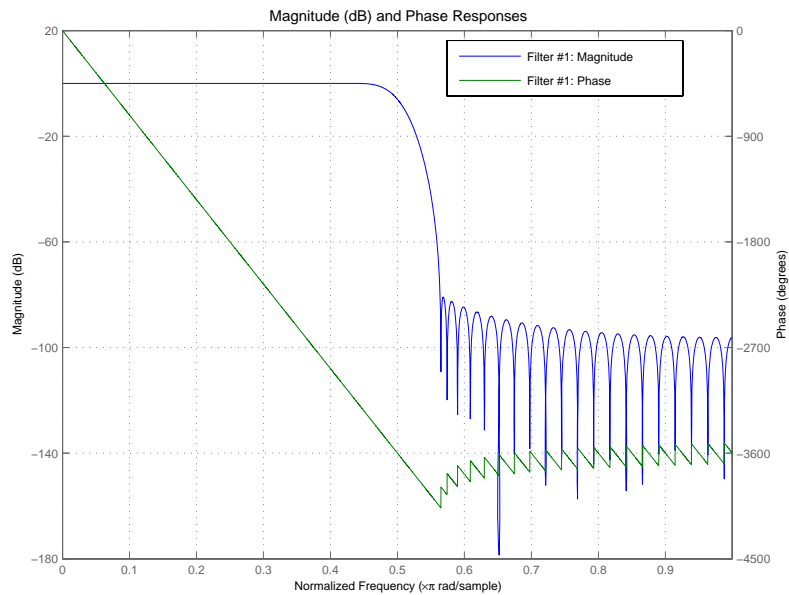
## Algorithm

`freqz` calculates the frequency response for a filter from the filter transfer function  $H_Q(z)$ . The complex-valued frequency response is calculated by evaluating  $H_Q(e^{j\omega})$  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

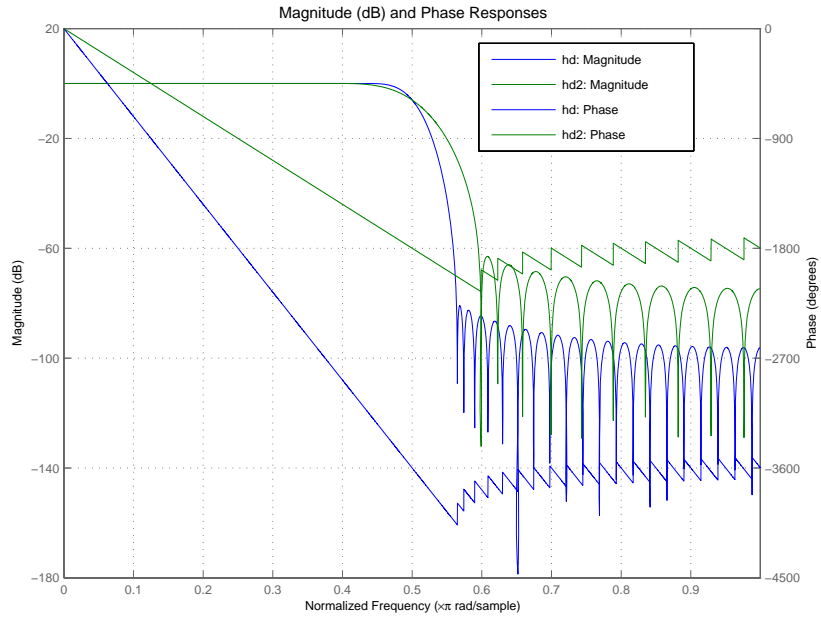
Plot the estimated frequency response of a filter. This example uses discrete-time filters, but any `adaptfilt`, `dfilt`, or `mfilt` object would work. First plot the results for one filter.

```
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

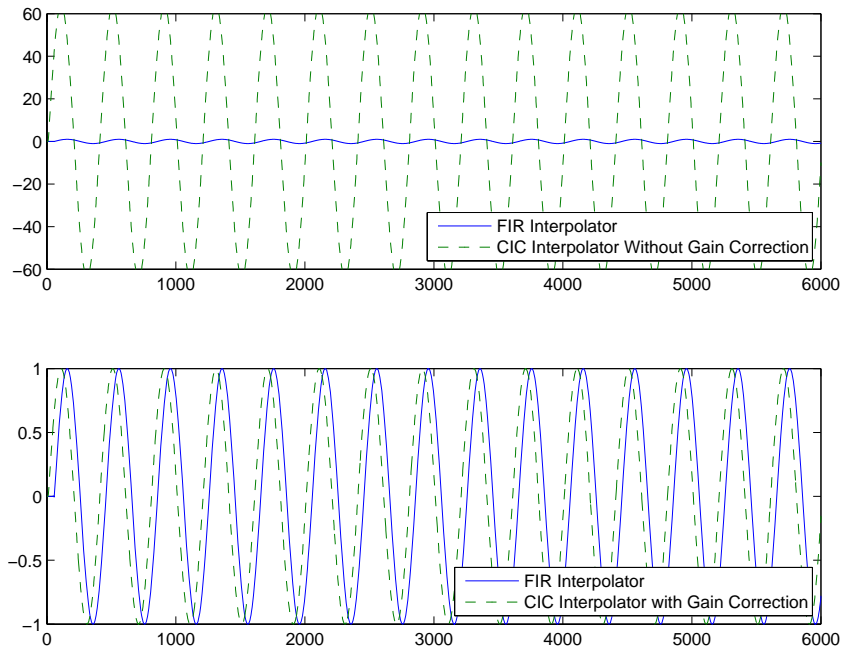
`adaptfilt`, `dfilt`, `mfilt`  
`fvtool` in your Signal Processing Toolbox documentation

<b>Purpose</b>	Gain of cascaded integrator-comb (CIC) filter
<b>Syntax</b>	<code>gain(hm)</code> <code>gain(hm,j)</code>
<b>Description</b>	<p><code>gain(hm)</code> returns the gain of <code>hm</code>, the CIC decimation or interpolation filter.</p> <p>When <code>hm</code> is a decimator, <code>gain</code> returns the gain for the overall CIC decimator.</p> <p>When <code>hm</code> is an interpolator, the CIC interpolator inserts zeros into the input data stream, reducing the filter overall gain by <math>1/R</math>, where <math>R</math> is the interpolation factor, to account for the added zero valued samples. Therefore, the gain of a CIC interpolator is <math>(RM)^N/R</math>, where <math>N</math> is the number of filter sections and <math>M</math> is the filter differential delay. <code>gain(hm)</code> returns this value. The example below presents this case.</p> <p><code>gain(hm,j)</code> returns the gain of the <math>j</math>th section of a CIC interpolation filter. When you omit <math>j</math>, <code>gain</code> assumes that <math>j</math> is <math>2*N</math>, where <math>N</math> is the number of sections, and returns the gain of the last section of the filter. This syntax does not apply when <code>hm</code> is a decimator.</p>
<b>Examples</b>	<p>To compare the performance of two interpolators, one a CIC filter and the other an FIR filter, use <code>gain</code> to adjust the CIC filter output amplitude to match the FIR filter output amplitude. Start by creating an input data set—a sinusoidal signal <code>x</code>.</p> <pre>fs = 1000;           % Input sampling frequency. t = 0:1/fs:1.5;     % Signal length = 1501 samples. x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.  l = 4; % Interpolation factor for FIR filter. d = fdesign.interpolator(l); hm = design(d,'multistage'); ym = filter(hm,x);  r = 4; % Interpolation factor for the CIC filter. d = fdesign.interpolator(r,'cic'); hcic = design(d,'multisection'); ycic = filter(hcic,x); gaincic = gain(hcic);</pre>

# gain

```
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

`filtmsb`



**Purpose** Group delay of adaptive, discrete-time, and multirate filters

**Syntax**

```
[gd,w] = grpdelay(ha)
[gd,w] = grpdelay(ha,n)
[gd,w] = grpdelay(...,f)
grpdelay(ha)
[gd,w] = grpdelay(hd)
[gd,w] = grpdelay(hd,n)
[gd,w] = grpdelay(...,f)
grpdelay(hd)
[gd,w] = grpdelay(hm)
[gd,w] = grpdelay(hm,n)
[gd,w] = grpdelay(...,f)
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 the Signal Processing Toolbox.

### 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(h,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(hm)` uses `FVTool` to plot the magnitude and unwrapped phase of the group delay of the multirate filter `hm`. If `hm` is a vector of filters, `grpdelay` plots the group delay for each filter in the vector.

# grpdelay

---

## See Also

phasez, zerophase

<b>Purpose</b>	Help text for design method with filter specification object
<b>Syntax</b>	<code>help(d, 'designmethod')</code>
<b>Description</b>	<code>help(d, 'designmethod')</code> displays help in the Command Window for the design algorithm <code>designmethod</code> for the current specifications of the filter specification object <code>d</code> . The string you enter for <code>designmethod</code> must be one of the strings returned by <code>designmethods</code> for <code>d</code> , the design object.
<b>Examples</b>	<p>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.</p> <pre>d = fdesign.lowpass; designmethods(d)</pre> <p>Design Methods for class <code>fdesign.lowpass (Fp,Fst,Ap,Ast)</code>:</p> <pre>butter cheby1 cheby2 ellip equiripple ifir kaiserwin multistage</pre> <pre>help(d, 'butter')</pre> <p>DESIGN Design a Butterworth IIR filter. <code>HD = DESIGN(D, 'butter')</code> designs a Butterworth filter specified by the <code>FDESIGN</code> object <code>D</code>.</p> <p><code>HD = DESIGN(..., 'FilterStructure', STRUCTURE)</code> returns a filter with the structure <code>STRUCTURE</code>. <code>STRUCTURE</code> is 'df2sos' by default and can be any of the following.</p> <pre>'df1sos' 'df2sos'</pre>

```
'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

# ifir

---

**Purpose** Use interpolated FIR method to design FIR filter from filter specification object

**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. In this syntax, `ifir` only creates minimum 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 the Filter Design Toolbox 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.

---

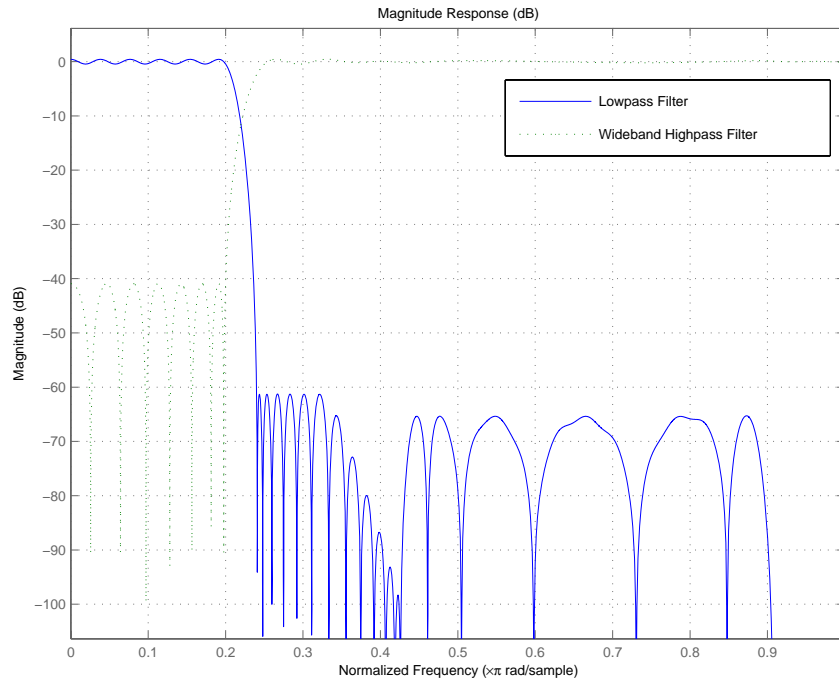
## Examples

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)
```



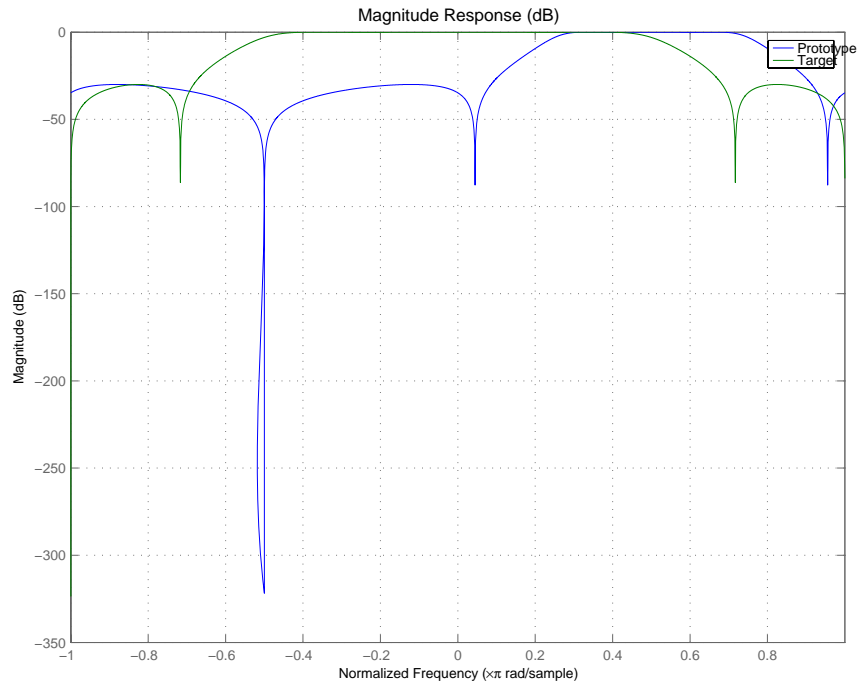
**See Also**

fdesign, firgr  
 fir1, fir1s, firpm in your Signal Processing Toolbox documentation

<b>Purpose</b>	Transform IIR complex bandpass filter to IIR complex bandpass filter with different frequency response 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> <p>Create a complex passband from 0.25 to 0.75:</p> <pre>[b, a] = iir1p2bpc(b, a, 0.5, [0.25,0.75]); [num, den] = iirbpc2bpc(b, a, [0.25, 0.75], [-0.5, 0.5]);</pre> <p>Verify the result by comparing the prototype filter with the target filter:</p>

```
fvtool(b, a, num, den);
```

Using FVTool to plot the filters shows you the comparison, presented in this figure.



## Arguments

B

Numerator of the prototype lowpass filter

A

Denominator of the prototype lowpass filter

Wo

Frequency values to be transformed from the prototype filter

Wt

Desired frequency locations in the transformed target filter

Num  
Numerator of the target filter

Den  
Denominator of the target filter

AllpassNum  
Numerator of the mapping filter

AllpassDen  
Denominator of the mapping filter

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

## See Also

`iirfttransf`, `allpassbpc2bpc`, `zpkbpc2bpc`

# iircomb

---

**Purpose** Design IIR comb notching or peaking digital 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 -3dB 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

$$H(z) = b \times \frac{1 - z^{-n}}{1 - az^{-n}}$$

where  $a$  and  $b$  are the filter coefficients and  $n$  is the filter order or the number of notches in the filter minus 1.

The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = \omega_0/bw$  where  $\omega_0$  is the frequency to remove from the signal.

[num,den] = iircomb(n,bw,ab) returns a digital notching filter whose bandwidth,  $bw$ , is specified at a level of  $-ab$  decibels. Including the optional input argument  $ab$  lets you specify the magnitude response bandwidth at a level that is not the default -3dB 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

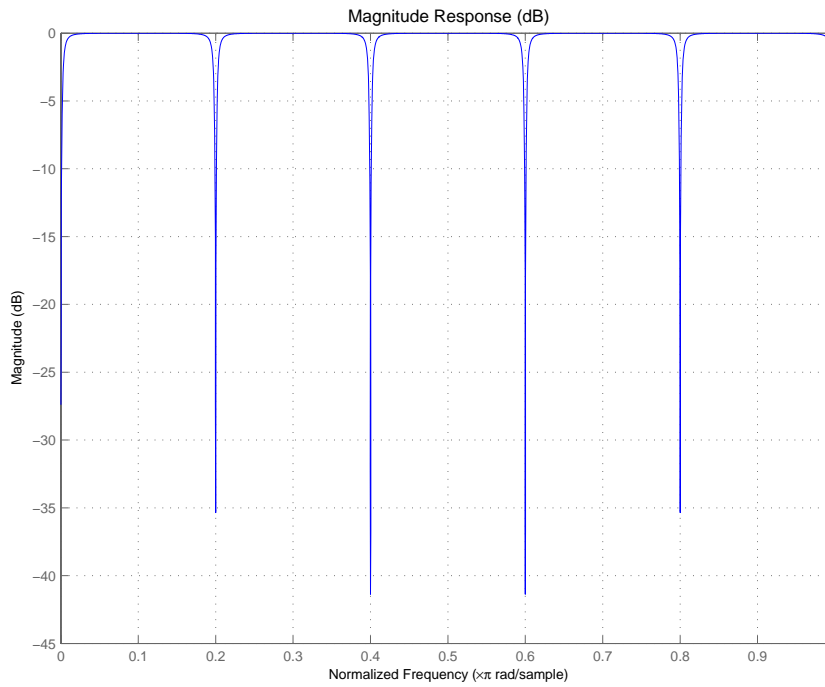
$$H(z) = b \times \frac{1 + z^{-n}}{1 - az^{-n}}$$

## 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 the type flag 'notch'  
fvtool(b,a);
```

Using the Filter Visualization Tool (FVTool) generates the following plot showing the filter notches. Note the notches are evenly spaced and one falls at exactly 60 Hz.



## See Also

`firgr`, `iirnotch`, `iirpeak`



**Purpose** IIR frequency transformation of digital filter

**Syntax** `[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)`

**Description** `[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)` returns the numerator and denominator vectors, `OutNum` and `OutDen`, of the target filter, which is the result of transforming the prototype filter specified by the numerator, `OrigNum`, and denominator, `OrigDen`, with the mapping filter given by the numerator, `FTFNum`, and the denominator, `FTFDen`. If the allpass mapping filter is not specified, then the function returns an original filter.

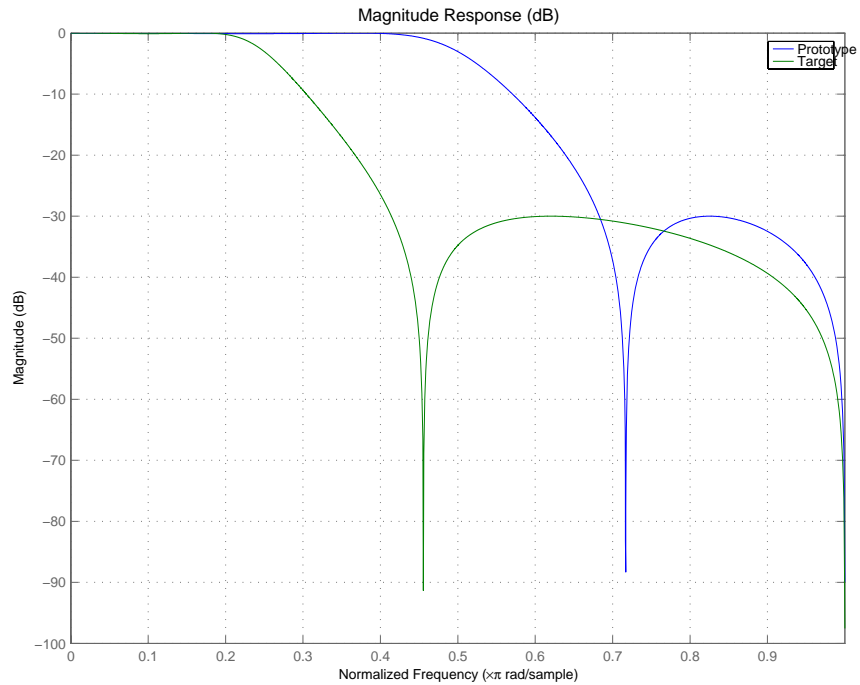
**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);  
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);  
[num, den] = iirftransf(b, a, AlpNum, AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Here's the comparison between the filters.



## Arguments

OrigNum

Numerator of the prototype lowpass filter

OrigDen

Denominator of the prototype lowpass filter

FTFNum

Numerator of the mapping filter

FTFDen

Denominator of the mapping filter

OutNum

Numerator of the target filter

OutDen

Denominator of the target filter

## See Also

[zpkftransf](#)

# iirgrpdelay

---

**Purpose** Optimal IIR filter design 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 the Signal Processing Toolbox 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.

**Purpose** Transform IIR real lowpass filter to IIR real bandpass filter frequency response

**Syntax** `[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)`  
`[G,AllpassNum,AllpassDen] = iirlp2bp(Hd,Wo,Wt)`, where Hd is a `dfilt` object

**Description** `[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)` returns the numerator and denominator vectors, Num and Den respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, AllpassNum, and the denominator AllpassDen, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by B and a denominator specified by A.

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . This transformation implements the “DC Mobility,” meaning that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of  $W_{ts}$ .

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

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

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

`[G,AllpassNum,AllpassDen] = iirlp2bp(Hd,Wo,Wt)` returns transformed `dfilt` object G with a real bandpass magnitude response. The coefficients AllpassNum and AllpassDen represent the allpass mapping filter for mapping the prototype filter frequency  $W_o$  and target frequencies vector  $W_t$ . 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);
```

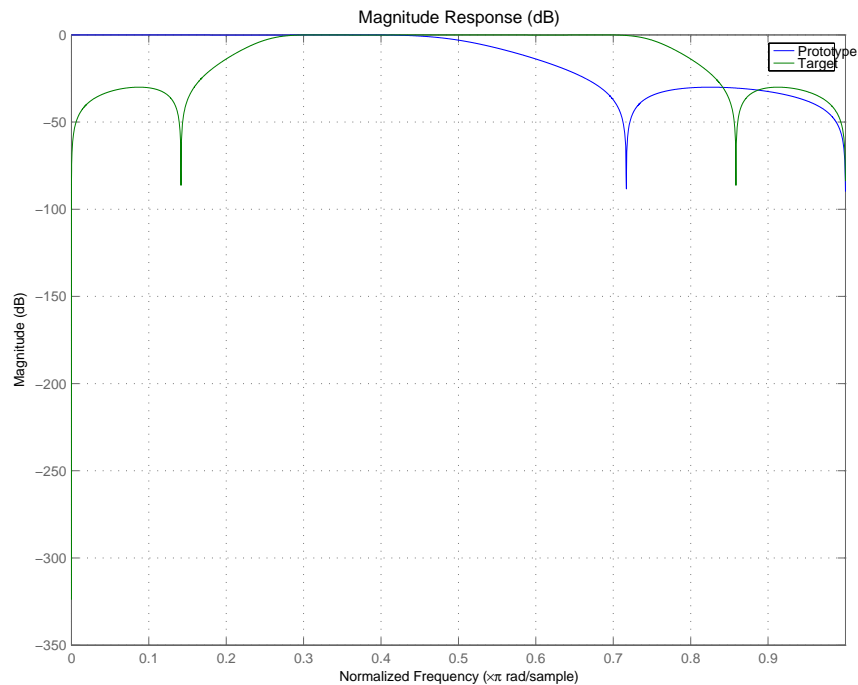
Create the real bandpass filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies  $W_{t1}=0.25$  and  $W_{t2}=0.75$ :

```
[num,den] = iirlp2bp(b,a,0.5,[0.25,0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b,a,num,den);
```

You can compare the results in this figure to verify the transformation.



## Arguments

B  
Numerator of the prototype lowpass filter

A

Denominator of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency locations in the transformed target filter

Num

Numerator of the target filter

Den

Denominator of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

iirftransf, allpasslp2bp, zpklp2bp

## References

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

[2] 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.

[3] 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 frequency transformation frequency response

**Syntax** `[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)`  
`[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt)`, where Hd is a `dfilt` object

**Description** `[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)` returns the numerator and denominator vectors, Num and Den respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandpass frequency transformation.

It also returns the numerator, AllpassNum, and the denominator, AllpassDen, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by B and a denominator specified by A.

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ .

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

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

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

`[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt)` returns transformed `dfilt` object G with a bandpass magnitude response. The coefficients AllpassNum and AllpassDen represent the allpass mapping filter for mapping

the prototype filter frequency  $\omega_0$  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);
```

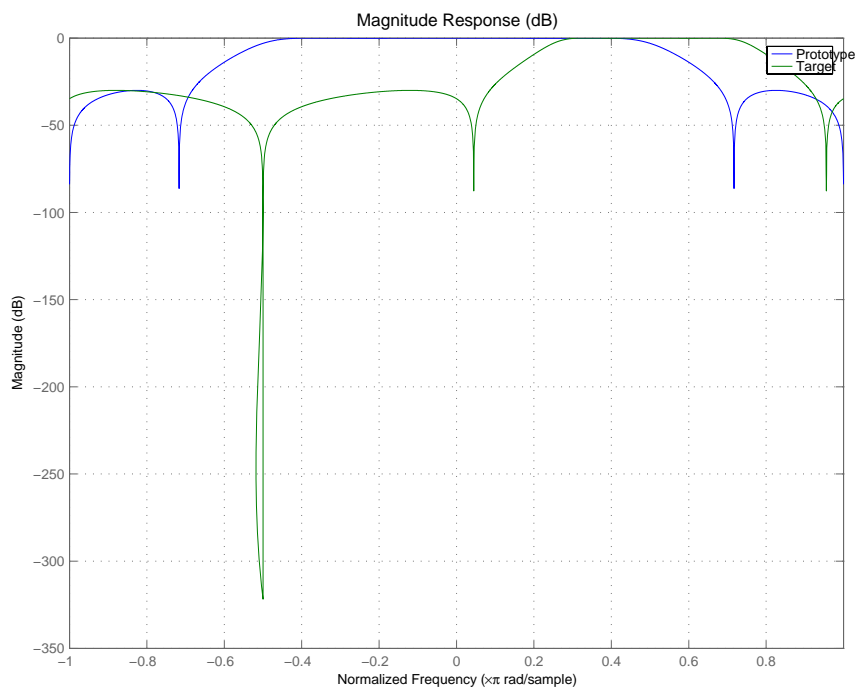
Move the cutoffs of the prototype filter to the new locations  $\omega_{t1}=0.25$  and  $\omega_{t2}=0.75$  creating a complex bandpass filter:

```
[num, den] = iirlp2bpc(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvttool(b, a, num, den);
```

Plotting the prototype and target filters together in FVTool lets you compare the filters.



## Arguments

B

Numerator of the prototype lowpass filter

A

Denominator of the prototype lowpass filter

W<sub>0</sub>

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<sub>t</sub>

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.

# iirlp2bpc

---

Num

Numerator of the target filter

Den

Denominator of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

## See Also

iirftransf, allpasslp2bpc, zpklp2bpc

**Purpose** Transform IIR real lowpass filter to IIR real bandstop filter frequency response

**Syntax** `[Num,Den,AllpassNum,AllpassDen] = iir1p2bs(B,A,Wo,Wt)`  
`[G,AllpassNum,AllpassDen] = iir1p2bs(Hd,Wo,Wt)`, where Hd is a `dfilt` object

**Description** `[Num,Den,AllpassNum,AllpassDen] = iir1p2bs(B,A,Wo,Wt)` returns the numerator and denominator vectors, Num and Den respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.

It also returns the numerator, AllpassNum, and the denominator, AllpassDen, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by B and a denominator specified by A.

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . This transformation implements the “Nyquist Mobility,” which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of  $W_o$  and  $W_{ts}$ .

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

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

`[G,AllpassNum,AllpassDen] = iir1p2bs(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  $W_o$  and the target frequencies vector  $W_t$ . 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:

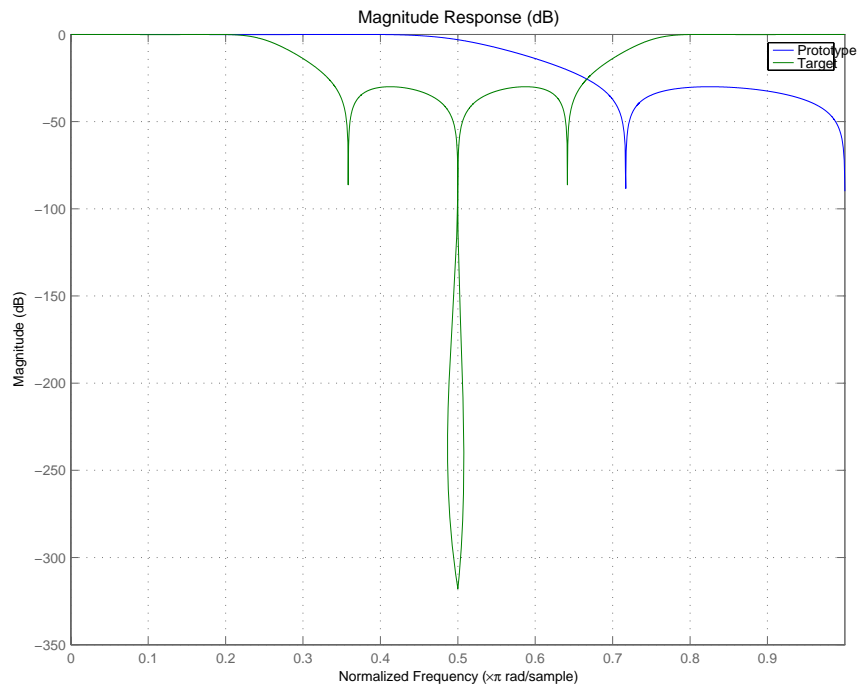
```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create the real bandstop filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies  $W_{t1}=0.25$  and  $W_{t2}=0.75$ :

```
[num, den] = iirlp2bs(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```



With both filters plotted in the figure, you see clearly the results of the transformation.

## Arguments

**B**  
Numerator of the prototype lowpass filter

A

Denominator of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency locations in the transformed target filter

Num

Numerator of the target filter

Den

Denominator of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

iirftransf, allpasslp2bs, zpklp2bs

## References

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

[2] 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.

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

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

**Purpose** Transform IIR real lowpass filter to IIR complex bandstop filter frequency response

**Syntax**  $[Num, Den, AllpassNum, AllpassDen] = iirlp2bsc(B, A, Wo, Wt)$   
 $[G, AllpassNum, AllpassDen] = iirlp2bsc(Hd, Wo, Wt)$ , where Hd is a `dfilt` object

**Description**  $[Num, Den, AllpassNum, AllpassDen] = iirlp2bsc(B, A, Wo, Wt)$  returns the numerator and denominator vectors, Num and Den respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, AllpassNum, and the denominator, AllpassDen, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by B and the denominator specified by A.

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . Additionally the transformation swaps passbands with stopbands in the target filter.

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

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

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

$[G, AllpassNum, AllpassDen] = iirlp2bsc(Hd, Wo, Wt)$  returns transformed `dfilt` object G with a bandstop magnitude response. The coefficients



AllpassNum and AllpassDen represent the allpass mapping filter for mapping the prototype filter frequency  $\omega_0$  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);
```

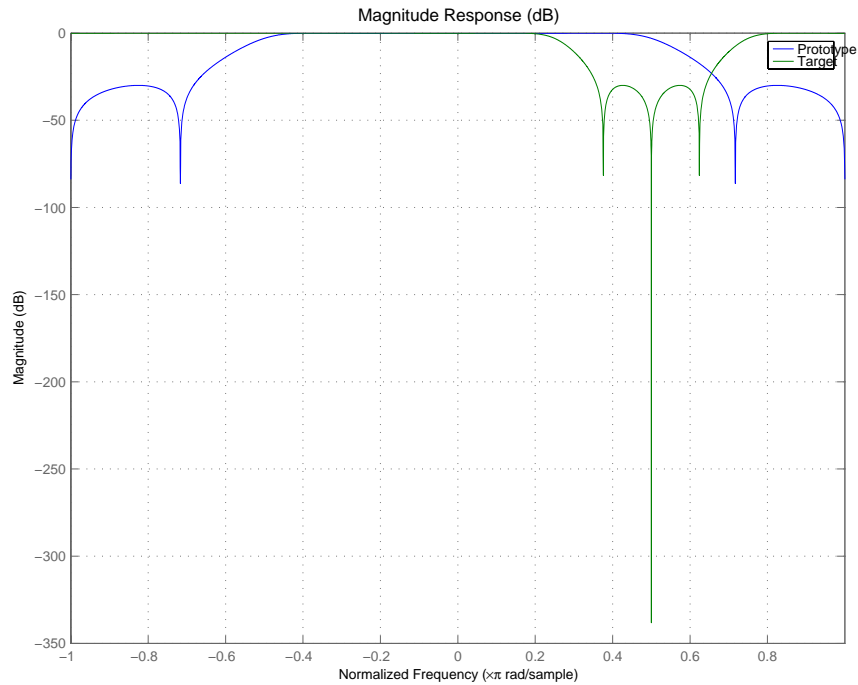
Move the cutoffs of the prototype filter to the new locations  $\omega_{t1}=0.25$  and  $\omega_{t2}=0.75$  creating a complex bandstop filter:

```
[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

**B**  
Numerator of the prototype lowpass filter

**A**  
Denominator of the prototype lowpass filter

**Wo**  
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

**Wt**  
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Num  
Numerator of the target filter

Den  
Denominator of the target filter

AllpassNum  
Numerator of the mapping filter

AllpassDen  
Denominator of the mapping filter

**See Also**

iirftransf, allpasslp2bsc, zpklp2bsc.

**Purpose** Transform discrete time 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.

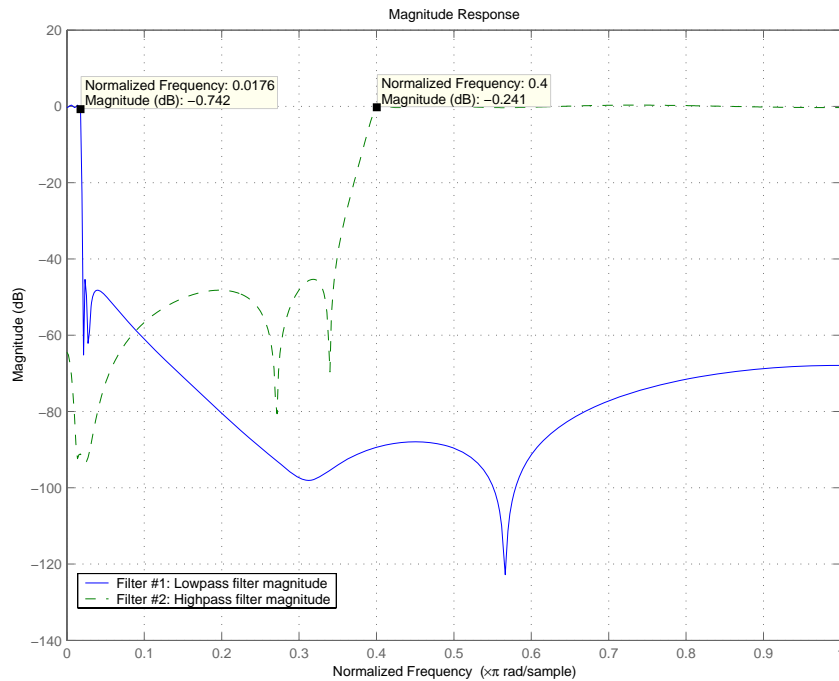
`[G,AllpassNum,AllpassDen] = iirlp2hp(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a highpass magnitude response. The coefficients

AllpassNum and AllpassDen represent the allpass mapping filter for mapping the prototype filter frequency  $W_0$  and the target frequencies vector  $W_t$ . 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, we select the frequency in the lowpass filter where the passband starts to rolloff ( $w_c = 0.0175$ ) and move it to the new location at  $w_d = 0.4$ .

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1],...  
[0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0],[1 1 1 1 10 10]);  
wc = 0.0175;  
wd = 0.4;  
[num,den] = iirlp2hp(b,a,wc,wd);  
fvtool(b,a,num,den);
```



In the figure showing the magnitude responses for the two filters, the transition band for the highpass filter is essentially the mirror image of the transition for the lowpass filter from 0.0175 to 0.025, stretched out over a wider frequency range. In the passbands, the filter share common ripple characteristics and magnitude.

## See Also

`iirlp2bp`, `iirlp2bs`, `iirlp2lp`, `firlp2lp`, `firlp2hp`

## References

Sanjit K. Mitra, *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

**Purpose** Transform discrete time lowpass IIR filter to different lowpass filter

**Syntax** `[ num,den ] = iir1p21p(b,a,wc,wd)`  
`[ G,AllpassNum,AllpassDen ] = iir1p21p(Hd,Wo,Wt)`, where Hd is a `dfilt` object

**Description** `[ num,den ] = iir1p21p(b,a,wc,wd)` with input arguments b and a, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, `iir1p21p` 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 ] = iir1p21p(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  $\omega_0$  and the target frequencies vector  $\omega_t$ . Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

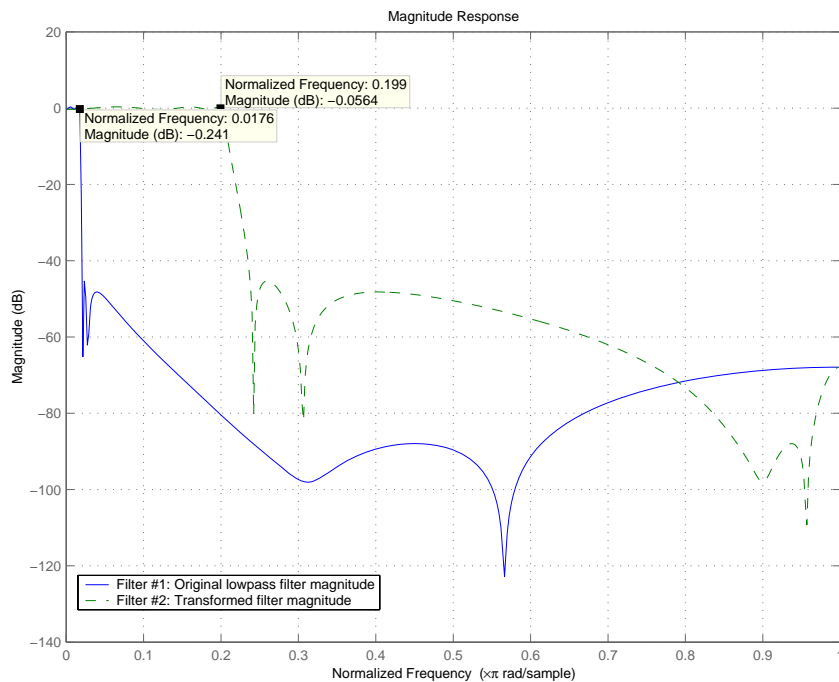
## Examples

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, we select the frequency in the lowpass filter where the passband starts to rolloff ( $\omega_c = 0.0175$ ) and move it to the new location at  $\omega_d = 0.2$ .

```
[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.





Notice that 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

Sanjit K. Mitra, *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

# iirlp2mb

---

**Purpose** Transform IIR real lowpass filter to IIR real M-band filter frequency response

**Syntax**

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt,Pass)
[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt), where Hd is a dfilt object
[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 multibandpass 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  $W_0$ , 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] = iirlp2bs(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(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.

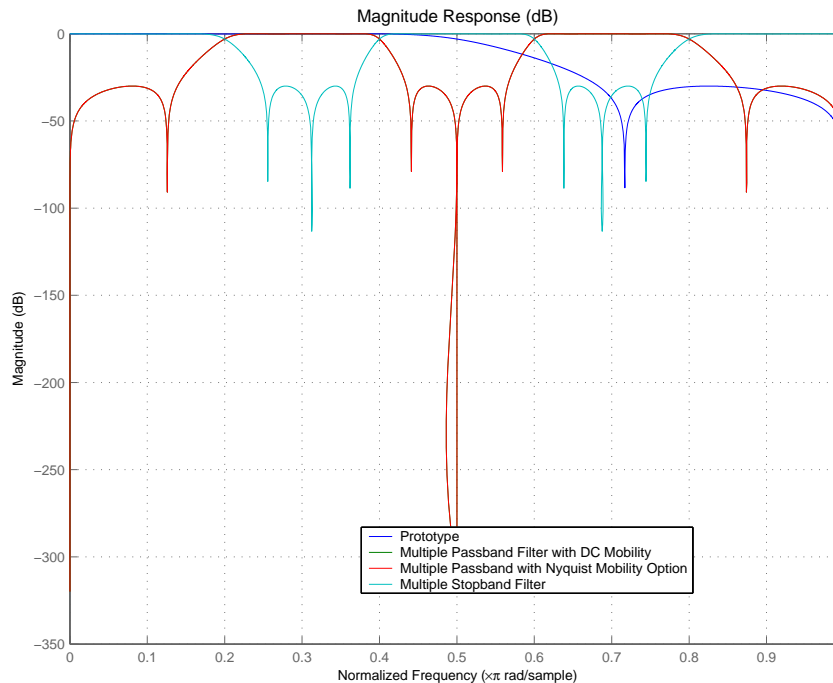
**Example 2:** Create the real multiband filter with two stopbands:

```
[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

B  
Numerator of the prototype lowpass filter

A  
Denominator of the prototype lowpass filter

Wo  
Frequency value to be transformed from the prototype filter

Wt  
Desired frequency locations in the transformed target filter

Pass  
Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default

Num  
Numerator of the target filter

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, allpass1p2mb, zpk1p2mb

### References

- [1] Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," MSc Thesis, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.
- [2] 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.
- [3] Mullis, C.T. and R. A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Mass., Addison-Wesley, 1987.
- [4] Feyh, G., W.B. Jones and C.T. Mullis, "An extension of the Schur Algorithm for frequency transformations," *Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

# iirlp2mbc

---

**Purpose** Transform IIR real lowpass filter to IIR complex M-band filter frequency response

**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 Mth-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  $W_0$ , 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  $W_0$  and the target frequencies vector  $W_t$ . 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:

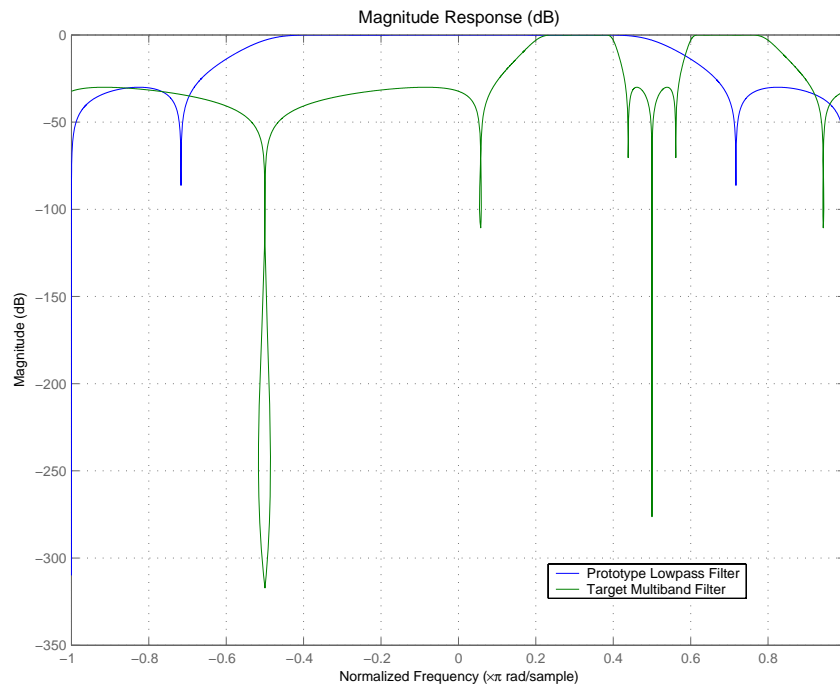
```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

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.

# iirlp2mbc

---

## Arguments

B

Numerator of the prototype lowpass filter.

A

Denominator of the prototype lowpass filter.

Wo

Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Wc

Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Num

Numerator of the target filter.

Den

Denominator of the target filter.

AllpassNum

Numerator of the mapping filter.

AllpassDen

Denominator of the mapping filter.

## See Also

iirftransf, allpasslp2mbc, zpklp2mbc



<b>Purpose</b>	Transform IIR real lowpass filter to IIR complex N-point filter frequency response
<b>Syntax</b>	<pre>[Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt) [G,AllpassNum,AllpassDen] = iirlp2xc(Hd,Wo,Wt), where Hd is a dfilt object</pre>
<b>Description</b>	<p><code>[Num,Den,AllpassNum,AllpassDen] = iirlp2xc(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 an Nth-order real lowpass to complex multipoint 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 a denominator specified by A.</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_{01}, \dots, W_{0N}</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., 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.</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>

`[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);
```

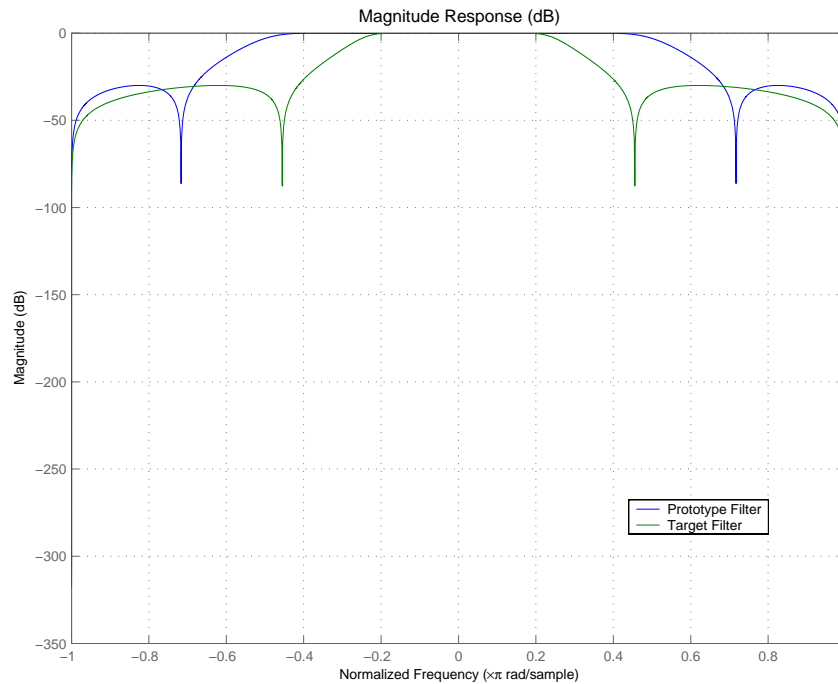
Create the complex bandpass filter from the real lowpass filter:

```
[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

B

Numerator of the prototype lowpass filter

A

Denominator of the prototype lowpass filter

Wo

Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Wt

Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

# iirlp2xc

---

Num

Numerator of the target filter

Den

Denominator of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

## See Also

iirfttransf, allpasslp2xc, zpklp2xc

**Purpose** Transform IIR real lowpass filter to IIR real N-point filter frequency response

**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  $W_{01}, \dots, W_{0N}$ , 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);
```

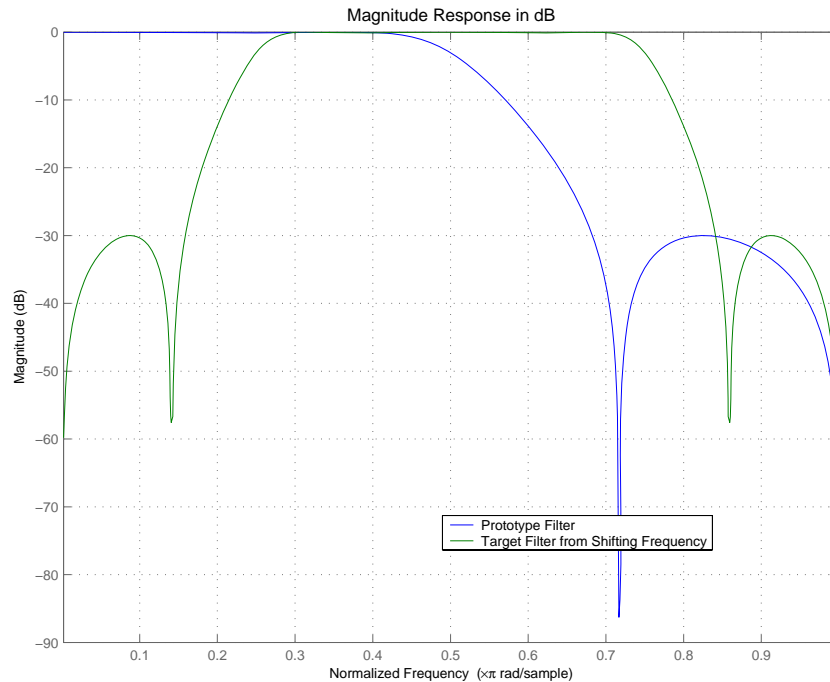
Move the cutoffs of the prototype filter to the new locations  $W_{t1}=0.25$  and  $W_{t2}=0.75$  creating a real bandpass filter:

```
[num, den] = iirlp2xn(b, a, [-0.5 0.5], [0.25 0.75], 'pass');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

iirlp2xn has created the desired bandpass filter with the cutoff locations specified in the command.



## Arguments

B

Numerator of the prototype lowpass filter

A

Denominator of the prototype lowpass filter

Wo

Frequency values to be transformed from the prototype filter

Wt

Desired frequency locations in the transformed target filter

Pass

Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default

Num

Numerator of the target filter

Den

Denominator of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

iirftransf, allpasslp2xn, zpklp2xn

## References

- [1] 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.
- [2] Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.



**Purpose** Least P-norm optimal IIR filter design

**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” 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.

# iirlpnorm

---

`[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 the Signal Processing Toolbox 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** Design 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- $p$ th sense. The vector  $edges$  specifies the band-edge frequencies for multi-band designs. A constrained Newton-type algorithm is employed.  $n$  and  $d$  should be chosen so that the zeros and poles are used effectively. See the “Hints” section. Always check the resulting filter using `fvtool`.

`[num,den] = iirlpnormc(n,d,f,edges,a,w)` uses the weights in  $w$  to weight the error.  $w$  has one entry per frequency point (the same length as  $f$  and  $a$ ) which tells `iirlpnormc` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points.  $f$  and  $a$  must have the same number of elements, which can exceed the number of elements in  $edges$ . This allows for the specification of filters having any gain contour within each band. The frequencies specified in  $edges$  must also appear in the vector  $f$ . For example,

```
[num,den] = iirlpnormc(5,5,[0 .15 .4 .5 1],[0 .4 .5 1],...
    [1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)` returns a filter having a maximum pole radius of  $radius$  where  $0 < radius < 1$ .  $radius$  defaults to 0.999999. Filters that have a reduced pole radius may retain better transfer function accuracy after you quantize them.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)` where `p` is a two-element vector `[pmin pmax]` allows for the specification of the minimum and maximum values of `p` used in the least-`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] = 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 the Signal Processing Toolbox can be used for generating `initnum` and `initden`.

`[num,den,err] = iirlpnormc(...)` returns the least-`P`th 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-`p`th 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:

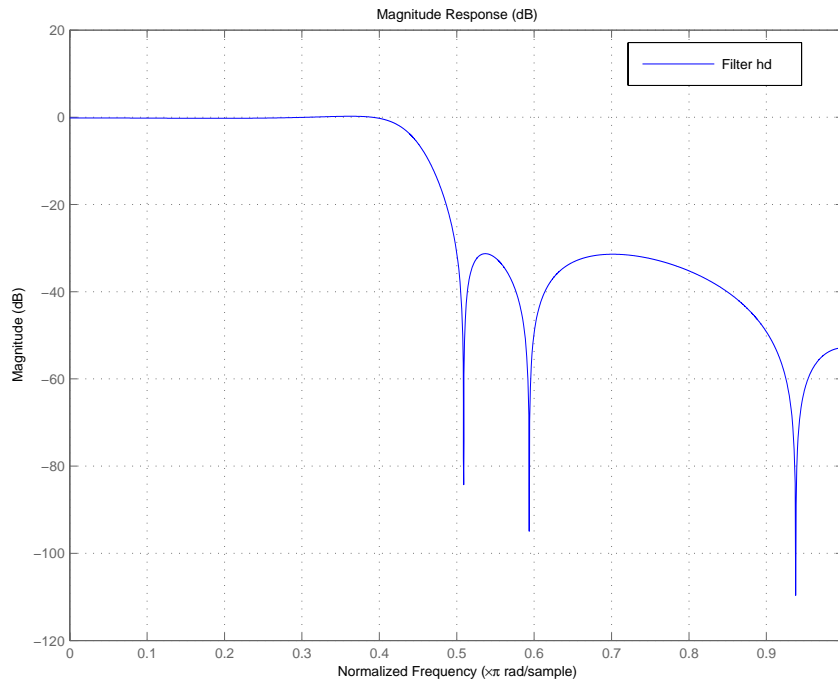
- a** The approximation error is extremely small (try reducing the number of poles or zeros—refer to the hints above).
- b** The filter specifications have huge variation, such as `a=[1 1e9 0 0]`.

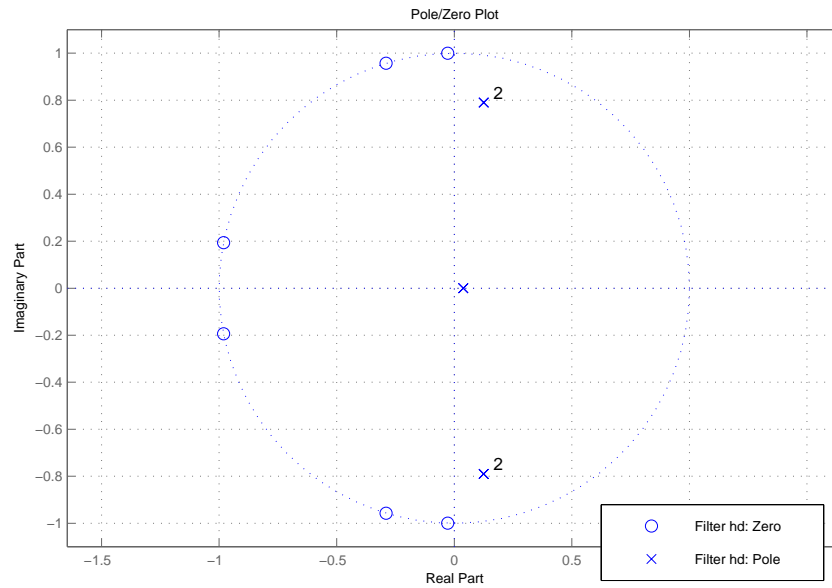
## Examples

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.



**See Also**

freqz, filter, iirgrpdelay, iirlpnorm, zplane

**References**

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

# iirls

---

**Purpose** Design least-squares IIR filter from filter specification object

**Syntax**  
`hd = design(d,'iirls')`  
`hd = design(d,'iirls',designoption,value,designoption,value,...)`

**Description** `hd = design(d,'iirls')` designs a least-squares filter specified by the filter specification object `d`.

---

**Note** The `iirls` algorithm might not be well behaved in all cases. Experience is your best guide to determining if the resulting filter meets your needs. When you use `iirls` to design a filter, review the filter carefully to ensure that it is appropriate for your use.

---

`hd = design(d,'iirls',designoption,value,designoption,value,...)` returns a least-squares IIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

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

```
help(d,'iirls')
```

## Examples

Starting from an arbitrary magnitude and phase design object `d`, generate a complex bandpass filter of order = 5. To make the example a little easier to do, use the default values for `F`, and `H`, the frequency vector and the complex desired frequency response.

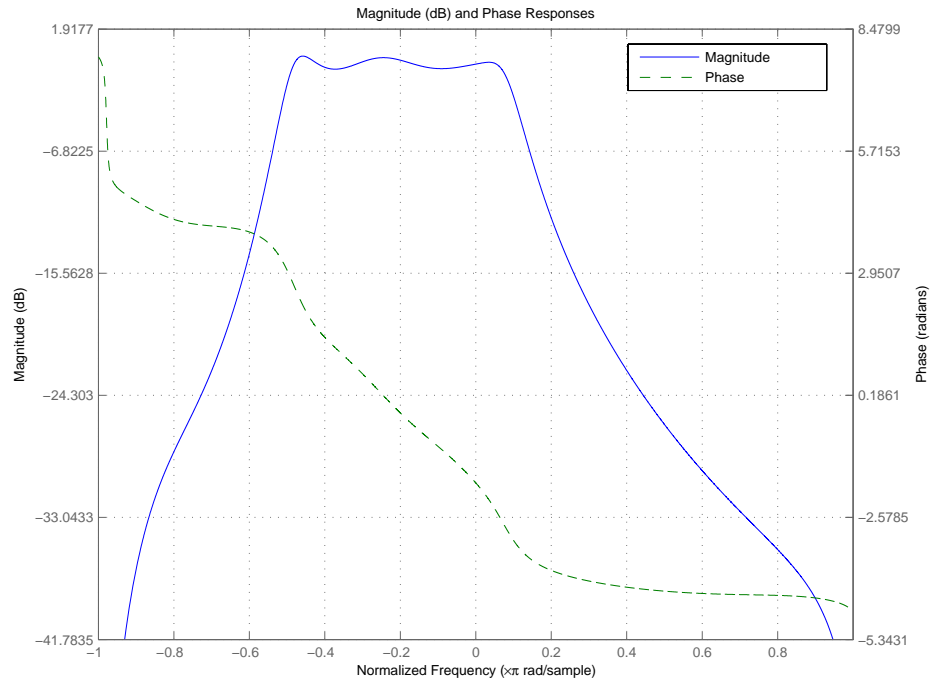
```
d = fdesign.arbmagnphase('N,F,H',5);  
d =  
  
    Response: 'Arbitrary Magnitude and Phase'  
    Specification: 'N,F,H'  
    Description: {'Filter Order';'Frequency Vector';'Complex Desired Frequency  
                Response'}  
    NormalizedFrequency: true
```



```
FilterOrder: 5  
Frequencies: [1x655 double]  
FreqResponse: [1x655 double]
```

```
design(d,'iirls'); % Opens FVTool to show the filter.
```

Displaying both the phase and magnitude response in FVTool shows you the filter.



## See Also

`fdesign.arbmag`, `fdesign.arbmagnphase`, `firls`

# iirnotch

---

**Purpose** Design second-order IIR notch digital 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.

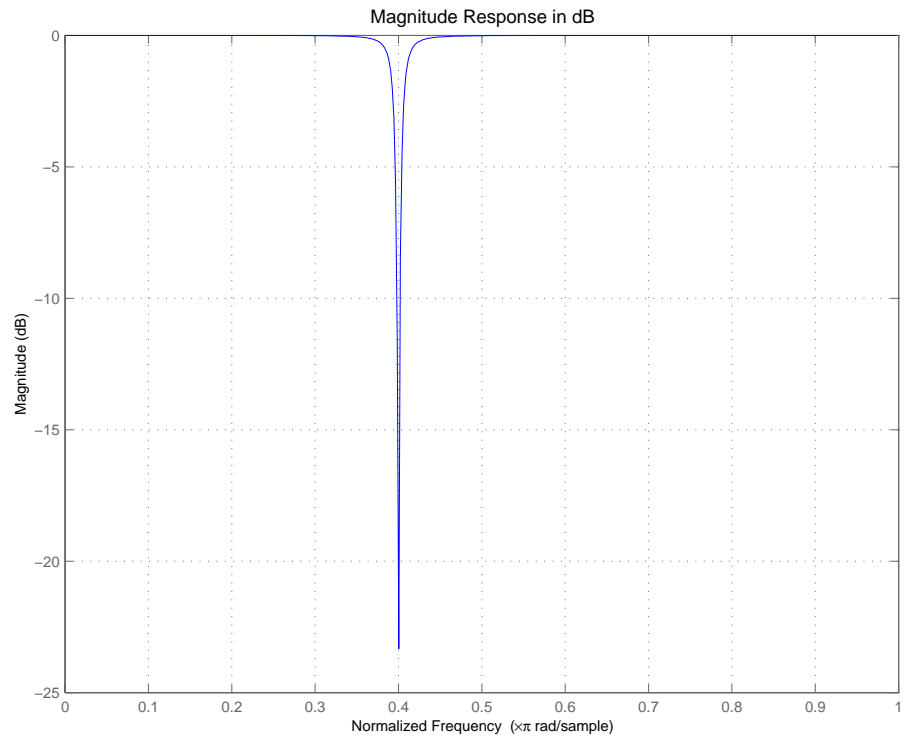
The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = w_0/bw$  where  $\omega_0$  is  $w_0$ , the frequency to remove from the signal.

`[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 -3dB 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** Design second-order IIR peak or resonator digital 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 +3dB 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.

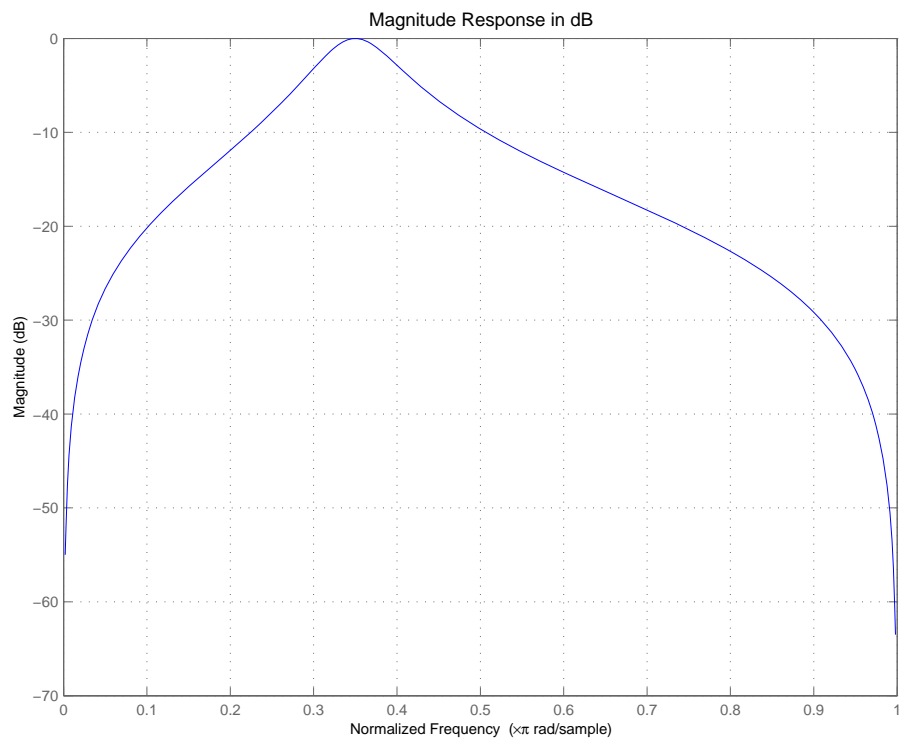
The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = w_0/bw$  where  $\omega_0$  is  $w_0$  the signal frequency to boost.

`[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 +3dB 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** Compute power complementary filter

**Syntax** `[bp,ap] = iirpowcomp(b,a)`  
`[bp,ap,c] = iirpowcomp(b,a)`

**Description** `[bp,ap] = iirpowcomp(b,a)` returns the coefficients of the power complementary IIR filter  $g(z) = bp(z)/ap(z)$  in vectors `bp` and `ap`, given the coefficients of the IIR filter  $h(z) = b(z)/a(z)$  in vectors `b` and `a`. `b` must be symmetric (Hermitian) or antisymmetric (antihermitian) and of the same length as `a`. The two power complementary filters satisfy the relation

$$|H(w)|^2 + |G(w)|^2 = 1.$$

`[bp,ap,c] = iirpowcomp(b,a)` where `c` is a complex scalar of magnitude =1, forces `bp` to satisfy the generalized hermitian property

`conj(bp(end:-1:1)) = c*bp.`

When `c` is omitted, it is chosen as follows:

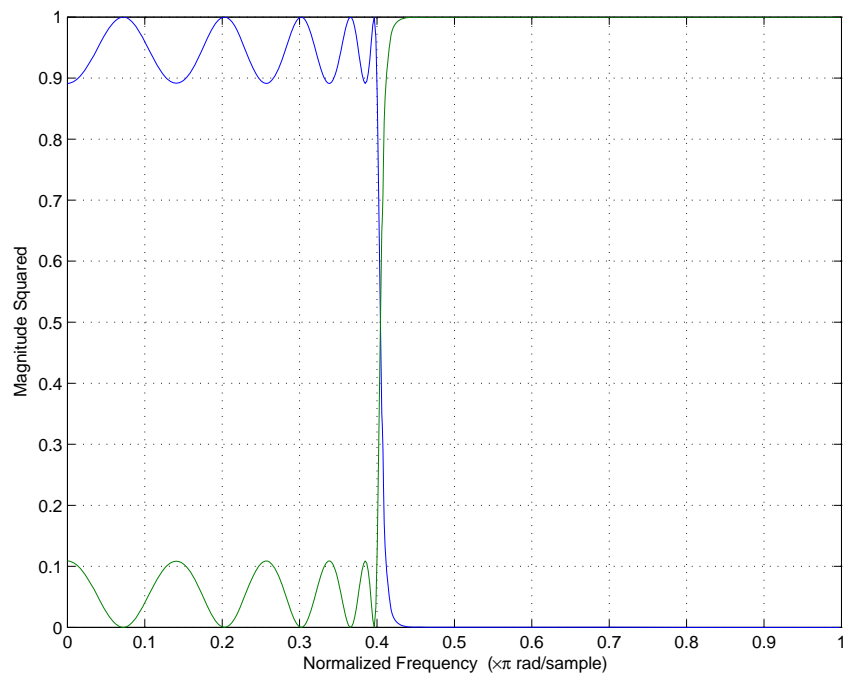
- When `b` is real, chooses `C` as 1 or -1, whichever yields `bp` real
- When `b` is complex, `C` defaults to 1

`ap` is always equal to `a`.

## Examples

```
[b,a]=cheby1(10,.5,.4);  
[bp,ap]=iirpowcomp(b,a);  
[h,w,s]=freqz(b,a); [h1,w,s]=freqz(bp,ap);  
s.plot='mag'; s.yunits='sq';freqzplot([h h1],w,s)
```

The next figure presents the results of applying `iirpowcomp` to the Chebyshev filter—the power complementary version of the original filter.

**See Also**`tf2ca`, `tf2cl`, `ca2tf`, `cl2tf`

# iirateup

---

**Purpose** Upsample IIR filter by integer factor

**Syntax** `[Num,Den,AllpassNum,AllpassDen] = iirateup(B,A,N)`

**Description** `[Num,Den,AllpassNum,AllpassDen] = iirateup(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:

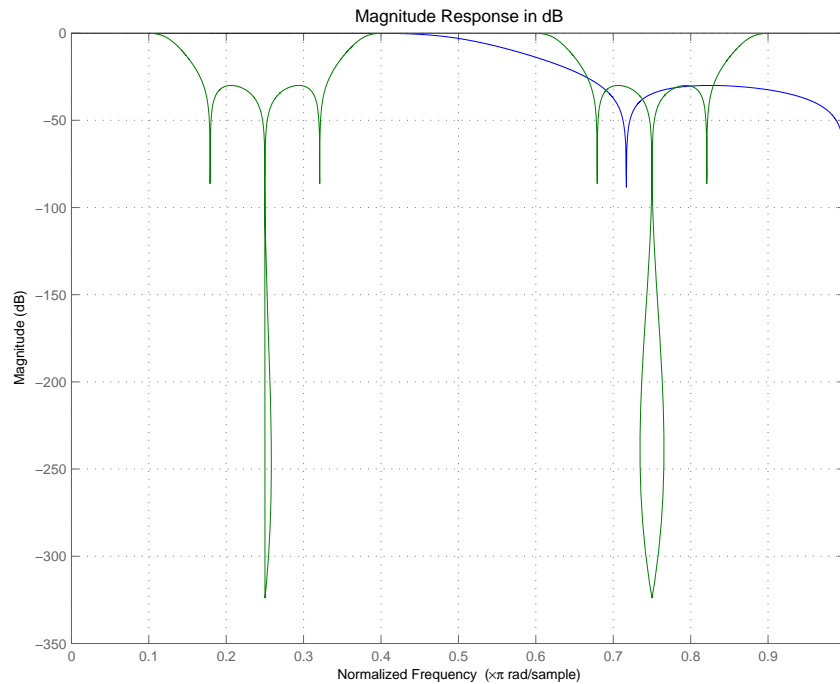
```
[b, a] = ellip(3, 0.1, 30, 0.409);  
[num, den] = iirateup(b, a, 4);
```

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

B  
Numerator of the prototype lowpass filter

A  
Denominator of the prototype lowpass filter

N  
Frequency multiplication ratio

Num  
Numerator of the target filter

Den  
Denominator of the target filter

AllpassNum  
Numerator of the mapping filter

# iirateup

---

AllpassDen  
Denominator of the mapping filter

**See Also**      iirfttransf, allpassrateup, zpkrateup

<b>Purpose</b>	Shift frequency response of IIR real filter
<b>Syntax</b>	<code>[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)</code>
<b>Description</b>	<p><code>[Num,Den,AllpassNum,AllpassDen] = iirshift(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 second-order real shift frequency mapping.</p> <p>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.</p> <p>This transformation places one selected feature of an original filter located at frequency <math>W_0</math> to the required target frequency location, <math>W_t</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_0</math> and <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 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.</p> <p>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.</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> <p>Perform the real frequency shift by defining where the selected feature of the prototype filter, originally at <math>W_0=0.5</math>, should be placed in the target filter, <math>W_t=0.75</math>:</p>

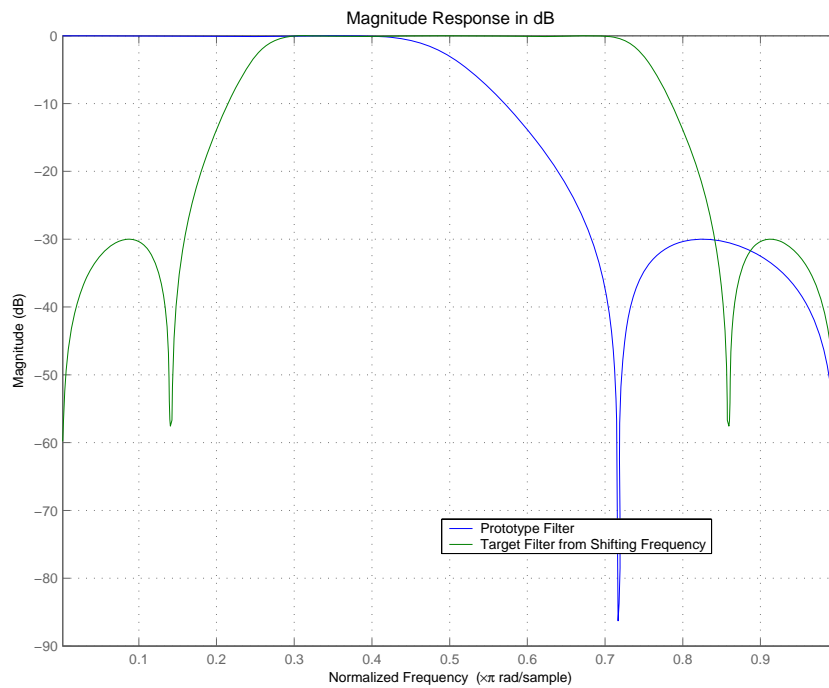
```
Wo = 0.5; Wt = 0.75;
```

```
[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  
Numerator of the prototype lowpass filter

A  
Denominator of the prototype lowpass filter

$W_o$

Frequency value to be transformed from the prototype filter

$W_t$

Desired frequency location in the transformed target filter

Num

Numerator of the target filter

Den

Denominator of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

**See Also**

iirftransf, allpassshift, zpkshift.

# iirshifc

---

**Purpose** Shift frequency response of IIR complex filter

**Syntax** `[Num,Den,AllpassNum,AllpassDen] = iirshifc(B,A,Wo,Wt)`

**Description** `[Num,Den,AllpassNum,AllpassDen] = iirshifc(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] = iirshifc(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] = iirshifc(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);
```

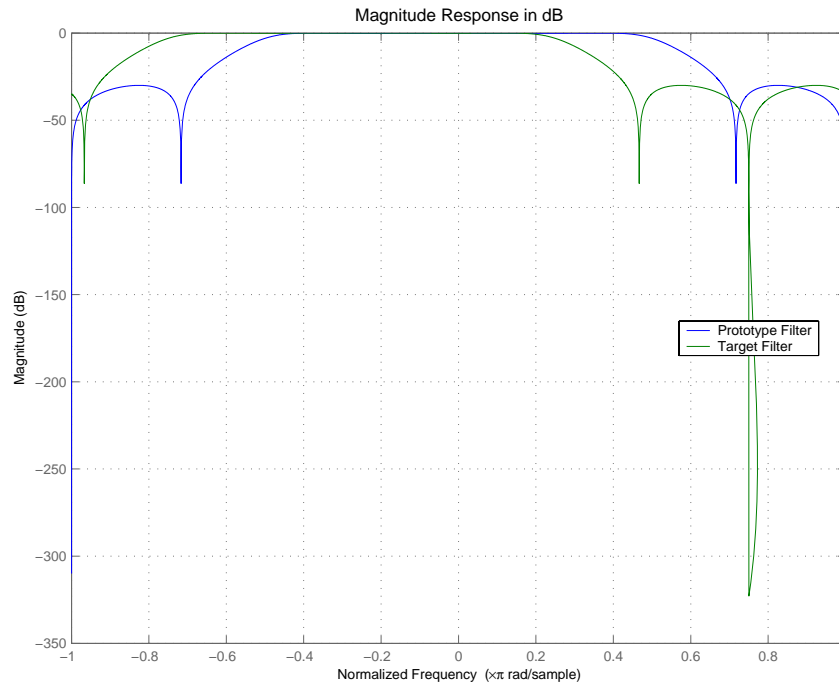
Rotate all features of the prototype filter in the frequency domain by the same amount by specifying where the selected feature of an original filter,  $W_o=0.5$ , should appear in the target filter,  $W_t=0.25$ :

```
[num, den] = iirshifc(b, a, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvttool(b, a, num, den);
```

After applying the shift, the selected feature from the original filter is just where it should be, at  $W_t = 0.25$ .



## Arguments

B

Numerator of the prototype lowpass filter

A

Denominator of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

Num

Numerator of the target filter

Den

Denominator of the target filter

# iirshiftc

---

AllpassNum  
Numerator of the mapping filter

AllpassDen  
Denominator of the mapping filter

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

## See Also

iirftransf, allpassshiftc, zpkshiftc

## References

- [1] Oppenheim, A.V., R.W. Schaffer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.
- [2] 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** Compute impulse response of filters

**Syntax**

```
[h,t] = impz(ha)
[h,t] = impz(...,fs)
impz(ha,...)
[h,t] = impz(hd)
[h,t] = impz(...,fs)
impz(hd,...)
[h,t] = impz(hm)
[h,t] = impz(...,fs)
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 the Signal Processing Toolbox.

- “Discrete-Time Filters” on page 8-796
- “Multirate Filters” on page 8-796

### 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, low-pass elliptic filter with a cutoff frequency of 0.4 times the Nyquist frequency. Use a second-order sections structure to resist quantization errors. Plot the first 50 samples of the impulse response, along with the reference impulse response.

```
% Create a design object for the prototype filter.
```

```
d = fdesign.lowpass(.4,.5,1,80)
```

```
d =
```

```

           Response: 'Minimum-order lowpass'
    Specification: 'Fp,Fst,Ap,Ast'
      Description: {4x1 cell}
NormalizedFrequency: true
              Fs: 'Normalized'
             Fpass: 0.4000
             Fstop: 0.5000
             Apass: 1
             Astop: 80

```

Use `ellip` to design the discrete-time filter in second-order section form, with minimum-order.

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

```
hd =
```

```

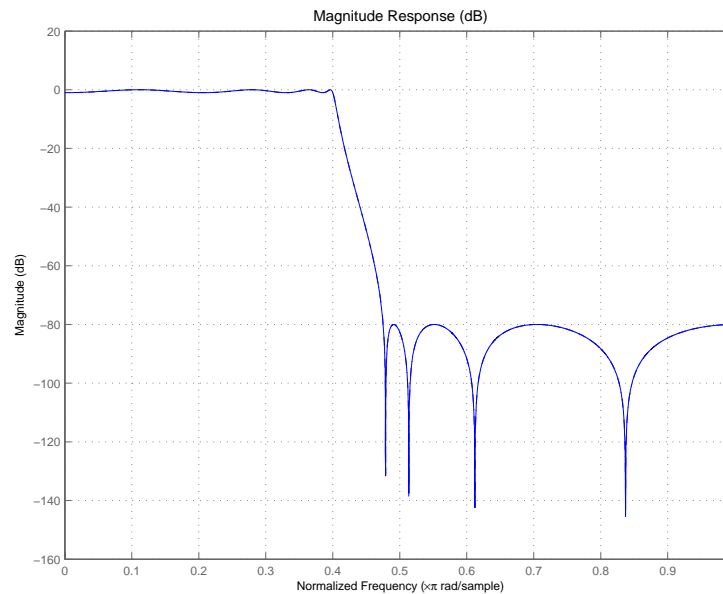
FilterStructure: 'Direct-Form II, Second-Order Sections'
  Arithmetic: 'double'
    sosMatrix: [4x6 double]

```

```
ScaleValues: [5x1 double]  
ResetBeforeFiltering: 'on'  
States: [2x4 double]
```

Convert hd to fixed-point and check the impulse response  
hd.arithmetic = 'fixed';

```
impz(hd)
```



**See Also**

`filter`

**Purpose** Information about filter objects

**Syntax**

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

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

`info` returns a variety of information about filters:

- 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. Filter measurement data is the same as the information returned by `measure` for the filter. Specific measurements appearing in the display depend on the filter and on the specifications you use when you construct the filter.
- Cost of implementing the filter in terms of operations required to apply the filter to data. Cost information is the same as you get from the `cost` method.

When the filter object uses fixed-point arithmetic (fixed-point `dfilt` objects or `mfilt` objects), `info` returns additional information about the filter, including the arithmetic setting and details about the filter internals.

You do not need to assign the output of `info` to a variable. Omitting the output value displays the same information in the Command Window.

### Adaptive Filters

`s = info(ha)` returns a string matrix with information about the filter `ha`.

Generally, `info` returns more information than the default display for the filter.

### Discrete-Time Filters

`s = info(hd)` returns a string matrix with information about the filter `hd`.

Generally, `info` returns more information than the default display for the filter.

## Multirate Filters

`s = info(hm)` returns a string matrix with information about the filter `hm`.

Generally, `info` returns more information than the default display for the filter.

## Examples

Given two filters—`hd` and `hm`, use `info` to learn more about each filter. Here is `hd`, a discrete-time direct-form FIR filter.

```
d = fdesign.lowpass('N,Fc,Ap,Ast',80,0.45,0.05,60);  
designmethods(d)
```

```
Design Methods for class fdesign.lowpass (N,Fc,Ap,Ast):
```

```
equiripple
```

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

```
hd =
```

```
    FilterStructure: 'Direct-Form FIR'  
      Arithmetic: 'double'  
      Numerator: [1x81 double]  
 PersistentMemory: false
```

Similarly, here is a multirate CIC filter `hm`. Note the differential delay value 2.

```
d1 = fdesign.interpolator(6,'cic',2,'fp,ast',0.40,60);  
designmethods(d1)
```

```
FIR Design Methods for class fdesign.interpolator (Fp,Ast):
```

```
multisection
```

```
hm=design(d1,'multisection')
```

```
hm =
```

```
    FilterStructure: 'Cascaded Integrator-Comb Interpolator'  
      Arithmetic: 'fixed'  
DifferentialDelay: 2  
  NumberOfSections: 5  
InterpolationFactor: 6  
  PersistentMemory: false  
  
    InputWordLength: 16  
    InputFracLength: 15  
  
    FilterInternals: 'FullPrecision'
```

Now use info to get more details about both filters.

```
s = info(hd)
```

```
Discrete-Time FIR Filter (real)
```

```
-----  
Filter Structure : Direct-Form FIR  
Filter Length   : 81  
Stable          : Yes  
Linear Phase    : Yes (Type 1)
```

```
Design Method Information  
Design Algorithm : equiripple
```

```
Design Options  
MinPhase       : false  
StopbandDecay  : 0  
StopbandShape  : flat
```

```
Design Specifications  
Sampling Frequency : N/A (normalized frequency)  
Response           : Lowpass  
Specification      : N,Fc,Ap,Ast  
FilterOrder       : 80
```

Fcutoff : 0.45  
Passband Ripple : 0.05 dB  
Stopband Atten. : 60 dB

#### Measurements

Sampling Frequency : N/A (normalized frequency)  
Passband Edge : 0.41517  
3-dB Point : 0.44091  
6-dB Point : 0.45  
Stopband Edge : 0.48968  
Passband Ripple : 0.05 dB  
Stopband Atten. : 60 dB  
Transition Width : 0.074506

#### Implementation Cost

Number of Multipliers : 81  
Number of Adders : 80  
Number of States : 80  
MultPerInputSample : 81  
AddPerInputSample : 80

s = info(hm)

#### Discrete-Time FIR Multirate Filter (real)

-----  
Filter Structure : Cascaded Integrator-Comb Interpolator  
Interpolation Factor : 6  
Differential Delay : 2  
Number of Sections : 5  
Stable : Yes  
Linear Phase : Yes (Type 2)

Input : s16,15  
Output : s32,15  
Filter Internals : Full Precision  
Integrator Section 1 : s17,15  
Integrator Section 2 : s18,15  
Integrator Section 3 : s19,15  
Integrator Section 4 : s20,15  
Integrator Section 5 : s21,15



```

Comb Section 1      : s21,15
Comb Section 2      : s24,15
Comb Section 3      : s27,15
Comb Section 4      : s29,15
Comb Section 5      : s32,15
    
```

```

Design Method Information
Design Algorithm : multisection
    
```

```

Design Specifications
Sampling Frequency : N/A (normalized frequency)
Response           : CIC
Specification      : Fp,Ast
MultirateType      : Interpolator
InterpolationFactor : 6
DifferentialDelay   : 2
Passband Edge      : 0.4
Stopband Atten.    : 60 dB
    
```

```

Measurements
Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.4
Stopband Edge      : -0.23333
Fnulls             : 0.16667  0.33333  0.5  0.66667  0.83333  1
Passband Ripple    : 87.0194 dB
Stopband Atten.    : 65.5304 dB
    
```

```

Implementation Cost
Number of Multipliers : 0
Number of Adders      : 10
Number of States      : 15
MultPerInputSample    : 0
AddPerInputSample     : 35
    
```

If you convert your filter object, such as a `dfilt` or `mfilt`, to a fixed-point filter, `info` returns more information about the ranges provided by the fixed-point formats in the filter. After converting `hd` to fixed arithmetic, `info` returns this display:

```

Discrete-Time FIR Filter (real)
-----
Filter Structure : Direct-Form FIR
Filter Length    : 81
    
```

Stable : Yes  
Linear Phase : Yes (Type 1)  
Arithmetic : fixed  
Numerator : s16,16 -> [-5.000000e-001 5.000000e-001)  
Input : s16,15 -> [-1 1)  
Filter Internals : Full Precision  
Output : s34,31 -> [-4 4) (auto determined)  
Product : s31,31 -> [-5.000000e-001 5.000000e-001) (auto determined)  
Accumulator : s34,31 -> [-4 4) (auto determined)  
Round Mode : No rounding  
Overflow Mode : No overflow

Design Method Information  
Design Algorithm : equiripple

Design Options  
MinPhase : false  
StopbandDecay : 0  
StopbandShape : flat

Design Specifications  
Sampling Frequency : N/A (normalized frequency)  
Response : Lowpass  
Specification : N,Fc,Ap,Ast  
FilterOrder : 80  
Fcutoff : 0.45  
Passband Ripple : 0.05 dB  
Stopband Atten. : 60 dB

Measurements  
Sampling Frequency : N/A (normalized frequency)  
Passband Edge : 0.41517  
3-dB Point : 0.44091  
6-dB Point : 0.45  
Stopband Edge : 0.48962  
Passband Ripple : 0.05 dB  
Stopband Atten. : 60 dB  
Transition Width : 0.07445

Implementation Cost  
Number of Multipliers : 81  
Number of Adders : 80  
Number of States : 80  
MultPerInputSample : 81  
AddPerInputSample : 80

**See Also**

`coefficients`, `isfir`, `isstable`, `islinphase`

`dfilt` in the Signal Processing Toolbox documentation

# int

---

**Purpose** Return states from CIC filter as signed integer matrix containing the numerator and denominator states for all filter sections

**Syntax** `integerstates = int(hm.states)`

**Description** `integerstates = int(hm.states)` returns the states of a CIC filter in matrix form, rather than as the native `filtstates` object. An important point about `int` is that it quantizes the state values to the smallest number of bits possible while maintaining the values accurately.

**Examples** For many users, the states of multirate filters are most useful as a matrix, but the CIC filters store the states as objects. Here is how you get the states from you CIC filter as a matrix.

```
hm = mfilt.cicdecim;
hs = hm.states; % Returns a FILTSTATES.CIC object hs.
states = int(hs); % Convert object hs to a signed integer matrix.
```

After using `int` to convert the states object to a matrix, here is what you get.

Before converting:

```
hm.states

ans =

    Integrator: [2x1 States]
    Comb: [2x1 States]
```

After the conversion and assigning the states to `states`:

```
states

states =

     0     0
     0     0
```

**See Also** `filtstates.cic`, `mfilt.cicdecim`, `mfilt.cicinterp`

**Purpose** Determine whether filters are allpass structures

**Syntax** `isallpass(hd)`  
`isallpass(hd,tolerance)`

**Description** `isallpass(hd)` determines whether the filter object `hd` is an allpass filter, returning 1 if true and 0 if false.

`isallpass(hd,tolerance)` uses input argument `tolerance` to determine whether the numerator and denominator transfer functions for the filter are close enough in value to be considered equal, and thus allpass, returning 1 if true (the difference between the numbers is less than `tolerance`) and 0 if not.

Given an allpass filter with this transfer function

$$H(z) = \frac{a_n + \dots + a_1 z^{-(n-1)} + z^{-n}}{1 + a_1 z^{-1} + \dots + a_n z^{-n}}$$

if the numerator and denominator transfer functions are equal, the filter is allpass. The `tolerance` input argument lets you determine how closely the transfer functions have to match to be considered equal. This might be most helpful in fixed-point allpass filters.

Lattice coupled allpass filters always have allpass sections, this function always returns 1 for filters whose structure is `latticeca`.

**Examples** Use `dfilt.allpass` to construct an allpass filter and test whether the filter is allpass.

```
c=[.8,1.5,0.4, 0.7]; % Allpass coefficients.
hd=dfilt.allpass(c)
```

```
hd =
```

```
    FilterStructure: 'Minimum-Multiplier Allpass'
AllpassCoefficients: [.8,1.5,0.4, 0.7]
    PersistentMemory: false
                States: [0;0;0;0;0;0;0]
    NumSamplesProcessed: 0
```

# isallpass

---

```
isallpass(hd)
```

```
ans =
```

```
1
```

## See Also

isfir, islinphase, ismaxphase, isminphase, isreal, issos, isstable

---

<b>Purpose</b>	Determine whether filters are FIR filters
<b>Syntax</b>	<code>isfir(h)</code>
<b>Description</b>	<p><code>isfir(h)</code> determines whether filter <code>h</code> is an FIR filter, returning 1 when the filter is an FIR filter, and 0 when it is IIR. <code>isfir</code> applies to <code>dfilt</code>, <code>mfilt</code>, and <code>adaptfilt</code> objects.</p> <p>To determine whether <code>h</code> is an FIR filter, <code>isfir(h)</code> inspects filter <code>h</code> and determines whether the filter, in transfer function form, has a scalar denominator. If it does, it is an FIR filter.</p>
<b>Examples</b>	<pre>d = fdesign.lowpass; h = design(d); isfir(h) ans =      1</pre> <p>returns 1 for the status of filter <code>h</code>; the filter is an FIR structure with denominator reference coefficient equal to 1.</p> <p>For multirate filters, <code>isfir</code> works the same way.<pre>d = fdesign.interpolator(5); % Interpolate by 5. h = design(d); % Use the default design method. isfir(h)  ans =      1</pre><p>Use <code>isfir</code> with adaptive filters as well.</p></p>
<b>See Also</b>	<code>isallpass</code> , <code>islinphase</code> , <code>ismaxphase</code> , <code>isminphase</code> , <code>isreal</code> , <code>issos</code> , <code>isstable</code>

# islinphase

---

**Purpose** Determine whether filters are linear phase

**Syntax** `islinphase(h)`  
`islinphase(h,tolerance)`

**Description** `islinphase(h)` determines if the filter object `h` is linear phase, and returns 1 if true and 0 if false. `adapfilt`, `dfilt`, and `mfilt` objects work with `islinphase`.

`islinphase(h,tolerance)` uses input argument `tolerance` to determine whether the filter coefficients are close enough in value to be considered symmetric or antisymmetric, returning 1 if true (the difference between the values is less than `tolerance`) and 0 if not.

The phase determination is based on the reference coefficients. A filter has linear phase if it is FIR and its transfer function coefficients are symmetric or antisymmetric. If it is IIR and it has poles on or outside the unit circle and both numerator and denominator are symmetric or antisymmetric, it is linear phase also.

**Examples** This IIR filter has linear phase.

```
d = fdesign.lowpass('n,fc',10,0.55);
h = design(d,'window');
islinphase(h)
ans =

     1
```

Using the specification `nb,na,fp,fst` results in an IIR filter that is not linear phase in this design.

```
nb=15
na=10
d=fdesign.lowpass('nb,na,fp,fst',nb,na,0.45,0.55)

d =

    Response: 'Lowpass'
  Specification: 'Nb,Na,Fp,Fst'
    Description: {4x1 cell}
  NormalizedFrequency: true
```



```
NumOrder: 15
DenOrder: 10
Fpass: 0.45
Fstop: 0.55
```

```
h=design(d) % Use the default design method iirlpnorm.
```

```
h =
```

```
FilterStructure: 'Direct-Form II, Second-Order Sections'
Arithmetic: 'double'
sosMatrix: [8x6 double]
ScaleValues: [-0.0051749857036492;1;1;1;1;1;1;1;1]
PersistentMemory: false
```

```
islinphase(h)
```

```
ans =
```

```
0
```

## See Also

`isallpass`, `isfir`, `ismaxphase`, `isminphase`, `isreal`, `issos`, `isstable`

# ismaxphase

---

**Purpose** Determine whether filters are maximum phase

**Syntax** `ismaxphase(h)`  
`ismaxphase(h,tolerance)`

**Description** `ismaxphase(h)` determines if the filter object `h` is maximum phase, and returns 1 if true and 0 if false. `adapfilt`, `dfilt`, and `mfilt` objects work with `ismaxphase`.

`ismaxphase(h,tolerance)` uses input argument `tolerance` to determine whether the zeros of the filter transfer function have values that are close enough to 1 to be considered 1 or greater (on or outside the unit circle, returning 1 if true (the difference between the coefficient value and 1 is less than `tolerance`) and 0 if not.

The phase determination is based on the reference coefficients. A filter is maximum phase when the zeros of its transfer function are on or outside the unit circle, or when the numerator is a scalar.

**Examples** Two examples show `ismaxphase` in use. The first is a discrete-time `dfilt` object and the second an adaptive filter.

```
fp = 100;
fst= 120;
fs = 800;
ap = 1;
ast= 80;
d = fdesign.lowpass('fp,fst,ap,ast',fp,fst,ap,ast,fs);
h = design(d,'equiripple','minphase',true);

isminphase(h)

ans =

     1
```

To make this a maximum phase filter, use `fliplr` to change the coefficient order. Reordering the coefficients this way changes the phase from minimum to maximum.

```
h.numerator=fliplr(h.numerator);
```

```
ismaxphase(h)
```

```
ans =
```

```
1
```

returns 1 so this is a maximum phase filter. Compare to `isminphase`.

For the adaptive filter example, try the following code:

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

After adapting, `ha` is an FIR filter that does not exhibit maximum phase.

```
ismaxphase(ha)
```

```
ans =
```

```
0
```

## See Also

`isallpass`, `isfir`, `islinphase`, `isminphase`, `isreal`, `issos`, `isstable`

# isminphase

---

**Purpose** Determine whether filters are minimum phase

**Syntax** `isminphase(h)`  
`isminphase(h,tolerance)`

**Description** `isminphase(h)` determines if the filter object `h` is maximum phase, and returns 1 if true and 0 if false. `adapfilt`, `dfilt`, and `mfilt` objects work with `isminphase`.

`isminphase(h,tolerance)` uses input argument `tolerance` to determine whether the values of the filter transfer function zeros are close enough to 1 to be considered to be on the unit circle, returning 1 if true (the difference between the transfer function zero values and 1 is less than `tolerance`) and 0 if not.

The determination is based on the reference coefficients. A filter is minimum phase when the zeros of its transfer function are on or inside the unit circle, or the numerator is a scalar.

**Examples** This example creates a minimum-phase filter.

```
fp = 200;  
fst= 230;  
fs = 900;  
ap = 1;  
ast= 80;  
d = fdesign.lowpass('fp,fst,ap,ast',fp,fst,ap,ast,fs);  
h = design(d,'equiripple','minphase',true);  
isminphase(h)
```

```
ans =
```

```
1
```

When you make `h` a fixed-point filter, the quantization process results in the filter no longer being minimum phase.

```
h.arithmetic='fixed';  
isminphase(h)
```

```
ans =
```

```
0
```

**See Also**

isallpass, isfir, islinphase, ismaxphase, isreal, issos, isstable,

# isreal

---

**Purpose** Determine whether discrete-time filters use purely real coefficients

**Syntax** `isreal(hd)`

**Description** `isreal(hd)` returns 1 (or true) if all filter coefficients for the filter `hd` are real, and returns 0 (or false) otherwise.

`isreal(hd)` returns 1 if all filter coefficients in filter `hd` have zero imaginary part. Otherwise, `isreal(hd)` returns a 0 indicating that the filter is complex. Complex filters have one or more coefficients with nonzero imaginary parts.

---

**Note** Quantizing a filter cannot make a real filter into a complex filter.

---

**Examples** To demonstrate the `isreal` test, this example creates a double-precision filter and fixed-point filter, and tests the coefficients of the fixed-point filter to see if they are strictly real.

```
d=fdesign.lowpass('n,fp,ap,ast',5,0.4,0.5,20);
hd=design(d,'ellip')

hd =

    FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
      sosMatrix: [3x6 double]
      ScaleValues: [0.362583368859661;0.918321077151039;0.496533475964919;1]
 PersistentMemory: false

hq=design(d,'ellip'); % Use d to design the fixed-point filter.
hq.arithmetic='fixed'; % Convert to fixed-point arithmetic.
isreal(hq)

ans =

     1
```

**See Also** `isfir`, `islinphase`, `ismaxphase`, `isminphase`, `issos`, `isstable`, `isallpass`

**Purpose** Determine whether discrete-time filters are composed of second-order sections

**Syntax** `issos(hd)`

**Description** `issos(hd)` determines whether quantized filter `hq` consists of second-order sections. Returns 1 if all sections of quantized filter `hq` have order less than or equal to two, and 0 otherwise.

**Examples** By default, `fdesign` and `design` return SOS filters when possible. This example designs a lowpass SOS filter that uses fixed-point arithmetic.

```
d=fdesign.lowpass('n,fp,ap,ast',40,0.55,0.1,60)
```

```
d =
```

```

           Response: 'Lowpass'
    Specification: 'N,Fp,Ap,Ast'
      Description: {4x1 cell}
NormalizedFrequency: true
      FilterOrder: 40
           Fpass: 0.55
           Apass: 0.1
           Astop: 60

```

```
designmethods(d)
```

```
Design Methods for class fdesign.lowpass (N,Fp,Ap,Ast):
```

```

    ellip
    equiripple

```

```

hd=design(d,'ellip');
hd.arithmetic='fixed';

```

```
issos(hd)
```

# issos

---

ans =

1

Fixed-point filter `hd` is in second-order section form, as is the double-precision version.

## See Also

`isallpass`, `isfir`, `islinphase`, `ismaxphase`, `isminphase`, `isreal`, `isstable`



**Purpose** Determine whether discrete-time filter is stable

**Syntax** `isstable(hd)`

**Description** `isstable(hq)` tests quantized filter `hq` to determine whether its poles are inside the unit circle. If the poles lie on or outside the circle, `isstable` returns 0. If the poles are inside the circle, `isstable` returns 1.

To determine the filter stability, `isstable` checks the filter coefficients. When the poles lie on or inside the unit circle, the filter is stable. FIR filters are stable by design since the defining transfer functions do not have denominator polynomials, thus no feedback to cause instability.

**Examples** Since filter stability is very important in your design process, use `isstable` to determine whether your double-precision or fixed-point IIR filter is stable.

```
d=fdesign.nyquist(2,'n,tw',24,0.1);
hd=design(d,'iirlinphase')

hd =

    FilterStructure: Cascade
           Stage(1): Scalar
           Stage(2): Parallel
                   Stage(1): Delay
                   Stage(2): Cascade
                           Stage(1): Delay
                           Stage(2): Cascade
    PersistentMemory: false

isstable(hd)

ans =

     1

hd2=design(d,'equiripple');
isstable(hd2)
```

# isstable

---

```
ans =
```

```
1
```

## See Also

isallpass, isfir, islinphase, ismaxphase, isminphase, isreal, issos, zplane

<b>Purpose</b>	Design discrete-time or multirate filter from filter specification object and Kaiser window
<b>Syntax</b>	<pre>h = design(d, 'kaiserwin') h = design(d, 'kaiserwin', designoption, value, designoption, ... value, ...)</pre>
<b>Description</b>	<p><code>h = design(d, 'kaiserwin')</code> designs a digital filter <code>hd</code>, or a multirate filter <code>hm</code> that uses a Kaiser window. For <code>kaiserwin</code> 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. <code>kaiserwin</code> returns a warning when your filter order may be too low to design your filter accurately.</p> <p><code>h = design(d, 'kaiserwin', designoption, value, designoption, ... value, ...)</code> returns a filter where you specify design options as input arguments and the design process uses the Kaiser window technique.</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>kaiserwin</code>, refer to the command line help system. For example, to get specific information about using <code>kaiserwin</code> with <code>d</code>, the specification object, enter the following at the MATLAB prompt.</p> <pre>help(d, 'kaiserwin')</pre>
<b>Examples</b>	<p>This example designs a direct form FIR filter from a halfband filter specification object.</p> <pre>d=fdesign.halfband('n,tw',100,0.004)  d =      Response: 'Halfband with filter order and transition width'   Specification: 'N,TW'   Description: {2x1 cell} NormalizedFrequency: true                 Fs: 'Normalized'       FilterOrder: 100   TransitionWidth: 0.0040</pre>

```
designopts(d,'kaiserwin')

ans =

    FilterStructure: 'dffir'

hd= design(d,'kaiserwin','filterstructure','dffir')
Warning: Filter order is too low. Design may be inaccurate.

hd =

    FilterStructure: 'Direct-Form FIR'
    Arithmetic: 'double'
    Numerator: [1x101 double]
    ResetBeforeFiltering: 'on'
    States: [100x1 double]
```

In this example, `kaiserwin` uses an interpolating filter specification object to implement a multirate filter.

```
d=fdesign.interp(4,'pl,tw',120,0.004)

d =

    Response: [1x46 char]
    Specification: 'PL,TW'
    Description: {2x1 cell}
    InterpolationFactor: 4
    NormalizedFrequency: true
    Fs: 'Normalized'
    PolyphaseLength: 120
    TransitionWidth: 0.0040

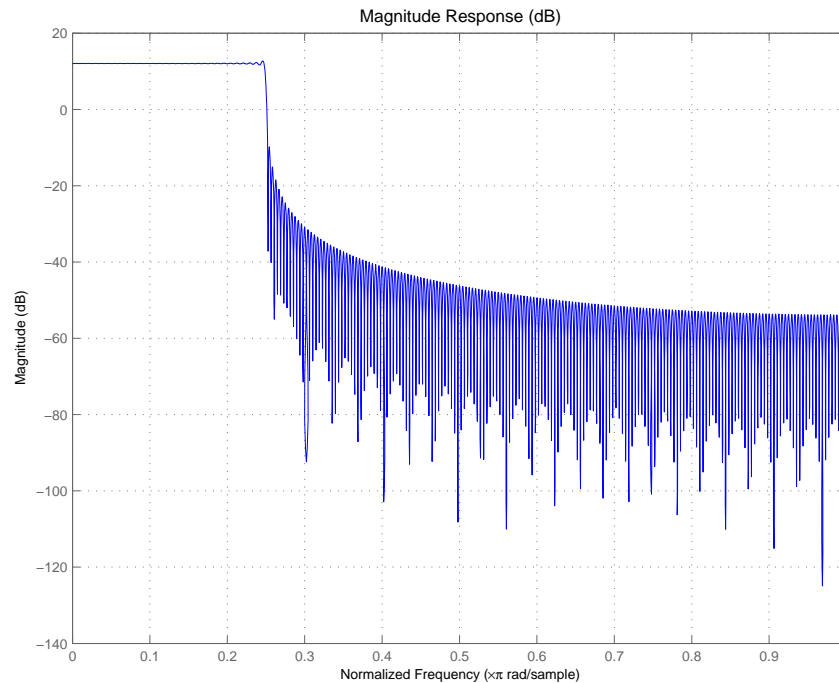
hm = design(d,'kaiserwin')

hm =

    FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
    Numerator: [1x480 double]
    InterpolationFactor: 4
    ResetBeforeFiltering: 'on'
    States: [119x1 double]
```

With the polyphase length of 120 you do not see the warning about the filter accuracy. Increasing the transition width `tw` can also reduce the possible inaccuracies.

FVTool shows clearly the multirate filter `hm`.



## See Also

`equiripple`, `firls`

# limitcycle

---

**Purpose** Explore steady-state response of single-rate, fixed-point IIR filter to zero-valued input

**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>• Granular—indicates that a granular overflow occurred.</li><li>• Overflow—indicates that an overflow limit cycle occurred.</li><li>• None—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)
```

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:

# limitcycle

---

- `ntrials` — Number of Monte Carlo trials (default is 20).
- `inputlengthfactor` — integer factor used to calculate the length of the input vector. The length of the input vector comes from `(impzlength(hd) * inputlengthfactor)`, where `inputlengthfactor = 2` is the default value.
- `stopcriterion` — the criterion for stopping the Monte Carlo trial processing. `stopcriterion` can be set to **either** (the default), **granular**, **overflow**. This table describes the results of each stop criterion.

stopcriterion Setting	Description
<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

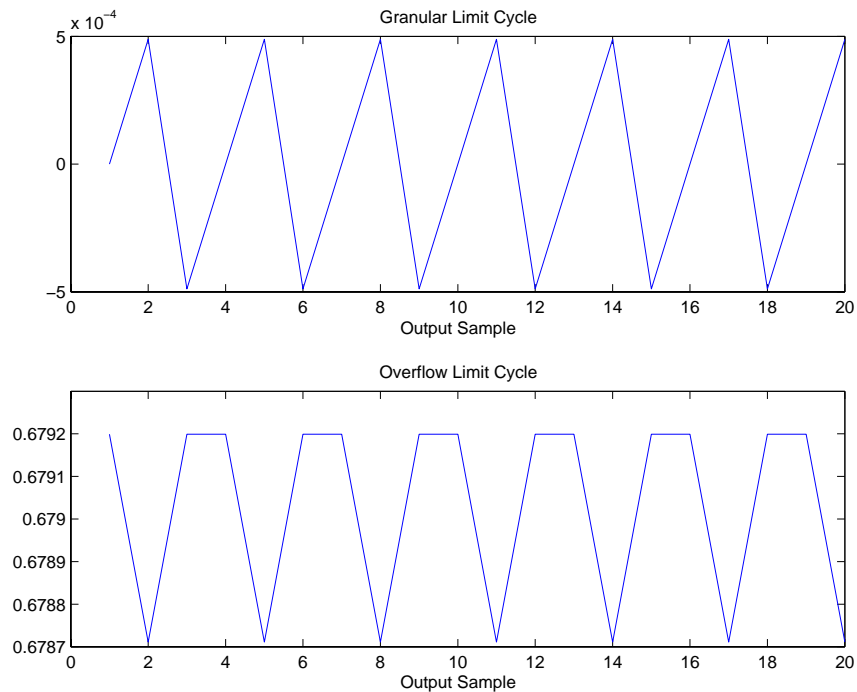
oreport =

    LimitCycle: 'overflow'
             Zi: [2x1 double]
             Output: [1303x1 embedded.fi]
             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.

# limitcycle

---



As you see from the plots, and as is generally true, overflow limit cycles are much greater magnitude than granular limit cycles. This is why `limitcycle` favors overflow limit cycle detection and reporting.

## See Also

`freqz`, `noisepsd`

**Purpose** Maximum step size that allows 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  $0 < \text{ha.stepsize} < \text{mumaxmse}/2$ .

---

**Note** `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`

---

For `adaptfilt.nlms` filter objects, `maxstep` uses a 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);
```

## maxstep

---

```
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  $0 < \mu < \text{mumaxmse}/2$ :  
Erratic behavior might result.

mumax

mumax =

0.0623

mumaxmse

mumaxmse =

0.0530

### See Also

msepred, msesim, filter

**Purpose** Magnitude response measurement for discrete-time and multirate filter created from filter specification object

**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.
6-dB Point	Location of the -6 dB point on the response curve.

## measure

<b>Lowpass Filter Specification</b>	<b>Description</b>
Stopband Edge	Location of the edge of the transition band as it enters the stopband.
Passband Ripple	Ripple in the passband.
Seopband Atten.	Attenuation in the stopband.
Transition Width	Width of the transition between the pass- 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.
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.

<b>Bandstop Filter Specification</b>	<b>Description</b>
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.

<b>Interpolator Filter Specification</b>	<b>Description</b>
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.

## measure

Interpolator Filter Specification	Description
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 pass- and stopband, in normalized frequency or absolute frequency. Measured between Fpass and Fstop.

For reference, this is the specification object `d` that created the interpolator specifications shown in the preceding table.

```
d=fdesign.interpolator(6,'lowpass')

d =

    MultirateType: 'Interpolator'
  InterpolationFactor: 6
         Response: 'Lowpass'
  Specification: 'Fp,Fst,Ap,Ast'
   Description: {4x1 cell}
NormalizedFrequency: true
         Fpass: 0.13333333333333333
         Fstop: 0.16666666666666667
         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'  
        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
```

## measure

---

```
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`

# mfilt

---

**Purpose** Construct multirate filter object

**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, meaning 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.

<b>Filter Structure String</b>	<b>Description of Resulting Multirate Filter</b>
<code>mfilt.cascade</code>	Cascade multirate filters to form another filter
<code>mfilt.cicdecim</code>	Cascaded integrator-comb decimator
<code>mfilt.cicinterp</code>	Cascaded integrator-comb interpolator
<code>mfilt.fftfirinterp</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>	Direct-form FIR polyphase interpolator
<code>mfilt.firsrc</code>	Direct-form FIR polyphase sample rate converter

Filter Structure String	Description of Resulting Multirate Filter
<code>mfilt.firtdecim</code>	Direct-form transposed FIR polyphase decimator
<code>mfilt.holdinterp</code>	FIR hold interpolator
<code>mfilt.iirdecim</code>	IIR decimator
<code>mfilt.iirinterp</code>	IIR interpolator
<code>mfilt.linearinterp</code>	FIR Linear interpolator
<code>mfilt.iirwdfdecim</code>	IIR wave digital filter decimator
<code>mfilt.iirwdfinterp</code>	IIR wave digital filter interpolator

### Copying mfilt Objects

To create a copy of an `mfilt` object, use the `copy` method.

```
h2 = copy(hd)
```

---

**Note** The syntax `hd2 = hd` copies only the object handle. It does not create a new object. `hd2` and `hd` are not independent. If you change the property value for one of the two, such as `hd2`, you are changing the property for both.

---

### Examples

Create an FIR decimator that uses a decimation factor equal to three. In this case, the only input argument needed is `m`, the decimation factor. Other input arguments are available to you—refer to the reference page for the structure that interests you for more information.

```
m=3;
```

```
hm=mfilt.firdecim(m)
```

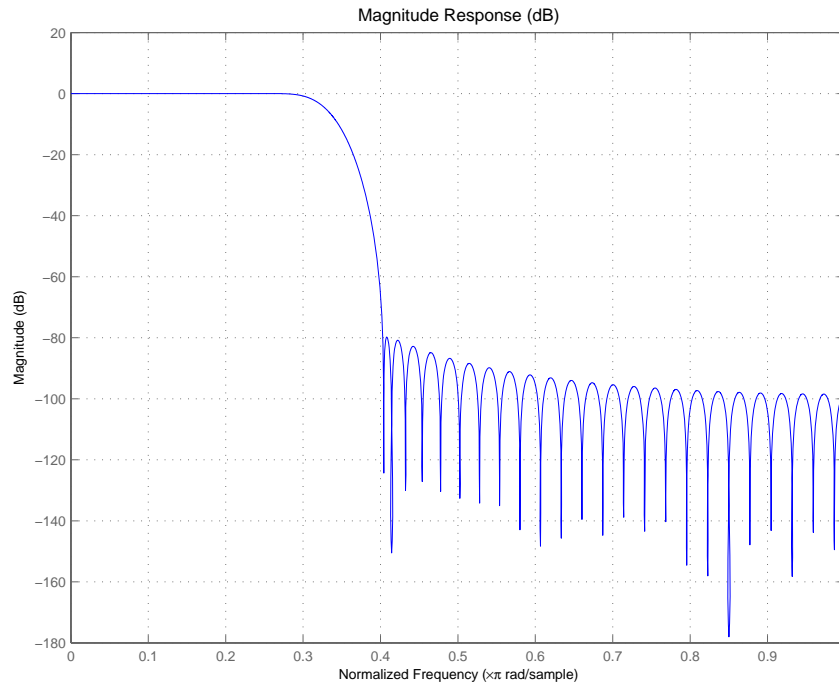
```
hm =
```

```
FilterStructure: 'Direct-Form FIR Polyphase Decimator'
      Numerator: [1x73 double]
      DecimationFactor: 3
```

```
NumberOfSamplesProcessed: 0
ResetStates: 'on'
States: [72x1 double]
```

To demonstrate a few of the methods that apply to multirate filters, here are two examples of using `hm`, your FIR decimator.

Use the Filter Visualization tool to review the magnitude response of your decimator.



Now check to see if your filter is stable.

```
isstable(hm)
```

```
ans =
```

```
1
```

Finally, pass a signal through the filter to see if it indeed decimates by three.

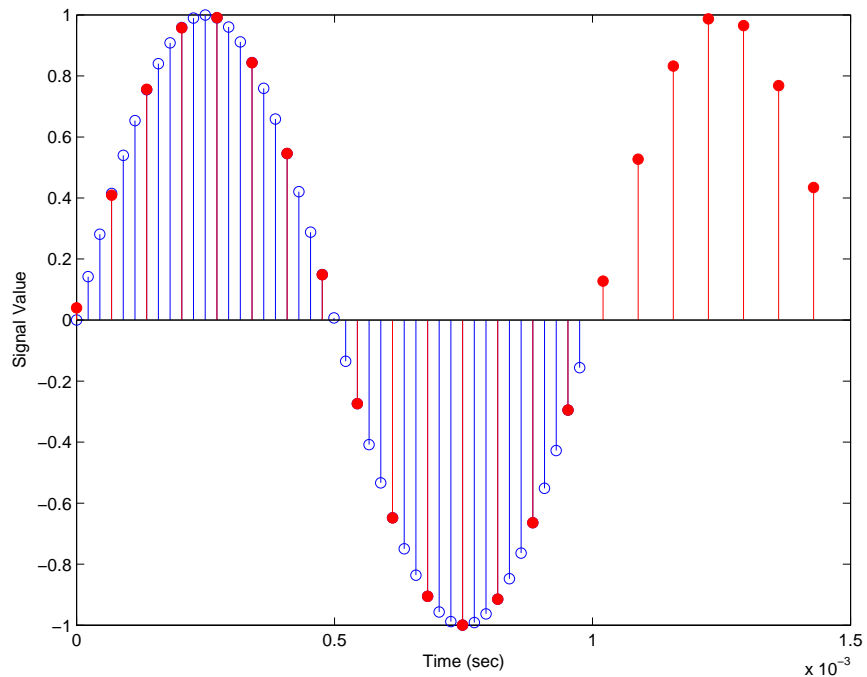
```

m = 3; % Decimation factor
hm = mfiltr.firdecim(m); % We use the default filter
fs = 44.1e3; % Original sample freq: 44.1kHz.
n = 0:10239; % 10240 samples, 0.232 second long
% signal

x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz
y = filter(hm,x); % 5120 samples, still 0.232 seconds
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1kHz
hold on % Plot decimated signal (22.05kHz) in red
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')

```

Here is the stem plot that shows the result of the decimation process.



hm =

```
    FilterStructure: 'Direct-Form FIR Polyphase Decimator'  
      Numerator: [1x73 double]  
DecimationFactor: 3  
PersistentMemory: 'on'  
      States: [72x1 double]
```

The filter processes 10239 samples with 1 unprocessed sample whose value is 0.8963. One nonprocessed sample results from dividing the number of samples, 10240, by the decimation factor, 3, to get 3413 output samples and one left over.

## See Also

`mfilt.firfracdecim`, `mfilt.firfracinterp`, `mfilt.firinterp`,  
`mfilt.firsrc`, `mfilt.firtdecim`

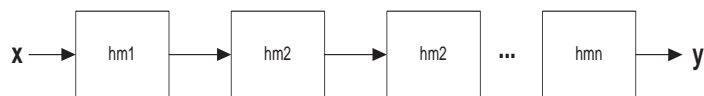


**Purpose** Cascade `dfilt` and `mfilt` object(s) into filter

**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`:



**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 your Signal Processing Toolbox documentation

# mfilt.cicdecim

---

**Purpose** Construct fixed-point cascaded integrator-comb (CIC) decimator filter object

**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. When you omit one or more input options, the object applies default values for the omitted input argument as shown in the next table.

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 <math>2*n</math>, where <math>n</math> 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 8-859 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 8-859.

## Properties of the Object

Objects have properties that control the way the object behaves. This table lists all the properties for the filter, with a description of each.

Name	Values	Default	Description
Arithmetic	fixed	fixed	Reports the kind of arithmetic the filter uses. CIC decimators are always fixed-point filters.
DecimationFactor	Any positive integer	2	Amount to reduce the input sampling rate.
DifferentialDelay	Any integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.
FilterStructure	mfilt structure string	None	Reports the type of filter object. You cannot set this property—it is always read only and results from your choice of mfilt objects.
FilterInternals	FullPrecision, MinWordLengths, SpecifyPrecision, SpecifyWordLengths	FullPrecision	Set the usage mode for the filter. Refer to “Usage Modes” below for details.

Name	Values	Default	Description
InputFracLength	Any positive integer	15	The number of bits applied to the fraction length to interpret the input data to the filter.
InputOffset	0 -> r.	0	Indicates the length of the output signal given the length of the input signal. InputOffset starts at zero and cycles through the phases as follows for each input sample: 0->r->(r-1)->(r-2)->(r-p)->0, where $p = r-1$ .
InputWordLength	Any positive integer	16	The number of bits applied to the word length to interpret the input data to the filter.
NumberOfSections	Any positive integer	2	Number of sections used in the decimator. Generally called n. Reflects either the number of decimator or comb sections, not the total number of sections in the filter.
OutputFracLength	Any positive integer	15	The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.

## mfilt.cicdecim

Name	Values	Default	Description
OutputWordLength	Any positive integer	16	The number of bits applied to the word length to interpret the output data from the filter.
PersistentMemory	false or true	false	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When PersistentMemory is false, you cannot access the filter states. Setting PersistentMemory to true reveals the States property so you can modify the filter states.

Name	Values	Default	Description
SectionWordLengths	Any integer or a vector of length $2*n$ .	16	<p>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 <math>2*n</math>, where <math>n</math> 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.</p>

# mfilt.cicdecim

Name	Values	Default	Description
SectionWordLengthMode	MinWordLengths or SpecifyWordLengths	MinWordLength	Determines whether the filter object sets the section word lengths or you provide the word lengths explicitly. By default, the filter uses the input and output word lengths in the command to determine the optimal word lengths for each section, according to the information in [1]. When you choose SpecifyWordLengths, you provide the word length for each section. In addition, choosing SpecifyWordLengths exposes the SectionWordLengths property for you to modify as needed.



Name	Values	Default	Description
States	filtstates.cic object	m+1-by-n matrix of zeros, after you call function int.	Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. m is the differential delay of the comb section and n is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when PersistentMemory is true. Refer to the filtstates object in the Signal Processing Toolbox 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  $B_{\text{accum}}$  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  $B_{\text{accum}}$  as defined by

$$B_{\text{accum}} = \text{ceil}(N_{\text{secs}}(\text{Log}_2(D \times M)) + \text{InputWordLength})$$

where  $N_{\text{secs}}$  is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display looks for this mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'  
Arithmetic: 'fixed'  
DifferentialDelay: 1  
NumberOfSections: 2  
DecimationFactor: 4  
PersistentMemory: false  
  
InputWordLength: 16  
InputFracLength: 15  
  
FilterInternals: 'FullPrecision'
```

## Minimum Wordlengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than  $B_{\text{accum}}$ , 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  $B_{\text{accum}}$  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  $B_{\text{accum}}$  to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the

fraction length of that section by one bit compared to the input fraction length. Setting the output wordlength for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false
```

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

```
FilterInternals: 'MinWordLengths'
```

```
OutputWordLength: 16
```

### Specify word lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length  $2*(NumberOfSections)$ , where each vector element represents the word length for a section. If you specify a scalar, such as  $B_{accum}$ , the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display looks like for the SpecifyWordLengths mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'  
Arithmetic: 'fixed'  
DifferentialDelay: 1  
NumberOfSections: 2  
DecimationFactor: 4  
PersistentMemory: false
```

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

```
FilterInternals: 'SpecifyWordLengths'
```

```
SectionWordLengths: [19 18 18 17]
```

```
OutputWordLength: 16
```

## Specify precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the SpecifyPrecision mode, you must enter a vector of length  $2 \times (\text{NumberOfSections})$  with elements that represent the word length for each section. When you enter a scalar such as  $B_{\text{accum}}$ , `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  $2 \times (\text{NumberOfSections})$  with elements that represent the fraction length for each section as well. When you enter a scalar such as  $B_{\text{accum}}$ , `mfilt.cicdecim` applies scalar expansion as done for the word lengths.

Here is the SpecifyPrecision display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'  
Arithmetic: 'fixed'  
DifferentialDelay: 1
```

```

NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11

```

### About the States of the Filter

In the `states` property you find the states for both the integrator and comb portions of the filter. `states` is a matrix of dimensions  $m+1$ -by- $n$ , with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.
- States for the comb portion fill the remaining rows in the state matrix..

To review the states of a CIC filter, use `int` to assign the states to a variable in MATLAB. As an example, here are the states for a CIC decimator `hm` before and after filtering a data set.

```

x = fi(ones(1,10),true,16,0); % Fixed-point input data.
hm = mfilt.cicdecim(2,1,2,16,16,16);
sts=int(hm.states)

sts =

     0     0
     0     0

set(hm,'InputFracLength',0); % Integer input specified.
y=filter(hm,x)

```

```
sts=int(hm.states)

sts =

    10    45
    28    13
```

STS is an integer matrix that `int` returns from the contents of the `filtstates.cic` object in ```.

## 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

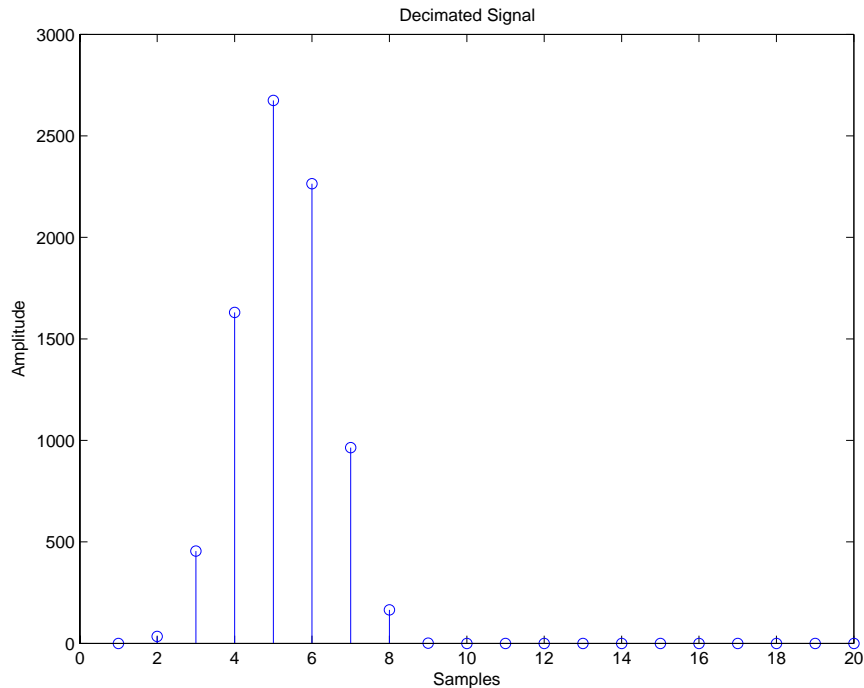
This example applies a decimation factor  $r$  equal to 8 to a 160-point impulse signal. The signal output from the filter has  $160/r$ , or 20, points or samples. Choosing 10 bits for the word length represents a fairly common setting for analog to digital converters. The plot shown after the code presents 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 the output as ...
                % a stem plot.
xlabel('Samples'); ylabel('Amplitude');
title('Decimated Signal');

```



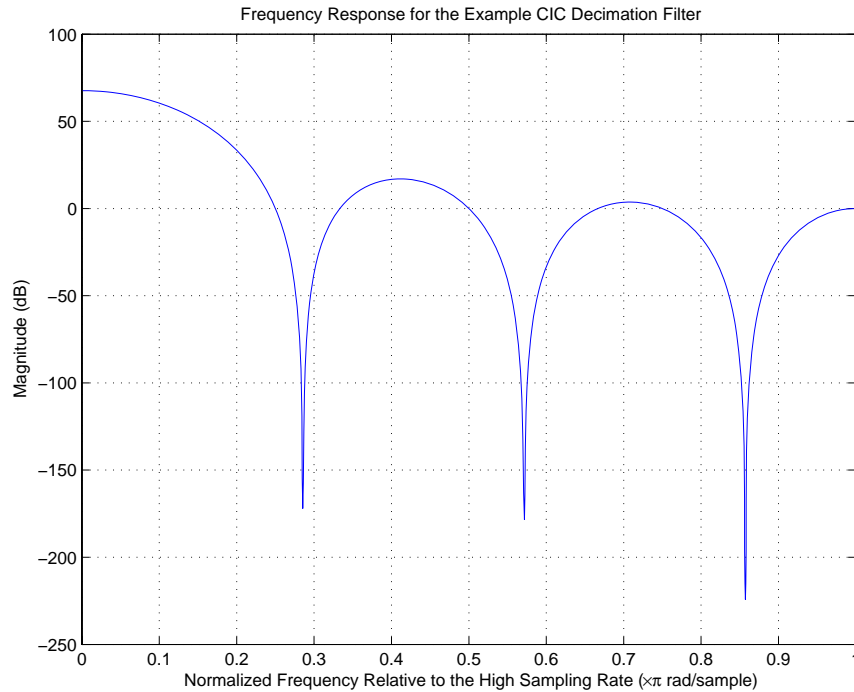
The next example demonstrates one way to compute the filter frequency response, using a 4-section decimation filter with the decimation factor set to 7:

```

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.

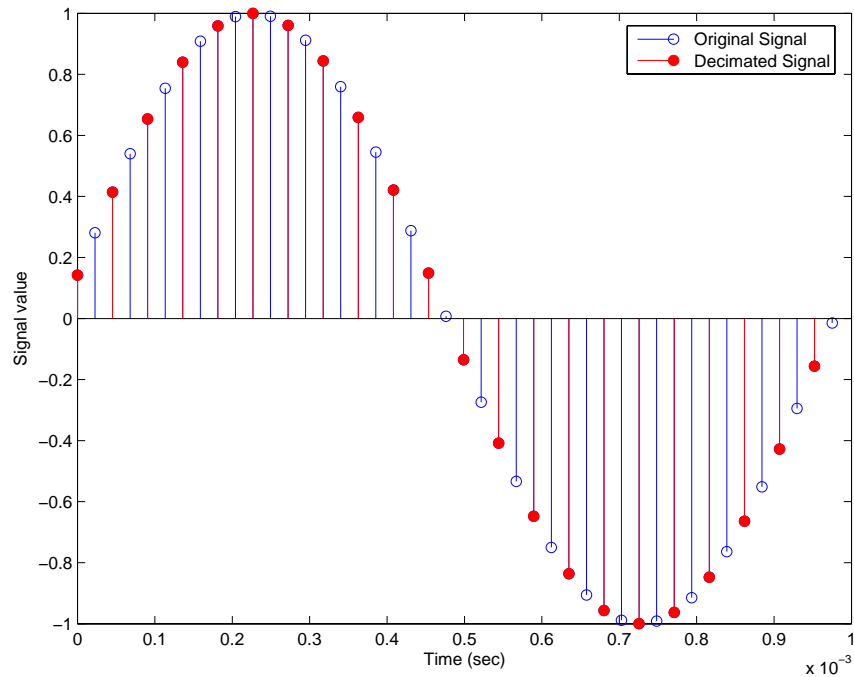
% Scale the output to overlay the stem plots.
x = double(x);
y = double(y_fi);
```



```

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');

```

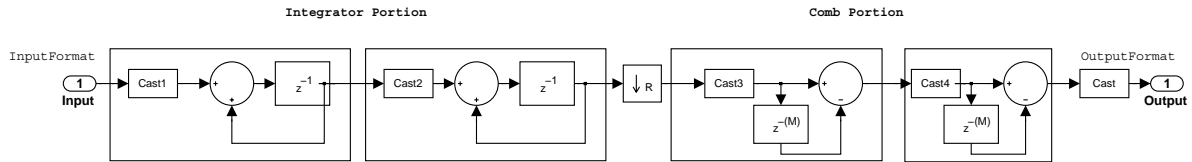


### Algorithm

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



`mfilt.cicdecim` calculates the fraction length at each section of the decimator to avoid overflows at the output of the filter.

## See Also

`mfilt`, `mfilt.cicinterp`

## References

- [1] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(2): pp. 155-162, 1981
- [2] Meyer-Baese, Uwe, "Hogenauer CIC Filters," in *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, pp. 155-172
- [3] harris, fredric j, *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004 , pp. 343

**Purpose** Construct fixed-point cascaded integrator-comb (CIC) interpolator filter object

**Syntax** `hm = mfilt.cicinterp(r,m,n,ilw,owl,wlps)`

**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. When you omit one or more input options, the omitted option applies default values shown in the table below.

The following table describes the input arguments for creating `hm`.

Input Arguments	Description
<code>r</code>	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.
<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 as values for <code>m</code> . 1 is the default.
<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. By default, the filter has two sections.
<code>ilw</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 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>

## 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 nondecreasing. 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

$$W_N = B_{in} + N - 1 \quad \text{if } M = 1$$

The conversions denoted by the cast blocks in the integrator diagrams in “Algorithm” on page 8-874 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints described in this section, `mfilt.cicinterp` returns an error.

The fraction lengths and scalings of the filter sections do not change. At each section the word length is either staying the same or increasing. The signal scaling can change at the output after the final filter section if you choose the output word length to be less than the word length of the final filter section.

### Properties of the Object

Objects have properties that control the way the object behaves. This table lists all the properties for the filter, with a description of each.

Name	Values	Default	Description
Arithmetic	fixed	fixed	Reports the kind of arithmetic the filter uses. CIC interpolators are always fixed-point filters.
InterpolationFactor	Any positive integer	2	Amount to increase the input sampling rate.
DifferentialDelay	Any integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.

# mfilt.cicinterp

Name	Values	Default	Description
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 <code>mfilt</code> objects.
FilterInternals	FullPrecision, MinWordLengths, SpecifyPrecision, SpecifyWordLengths	FullPrecision	Set the usage mode for the filter. Refer to “Usage Modes” 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.
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 <i>n</i> . Reflects either the number of interpolator or comb sections, not the total number of sections in the filter.
OutputFracLength	Any positive integer	15	The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.

Name	Values	Default	Description
OutputWordLength	Any positive integer	16	The number of bits applied to the word length to interpret the output data from the filter.
PersistentMemory	false or true	false	<p>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.</p> <p>PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When PersistentMemory is false, you cannot access the filter states. Setting PersistentMemory to true reveals the States property so you can modify the filter states.</p>

# mfilt.cicinterp

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 interpolator. Available when SectionWordLengthMode is SpecifyWordLengths.



<b>Name</b>	<b>Values</b>	<b>Default</b>	<b>Description</b>
SectionWordLengthMode	MinWordLengths, SpecifyWordLengths	MinWordLength	<p>Determines whether the filter object sets the section word lengths or you provide the word lengths explicitly. By default, the filter uses the input and output word lengths in the command to determine the proper word lengths for each section, according to the information in [1]. When you choose SpecifyWordLengths, you provide the word length for each section. In addition, choosing SpecifyWordLengths exposes the SectionWordLengths property for you to modify as needed.</p>

# mfilt.cicinterp

Name	Values	Default	Description
States	filtstates.cic object	m+1-by-n matrix of zeros, after you call function int.	Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. m is the differential delay of the comb section and n is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when PersistentMemory is true. Refer to the filtstates object in the Signal Processing Toolbox 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  $B_{\text{accum}}$  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  $B_{\text{accum}}$  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 Interpolator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
InterpolationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

### Minimum Wordlengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than  $B_{accum}$ , roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [1]) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than  $B_{accum}$  as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits computed using eq. 21 in Hogenauer's paper from  $B_{accum}$  to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the 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 Interpolator'  
Arithmetic: 'fixed'  
DifferentialDelay: 1  
NumberOfSections: 2  
InterpolationFactor: 4  
PersistentMemory: false
```

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

```
FilterInternals: 'MinWordLengths'
```

```
OutputWordLength: 16
```

## Specify wordlengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length  $2 * (\text{NumberOfSections})$ , where each vector element represents the word length for a section. If you specify a scalar, such as  $B_{\text{accum}}$ , the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicinterp;  
hcic.FilterInternals='minwordlengths';  
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display looks like for the SpecifyWordLengths mode.

```

FilterStructure: 'Cascaded Integrator-Comb Interpolator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
InterpolationFactor: 4
PersistentMemory: false

```

```

InputWordLength: 16
InputFracLength: 15

```

```

FilterInternals: 'SpecifyWordLengths'

```

```

SectionWordLengths: [19 18 18 17]

```

```

OutputWordLength: 16

```

### Specify precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the SpecifyPrecision mode, you must enter a vector of length  $2*(NumberOfSections)$  with elements that represent the word length for each section. When you enter a scalar such as  $B_{accum}$ , `mfilt.cicinterp` expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length  $2*(NumberOfSections)$  with elements that represent the fraction length for each section as well. When you enter a scalar such as  $B_{accum}$ , `mfilt.cicinterp` applies scalar expansion as done for the word lengths.

Here is the SpecifyPrecision display.

```

FilterStructure: 'Cascaded Integrator-Comb Interpolator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

```

```
InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11
```

## About the States of the Filter

In the `states` property you find the states for both the integrator and comb portions of the filter. `states` is a matrix of dimensions  $m+1$ -by- $n$ , with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.
- States for the comb portion fill the remaining rows in the state matrix..

To review the states of a CIC filter, or any filter object states, use `int` to assign the states to a variable in MATLAB. As an example, here are the states for a CIC interpolator `hm` before and after filtering a data set.

```
x = fi(ones(1,10),true,16,0); % Fixed-point input data.
hm = mfilt.cicinterp(2,1,2,16,16,16);
sts=int(hm.states)

sts =

     0     0
     0     0

set(hm,'InputFracLength',0); % Integer input specified.
y=filter(hm,x)

sts=int(hm.states)

sts =
```

```

10    45
28    13

```

### Design Considerations

When you design your CIC interpolation filter, remember the following general points:

- The filter output spectrum has nulls at  $\omega = k * 2\pi/rm$  radians,  $k = 1,2,3,\dots$
- 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.

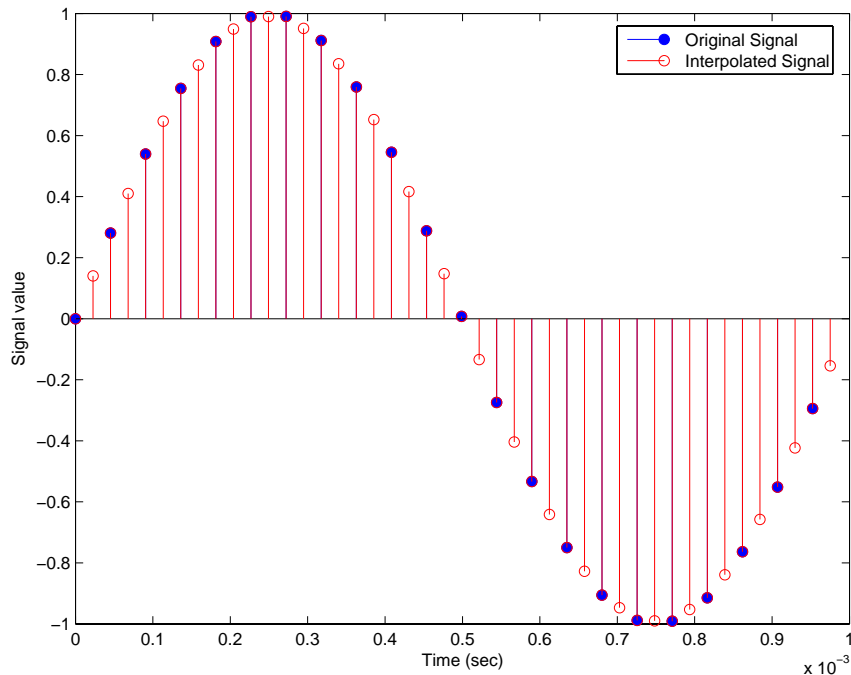
hold on;
stem(n(1:44)/(fs*R),y(4:47),'r'); % Plot interpolated signal
% (44.1 kHz) in red.

```

# mfilt.cicinterp

```
xlabel('Time (sec)');ylabel('Signal Value');
```

As you expect, the plot shows that the interpolated signal matches the input sine shape, with additional samples between each original sample.



Use the filter visualization tool (FVTool) to plot the response of the interpolator object. For example, to plot the response of an interpolator with an interpolation factor of 7, 4 sections, and 1 differential delay, do something like the following:

```
hm = mfilt.cicinterp(7,1,4)
fvtool(hm)
```

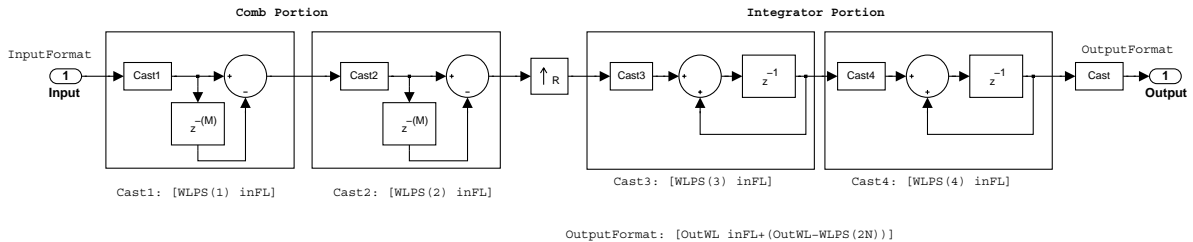
## Algorithm

To show how the CIC interpolation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC



interpolation filter ( $n = 2$ ).  $fs$  is the high sampling rate, the output from the interpolation process.

For details about the bits that are removed in the integrator section, refer to [1] in References.



When you select `MinWordLengths`, the filter section word lengths are automatically set to the minimum number of bits possible in a valid CIC interpolator. `mfiltr.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.fftfirinterp

---

**Purpose** Construct an overlap-add FIR polyphase interpolator filter object

**Syntax** `hm = mfilt.fftfirinterp(l,num,b1)`

**Description** `hm = mfilt.fftfirinterp(l,num,b1)` returns a discrete-time FIR filter object that uses the overlap-add method for filtering input data.

The number of FFT points is given by  $[b1 + \text{ceil}(\text{length}(\text{num})/l) - 1]$ . 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>l</code>	Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. 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 interpolation. When <code>num</code> is not provided as an input, <code>fftfirinterp</code> uses a lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to $\pi/l$ 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 100

## mfilt.fftfirinterp Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for

creating `mfilt.fftfirinterp` objects. The next table describes each property for an `mfilt.fftfirinterp` filter object.

Name	Values	Description
FilterStructure		Reports the type of filter object. You cannot set this property—it is always read only and results from your choice of <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.
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. <code>PersistentMemory</code> 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.

# mfilt.fftfirinterp

---

## Examples

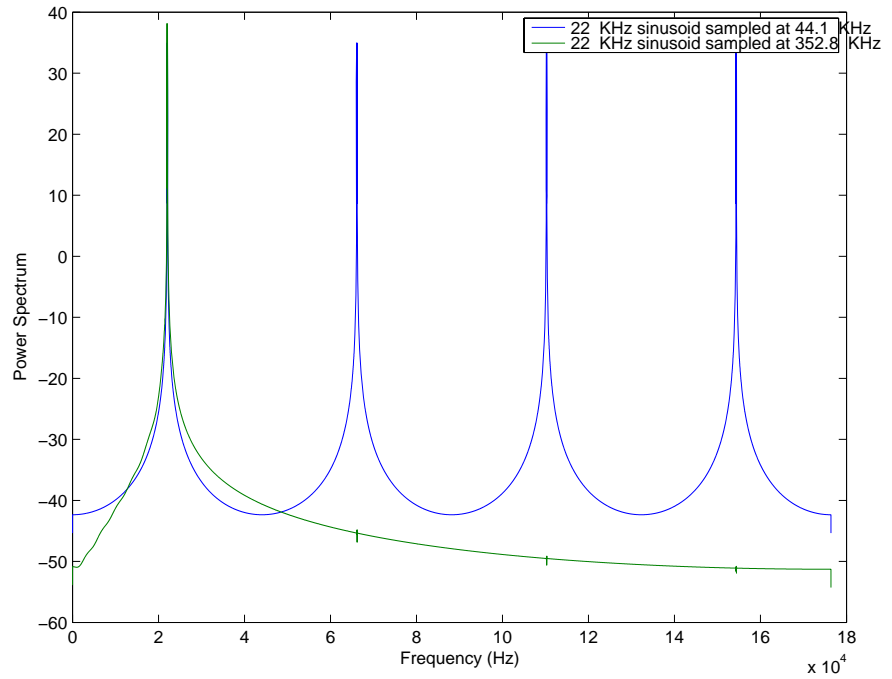
Interpolation by a factor of 8. Notice that this object removes the spectral replicas in the signal after interpolation.

```
l = 8; % Interpolation factor
hm = mfilt.fftfirinterp(l); % We use the default filter
n = 8192; % Number of points
hm.blocklength = n; % Set block length to number of points
fs = 44.1e3; % Original sample freq: 44.1 kHz.
n = 0:n-1; % 0.1858 secs of data
x = sin(2*pi*n*22e3/fs); % Original signal, sinusoid at 22 kHz
y = filter(hm,x); % Interpolated sinusoid
xu = l*upsample(x,8); % Upsample to compare--the spectrum
% does not change

[px,f]=periodogram(xu,[],65536,l*fs);% Power spectrum of original
% signal
[py,f]=periodogram(y,[],65536,l*fs); % Power spectrum of
% interpolated signal

plot(f,10*log10([fs*px,l*fs*py]))
legend('22 kHz sinusoid sampled at 44.1 kHz',...
'22 kHz sinusoid sampled at 352.8 kHz')
xlabel('Frequency (Hz)'); ylabel('Power Spectrum');
```

To see the results of the example, look at this figure.



## See Also

`mfilt.firinterp`, `mfilt.holdinterp`, `mfilt.linearinterp`,  
`mfilt.firfracinterp`, `mfilt.cicinterp`

# mfilt.firdecim

---

**Purpose** Construct direct-form FIR polyphase decimator filter

**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 the input signal. It must be an integer.

## mfilt.firdecim

---

<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. 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 <code>true</code> allows you to modify the <code>States</code> , <code>InputOffset</code> , and <code>PolyphaseAccum</code> properties.



Name	Values	Description
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. Notice that 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.

## mfilt.firdecim

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 7-117.

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	[ <code>true</code> ], <code>false</code>	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.
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

---

<b>Name</b>	<b>Values</b>	<b>Description</b>
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.

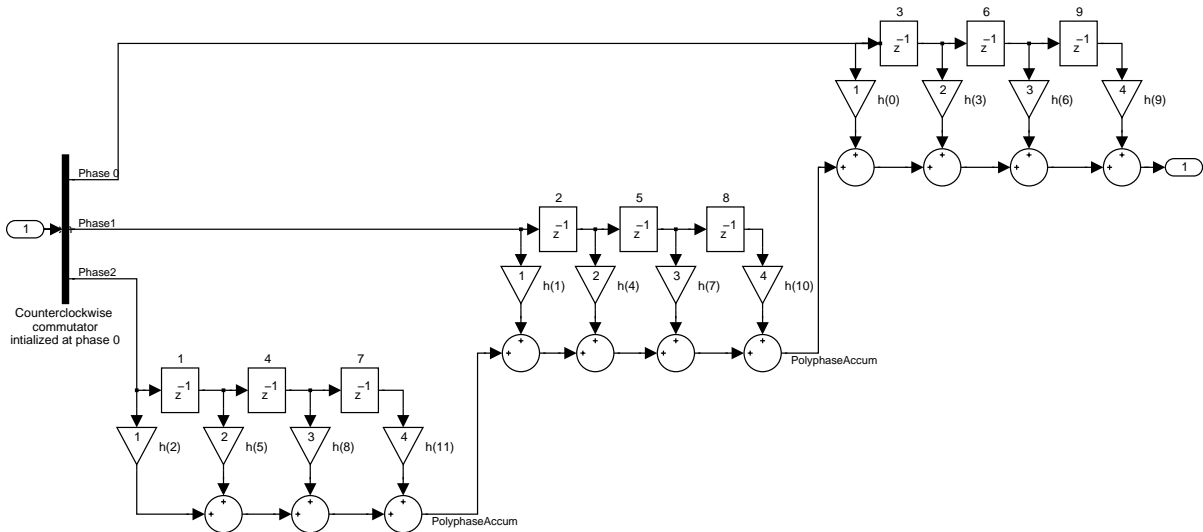
Name	Values	Description
RoundMode	[convergent], ceil,fix,floor, round	<p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"><li>• <b>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

# mfilt.firdecim

Name	Values	Description
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system. For information about the ordering of the states, refer to the filter structure section.

**Filter Structure** To provide decimation, `mfilt.firdecim` uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter.

The figure below details the signal flow for the direct form FIR filter implemented by `mfilt.firdecim`.



Notice the order of the states in the filter flow diagram. States 1 through 9 appear in the diagram above each delay element. State 1 applies to the first delay element in phase 2. State 2 applies to the first delay element in phase 1. State 3 applies to the first delay element in phase 0. State 4 applies to the second delay in phase 2, and so on. When you provide the states for the filter as a vector to the States property, the above description explains how the filter assigns the states you specify.

In property value form, the states for a filter `hm` are

```
hm.states=[1:9];
```

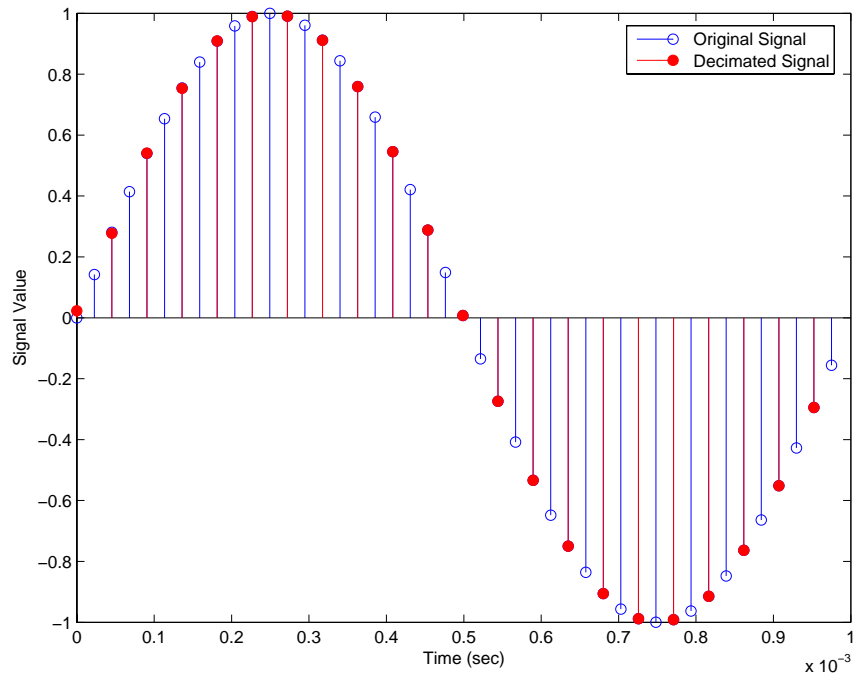
## Examples

Convert an input signal from 44.1 kHz to 22.05 kHz using decimation by a factor of 2. In the figure that appears after the example code, you see the results of the decimation.

```
m = 2; % Decimation factor.
hm = mfilt.firdecim(m); % Use the default filter.
fs = 44.1e3; % Original sample freq: 44.1kHz.
n = 0:10239; % 10240 samples, 0.232 second long
% signal.
```

# mfilt.firdecim

```
x = sin(2*pi*1e3/fs*n); % Original signal--sinusoid at 1kHz.
y = filter(hm,x); % 5120 samples, 0.232 seconds.
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1 kHz.
hold on % Plot decimated signal (22.05 kHz)
% in red.
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```



## See Also

`mfilt.firtdecim`, `mfilt.firfracdecim`, `mfilt.cicdecim`



**Purpose** Construct direct-form FIR polyphase fractional decimator filter object

**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 omitted, 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  $3/2$  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 interpolation. 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.
<code>m</code>	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 <code>l + 1</code> .

# mfilt.firfracdecim

---

## 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.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for interpolation.
RateChangeFactors	[1,m]	Reports the decimation ( <i>m</i> ) and interpolation ( <i>l</i> ) factors for the filter object. Combining these factors results in the final rate change for the signal.

Name	Values	Description
PersistentMemory	false or true	<p>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.</p> <p>PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.</p>
States	Matrix	<p>Stored conditions for the delays between each interpolator phase, the filter states, and the states at the output of each phase in the filter.</p> <p>The number of states is <math>(lh-1)*m+(l-1)*(lo+mo)</math> where <math>lh</math> is the length of each subfilter, and <math>l</math> and <math>m</math> are the interpolation and decimation factors. <math>lo</math> and <math>mo</math>, the input and output delays between each interpolation phase, are integers from Euclid's theorem such that <math>lo*l-mo*m = -1</math> (refer to the reference for more details). Use <code>euclidfactors</code> to get <math>lo</math> and <math>mo</math> for an <code>mfilt.firfracdecim</code> object</p>

# mfilt.firfracdecim

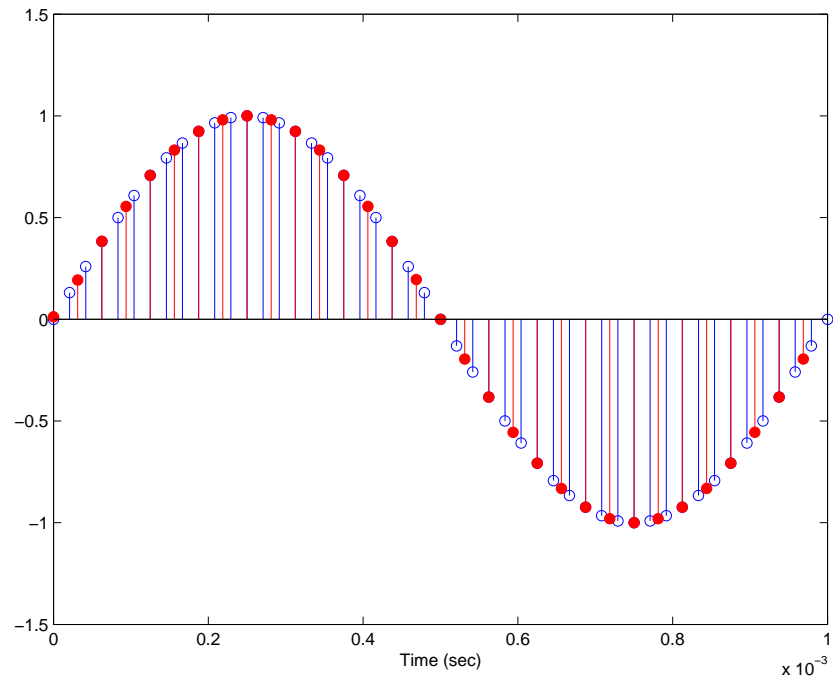
---

## Example

To demonstrate `firfracdecim`, perform a fractional decimation by a factor of  $2/3$ . This is one way to downsample a 48 kHz signal to 32 kHz, commonly done in audio processing.

```
l = 2; m = 3; % Interpolation/decimation factors.
hm = mfilt.firfracdecim(l,m); % We use the default
fs = 48e3; % Original sample freq: 48 kHz.
n = 0:10239; % 10240 samples, 0.213 second long
% signal
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz
y = filter(hm,x); % 9408 samples, still 0.213 seconds
stem(n(1:49)/fs,x(1:49)); hold on; % Plot original signal sampled
% at 48 kHz
stem(n(1:32)/(fs*l/m),y(13:44),'r','filled') % Plot decimated
% signal at 32 kHz
xlabel('Time (sec)');
```

As shown, the plot clearly demonstrates the reduced sampling frequency of 32 kHz.



## See Also

`mfilt.firsrc`, `mfilt.firfracinterp`, `mfilt.firinterp`, `mfilt.firdecim`

## References

Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley & Sons, Ltd., 1994

# mfilt.firfracinterp

---

**Purpose** Construct direct-form FIR polyphase fractional interpolator filter object

**Syntax** `hm = mfilt.firfracinterp(1,m,num)`

**Description** `hm = mfilt.firfracinterp(1,m,num)` returns a direct-form FIR polyphase fractional interpolator `mfilt` 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> .

Input Argument	Description
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, firfracinterp uses a lowpass Nyquist filter with gain equal to 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 mfilt 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.

# mfilt.firfracinterp

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

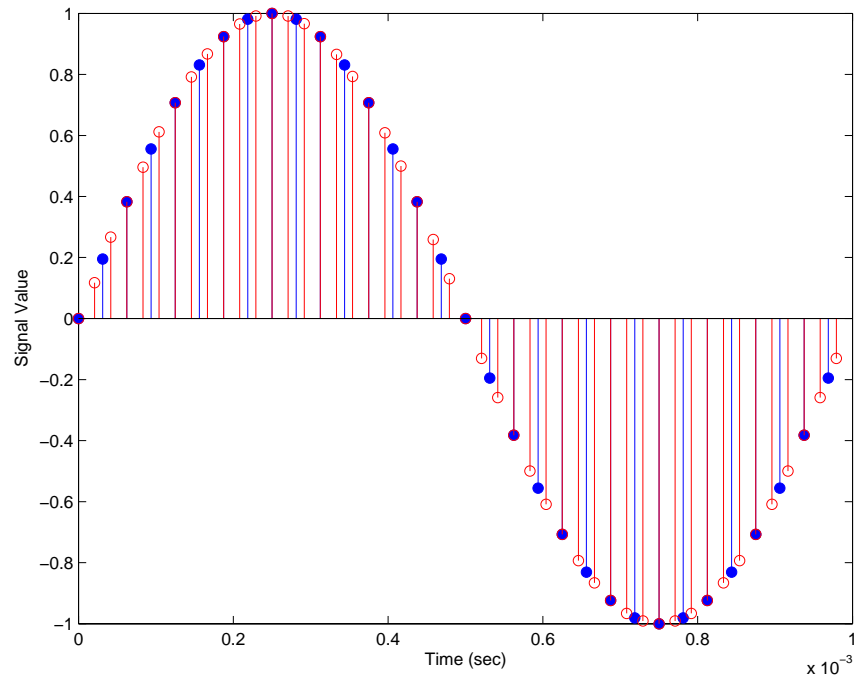
To convert a signal from 32 kHz to 48 kHz requires fractional interpolation. This example uses the `mfilt.firfracinterp` object to upsample an input signal. Setting `l = 3` and `m = 2` returns the same `mfilt` object as the default `mfilt.firfracinterp` object.

```
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')
xlabel('Time (sec)');ylabel('Signal Value')
```



Having the ability to interpolate by fractional amounts lets us raise the sampling rate from 32 to 48 kHz, something you cannot do with integral interpolators. Both signals appear in the following figure.



## See Also

`mfilt.firsrc`, `mfilt.firfracdecim`, `mfilt.firinterp`, `mfilt.firdecim`

# mfilt.firinterp

---

**Purpose** Construct FIR filter-based interpolator

**Syntax**  
`hm = mfilt.firinterp(1)`  
`hm = mfilt.firinterp(1,num)`

**Description** `hm = mfilt.firinterp(1)` returns an FIR-based interpolator object `hm` with an interpolation factor of 1. A lowpass Nyquist filter of gain 1 and cutoff frequency of  $\pi/1$  is the default if you do not include 1 as an input.

`hm = mfilt.firinterp(1,num)` uses the coefficients specified by `num` for the numerator coefficients of the interpolation filter.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter  
`set(hm,'arithmetic','single');`
- To change to fixed-point filtering, enter  
`set(hm,'arithmetic','fixed');`

## Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
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 $24*1$ . 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. PersistentMemory set to false returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to true allows you to modify the States property.
States	Double, single, matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

## Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.firinterp` filter.

---

**Note** The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties.

To view all of the characteristics for a filter at any time, use  
`info(hm)`

where `hm` is a filter.

---

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 7-117.

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— <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> —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 <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.

# mfilt.firinterp

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.

<b>Name</b>	<b>Values</b>	<b>Description</b>
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.firinterp

Name	Values	Description
RoundMode	[convergent], ceil,fix,floor, 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>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

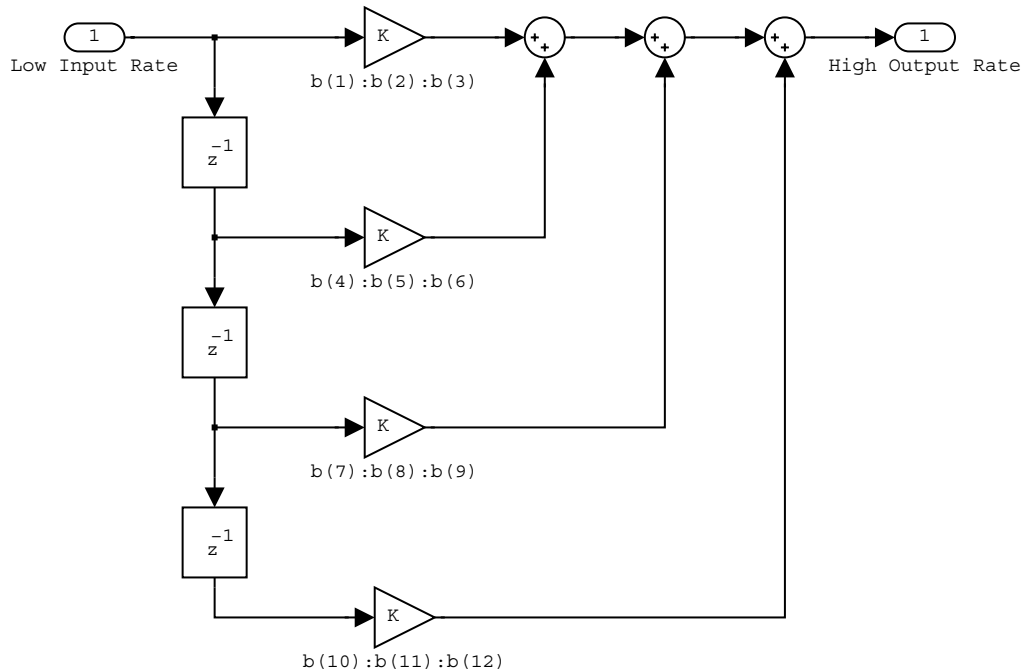


Name	Values	Description
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object to match the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system.

**Filter Structure** To provide interpolation, `mfilt.firinterp` uses the following structure.

The figure below 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

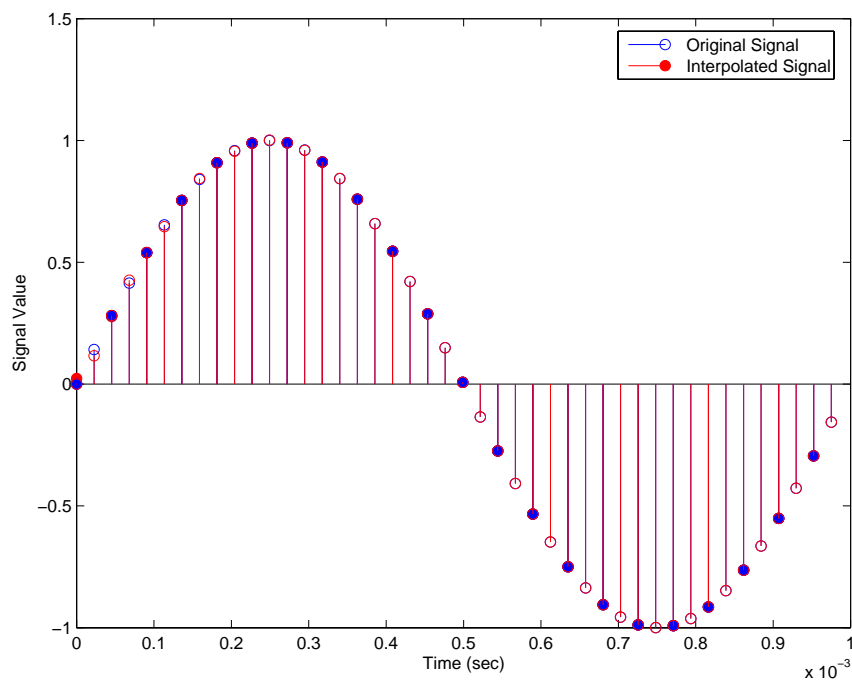
This example uses `mfilt.firinterp` to double the sample rate of a 22.05 kHz input signal. The output signal ends up at 44.1 kHz. Although `l` is set explicitly to 2, this represents the default interpolation value for `mfilt.firinterp` objects.

```
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*1),y(25:68),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```

With interpolation by 2, the resulting signal perfectly matches the original, but with twice as many samples—one between each original sample, as shown in the following figure.



## See Also

`mfilt.holdinterp`, `mfilt.linearinterp`, `mfilt.fftfirinterp`,  
`mfilt.firfracinterp`, `mfilt.cicinterp`

**Purpose** Construct direct-form FIR polyphase sample rate converters

**Syntax** `hm = mfilt.firsrc(l,m,num)`

**Description** `hm = mfilt.firsrc(l,m,num)` returns a direct-form FIR polyphase sample rate converter. `l` specifies the interpolation factor. It must be an integer and when omitted in the calling syntax, it defaults to 2.

`m` is the decimation factor. It must be an integer. If not specified, `m` defaults to 1. If `l` is also not specified, `m` defaults to 3 and the overall rate change factor is 2/3.

You specify the coefficients of the FIR lowpass filter used for sample rate conversion in `num`. If omitted, a lowpass Nyquist filter with gain 1 and cutoff frequency of  $\pi/\max(l,m)$  is the default.

Combining an interpolation factor and a decimation factor lets you use `mfilt.firsrc` to perform fractional interpolation or decimation on an input signal. Using an `mfilt.firsrc` object applies a rate change factor defined by  $1/m$  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');`

## 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, mfilt.firsrc 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 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 double, single, and fixed. In short, this property defines the operation mode for your filter.
FilterStructure	String	Reports the type of filter object. You cannot set this property—it is always read only and results from your choice of <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.

Name	Values	Description
PersistentMemory	false, true	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory set to false returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to true allows you to modify the States, InputOffset, and PolyphaseAccum properties.
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 7-117.

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.



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.

## mfilt.firsrc

---

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
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, 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>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system. For information about the ordering of the states, refer to the filter structure section.

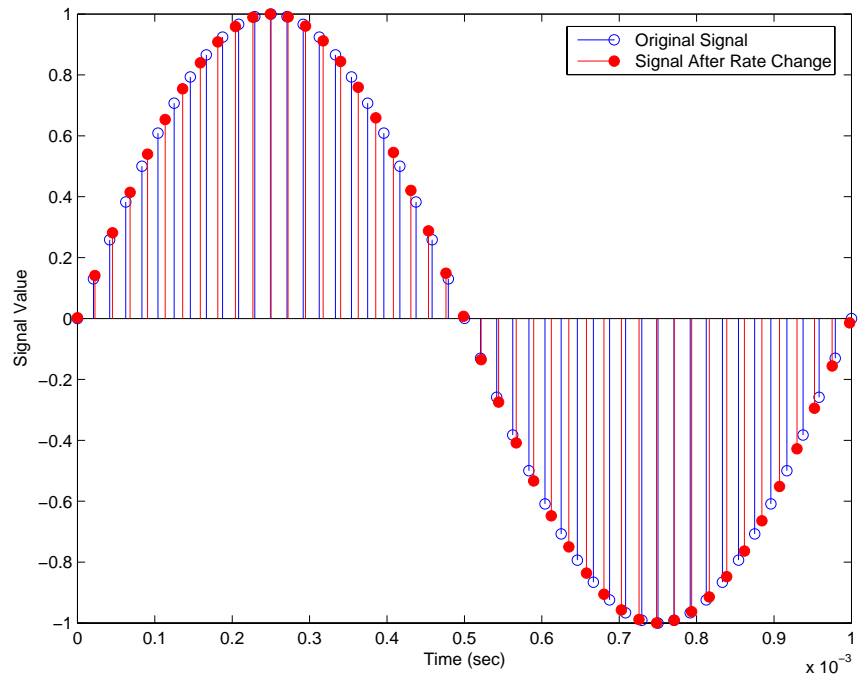
## Examples

This is an example of a common audio rate change process—changing the sample rate of a high end audio (48 kHz) signal to the compact disc sample rate (44.1 kHz). This conversion requires a rate change factor of 0.91875, or  $l = 147$  and  $m = 160$ .

```
l = 147; m = 160;           % Interpolation/decimation factors.
hm = mfilt.firsrc(l,m);    % Use the default FIR filter.
fs = 48e3;                 % Original sample freq: 48 kHz.
n = 0:10239;               % 10240 samples, 0.213 seconds long.
x = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz.
y = filter(hm,x);          % 9408 samples, still 0.213 seconds.
stem(n(1:49)/fs,x(1:49))   % Plot original sampled at 48 kHz.
hold on

% Plot fractionally decimated signal (44.1 kHz) in red
stem(n(1:45)/(fs*l/m),y(13:57),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```

Fractional decimation provides you the flexibility to pick and choose the sample rates you want by carefully selecting  $l$  and  $m$ , the interpolation and decimation factors, that result in the final fractional decimation. The following figure shows the signal after applying the rate change filter `hm` to the original signal.

**See Also**

`mfilt.firfracinterp`, `mfilt.firfracdecim`, `mfilt.firinterp`,  
`mfilt.firdecim`

# mfilt.firtdecim

---

**Purpose** Construct 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`.

<b>Input Argument</b>	<b>Description</b>
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.

<b>Name</b>	<b>Values</b>	<b>Description</b>
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.

## mfilt.firtdecim

---

<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. 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. <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
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.

## mfilt.firtdecim

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 7-117.

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— <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> —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 <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 [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.firtdecim

---

Name	Values	Description
OverflowMode	saturate, [wrap]	<p>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.</p>
PolyphaseAccum	fi object with zeros to start	<p>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.</p>

Name	Values	Description
RoundMode	[convergent], ceil,fix,floor, 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>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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>

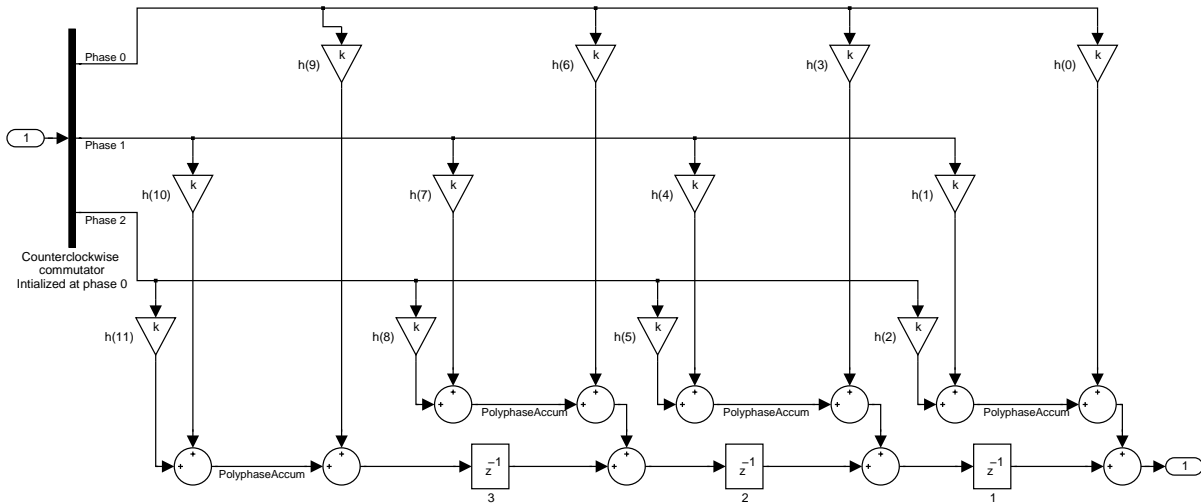
# mfilt.firtdecim

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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system. For information about the ordering of the states, refer to the filter structure section.

## Filter Structure

To provide sample rate changes, `mfilt.firtdecim` uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter. To keep track of the position of the commutator, the `mfilt` object uses the property `InputOffset` which reports the current position of the commutator in the filter.

The figure below 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 diagram below 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];
```

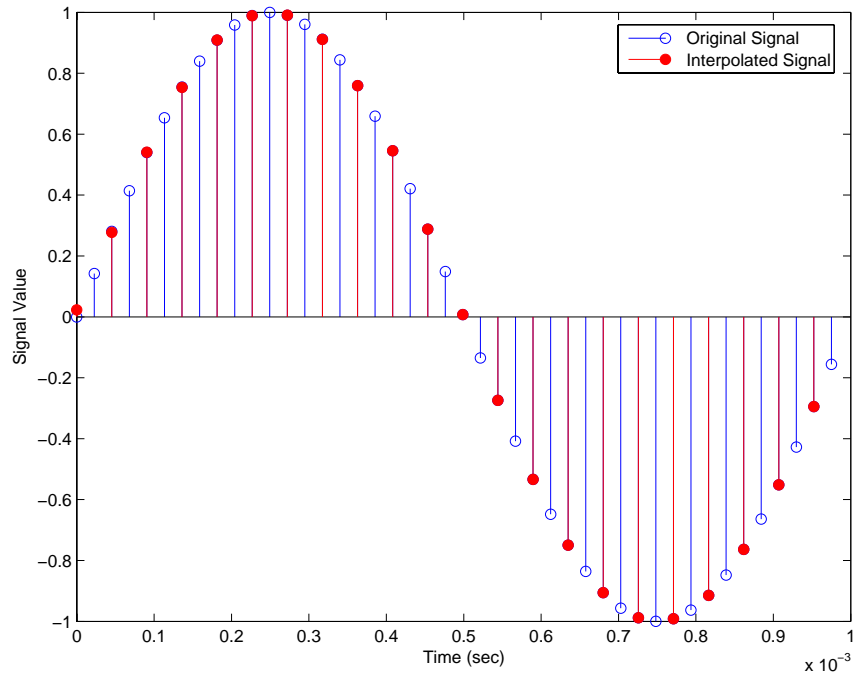
### Examples

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 = mfilter.firtdecim(m); % Use the default filter coeffs.
fs = 44.1e3; % Original sample freq: 44.1 kHz.
n = 0:10239; % 10240 samples, 0.232 second long signal
x = sin(2*pi*1e3/fs*n); % Original signal--sinusoid at 1 kHz.
y = filter(hm,x); % 5120 samples, 0.232 seconds.
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1 kHz.
hold on % Plot decimated signal (22.05 kHz) in red
```

# mfilt.firtdecim

```
stem(n(1:22)/(fs/m),y(13:34),'r','filled')  
xlabel('Time (sec)');ylabel('Signal Value')
```



## See Also

[mfilt.firdecim](#), [mfilt.firfracdecim](#), [mfilt.cicdecim](#)



**Purpose** Construct 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`.

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.

### Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

# mfilt.holdinterp

## Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.holdinterp` objects. The next table describes each property for an `mfilt.interp` filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Specifies the arithmetic the filter uses to process data while filtering.
FilterStructure	String	Reports the type of filter object. You cannot set this property—it is always read only and results from your choice of <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'	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(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 7-117.

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.

# mfilt.holdinterp

Name	Values	Description
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	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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system.

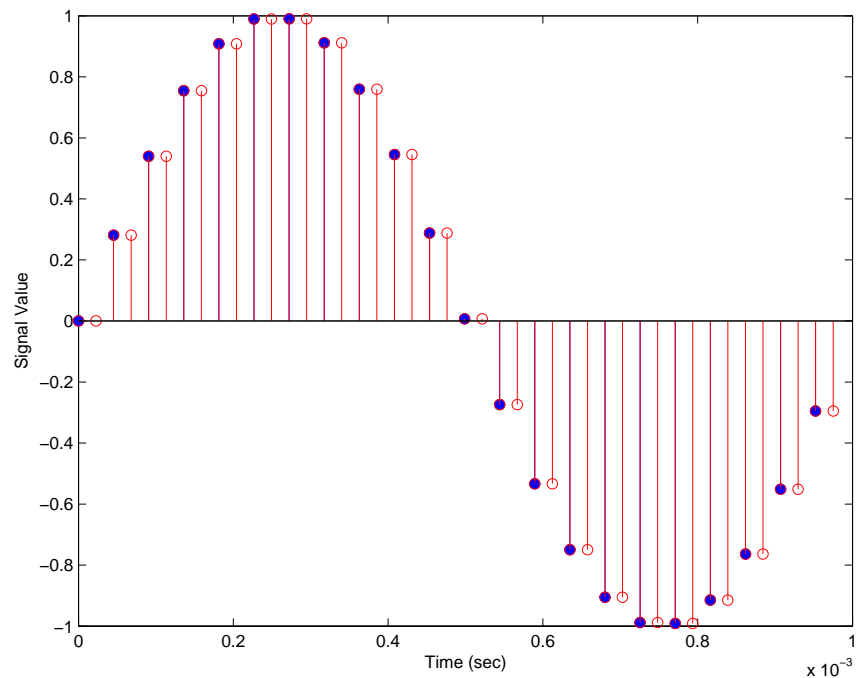
**Filter Structure** Hold interpolators do not have structures or filter coefficients.

**Examples** To see the effects of hold-based interpolation, interpolate an input sine wave from 22.05 to 44.1 kHz. Note that each added sample retains the value of the most recent original sample.

```
l = 2; % Interpolation factor
hm = mfilt.holdinterp(l);
fs = 22.05e3; % Original sample freq: 22.05 kHz.
n = 0:5119; % 5120 samples, 0.232 second long signal
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz
y = filter(hm,x); % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
% 22.05 kHz
```

```
hold on % Plot interpolated signal (44.1 kHz)
in red
stem(n(1:44)/(fs*1),y(1:44),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```

The following figure shows clearly the step nature of the signal that comes from interpolating the signal using the hold algorithm approach. Compare the output to the linear interpolation used in `mfilt.linearinterp`.



## See Also

`mfilt.linearinterp`, `mfilt.firinterp`, `mfilt.firfracinterp`,  
`mfilt.cicinterp`

# mfilt.iirdecim

---

**Purpose** Construct IIR decimator filter object

**Syntax** `hm = mfilt.iirdecim(c1,c2,...)`

**Description** `hm = mfilt.iirdecim(c1,c2,...)` constructs an IIR decimator filter given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The resulting IIR decimator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array—if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR decimators.

Usually you do not construct IIR decimators explicitly. Instead, you obtain an IIR decimator filter as a result of designing a halfband decimator. The first example below illustrates this case.

## Examples

Design an elliptic halfband decimator with a decimation factor of 2. Notice that the example specifies the optional sampling frequency argument.

```
tw = 100; % Transition width of filter to design, 100 Hz.
ast = 80; % Stopband attenuation of filter to design, 80 dB.
fs = 2000; % Sampling frequency of signal to filter.
m = 2; % Decimation factor.
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

`d` contains the specifications for a decimator defined by `tw`, `ast`, `m`, and `fs`.

Use the specification object `d` to perform an actual filter design. `hm` is an `mfilt.iirdecim` filter object.

```
hm = design(d,'ellip','filterstructure','iirdecim');
realizemdl(hm) % Requires Simulink to build model for 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 to design, 100 Hz.
ast = 60; % Stopband attenuation of filter to design, 80 dB.
fs = 2000; % Sampling frequency of signal to filter.
m = 2; % Decimation factor.
```

Create a specification object for the decimator.

```
d = fdesign.decimator(m, 'halfband', 'tw,ast',tw,ast,fs);
```

Finally, design the actual filter hm. As designed, hm is an `mfilt.iirdecim` filter object.

```
hm = design(d, 'iirlinphase', 'filterstructure', 'iirdecim');
realizemdl(hm) % Requires Simulink to visualize the structure.
```

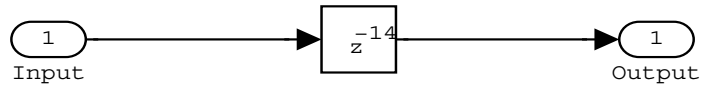
The filter implementation appears in this model, generated by `realizemdl` and `Simulink`.

Given the design specifications shown here

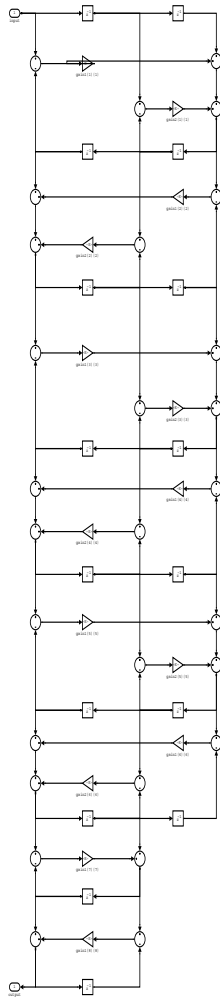
```
hm =
    FilterStructure: 'IIR Polyphase Decimator'
    Polyphase: Phase1: Section1: [0 0 0 0 0 0 0 0 0 0 0 0 0 1]
    Phase2: Section1: [1.14740498857167 0.409481636102326]
    Section2: [0.751016281415127 0.36048597074495]
    Section3: [0.272921271612044 0.343931116911137]
    Section4: [-0.244601181956782 0.33691092991289]
    Section5: [-0.711317191438094 0.333590883744604]
    Section6: [-1.03562723857273 0.332039064718955]
    Section7: 0.893704991634848
    Section8: -0.575824830892574
    DecimationFactor: 2
    PersistentMemory: false
```

the first phase is a delay section with 0s and a 1 for coefficients and the second phase is a linear phase decimator, shown in the next models.

## Phase 1 model

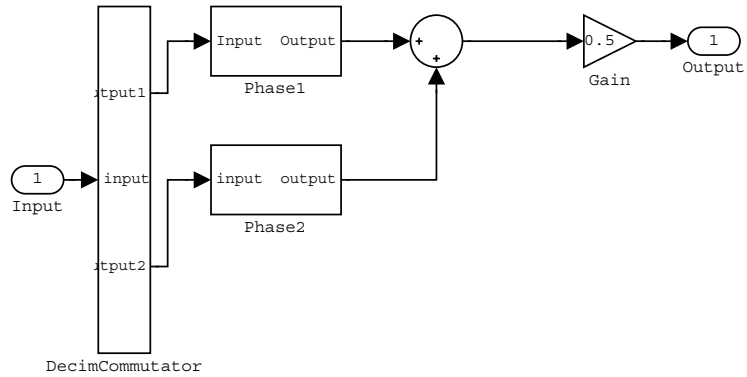


## Phase 2 model





**Overall model**



**See Also**

`dfilt.cascadeallpass`, `mfilt`, `mfilt.iirinterp`, `mfilt.iirwdfdecim`

# mfilt.iirinterp

---

**Purpose** Construct IIR interpolator filter object

**Syntax** `hm = mfilt.iirinterp(c1,c2,...)`

**Description** `hm = mfilt.iirinterp(c1,c2,...)` constructs an IIR interpolator filter given the coefficients specified in the cell arrays C1, C2, etc.

The IIR interpolator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array—if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and a unique element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR interpolators.

Usually you do not construct IIR interpolators explicitly. Instead, you obtain an IIR interpolator filter as a result of designing a halfband interpolator. The first example below illustrates this case.

## Examples

Design an elliptic halfband interpolator with a interpolation factor of 2.

```
tw = 100; % Transition width of filter to design, 100 hz.  
ast = 80; % Stopband attenuation of filter to design, 80 dB.  
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');  
realizemdl(hm) % Requires Simulink to build model for filter.
```

Designing a linear phase halfband interpolator follows the same pattern.

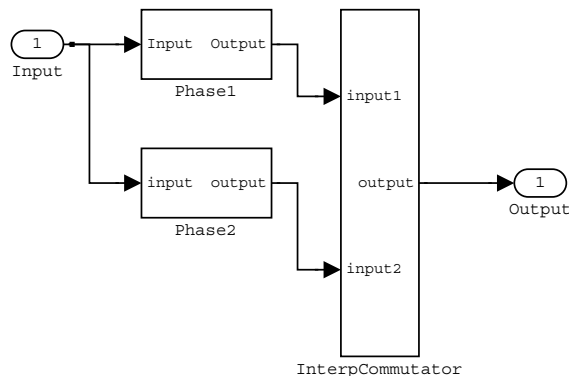
```
tw = 100; % Transition width of filter to design, 100 Hz.
ast= 60; % Stopband attenuation of filter to design, 80 dB.
fs = 2000; % Sampling frequency of filter.
l = 2; % Interpolation factor.
d = fdesign.interpolator(1,'halfband', 'tw,ast',tw,ast,fs);
```

`fdesign.interpolator` returns a specification object that stores the design features for an interpolator.

Now perform the actual design that results in an `mfilt.iirinterp` filter, `hm`.

```
hm = design(d,'iirlinphase','filterstructure','iirinterp');
realizemdl(hm)
```

The toolbox creates a Simulink model for `hm`, shown here. `Phase1`, `Phase2`, and `InterpCommutator` are all subsystem blocks.



## See Also

`dfilt.cascadeallpass`, `mfilt`, `mfilt.iirdecim`, `mfilt.iirwdfinterp`

# mfilt.iirwdfdecim

---

**Purpose** Construct IIR wave digital filter decimator object

**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 below 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');
realizemdl(hm) % Requires Simulink to build and visualize the
structure.
```

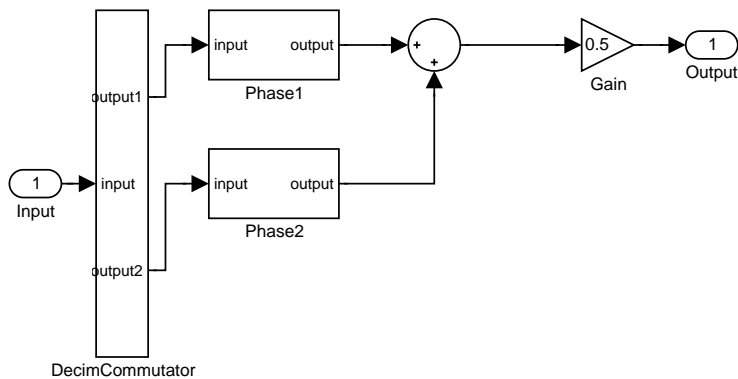
Design a linear phase halfband decimator for decimating a signal by a factor of 2.

```
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);
```

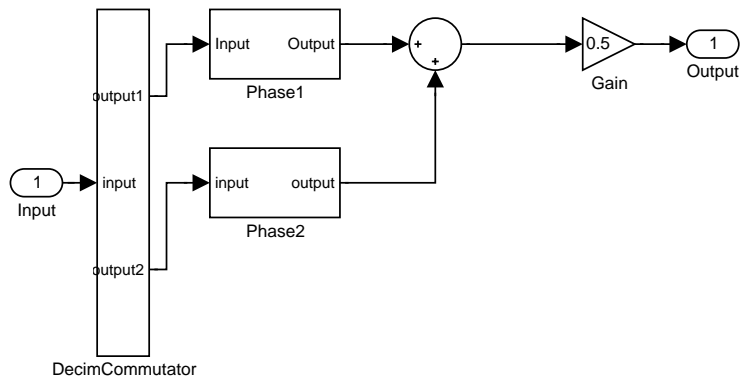
Use `d` to design the final filter `hm`, an `mfilt.iirwdfdecim` object.

```
hm = design(d,'iirlinphase','filterstructure','iirwdfdecim');
realizemdl(hm) % Requires Simulink to be able to build model.
```

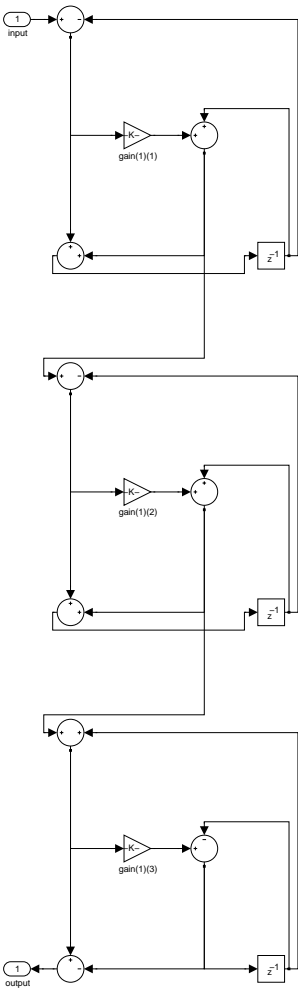
The models that `realizemdl` returns for each example appear below. At this level, the realizations of the filters are identical. The differences appear in the subsystem blocks Phase1 and Phase2.



# mfilt.iirwdfdecim



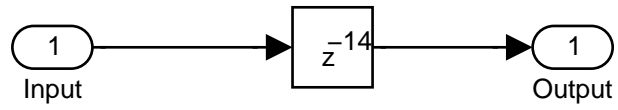
This is the Phase1 subsystem from the halfband model.



Phase1 subsystem from the linear phase model is less revealing—an allpass filter.

# mfilt.iirwdfdecim

---



## See Also

`dfilt.cascadewdfallpass`, `mfilt`, `mfilt.iirdecim`, `mfilt.iirwdfinterp`



**Purpose** Construct IIR wave digital interpolator filter

**Syntax** `hm = iirwdfinterp(c1,c2,...)`

**Description** `hm = mfilt.iirwdfinterp(c1,c2,...)` constructs an IIR wave digital interpolator filter given the coefficients specified in the cell arrays `C1`, `C2`, etc.

`hm = mfilt.iirwdfinterp(c1,c2,...)` constructs an IIR wave digital interpolator given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The IIR interpolator `hm` is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirwdfinterp` interprets them same way as `mfilt.cascadewdfallpass`.

The following exception applies to interpreting the contents of a cell array—if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadewdfallpass` section. This exception occurs with quasi-linear phase IIR interpolators.

Usually you do not construct IIR wave digital filter interpolators explicitly. Rather, you obtain an IIR wave digital interpolator as a result of designing a halfband interpolator. The first example below 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 of signal 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 the Signal Processing Blockset.

```
realizemdl(hm) % Requires Simulink to build model for 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 of signal 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');  
realizemdl(hm) % Requires Simulink to visualize the signal flow.
```

## See Also

`dfilt.cascadewdfallpass`, `mfilt.iirinterp`, `mfilt.iirwdfdecim`

**Purpose** Construct linear interpolator filter

**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 staircase 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.

## Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

# mfilt.linearinterp

## 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 7-117.

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.
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.

# mfilt.linearinterp

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]	<p>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.)</p> <p>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>

# mfilt.linearinterp

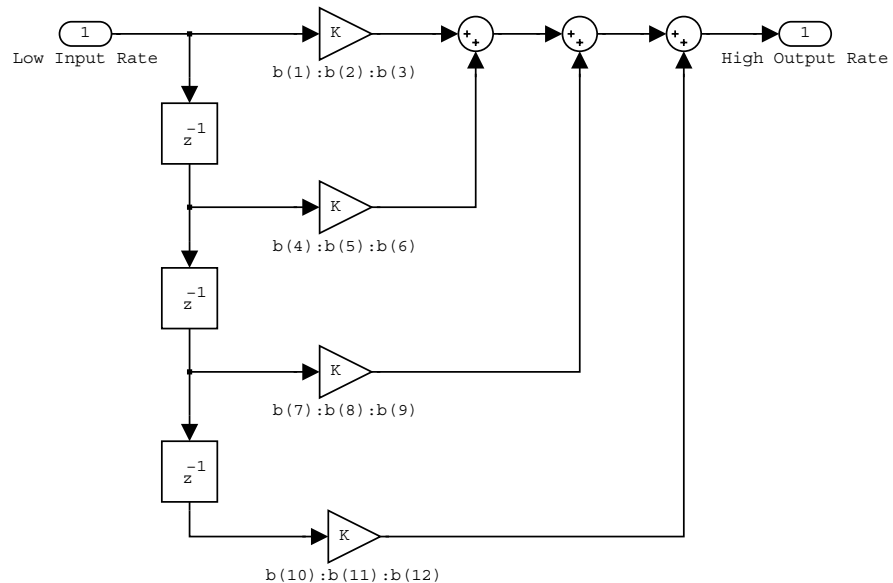
Name	Values	Description
RoundMode	[convergent], ceil,fix,floor, round	<p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"><li>• <b>convergent</b>—Round up to the next allowable quantized value.</li><li>• <b>ceil</b>—Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.</li><li>• <b>fix</b>—Round negative numbers up and positive numbers down to the next allowable quantized value.</li><li>• <b>floor</b>—Round down to the next allowable quantized value.</li><li>• <b>round</b>—Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.</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. Notice that the states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in your Fixed-Point Toolbox documentation or in the online Help system. For information about the ordering of the states, refer to the filter structure below.

**Filter Structure** Linear interpolator structures depend on the FIR filter you use to implement the filter. By default, the structure is direct-form FIR.

# mfilt.linearinterp

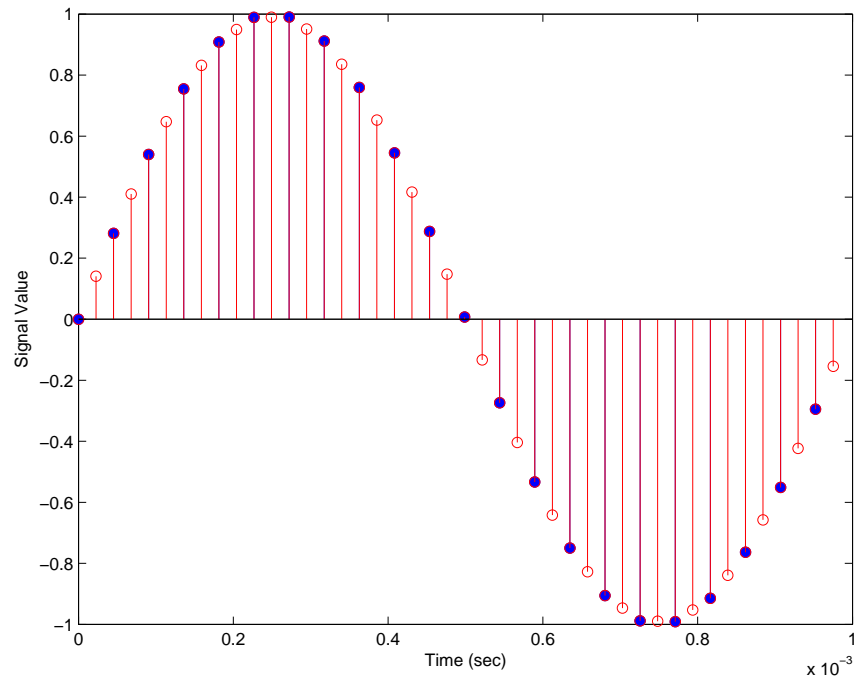


## Examples

Interpolation by a factor of 2 (used to convert the input signal sampling rate from 22.05 kHz to 44.1 kHz).

```
l = 2; % Interpolation factor
hm = mfilt.linearinterp(l);
fs = 22.05e3; % Original sample freq: 22.05 kHz.
n = 0:5119; % 5120 samples, 0.232 second long signal
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz
y = filter(hm,x); % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
% 22.05 kHz
hold on % Plot interpolated signal (44.1
% kHz) in red
stem(n(1:44)/(fs*l),y(2:45),'r')
xlabel('Time (s)');ylabel('Signal Value')
```

Using linear interpolation, as compared to the hold approach of `mfilt.holdinterp`, provides greater fidelity to the original signal.



## See Also

`mfilt.holdinterp`, `mfilt.firinterp`, `mfilt.firfracinterp`,  
`mfilt.cicinterp`

# msepred

---

**Purpose** Calculate predicted mean-squared error for selected 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 mth predicted value of each of these sequences. When you omit the optional argument m, it defaults to one.

---

**Note** msepred is available for the following adaptive filters only:

- adaptfilt.blms
- adaptfilt.blmsfft
- adaptfilt.lms
- adaptfilt.nlms

—adaptfilt.se

Using msepred is the same for any adaptfilt object constructed by the supported filters.

## Examples

Analyze and simulate a 32-coefficient adaptive filter using 25 trials of 2000 iterations each.

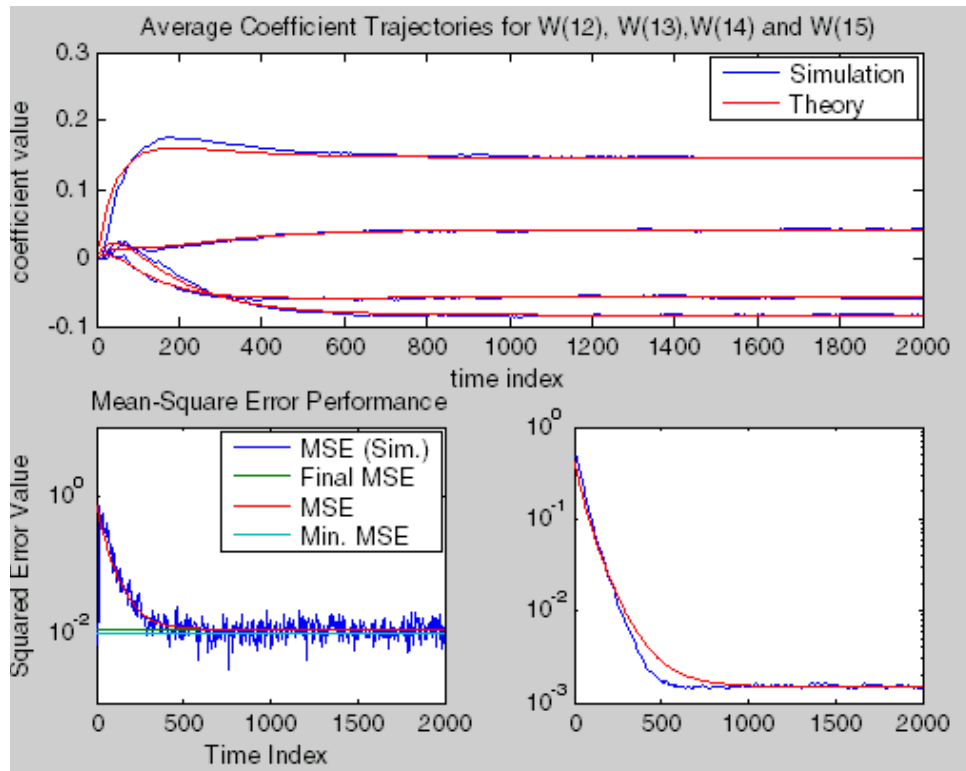
```
x = zeros(2000,25); d = x;           % Initialize variables
ha = fir1(31,0.5);                   % FIR system to be identified
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));              % observation noise signal
d = filter(ha,1,x)+n;                % desired signal
l = 32;                               % Filter length
mu = 0.008;                           % LMS step size.
m = 5;                                % Decimation factor for analysis
                                     % and simulation results

ha = adaptfilt.lms(l,mu);
[mmse,emse,meanW,mse,traceK] = msepred(ha,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r');
title('Average Coefficient Trajectories for W(12), W(13),...
W(14) and W(15)');
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
(emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse]);
title('Mean-Square Error Performance');
axis([0 size(x,1) 0.001 10]);
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim,nn,traceK,'r');
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)...
0.0001 1]);
```

# msepred

```
legend('Simulation','Theory');  
xlabel('Time Index'); ylabel('Squared Error Value');
```

Viewing the plots in this figure you see the various error values plotted in both simulation and theory. Each subplot reveals more information about the results as the simulation converges with the theoretical performance.



## See Also

`filter`, `maxstep`, `msesim`

**Purpose** Calculate and return measured mean-squared error for adaptive filter

**Syntax**

```
mse = msesim(ha,x,d)
[mse,meanw,w,tracek] = msesim(ha,x,d)
[mse,meanw,w,tracek] = msesim(ha,x,d,m)
```

**Description** `mse = msesim(ha,x,d)` returns the sequence of mean-square errors in column vector `mse`. The vector contains estimates of the mean-square error of the adaptive filter at each time instant during adaptation. The length of `mse` is equal to `size(x,1)`. The columns of matrix `x` contain individual input signal sequences, and the columns of the matrix `d` contain corresponding desired response signal sequences.

`[mse,meanw,w,tracek] = msesim(ha,x,d)` calculates three parameters that correspond to the simulated behavior of the adaptive filter defined by `ha`:

- `meanw`—sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the LMS adaptive filter coefficients at each time instant. The dimensions of `meanw` are `(size(x,1))-by-(ha.length)`.
- `w`—estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `ha`.
- `tracek`—sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the LMS adaptive filter at each time instant. The length of `tracek` is equal to `size(X,1)`.

`[mse,meanw,w,tracek] = msesim(ha,x,d,m)` specifies an optional input argument `m` that is the decimation factor for computing `meanw`, `mse`, and `tracek`. When `m > 1`, `msepsim` saves every `m`th predicted value of each of these sequences. When you omit the optional argument `m`, it defaults to one.

**Examples** Simulation of a 32-coefficient FIR filter using 25 trials, each trial having 2000 iterations of the adaptation process.

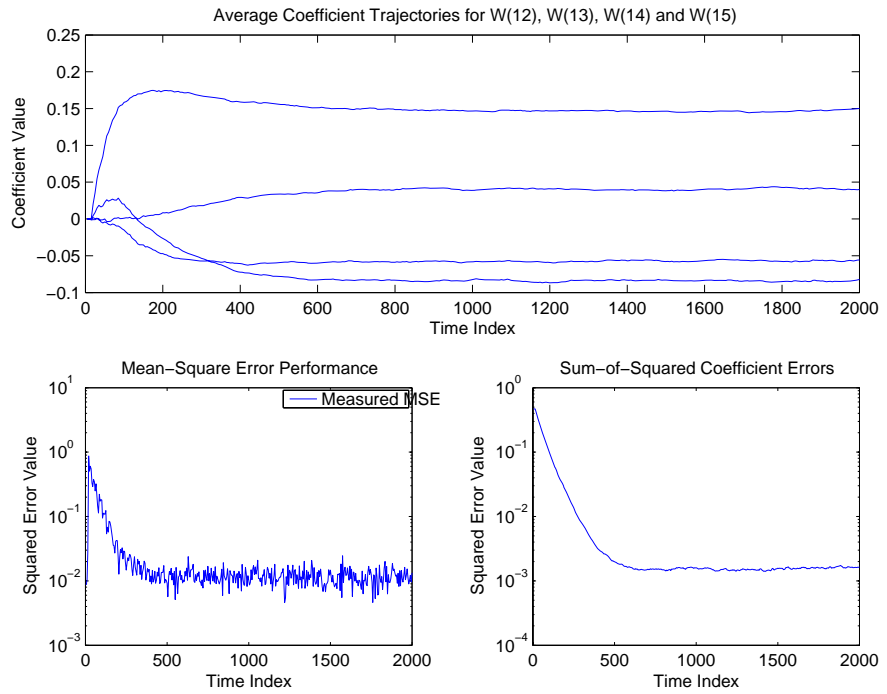
```
x = zeros(2000,25); d = x;           % Initialize variables
ha = fir1(31,0.5);                  % FIR system to be identified
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));              % Observation noise signal
d = filter(ha,1,x)+n;                % Desired signal
```

```
l = 32; % Filter length
mu = 0.008; % LMS Step size.
m = 5; % Decimation factor for analysis
% and simulation results

ha = adaptfilt.lms(l,mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanWsim(:,13:15),'b');
title('Average Coefficient Trajectories for W(12), W(13),
W(14) and W(15)');
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse);
title('Mean-Square Error Performance'); axis([0 size(x,1) 0.001
10]);
legend('Measured MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim);
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)
0.0001 1]);
xlabel('Time Index'); ylabel('Squared Error Value');
```

Calculating the mean squared error for an adaptive filter is one measure of the performance of the adapting algorithm. In this figure, you see a variety of measures of the filter, including the error values.





**See Also**

filter, msepred

# multistage

---

**Purpose** Design multistage filter from filter 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
<code>firdecim</code>	Lowpass or Nyquist response
<code>firtdecim</code>	Lowpass or Nyquist response
<code>firinterp</code>	Lowpass or Nyquist response
<code>lowpass</code>	All lowpass responses

In short, multistage design applies to all lowpass filter specifications objects 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`.

**Examples**

Design a minimum-order, multistage Nyquist interpolator. Use the `FilterStructure` property to specify the Nyquist response.

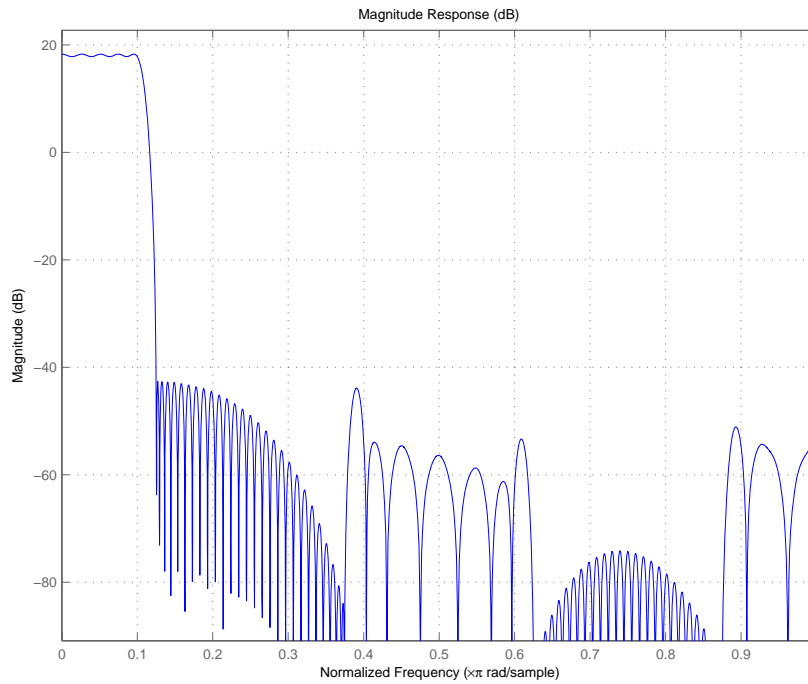
```
l = 15; % Interpolation factor. Also the Nyquist band.
tw = 0.05; % Normalized transition width
ast = 40; % Minimum stopband attenuation in dB
d = fdesign.interpolator(l,'filterstructure','nyquist',l,tw,ast);
hm = design(d,'multistage');
fvtool(hm);
```

Design a multistage lowpass interpolator with an interpolation factor of 8.

```
m = 8; % Interpolation factor;
d = fdesign.interpolator(m,'lowpass');
hm = design(d,'multistage','Usehalfbands',true); % Use halfband filters
                                                % if possible.
fvtool(hm);
```

This figure shows the response for `hm`.

# multistage



**See Also** [design](#), [designopts](#)

**Purpose** Compute power spectral density (PSD) of filter output caused by roundoff noise during quantization

**Syntax**

```
hpsd = noisepsd(hd,1);  
hpsd = noisepsd(hd,1, propertyname1, propertyvalue1,  
    propertyname2,propertyvalue2, );  
hpsd = 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 `avgpwr`, a method of `psd` data objects:

```
avgpwr = avgpower(hpsd).
```

# noisepsd

`hpsd = noisepsd(hd,l,p1,v1,p2,v2,...)` specifies optional parameters via `propertyname/propertyvalue` pairs. The properties of the `psd` object, and the valid entries are:

<b>Property Name</b>	<b>Default Value</b>	<b>Description and Valid Entries</b>
<code>Nfft</code>	512	Specifies the number of FFT points to use to calculate the PSD.
<code>NormalizedFrequency</code>	true	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, not a string.
<code>Fs</code>	normalized	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>

Property Name	Default Value	Description and Valid Entries
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 <code>SpectrumType</code> to <code>onesided</code>, it is changed to <code>twosided</code> and the data is converted to a two-sided spectrum.</li><li>• Setting <code>CenterDC</code> to <code>false</code> 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 <code>SpectrumType</code> 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,1,opts)` uses an options object `opts` to specify the optional input arguments instead of specifying property-value pairs in the command. Use `opts = noisepsopts(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.
```

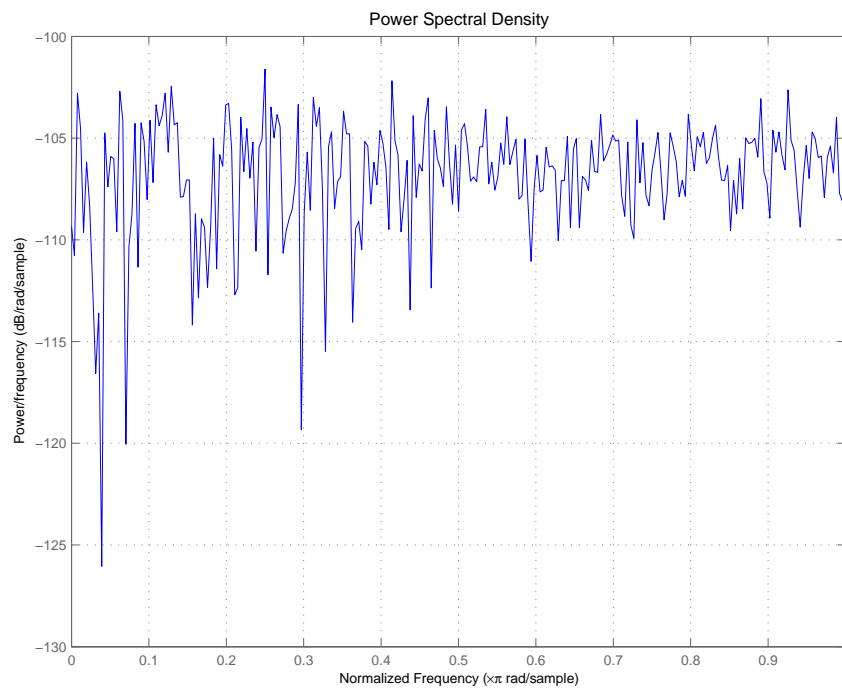
## Examples

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.  
h.arithmetic = 'fixed'; % Quantize the filter to fixed-point.  
hpsd = noisepsd(h);  
plot(hpsd)
```

hpsd looks like this—the data resulting from the noise PSD calculation. You can review the data in hpsd.data'.



Here is the specification for hpsd.

```
hpsd =
```

```
Name: 'Power Spectral Density'  
Data: [257x1 double]
```

# noisepsd

---

```
SpectrumType: 'Onesided'  
Frequencies: [257x1 double]  
NormalizedFrequency: true  
Fs: 'Normalized'
```

## See Also

filter, noisepsdopts, norm, reorder, scale  
spectrum.welch in the Signal Processing Toolbox

## References

McClellan, et al., *Computer-Based Exercises for Signal Processing Using MATLAB 5*, Prentice-Hall, 1998.

**Purpose** Create object containing options for running output noise power spectral density (PSD) computation `noisepsd` on filter

**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>	true	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>	normalized	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.

# noisepsdopts

---

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>

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>

# noisepsdopts

---

Property Name	Default Value	Description and Valid Entries
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>

## See Also

noisepsd

**Purpose** P-norm of `adaptfilt`, `dfilt`, or `mfilt` objects

**Syntax**

```
l = norm(ha)
l = norm(ha, pnorm)
l = norm(hd)
l = norm(hd, pnorm)
l = norm(hd, 'L2', tol)
l = norm(hm)
l = norm(hm, pnorm)
```

**Description** All of the variants of `norm` return the filter p-norm for the object in the syntax, either an adaptive filter, a digital filter, or a multirate filter. When you omit the `pnorm` argument, `norm` returns the L2-norm for the object.

Note that by Parseval's theorem, the L2-norm of a filter is equal to the `l2` norm. This equality is not true for the other norm variants.

#### For `adaptfilt` Objects

`l = norm(ha)` returns the L2-norm of an adaptive filter.

`l = norm(ha, pnorm)` adds the input argument `pnorm` to let you specify the norm returned. `pnorm` can be either

- Frequency-domain norms specified by one of `L1`, `L2`, or `Linf`
- Discrete-time domain norms specified by one of `l1`, `l2`, or `linf`

#### For `dfilt` Objects

`l = norm(hd)` returns the L2-norm of a discrete-time filter.

`l = norm(hd, pnorm)` includes input argument `pnorm` that lets you specify the norm returned. `pnorm` can be either

- Frequency-domain norms specified by one of `L1`, `L2`, or `Linf`
- Discrete-time domain norms specified by one of `l1`, `l2`, or `linf`

By Parseval's theorem, the L2-norm of a filter is equal to the `l2` norm. This equality is not true for the other norm variants.

IIR filters respond slightly differently to `norm`. When you compute the `l2`, `linf`, `L1`, and `L2` norms for an IIR filter, `norm(...,L2,tol)` lets you specify the tolerance for the accuracy in the computation. For `l1`, `l2`, `L2`, and `linf`, `norm` uses the tolerance to truncate the infinite impulse response that it uses to calculate the norm. For `L1`, `norm` passes the tolerance to the numerical integration algorithm. Refer to Examples to see this in use. You cannot specify `Linf` for the norm and include the `tol` option.

## For `mfilt` Objects

`l = norm(hm)` returns the L2-norm of a multirate filter.

`l = norm(hm,pnorm)` includes argument `pnorm` to let you specify the norm returned. `pnorm` can be either

- Frequency-domain norms specified by one of `L1`, `L2`, or `Linf`
- Discrete-time domain norms specified by one of `l1`, `l2`, or `linf`

Note that, by Parseval's theorem, the L2-norm of a filter is equal to the `l2` norm. This equality is not true for the other norm variants.

## Examples

### Adaptfilt Objects

For the adaptive filter example, compute the 2-norm of an `adaptfilt` object, here an LMS-based adaptive filter.

```
ha = adaptfilt.lms; % norm(ha) is zero because all coeffs are zero
% Create some data to filter to generate filter coeffs
x = randn(100,1);
d = x + randn(100,1);
[y,e] = filter(ha,x,d);
l2 = norm(ha); % Now norm(ha) is nonzero
l2 =

    1.1231
```

### Dfilt Objects

To demonstrate the tolerance option used with an IIR filter (`dfilt` object), compute the 2-norm of filter `hd` with a tolerance of `1e-10`.

```
d=fdesign.lowpass('n,fc',5,0.4)
```



```
d =  
  
        Response: 'Lowpass with cutoff'  
Specification: 'N,Fc'  
Description: {2x1 cell}  
NormalizedFrequency: true  
           Fs: 'Normalized'  
FilterOrder: 5  
           Fcutoff: 0.4000
```

```
hd = butter(d);  
l2=norm(hd, 'l2', 1e-10)
```

```
l2 =  
  
    0.6336
```

### Mfilt Objects

In this example, compute the infinity norm of an FIR interpolator, which is an `mfilt` object.

```
hm = mfilt.firinterp;  
linf = norm(hm,inf);  
linf =  
  
    2.0002
```

### See Also

`reorder`, `scale`, `scalecheck`

# normalize

---

**Purpose** Normalize filter numerator or feed-forward coefficients to values between -1 and 1

**Syntax**  
`normalize(hq)`  
`g = normalize(hq)`

**Description** `normalize(hq)` normalizes the filter numerator coefficients for a quantized filter to have values between -1 and 1. Notice that the coefficients of `hq` change—`normalize` does not copy `hq` and return the copy. To restore the coefficients of `hq` to the original values, use `denormalize`.

Note that for lattice filters, the feed-forward coefficients stored in the property `lattice` are normalized.

`g = normalize(hd)` normalizes the numerator coefficients for the filter `hq` to between -1 and 1 and returns the gain `g` due to the normalization operation. Calling `normalize` again does not change the coefficients. `g` always returns the gain returned by the first call to `normalize` the filter.

**Examples** Create a direct form II quantized filter that uses second-order sections. Then use `normalize` to maximize the use of the range of representable coefficients.

```
d=fdesign.lowpass('n,fp,ap,ast',8,.5,2,40);  
  
hd=ellip(d);  
  
hd =  
  
    FilterStructure: 'Direct-Form II, Second-Order Sections'  
    Arithmetic: 'double'  
    sosMatrix: [4x6 double]  
    ScaleValues: [5x1 double]  
    PersistentMemory: 'on'  
    States: [2x4 double]  
  
hd.arithmetic='fixed'  
  
hd =  
  
    FilterStructure: 'Direct-Form II, Second-Order Sections'
```

```
    Arithmetic: 'fixed'
      sosMatrix: [4x6 double]
      ScaleValues: [5x1 double]
PersistentMemory: 'on'
      States: [1x1 embedded.fi]

    CoeffWordLength: 16
      CoeffAutoScale: true
      Signed: true

    InputWordLength: 16
      InputFracLength: 15

    StageInputWordLength: 16
      StageInputAutoScale: true

    StageOutputWordLength: 16
      StageOutputAutoScale: true

    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'

    StateWordLength: 16
      StateFracLength: 15

      ProductMode: 'FullPrecision'

      AccumMode: 'KeepMSB'
    AccumWordLength: 40
      CastBeforeSum: true

      RoundMode: 'convergent'
      OverflowMode: 'wrap'

    InheritSettings: false
```

Check the filter coefficients to see that some of them are greater than 1.

```
hd.sosMatrix
```

# normalize

---

```
ans =  
  
    1.0000    1.5132    1.0000    1.0000   -0.9207    0.4373  
    1.0000    0.3867    1.0000    1.0000   -0.2779    0.8242  
    1.0000    0.0929    1.0000    1.0000   -0.0514    0.9610  
    1.0000    0.0339    1.0000    1.0000   -0.0020    0.9934
```

Use `normalize` to modify the coefficients into the range between -1 and 1. A quick check of the SOS matrix shows all of the numerator coefficients now within the limits. You see that `g` contains the gains applied to each section of the SOS filter.

```
g = normalize(hd)  
  
g =  
  
    1.5132  
    1.0000  
    1.0000  
    1.0000  
  
hd.sosMatrix  
  
ans =  
  
    0.6608    1.0000    0.6608    1.0000   -0.9207    0.4373  
    1.0000    0.3867    1.0000    1.0000   -0.2779    0.8242  
    1.0000    0.0929    1.0000    1.0000   -0.0514    0.9610  
    1.0000    0.0339    1.0000    1.0000   -0.0020    0.9934
```

Notice that none of the numerator coefficients exceed -1 or 1.

## See Also

`denormalize`

**Purpose** Switch filter specification object between normalized frequency specification and absolute frequency specification

**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  $f_s/2$  so they lie between 0 and 1, where  $f_s$  is specified in the object. Included in the normalization are the filter properties that define the filter pass and stopband edge locations by frequency:

- `F3dB`—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  $f_s$  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  $f_s$  and other related frequency specifications.  $f_s$  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  $f_s/2$  to remove the normalization. Use this syntax to switch your filter between using normalized frequency specifications and not using normalized frequency specifications.

# normalizefreq

---

`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 pass- 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 =
```

```
           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
```

# normalizefreq

---

```
Fstop1: 0.35
Fstop2: 0.55
Fpass2: 0.65
Apass1: 50
  Astop: 60
Apass2: 50
```

```
normalizefreq(d,false)
```

```
d
```

```
d =
```

```
      Response: 'Bandstop'
Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
Description: {7x1 cell}
NormalizedFrequency: false
      Fs: 1
      Fpass1: 0.125
      Fstop1: 0.175
      Fstop2: 0.275
      Fpass2: 0.325
      Apass1: 50
      Astop: 60
      Apass2: 50
```

## See Also

```
fdesign.lowpass, fdesign.halfband, fdesign.highpass,
fdesign.interpolator
```



**Purpose** Number of filter states in discrete-time or multirate filter

**Syntax** `n = nstates(hd)`  
`n = nstates(hm)`

**Description** **Discrete-Time Filters**

`n = nstates(hd)` returns the number of states `n` in the discrete-time filter `hd`. The number of states depends on the filter structure and the coefficients.

**Multirate Filters**

`n = nstates(hm)` returns the number of states `n` in the multirate filter `hm`. The number of states depends on the filter structure and the coefficients.

**Examples**

Check the number of states for two different filters, one a direct form FIR filter, the other a multirate filter.

```
h=firls(30,[0 .1 .2 .5]*2,[1 1 0 0])
```

```
hd=dfilt.dffir(h)
```

```
hd =
```

```

      FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'double'
      Numerator: [1x31 double]
PersistentMemory: 'on'
      States: [30x1 double]
```

```
n=nstates(hd)
```

```
n =
```

```
30
```

```
hm=mfilt.firfracdecim(2,3)
```

```
hm =
```

## nstates

---

```
FilterStructure: [1x46 char]
  Numerator: [1x72 double]
RateChangeFactors: [2 3]
PersistentMemory: false
  States: [35x1 double]
```

```
n=nstates(hm)
```

```
n =
```

```
35
```

### See Also

```
mfilt
```

**Purpose** Order of quantized 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
```

# phasedelay

---

**Purpose** Phase delay of discrete-time or multirate 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 the Signal Processing Toolbox.

## 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 the Signal Processing Toolbox documentation

# phasez

---

**Purpose** Unwrapped phase response for filter

**Syntax**

```
phasez(ha)
[phi,w] = phasez(ha,n)
[phi,w] = phasez(...,f)

phasez(hd)
[phi,w] = phasez(hd,n)
[phi,w] = phasez(...,f)

phasez(hm)
[phi,w] = phasez(hm,n)
[phi,w] = phasez(...,f)
[phi,w] = phasez(...,fs)
```

**Description** The following sections describe phasez operation for adaptive filters, discrete-time filters, and multirate filters. For more information about optional input arguments for phasez, refer to phasez in the Signal Processing Toolbox.

## 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.

# phasez

---

As an example, consider a 2-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. An equivalent single-stage filter with an overall interpolation factor of 8 can be found. `phasez` uses the equivalent filter for the analysis. If a sampling frequency `fs` is specified as an input argument to `phasez`, the function interprets `fs` as the rate at which the equivalent filter is running.

## See Also

`freqz`, `grpdelay`, `phasedelay`, `zerophase`, `zplane`

`freqz`, `fvtool`, `phasez` in the Signal Processing Toolbox documentation



**Purpose** Polyphase decomposition of multirate filter

**Syntax** `p = polyphase(hm)`  
`polyphase(hm)`

**Description** `p = polyphase(hm)` returns the polyphase matrix `p` of the multirate filter `hm`. Each row in the matrix represents one subfilter of the multirate filter. The first row of matrix `p` represents the first subfilter, the second row the second subfilter, and so on to the last subfilter.

`polyphase(hm)` called with no output argument launches the Filter Visualization Tool (FVTool) with all the polyphase subfilters to allow you to analyze each component subfilter individually.

**Examples** When you create a multirate filter that uses polyphase decomposition, `polyphase` lets you analyze the component filters individually by returning the components as rows in a matrix.

This example creates an interpolate by eight filter.

```
hm=mfilt.firinterp(8)
```

```
hm =
```

```

    FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
      Numerator: [1x192 double]
InterpolationFactor: 8
  PersistentMemory: false
           States: [23x1 double]
```

In this syntax, the matrix `p` contains all of the subfilters for `hm`, one filter per matrix row.

```
p=polyphase(hm)
```

```
p =
```

```
Columns 1 through 8
```

```

    0         0         0         0         0         0         0         0
-0.0000  0.0002 -0.0006  0.0013 -0.0026  0.0048 -0.0081  0.0133
-0.0001  0.0004 -0.0012  0.0026 -0.0052  0.0094 -0.0160  0.0261
-0.0001  0.0006 -0.0017  0.0038 -0.0074  0.0132 -0.0223  0.0361
```

# polyphase

```
-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
```

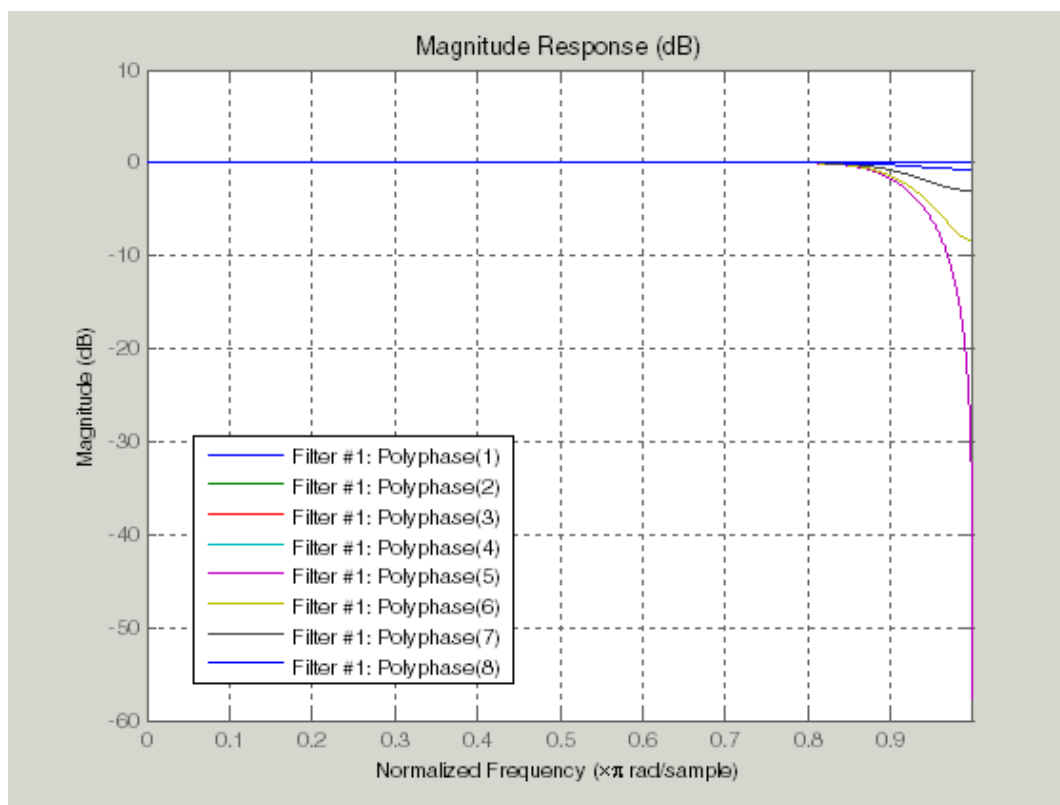
Columns 17 through 24

```
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 this figure, we switch FVTool to show the magnitude responses for the subfilters.



**See Also** `mfilt`

**Purpose** Results of most recent fixed-point filtering operation

**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 the Fixed-Point Toolbox 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 `fi`pref in the documentation for the Fixed-Point Toolbox in the online Help system.

---

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.  
`fi`pref('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** Realize Simulink subsystem block for quantized filter

**Syntax**  
`realizemdl(hq)`  
`realizemdl(hq, propertyname1, propertyvalue1, ...)`

**Description** `realizemdl(hq)` generates a model of filter `hq` in a Simulink subsystem block using sum, gain, and delay blocks from Simulink. The properties and values of `hq` define the resulting subsystem block parameters.

`realizemdl` requires Simulink. To accurately realize models of quantized filters, use Simulink Fixed-Point.

`realizemdl(hq,propertyname1,propertyvalue1,...)` generates the model or `hq` with the associated `propertyname/propertyvalue` pairs, and any other values you set in `hq`.

**Note** Subsystem filter blocks that you use `realizemdl` to create support sample-based input and output only. You cannot input or output frame-based signals with the block.

Using the optional `propertyname/propertyvalue` pairs lets you control more fully the way the block subsystem model gets built, such as where the block goes, what the name is, or how to optimize the block structure. Valid properties and values for `realizemdl` are listed in this table, with the default value noted and descriptions of what the properties do.

Property Name	Property Values	Description
Destination	'current' (default) or 'new'	Specify whether to add the block to your current Simulink model or create a new model to contain the block.
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 <code>propertyvalue</code> set to a string ' <i>blockname</i> '.

# realizemdl

Property Name	Property Values	Description
OverwriteBlock	'off' or 'on'	Specify whether to overwrite an existing block with the same name or create a new block.
OptimizeZeros	'off' (default) or 'on'	Specify whether to remove zero-gain blocks.
OptimizeOnes	'off' (default) or 'on'	Specify whether to replace unity-gain blocks with direct connections.
OptimizeNegOnes	'off' (default) or 'on'	Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.
OptimizeDelayChains	'off' (default) or 'on'	Specify whether to replace cascaded chains of delay blocks with a single integer delay block to provide an equivalent delay.

## Examples

To demonstrate how `realizemdl` works to create models, these two examples show the default and optional syntaxes in use. Both examples begin from a quantized filter designed by `butter` in the Signal Processing Toolbox.

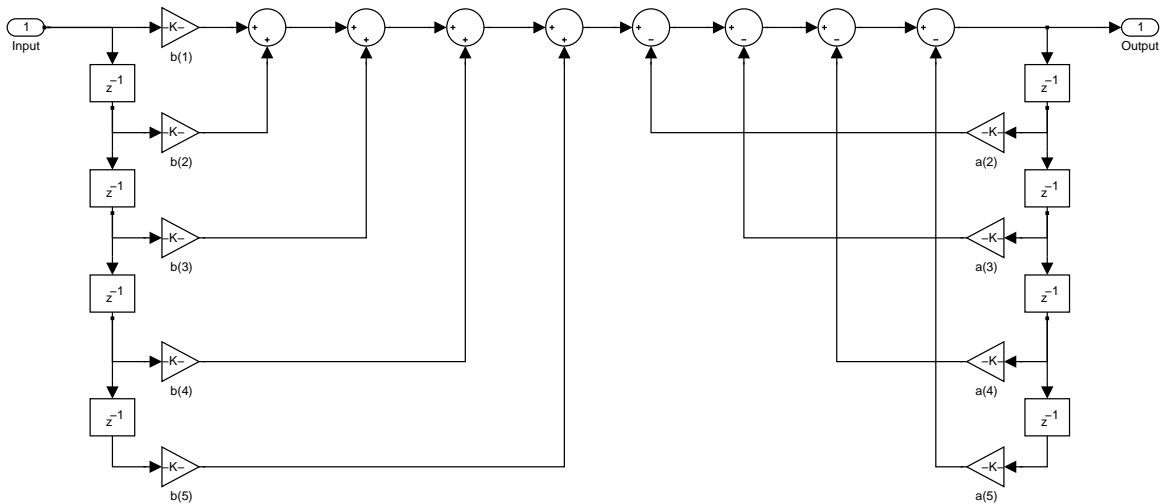
```
[b,a] = butter(4,.5);  
hq = dfilt.df1(b,a);
```

Example 1—Using the default syntax to realize a model of your quantized filter `hq`. When you use this syntax, `realizemdl` uses blocks from Simulink and Simulink Fixed-Point to realize the subsystem in your current Simulink model.

```
realizemdl(hq);
```

Look at the figure to see the model as realized by `realizemdl`.





Example 2—Using `propertyname/propertyvalue` pairs to specify the features of the subsystem block model created by `realizemdl`.

First, convert the filter to fixed-point arithmetic to ensure a few zero valued coefficients:

```
hq.arithmetic = 'fixed';
```

Your filter has two zero value denominators, `a(2)` and `a(4)`:

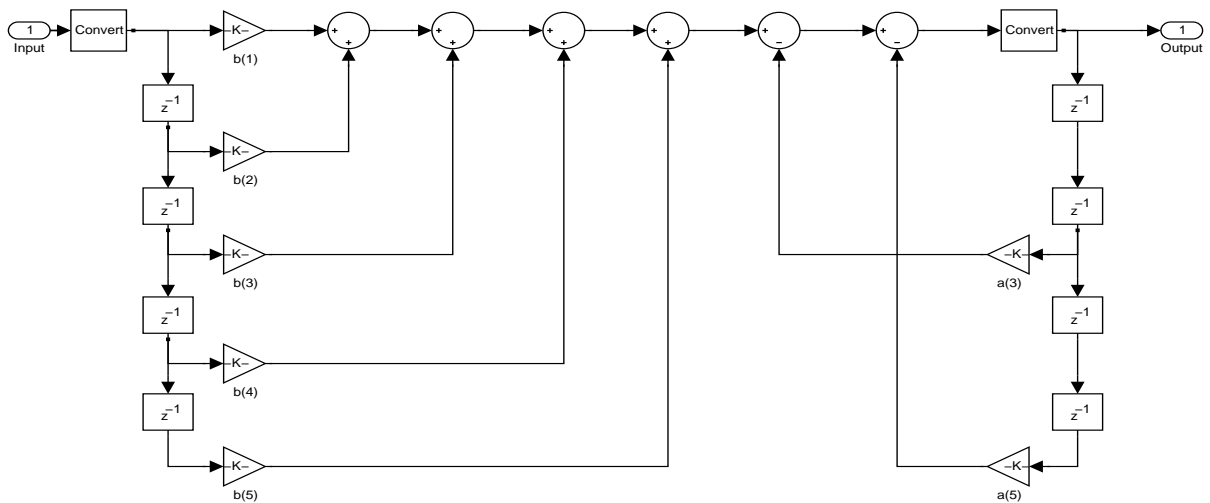
```
FilterStructure: 'Direct-Form I'
  Arithmetic: 'fixed'
  Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]
  Denominator: [1 0 0.4860 0 0.0176]
  PersistentMemory: false
  States: Numerator: [4x1 fi]
         Denominator:[4x1 fi]
```

Now realize the model implementation.

```
realizemdl(hq, 'optimizezeros', 'on', ...
  'blockname', 'newfiltermodel');
```

# realizemdl

Since this example uses the optional property name `optimizezeros`, set to 'on', the resulting block subsystem is slightly different—the zero-gain blocks for coefficients  $a(2)$  and  $a(4)$  are not included in the subsystem.



## See Also

`realizemdl` under the methods for `dfilt` in the Signal Processing Toolbox

**Purpose** Double-precision floating-point reference filter that corresponds to fixed-point or single-precision floating-point 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.

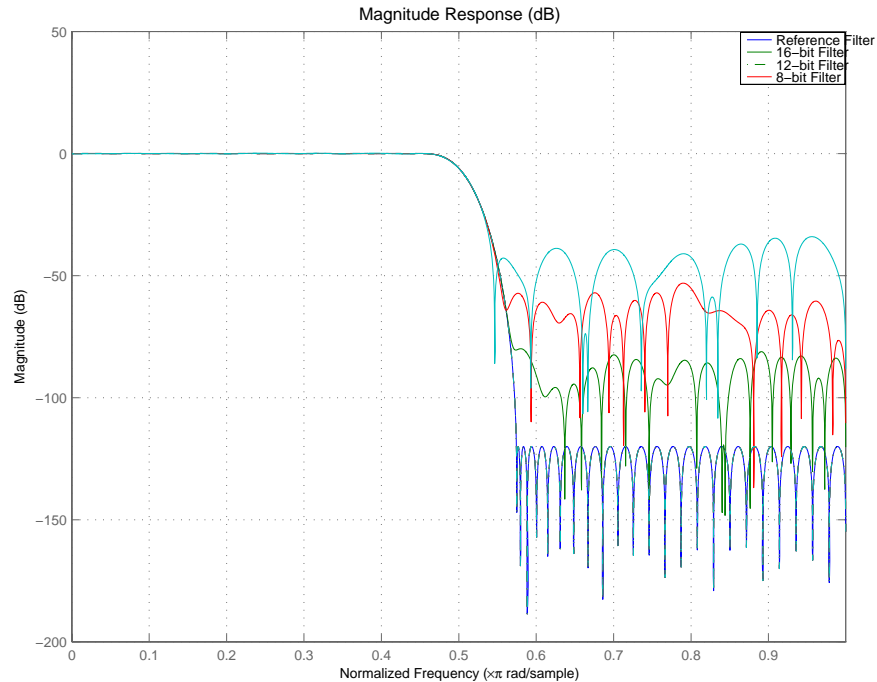
```
h = dfilt.dffir(firceqrip(87,.5,[1e-3,1e-6])); % Lowpass filter.
h1 = copy(h); h2 = copy(h); % Create copies of h.
h.arithmetic = 'fixed'; % Set h to filter using fixed-point
                        % arithmetic.
h1.arithmetic = 'fixed'; % Same for h1.
h2.arithmetic = 'fixed'; % Same for h2.
h.CoeffWordLength = 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.
```

# reffilter

```
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 second-order sections (SOS) filter

**Syntax**

```
reorder(hd,order)
reorder(hd,numorder,denorder)
reorder(hd,numorder,denorder,svorder)
reorder(hd,filter_type)
reorder(hd,dir_flag)
reorder(hd,dir_flag,sv)
```

**Description** `reorder(hd,order)` rearranges the sections of filter `hd` using the vector of indices provided in `order`.

`order` does not need to contain all of the indices of the filter. Omitting one or more filter section indices removes the omitted sections from the filter. You can use a logical array to remove sections from the filter, but not to reorder it (refer to the Examples to see this done).

`reorder(hd,numorder,denorder)` reorders the numerator and denominator separately using the vectors of indices in `numorder` and `denorder`. These two vectors must be the same length.

`reorder(hd,numorder,denorder,svorder)` the scale values can be independently reordered. When `svorder` is not specified, the scale values are reordered with the numerator. The output scale value always remains on the end when you use the argument `numorder` to reorder the scale values.

`reorder(hd,filter_type)` where `filter_type` is one of `auto`, `lowpass`, `highpass`, `bandpass`, or `bandstop`, reorders `hd` in a way suitable for the filter type you specify by `filter_type`. This reordering mode can be especially helpful for fixed-point implementations where the order of the filter sections can significantly affect your filter performance.

The `auto` option and automatic ordering only apply to filters that you used `fdesign` to create. With the `auto` option as an input argument, `reorder` automatically rearranges the filter sections depending on the specification response type of the design, such as `lowpass`, or `bandstop`. This technique appears in the first example.

`reorder(hd,dir_flag)` if `dir_flag` is `up`, the first filter section contains the poles closest to the origin, and the last section contains the poles closest to the

# reorder

---

unit circle. When `dir_flag` is down, the sections are ordered in the opposite direction. `reorder` always pairs zeros with the poles closest to them.

`reorder(hd, dir_flag, sv)` `sv` is either the string poles or zeros and describes how to reorder the scale values. By default the scale values are not reordered when you use the `dir_flag` option.

## Examples

Being able to rearrange the order of the sections in a filter can be a powerful tool for controlling the filter process. This example uses `reorder` to change the sections of a `df2sos` filter. Let `reorder` do the reordering automatically in the first example. In the second, use `reorder` to specify the new order for the sections.

First use the automatic reordering option on a lowpass filter.

```
d = fdesign.lowpass('n,f3db',15,0.75)
hd = design(d,'butter');
d =

    Response: 'Lowpass'
  Specification: 'N,F3dB'
  Description: {'Filter Order';'3dB Frequency'}
NormalizedFrequency: true
  FilterOrder: 15
      F3dB: 0.75

reorder(hd,'auto')
hd

hd =

    FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
      sosMatrix: [8x6 double]
      ScaleValues: [9x1 double]
  PersistentMemory: false
```

The SOS matrices show the reordering.

```
hd.sosMatrix
```

```
ans =
```

```

1.0000    2.0000    1.0000    1.0000    1.3169    0.8623
1.0000    2.0000    1.0000    1.0000    1.1606    0.6414
1.0000    2.0000    1.0000    1.0000    1.0448    0.4776
1.0000    2.0000    1.0000    1.0000    0.9600    0.3576
1.0000    2.0000    1.0000    1.0000    0.8996    0.2722
1.0000    2.0000    1.0000    1.0000    0.8592    0.2151
1.0000    2.0000    1.0000    1.0000    0.8360    0.1823
1.0000    1.0000         0    1.0000    0.4142         0

```

```
hdreorder.sosMatrix
```

```
ans =
```

```

1.0000    2.0000    1.0000    1.0000    1.0448    0.4776
1.0000    2.0000    1.0000    1.0000    0.8360    0.1823
1.0000    2.0000    1.0000    1.0000    0.8996    0.2722
1.0000    2.0000    1.0000    1.0000    1.3169    0.8623
1.0000    2.0000    1.0000    1.0000    0.9600    0.3576
1.0000    1.0000         0    1.0000    0.4142         0
1.0000    2.0000    1.0000    1.0000    0.8592    0.2151
1.0000    2.0000    1.0000    1.0000    1.1606    0.6414

```

For another example of using reorder, create an SOS filter in the direct form II implementation.

```

[z,p,k] = butter(15,.5);
[sos, g] = zp2sos(z,p,k);
hd = dfilt.df2sos(sos,g);

```

Reorder the sections by moving the second section to be between the seventh and eighth sections.

```

reorder(hd, [1 3:7 2 8]);
hfvf = 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(hfvf, 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

## Reference

Schlichthärle, Dietrich, *Digital Filters Basics and Design*, Springer-Verlag Berlin Heidelberg, 2000.



---

<b>Purpose</b>	Reset filter properties to initial conditions
<b>Syntax</b>	<pre>reset(ha) reset(hd) reset(hm)</pre>
<b>Description</b>	<p><code>reset(ha)</code> resets all the properties of the adaptive filter <code>ha</code> 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.</p> <p><code>reset(hd)</code> resets all the properties of the discrete-time filter <code>hd</code> to their factory values that are modified when you run the filter. In particular, the <code>States</code> property is reset to zero.</p> <p><code>reset(hm)</code> resets all the properties of the multirate filter <code>hm</code> to their factory value that are modified when the filter is run. In particular, the <code>States</code> property is reset to zero when <code>hm</code> is a decimator. Additionally, the filter internal properties are also reset to their factory values.</p>
<b>Examples</b>	<p>Denoise a sinusoid and reset the filter after filtering with it.</p> <pre>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)</pre>
<b>See Also</b>	<code>quantizer</code> , <code>set</code>

# scale

---

**Purpose** Scale sections of second-order sections (SOS) filter

**Syntax**

```
scale(hd)
scale(hd,pnorm)
scale(hd,pnorm,p1v,p2,v2, )
scale(hd,pnorm,opts)
```

**Description** `scale(hd)` scales the second-order section filter `hd` using peak magnitude response scaling (L-infinity, `Linf`), to reduce the possibility of overflows when your filter `hd` operates in fixed-point arithmetic mode.

`scale(hd,pnorm)` specifies the norm used to scale the filter. `pnorm` can be either a discrete-time-domain norm or a frequency-domain norm.

Valid time-domain norm values for `pnorm` are `l1`, `l2`, and `linf`. Valid frequency-domain norm values are `L1`, `L2`, and `Linf`. Note that `L2` norm is equal to `l2` norm (by Parseval's theorem) but this is not true for other norms—`l1` is not the same as `L1` and `Linf` is not the same as `linf`.

Filter norms can be ordered in terms of how stringent they are, as follows from most stringent to least:

```
l1 >= Linf >= L2 = l2 >= L1 >= linf
```

Using `l1`, the most stringent scaling, produces a filter that is least likely to overflow, but has the worst signal-to-noise ratio performance. `Linf` scaling, the least stringent, and the default scaling, is the most commonly used scaling norm.

`scale(hd,pnorm,p1,v1,p2,v2,...)` uses parameter name/parameter value pair input arguments to specify optional scaling parameters. Valid parameter names and options values appear in the table.

<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)
```

**See Also**

cumsec, norm, reorder, scalecheck, scaleopts

# scalecheck

---

**Purpose** Check scaling of a second-order sections (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.

```
hs = fdesign.lowpass; % Create a filter design specifications
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** Create object containing scaling options for second-order sections (SOS) scaling

**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 (Continued)	Default	Description and Valid Value
OverflowMode	wrap	Sets the way the filter handles arithmetic overflow situations during scaling. Choose either wrap or saturate
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 single-rate and multirate filters for integer filtering

**Syntax**

```
set2int(h)
set2int(h,coeffwl)
set2int(...,inwl)
g = set2int(...)
```

**Description** These sections apply to both discrete-time (`dfilt`) and multirate (`mfilt`) filters.

`set2int(h)` scales the filter coefficients to integer values and sets the filter coefficient and input fraction lengths to zero.

`set2int(h,coeffwl)` uses the number of bits specified by `coeffwl` as the word length it uses to represent the filter coefficients.

`set2int(...,inwl)` uses the number of bits specified by `coeffwl` as the word length it uses to represent the filter coefficients and the number of bits specified by `inwl` as the word length to represent the input data.

`g = set2int(...)` returns the gain `g` introduced into the filter by scaling the filter coefficients to integers. `g` is always calculated to be a power of 2.

---

**Note** `set2int` does not work with CIC decimators or interpolators because they do not have coefficients.

---

**Examples** These examples demonstrate some uses and ideas behind `set2int`.

The second parts of both examples depend on the following—after you filter a set of data, the input data and output data cover the same range of values, unless the filter process introduces gain in the output. Converting your filter object to integer form, and then filtering a set of data, does introduce gain into the system. When the examples refer to resetting the output to the same range as the input, the examples are accounting for this added gain feature.

## Discrete-Time Filter Example

Two parts comprise this example. Part 1 compares the step response of an FIR filter in both the fractional and integer filter modes. Fractional mode filtering

is essentially the opposite of integer mode. Integer mode uses a filter which has coefficients represented by integers. Fractional mode filters have coefficients represented in fractional form (nonzero fraction length).

```
b = firrcos(100,.25,.25,2,'rolloff','sqrt');
hd = dfilt.dffir(b);
hd.Arithmetic = 'fixed';
hd.InputFracLength = 0; % Integer inputs.
x = ones(100,1);
yfrac = filter(hd,x); % Fractional mode output.
g = set2int(hd); % Convert to integer coefficients.
yint = filter(hd,x); % Integer mode output.
```

Note that `yint` and `yfrac` are `fi` objects. Later in this example, you use the `fi` object properties `WordLength` and `FractionLength` to work with the output data.

Now use the gain `g` to rescale the output from the integer mode filter operation.

```
yints = double(yint)/g;
```

Verify that the scaled integer output is equal to the fractional output.

```
max(abs(yints-double(yfrac)))
```

In part 2, the example reinterprets the output binary data, putting the input and the output on the same scale by weighting the most significant bits in the input and output data equally.

```
WL = yint.WordLength;
FL = yint.Fractionlength + log2(g);
yints2 = fi(zeros(size(yint)),true,WL,FL);
yints2.bin = yint.bin;
max(abs(double(yints2)-double(yfrac)))
```

### Multirate Filter Example

This two-part example starts by comparing the step response of a multirate filter in both fractional and integer modes. Fractional mode filtering is essentially the opposite of integer mode. Integer mode uses a filter which has coefficients represented by integers. Fractional mode filters have coefficients in fractional form with nonzero fraction lengths.

```
hm = mfilt.firinterp;
```

## set2int

---

```
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`

<b>Purpose</b>	Set specifications for filter specification object
<b>Syntax</b>	<pre>setspecs(d,specvalue1,specvalue2,...) setspecs(d,Specification,specvalue1,specvalue2,...) setspecs(...,fs) setspecs(...,inputunits)</pre>
<b>Description</b>	<p><code>setspecs(d,specvalue1,specvalue2,...)</code> Set the specifications in the order that they appear in the <code>Specification</code> property for the design object <code>d</code>.</p> <p><code>setspecs(d,Specification,specvalue1,specvalue2,...)</code> 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 <code>Specification</code> string and the associated specification values for the object, rather than recreating the object to change it.</p> <p><code>setspecs(...,fs)</code> Set the <code>fs</code>. If you choose to specify the <code>fs</code>, it must be immediately after you provide all of the specifications for the current <code>Specification</code>. Refer to Examples to see this being used.</p> <p><code>setspecs(...,inputunits)</code> Specifying the <code>inputunits</code> option allows you to specify your filter magnitude specification values in different units. <code>inputunits</code> can be either of these strings:</p> <ul style="list-style-type: none"><li>• <b>'linear'</b>—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.</li><li>• <b>'squared'</b>—indicating that your input specification values represent squared magnitude values, usually dB. This is the default value. When you omit the <code>inputunits</code> argument, <code>setspecs</code> assumes all specification values are in square magnitude form.</li></ul> <p>You are not required to provide <code>fs</code>, the sampling frequency, as an input when you use the <code>inputunits</code> option. As you see from the syntax options, the <code>inputunits</code> option must be the rightmost input argument in the syntax—<code>inputunits</code> must be passed as the final input.</p>
<b>Examples</b>	To demonstrate using <code>setspecs</code> , the following examples show how to use various syntax forms to set the values in filter specifications objects.

## Example 1

Create a lowpass design object `d` using filter order and a cutoff value for the location of the edge of the passband. Then change the cutoff and order specifications of `d`.

```
d = fdesign.lowpass('n,fc')

d =

    ResponseType: 'Lowpass with cutoff'
    Specification: 'N,Fc'
    Description: {2x1 cell}
    NormalizedFrequency: true
                Fs: 'Normalized'
    FilterOrder: 10
    Fcutoff: 0.5000

setspecs(d, 20, .4);

d =

    ResponseType: 'Lowpass with cutoff'
    Specification: 'N,Fc'
    Description: {2x1 cell}
    NormalizedFrequency: true
                Fs: 'Normalized'
    FilterOrder: 20
    Fcutoff: 0.4000
```

## Example 2

Now specify a sampling frequency after you make `d`.

```
d = fdesign.lowpass('n,fc')

d =

    ResponseType: 'Lowpass with cutoff'
    Specification: 'N,Fc'
    Description: {2x1 cell}
    NormalizedFrequency: true
```

```
                Fs: 'Normalized'
FilterOrder: 10
                Fcutoff: 0.5000

setspecs(d, 20, 4, 20);

d

d =

        ResponseType: 'Lowpass with cutoff'
        Specification: 'N,Fc'
        Description: {2x1 cell}
NormalizedFrequency: false
                Fs: 20
        FilterOrder: 20
        Fcutoff: 4
```

### Example 3

This example uses the `inputunits` argument to change from the default setting of square to linear unit. Start with the default lowpass design object that specifies the edge locations for the passband and stopband, and the desired attenuation in the pass- and stopbands.

```
d=fdesign.lowpass

d =

        ResponseType: 'Minimum-order lowpass'
        Specification: 'Fp,Fst,Ap,Ast'
        Description: {4x1 cell}
NormalizedFrequency: true
                Fs: 'Normalized'
                Fpass: 0.4500
                Fstop: 0.5500
                Apass: 1
                Astop: 60
```

Convert to linear input values and reset the filter spec for d at the same time. With the linear argument included, the inputs for the response features now need to be in linear units.

```
setspecs(d, .4, .5, .1, .05, 'linear')
d

d =

    ResponseType: 'Minimum-order lowpass'
    Specification: 'Fp,Fst,Ap,Ast'
    Description: {4x1 cell}
    NormalizedFrequency: true
                Fs: 'Normalized'
                Fpass: 0.4000
                Fstop: 0.5000
                Apass: 1.7430
                Astop: 26.0206
```

## Example 4

Finally, use setspecs to change the Specification string and apply new filter specifications to d.

```
d=fdesign.decim(3)
d =

    ResponseType: 'Minimum-order nyquist'
    Specification: 'TW,Ast'
    Description: {2x1 cell}
    DecimationFactor: 3
    NormalizedFrequency: true
                Fs: 'Normalized'
    TransitionWidth: 0.1000
                Astop: 80

setspecs(d, 'n,ast',16,70)
d

d =

    ResponseType: 'Nyquist with filter order and stopband attenuation'
    Specification: 'N,Ast'
    Description: {2x1 cell}
    DecimationFactor: 3
```



```
NormalizedFrequency: true  
                    Fs: 'Normalized'  
PolyphaseLength: 16  
Astop: 70
```

**See Also**

`designmethods`, `fdesign.bandpass`, `fdesign.bandstop`, `fdesign.decimator`,  
`fdesign.halfband`, `fdesign.highpass`, `fdesign.interpolator`,  
`fdesign.lowpass`, `fdesign.nyquist`, `fdesign.rsrc`

# SOS

---

**Purpose** Convert quantized filter to second-order sections (SOS) form, order, and scaling

**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)
```

**See Also**

`convert`, `dfilt`

`tf2sos` in your Signal Processing Toolbox documentation

# specifyall

**Purpose** Access fixed-point scaling modes and features in direct-form FIR filter object

**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 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 = firband(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2], {'w' 'c'});
hd = dfilt.dffir(b);
hd.arithmetic = 'fixed'
hd =
```

```
    FilterStructure: 'Direct-Form FIR'
      Arithmetic: 'fixed'
      Numerator: [1x13 double]
PersistentMemory: false
      States: [1x1 embedded.fi]
```

```
    CoeffWordLength: 16
      CoeffAutoScale: 'true'
      Signed: 'on'
```

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

```
    OutputWordLength: 16
      OutputMode: 'AvoidOverflow'
```

```
      ProductMode: 'FullPrecision'
```

```
        AccumMode: 'KeepMSB'
    AccumWordLength: 40
      CastBeforeSum: 'on'
```

```
        RoundMode: 'convergent'
      OverflowMode: 'wrap'
```

```
    InheritSettings: 'off'
```

```
specifyall(hd)
hd
```

```
hd =  
  
    FilterStructure: 'Direct-Form FIR'  
        Arithmetic: 'fixed'  
        Numerator: [1x13 double]  
PersistentMemory: false  
        States: [1x1 embedded.fi]  
  
    CoeffWordLength: 16  
    CoeffAutoScale: false  
    NumFracLength: 16  
    Signed: true  
  
    InputWordLength: 16  
    InputFracLength: 15  
  
    OutputWordLength: 16  
        OutputMode: 'SpecifyPrecision'  
    OutputFracLength: 11  
  
        ProductMode: 'SpecifyPrecision'  
    ProductWordLength: 32  
    ProductFracLength: 31  
  
        AccumMode: 'SpecifyPrecision'  
    AccumWordLength: 40  
    AccumFracLength: 31  
    CastBeforeSum: true  
  
        RoundMode: 'convergent'  
    OverflowMode: 'wrap'  
  
    InheritSettings: false
```

The mode properties `InputMode`, `ProductMode`, and `AccumMode` now have the value `SpecifyPrecision` and the fraction length properties appear in the display. Now you use the properties (`InputFracLength`, `ProdFracLength`, `AccumFracLength`) to set the precision the filter applies to the input, product, and accumulator operations. `CoeffAutoScale` switches to `false`, meaning

autoscaling of the filter coefficients will not be done to prevent overflows. None of the other filter properties change when you apply `specifyall`.

## See Also

`double`, `refilter`  
`fi`, `fimath` in the Fixed-Point Toolbox

# stepz

---

**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 the Signal Processing Toolbox.

## 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** Convert transfer function to coupled allpass

**Syntax**

```
[d1,d2] = tf2ca(b,a)
[d1,d2] = tf2ca(b,a)
[d1,d2,beta] = 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

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

representing the generalized allpass decomposition.

In the above equations,  $H1(z)$  and  $H2(z)$  are real or complex allpass IIR filters given by

$$H1(z) = \frac{\text{fliplr}(\overline{D1(z)})}{D1(z)}, H2(z) = \frac{\text{fliplr}(\overline{D2(z)})}{D2(z)}$$

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, c12tf, iirpowcomp, latc2tf, tf2latc

# tf2cl

---

**Purpose** Convert transfer function to coupled allpass lattice

**Syntax** [k1,k2] = tf2cl(b,a)  
[k1,k2] = tf2cl(b,a)

**Description** [k1,k2] = tf2cl(b,a) where b is a real, symmetric vector of numerator coefficients and a is a real vector of denominator coefficients, corresponding to a stable digital filter, will perform the coupled allpass decomposition

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

of a stable IIR filter  $H(z)$  and convert the allpass transfer functions  $H1(z)$  and  $H2(z)$  to a coupled lattice allpass structure with coefficients given in vectors k1 and k2.

[k1,k2] = tf2cl(b,a) where b is a real, antisymmetric vector of numerator coefficients and a is a real vector of denominator coefficients, corresponding to a stable digital filter, performs the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

of a stable IIR filter  $H(z)$  and converts the allpass transfer functions  $H1(z)$  and  $H2(z)$  to a coupled lattice allpass structure with coefficients given in vectors k1 and k2.

In some cases, the decomposition is not possible with real  $H1(z)$  and  $H2(z)$ . In those cases, a generalized coupled allpass decomposition may be possible, using the command syntax

$$[k1,k2,beta] = tf2cl(b,a)$$

to perform the generalized allpass decomposition of a stable IIR filter  $H(z)$  and convert the complex allpass transfer functions  $H1(z)$  and  $H2(z)$  to corresponding lattice allpass filters

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

where beta is a complex scalar of magnitude equal to 1.

---

**Note** Coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to *Signal Processing Toolbox User's Guide*.

---

## Examples

```
[b,a]=cheby1(9,.5,.4);
[k1,k2]=tf2cl(b,a); % Get the reflection coeffs. for the lattices.
[num1,den1]=latc2tf(k1,'allpass'); % Convert each allpass lattice
[num2,den2]=latc2tf(k2,'allpass'); % back to transfer function.
num = 0.5*conv(num1,den2)+0.5*conv(num2,den1);
den = conv(den1,den2); % Reconstruct numerator and 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

# window

---

**Purpose** Design FIR filter using windowed impulse response method

**Syntax**  
`h = window(d,fcnhdl,fcnarg)`  
`h = window(d, win)`

**description** `h = window(d,fcnhdl,fcnarg)` designs an FIR filter using the specifications in filter specification object `d`. Depending on the specification type of `d`, the returned filter is either a single-rate digital filter—a `dfilt`, or a multirate digital filter—an `mfilt`.

`fcnhdl` is a handle to a filter design function that returns a window vector, such as the `hamming` or `blackman` functions. `fcnarg` is an optional argument that returns a window. You pass the function to `window`. Refer to example 1 below to see the function argument used to design the filter.

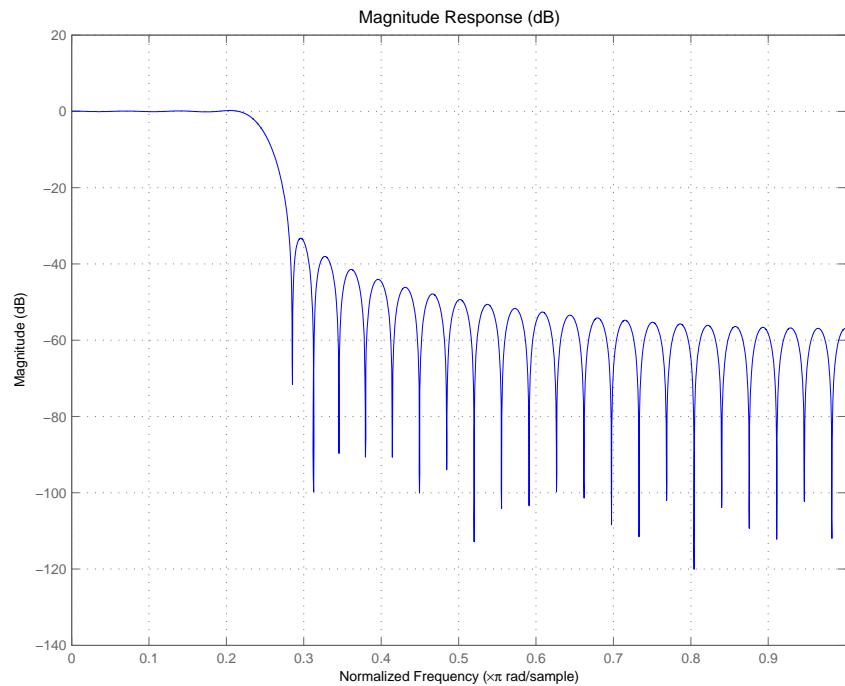
`h = window(d,win)` designs a filter using the vector you supply in `win`. The length of vector `win` must be the same as the impulse response of the filter, which is equal to the filter order plus one. Example 2 shows this being done.

**Examples** These examples design filters using the two design techniques of specifying a function handle or passing a window vector as an input argument.

## Example 1

Use a function handle and optional input arguments to design a multirate filter. We use a function handle to the function `Kaiser` to provide the window. Since this example creates a decimating filter specifications object, `window` returns a multirate filter.

```
d = fdesign.decim(4,'p1',14);  
hm = window(d,@kaiser,2.5);  
fvtool(hm)
```

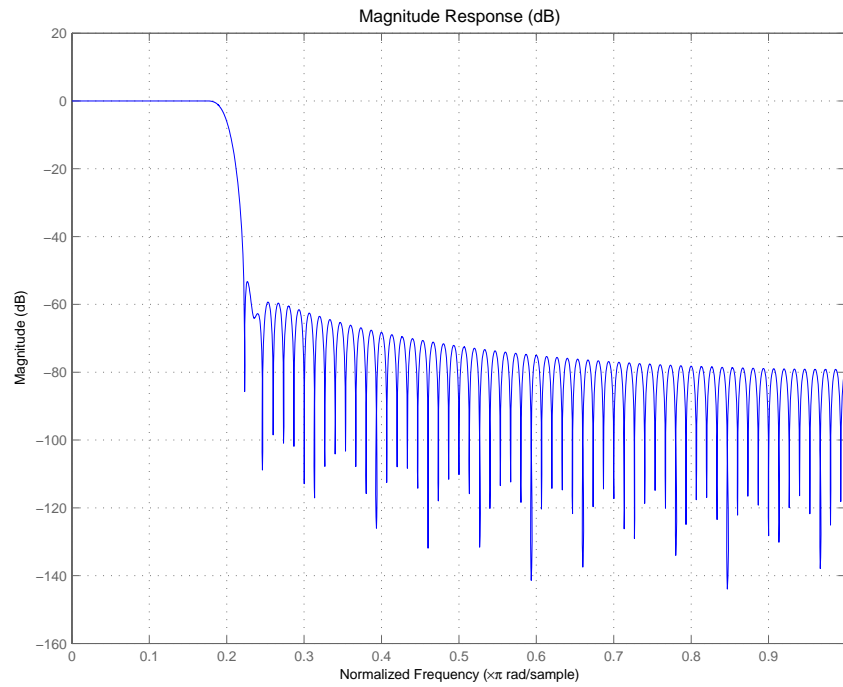


## Example 2

Use a window vector provided by the hamming window design function. For this example, the design object is a Nyquist filter, thus window returns `hd` as a discrete-time filter.

```
d = fdesign.nyquist(5, 'n', 150);  
hd = window(d, hamming(151));  
fvtool(hd)
```

# 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 the Signal Processing Toolbox.

### Adaptive Filters

For adaptive filters, zerophase returns the instantaneous zero-phase response based on the current filter coefficients.

zerophase(ha) displays the zero-phase response of ha in the Filter Visualization Tool (FVTool).

[hr,w] = zerophase(ha,n) returns length n vectors hr and w containing the instantaneous zero-phase response of the adaptive filter ha, and the frequencies in radians at which zerophase evaluated the response. The zero-phase response is evaluated at n points equally spaced around the upper half of the unit circle. For an FIR filter where n is a power of two, the computation is done faster using FFTs. If n is not specified, it defaults to 8192.

[hr,w] = zerophase(ha) returns a matrix hr if ha is a vector of filters. Each column of the matrix corresponds to each filter in the vector. If you provide a row vector of frequency points f as an input argument, each row of hr corresponds to one filter in the vector.

## Discrete-Time Filters

`zerophase(hd)` displays the zero-phase response of `hd` in the Filter Visualization Tool (FVTool).

`[hr,w] = zerophase(hd,n)` returns length `n` vectors `hr` and `w` containing the instantaneous zero-phase response of the adaptive filter `hd`, and the frequencies in radians at which `zerophase` evaluated the response. The zero-phase response is evaluated at `n` points equally spaced around the upper half of the unit circle. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs. If `n` is not specified, it defaults to 8192.

`[hr,w] = zerophase(hd)` returns a matrix `hr` if `hd` is a vector of filters. Each column of the matrix corresponds to each filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `hr` corresponds to one filter in the vector.

## Multirate Filters

`zerophase(hm)` displays the zero-phase response of `hd` in the Filter Visualization Tool (FVTool).

`[hr,w] = zerophase(hm,n)` returns length `n` vectors `hr` and `w` containing the instantaneous zero-phase response of the adaptive filter `hm`, and the frequencies in radians at which `zerophase` evaluated the response. The zero-phase response is evaluated at `n` points equally spaced around the upper half of the unit circle. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs. If `n` is not specified, it defaults to 8192.

`[hr,w] = zerophase(hm)` returns a matrix `hr` if `hm` is a vector of filters. Each column of the matrix corresponds to each filter in the vector. If you provide a row vector of frequency points `f` as an input argument, each row of `hr` corresponds to one filter in the vector.

Note that the response is computed relative to the rate at which the filter is running. If a sampling frequency is specified, it is assumed that the filter is running at that rate.

Note that the multirate filter delay response is computed relative to the rate at which the filter is running. When you specify `fs` (the sampling rate) as an input argument, `zerophase` assumes the filter is running at that rate.

For multistage cascades, `zerophase` forms a single-stage multirate filter that is equivalent to the cascade and computes the response relative to the rate at which the equivalent filter is running. `zerophase` does not support all multistage cascades. Only cascades for which it is possible to derive an equivalent single-stage filter are allowed for analysis.

As an example, consider a two-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. An equivalent single-stage filter with an overall interpolation factor of 8 can be found. `zerophase` uses the equivalent filter for the analysis. If a sampling frequency `fs` is specified as an input argument to `zerophase`, the function interprets `fs` as the rate at which the equivalent filter is running.

## See Also

`freqz`, `fvtool`, `grpdelay`, `impz`, `mfilt`, `phasez`, `zerophase`, `zplane`

# zpkbpc2bpc

---

**Purpose** Zero-pole-gain complex bandpass frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)`

**Description** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places two features of an original filter, located at frequencies  $W_{o1}$  and  $W_{o2}$ , at the required target frequency locations,  $W_{t1}$ , and  $W_{t2}$  respectively. It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

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

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

**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

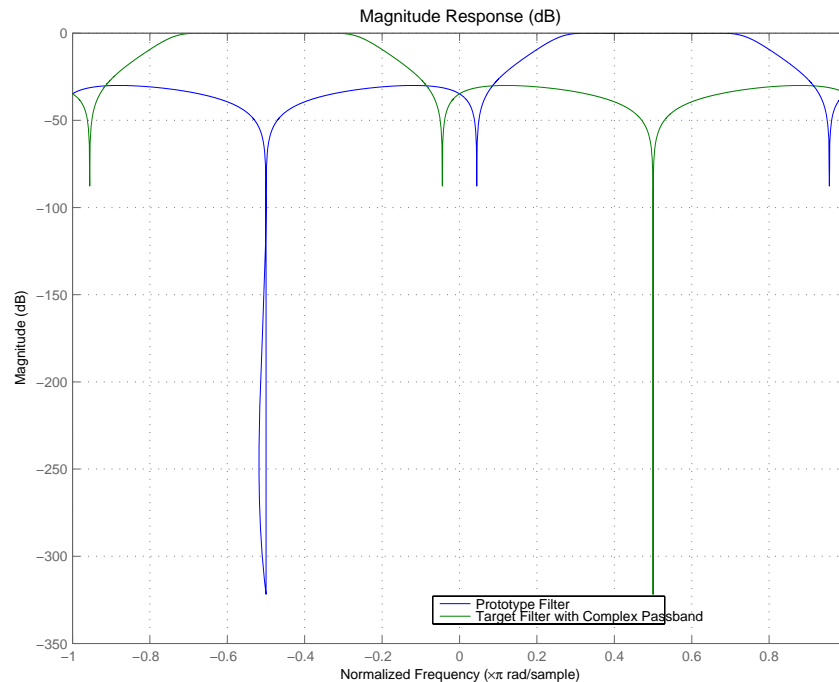
Create a complex passband from 0.25 to 0.75:

```
[b, a] = iir1p2bpc(b,a,0.5,[0.25,0.75]);  
z = roots(b);  
p = roots(a);  
k = b(1);  
[z2,p2,k2] = zpkbpc2bpc(z, p, k, [0.25, 0.75], [-0.75, -0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Comparing the filters in FVTool shows the example results. Use the features in FVTool to check the filter coefficients, or other filter analyses.



## Arguments

Z  
Zeros of the prototype lowpass filter

P  
Poles of the prototype lowpass filter

K  
Gain factor of the prototype lowpass filter

# zpkbpc2bpc

---

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

zpkftransf, allpassbpc2bpc, iirbpc2bpc

**Purpose** Zero-pole-gain frequency transformation of digital filter

**Syntax** `[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)`

**Description** `[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the transformed lowpass digital filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ . If `AllpassDen` is not specified it will default to 1. If neither `AllpassNum` nor `AllpassDen` is specified, then the function returns the input filter.

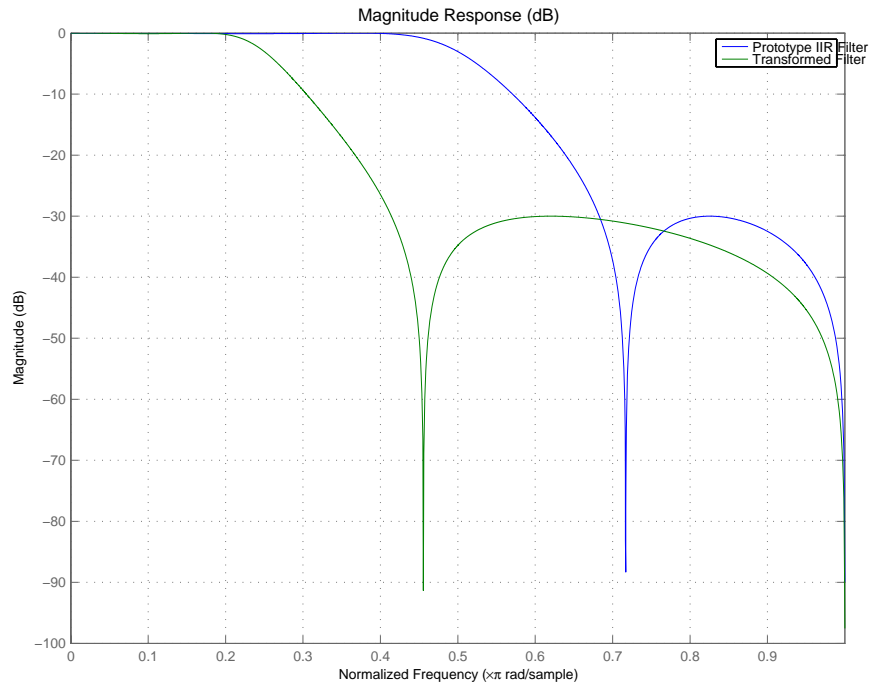
**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);  
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);  
[z2, p2, k2] = zpkftransf(roots(b),roots(a),b(1),AlpNum,AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

After transforming the filter, you get the response shown in the figure, where the passband has been shifted towards zero.



## Arguments

- 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

# zpk1p2bp

---

**Purpose** Zero-pole-gain lowpass to bandpass frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bp(Z,P,K,Wo,Wt)`

**Description** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bp(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $-W_0$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_0$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . This transformation implements the “DC Mobility,” which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of  $W_t$ .

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

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

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

**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);  
z = roots(b);  
p = roots(a);  
k = b(1);
```

```
[z2,p2,k2] = 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

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

zpkftransf, allpass1p2bp, iirlp2bp

## References

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

[2] 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.

[3] 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.

[4] 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 bandpass frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bpc(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bpc(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 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 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_0</math>, at the required target frequency location, <math>W_{t1}</math>, and the second feature, originally at <math>+W_0</math>, at the new location, <math>W_{t2}</math>. It is assumed that <math>W_{t2}</math> is greater than <math>W_{t1}</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>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.</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>

# zpk1p2bpc

---

```
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

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter. It should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Wt

Desired frequency locations in the transformed target filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

## See Also

zpkftransf, allpass1p2bpc, iir1p2bpc

<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_0</math>, at the required target frequency location, <math>W_{t1}</math>, and the second feature, originally at <math>+W_0</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_0</math> and <math>W_{ts}</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 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>
<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); [z2,p2,k2] = zpk1p2bs(z, p, k, 0.5, [0.2 0.3]);</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>

# zpklp2bs

---

## Arguments

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

zpkftransf, allpasslp2bs, iirlp2bs

## References

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

[2] 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.



[3] 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.

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

# zpk1p2bsc

---

**Purpose** Zero-pole-gain lowpass to complex bandstop frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bsc(Z,P,K,Wo,Wt)`

**Description** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2bsc(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . Additionally the transformation swaps passbands with stopbands in the target filter.

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

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

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators.

**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);  
z = roots(b);  
p = roots(a);  
k = b(1);
```

```
[z2,p2,k2] = 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

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Wt

Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

## See Also

zpkftransf, allpass1p2bsc, iir1p2bsc

**Purpose** Zero-pole-gain lowpass to highpass frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2hp(Z,P,K,Wo,Wt)`

**Description** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2hp(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real highpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency  $W_o$ , at the required target frequency location,  $W_t$ , at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and the gain factor,  $K$ .

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

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

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = 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

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

zpkftransf, allpass1p2hp, iir1p2hp

## References

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

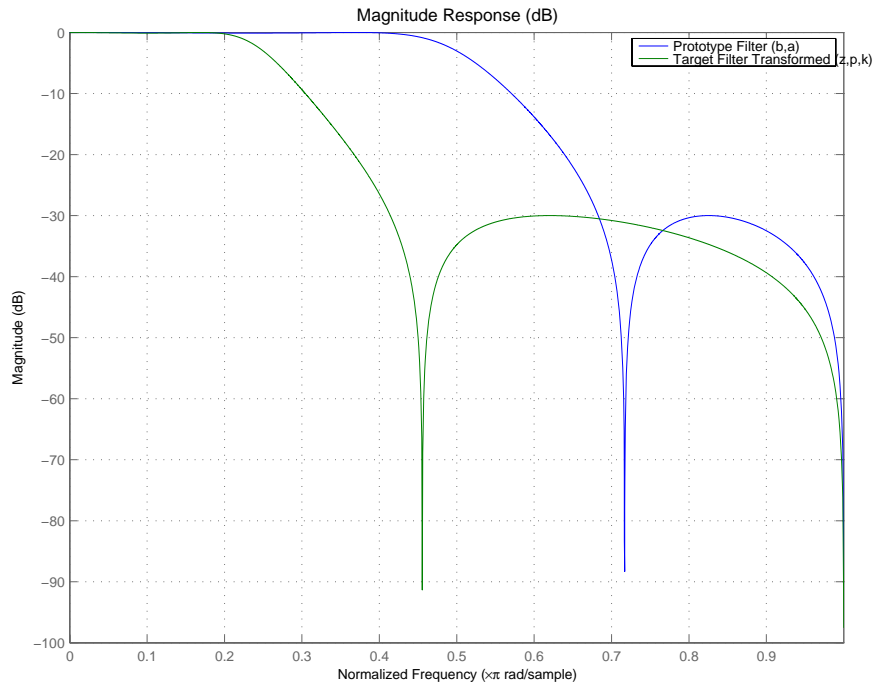
[2] Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

[3] 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.

[4] 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 lowpass frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2lp(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2lp(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 lowpass 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>.</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>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 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.</p> <p>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.</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); [z2,p2,k2] = zpk1p2lp(z, p, k, 0.5, 0.25);</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>

Using `zpk2lp` creates the desired half band IIR filter with the transformed features that you specify in the transformation function. This figure shows the results.



## Arguments

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter



Wt  
Desired frequency location in the transformed target filter

Z2  
Zeros of the target filter

P2  
Poles of the target filter

K2  
Gain factor of the target filter

AllpassNum  
Numerator of the mapping filter

AllpassDen  
Denominator of the mapping filter

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

## See Also

zpkftransf, allpass1p2lp, iir1p2lp

## References

- [1] Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.
- [2] 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.
- [3] 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.
- [4] Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

# zpk1p2mb

---

**Purpose** Zero-pole-gain lowpass to M-band frequency transformation

**Syntax**  $[Z_2, P_2, K_2, \text{AllpassNum}, \text{AllpassDen}] = \text{zpk1p2mb}(Z, P, K, W_0, W_t)$   
 $[Z_2, P_2, K_2, \text{AllpassNum}, \text{AllpassDen}] = \text{zpk1p2mb}(Z, P, K, W_0, W_t, \text{Pass})$

**Description**  $[Z_2, P_2, K_2, \text{AllpassNum}, \text{AllpassDen}] = \text{zpk1p2mb}(Z, P, K, W_0, W_t)$  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 Mth-order real lowpass to real multibandpass frequency mapping. By default the DC feature is kept at its original location.

$[Z_2, P_2, K_2, \text{AllpassNum}, \text{AllpassDen}] = \text{zpk1p2mb}(Z, P, K, W_0, W_t, \text{Pass})$  allows you to specify an additional parameter, *Pass*, which chooses between using the “DC Mobility” and the “Nyquist Mobility”. In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, *AllpassNum*, and the denominator, *AllpassDen*, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $W_0$ , 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.

**Examples**

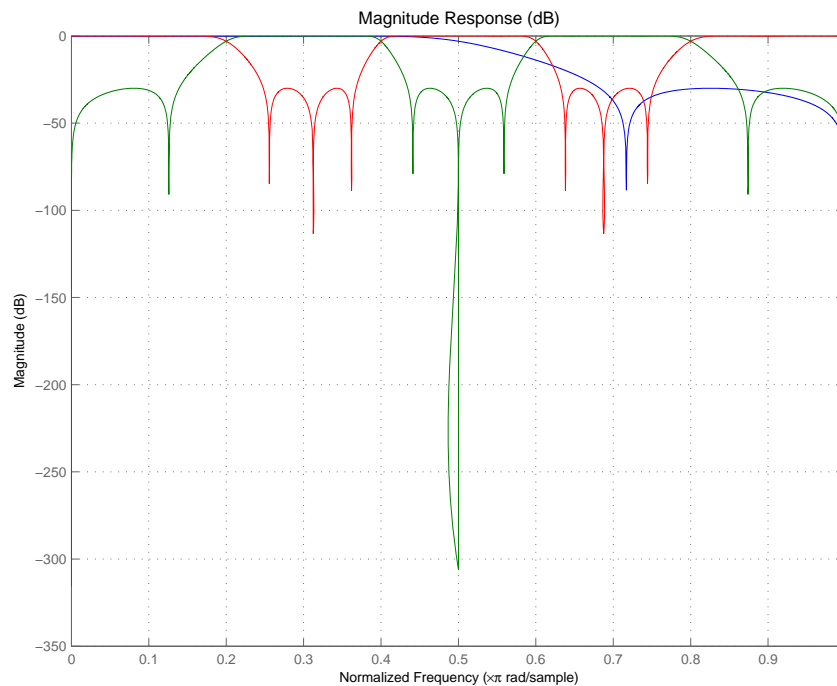
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[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

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

Pass

Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

zpkftransf, allpass1p2mb, iirlp2mb

## References

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

[2] 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.

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

[4] 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.

# zpk1p2mbc

---

**Purpose** Zero-pole-gain lowpass to complex M-band frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1pmbc(Z,P,K,Wo,Wt)`

**Description** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1pmbc(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 Mth-order real lowpass to complex multibandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $W_0$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

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

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

This transformation can also be used for transforming other types of filters; e.g., to replicate notch filters and resonators at any required location.

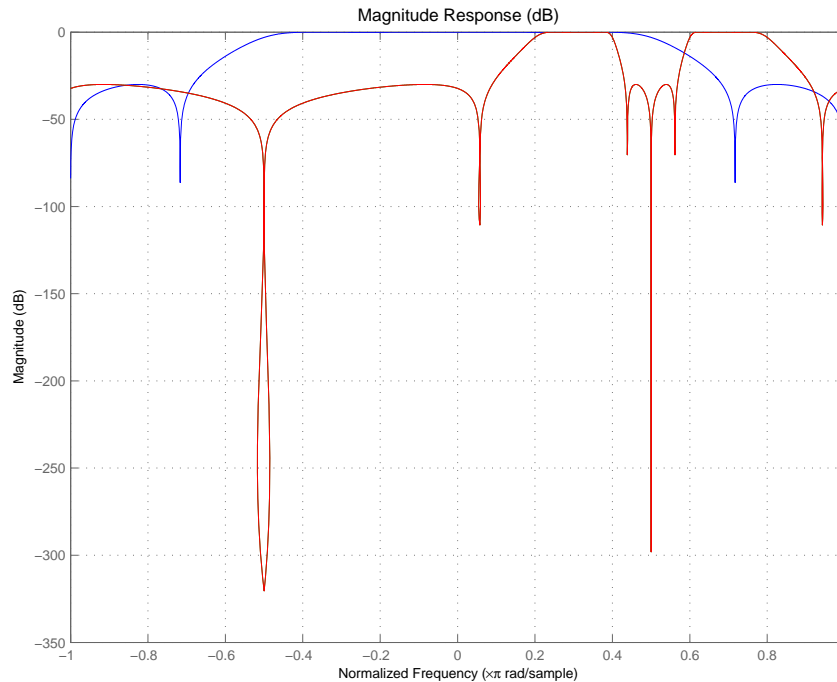
**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = 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

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Wt

Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassNum

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

## See Also

zpkftransf, allpasslp2mbc, iirlp2mbc



<b>Purpose</b>	Zero-pole-gain lowpass to complex N-point frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2xc(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpk1p2xc(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 Nth-order real lowpass to complex multipoint 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>Parameter <math>N</math> also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places <math>N</math> features of an original filter, located at frequencies <math>W_{01}, \dots, W_{0N}</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 <math>N</math> 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>

**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

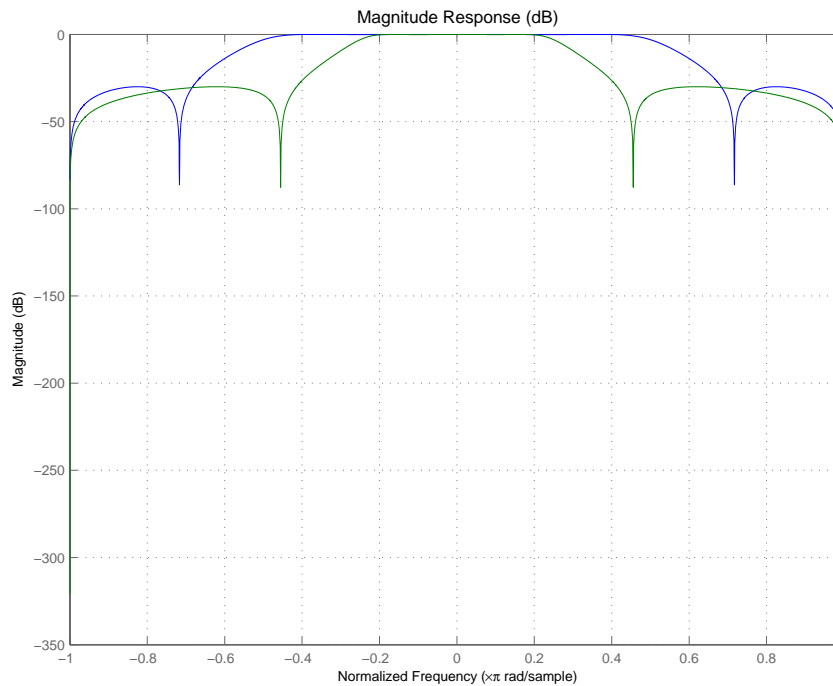
# zpk1p2xc

```
z = roots(b);  
p = roots(a);  
k = b(1);  
[z2,p2,k2] = zpk1p2xc(z, p, k, [-0.5 0.5], [-0.25 0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Plotting the filters on the same axes lets you compare the results graphically, shown here.



## Arguments

Z  
Zeros of the prototype lowpass filter

**P**  
Poles of the prototype lowpass filter

**K**  
Gain factor of the prototype lowpass filter

**Wo**  
Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

**Wt**  
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

**Z2**  
Zeros of the target filter

**P2**  
Poles of the target filter

**K2**  
Gain factor of the target filter

**AllpassNum**  
Numerator of the mapping filter

**AllpassDen**  
Denominator of the mapping filter

**See Also**

`zpkftransf`, `allpass1p2xc`, `iirlp2xc`

# zpk1p2xn

---

**Purpose** Zero-pole-gain lowpass to N-point frequency transformation

**Syntax**  $[Z_2, P_2, K_2, \text{AllpassNum}, \text{AllpassDen}] = \text{zpk1p2xn}(Z, P, K, W_0, W_t)$   
 $[Z_2, P_2, K_2, \text{AllpassNum}, \text{AllpassDen}] = \text{zpk1p2xn}(Z, P, K, W_0, W_t, \text{Pass})$

**Description**  $[Z_2, P_2, K_2, \text{AllpassNum}, \text{AllpassDen}] = \text{zpk1p2xn}(Z, P, K, W_0, W_t)$  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 real multipoint frequency transformation, where N is the number of features being mapped. By default the DC feature is kept at its original location.

$[Z_2, P_2, K_2, \text{AllpassNum}, \text{AllpassDen}] = \text{zpk1p2xn}(Z, P, K, W_0, W_t, \text{Pass})$  allows you to specify an additional parameter, *Pass*, which chooses between using the “DC Mobility” and the “Nyquist Mobility”. In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, *AllpassNum*, and the denominator, *AllpassDen*, of the allpass mapping filter. The prototype lowpass filter is given with zeros, *Z*, poles, *P*, and gain factor, *K*.

Parameter N also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places N features of an original filter, located at frequencies  $W_{01}, \dots, W_{0N}$ , 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.

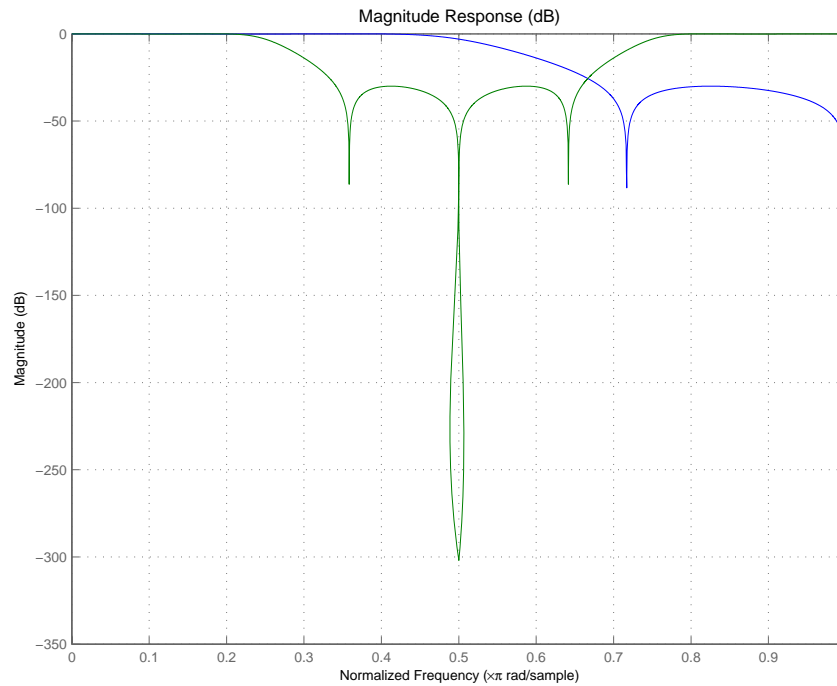
## Examples

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] = 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

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

Pass

Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default

Z2

Zeros of the target filter

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

- [1] 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.
- [2] 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.

# zpkrateup

---

**Purpose** Zero-pole-gain complex bandpass frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)`

**Description** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter being transformed from any prototype by applying an 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 original lowpass filter is given with zeros, Z, poles, P, and gain factor, K.

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

**Examples** Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);  
z = roots(b);  
p = roots(a);  
k = b(1);
```

Upsample the prototype filter four times:

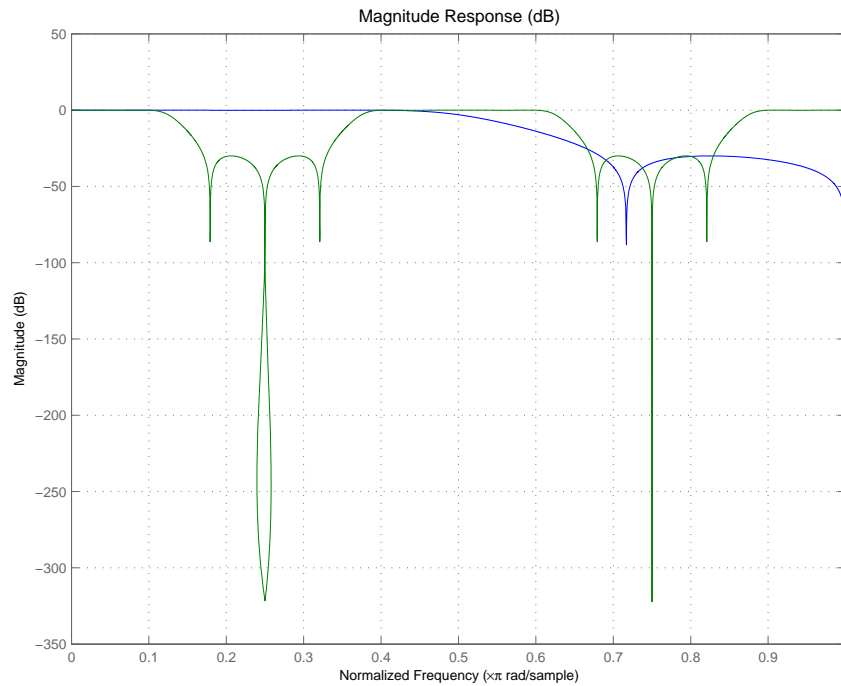
```
[z2,p2,k2] = zpkrateup(z, p, k, 4);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Applying the upsample process creates a bandpass filter, as shown here.





## Arguments

- 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

# zpkrateup

---

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

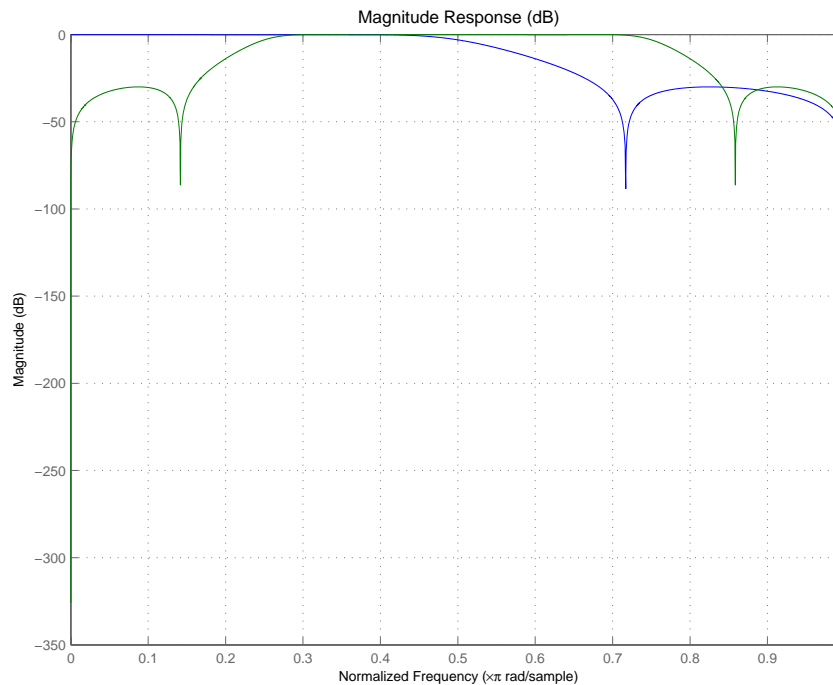
zpkrateup, allpassrateup, iirateup

<b>Purpose</b>	Zero-pole-gain real shift frequency transformation
<b>Syntax</b>	<code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)</code>
<b>Description</b>	<p><code>[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(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 shift frequency mapping.</p> <p>It also returns the numerator, <code>AllpassNum</code>, and the denominator of the allpass mapping filter, <code>AllpassDen</code>. 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 places one selected feature of an original filter, located at frequency <math>W_0</math>, at the required target frequency location, <math>W_t</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_0</math> and <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 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.</p> <p>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.</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); [z2,p2,k2] = zpkshift(z, p, k, 0.5, 0.25);</pre>

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

It is clear from the following figure that the shift process has taken the response value at 0.5 in the prototype and replicated it in the target at 0.25, as specified.



## Arguments

Z  
Zeros of the prototype lowpass filter

P  
Poles of the prototype lowpass filter

K  
Gain factor of the prototype lowpass filter

**Wo**  
Frequency value to be transformed from the prototype filter

**Wt**  
Desired frequency location in the transformed target filter

**Z2**  
Zeros of the target filter

**P2**  
Poles of the target filter

**K2**  
Gain factor of the target filter

**AllpassNum**  
Numerator of the mapping filter

**AllpassDen**  
Denominator of the mapping filter

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

**See Also**

`zpkftransf`, `allpassshift`, `iirshift`

# zpkshifc

---

**Purpose** Zero-pole-gain complex shift frequency transformation

**Syntax** `[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,Wo,Wt)`

**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:

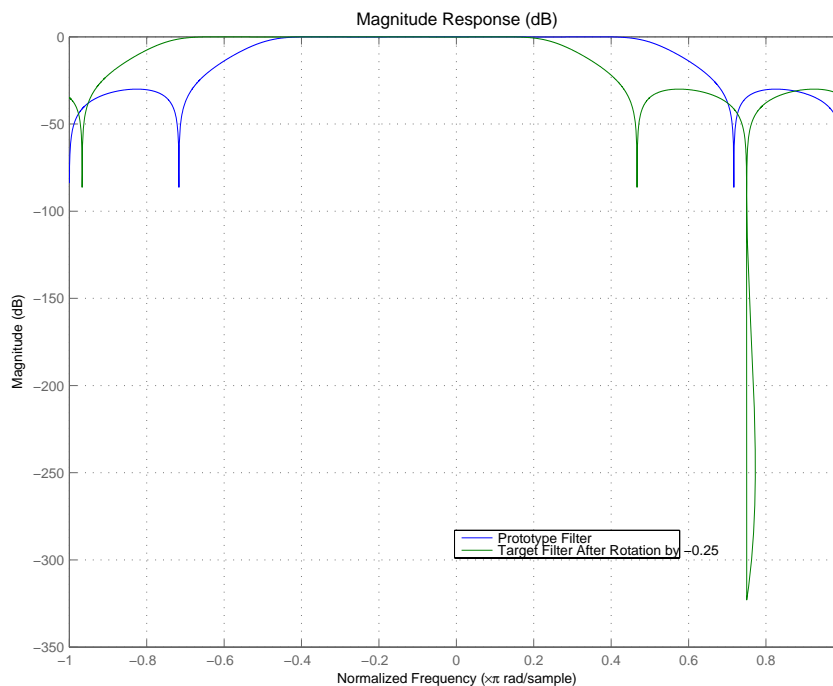
```
[z2,p2,k2] = zpkshifc(z, p, k, 0, 0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

**Example 3:** Inverse Hilbert transform:

```
[z2,p2,k2] = zpkshftc(z, p, k, 0, -0.5);  
fvtool(b, a, k2*poly(z2), poly(p2));
```

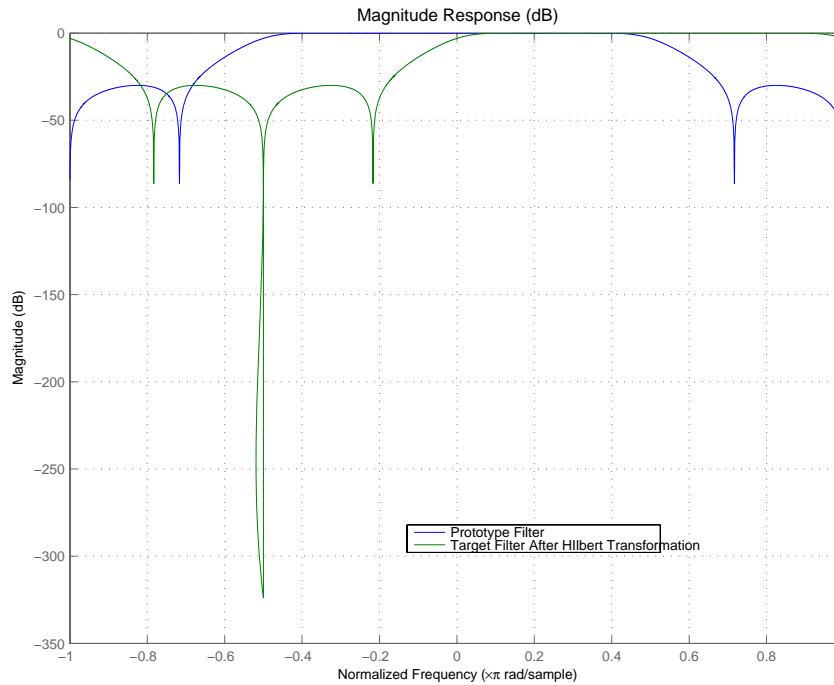
### Result of Example 1

After performing the rotation, the resulting filter shows the features desired.



### Result of Example 2

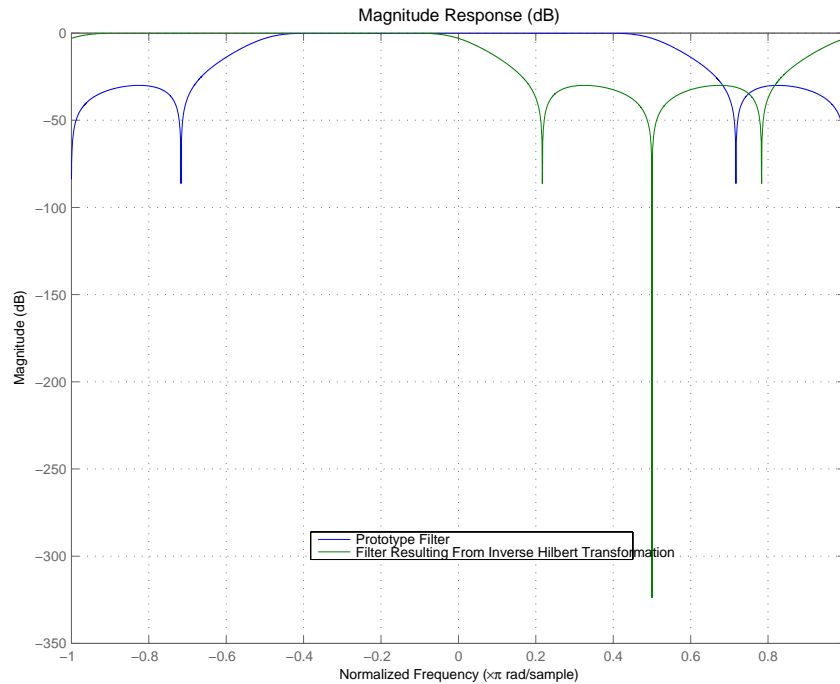
Similar to the first example, performing the Hilbert transformation generates the desired target filter, shown here.



### Result of Example 3

Finally, using the inverse Hilbert transformation creates yet a third filter, as the figure shows.





## Arguments

Z

Zeros of the prototype lowpass filter

P

Poles of the prototype lowpass filter

K

Gain factor of the prototype lowpass filter

Wo

Frequency value to be transformed from the prototype filter

Wt

Desired frequency location in the transformed target filter

Z2

Zeros of the target filter

P2

Poles of the target filter

K2

Gain factor of the target filter

AllpassDen

Numerator of the mapping filter

AllpassDen

Denominator of the mapping filter

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

## See Also

zpkftransf, allpassshiftc, iirshiftc

## References

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

[2] 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**

Compute zero-pole plot for quantized 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 below:

- **'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

# zplane

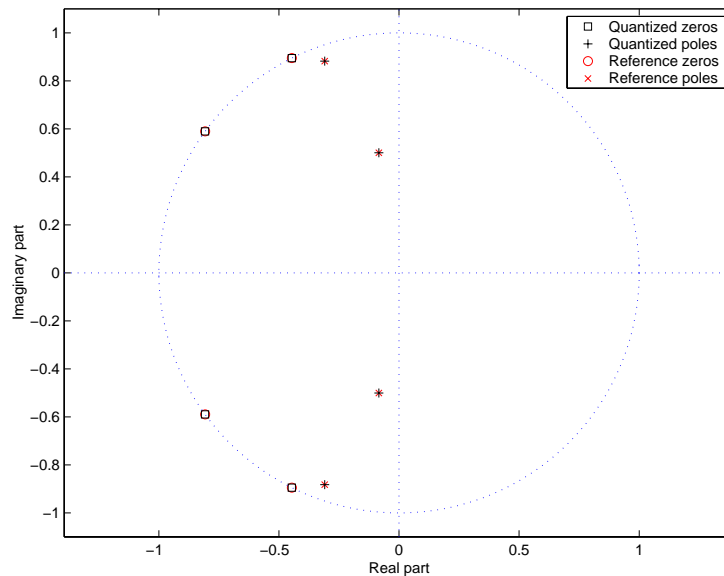
`[zq,pq,kq] = zplane(Hq)` returns the vectors of zeros `zq`, poles `pq`, and gains `kq`. If `Hq` has  $n$  sections, `zq`, `pq`, and `kq` are returned as 1-by- $n$  cell arrays. If there are no zeros (or no poles), `zq` (or `pq`) is set to the empty matrix `[]`.

`[zq,pq,kq,zr,pr,kr] = zplane(Hq)` returns the vectors of zeros `zr`, poles `pr`, and gains `kr` of the reference filter associated with the quantized filter `Hq`, and returns the vectors of zeros `zq`, poles `pq`, and gains `kq` for the quantized filter `Hq`.

## Examples

Create a quantized filter `Hq` from a fourth-order digital filter with cutoff frequency of 0.6. Scale the transfer function parameters to avoid overflows due to coefficient quantization. Plot the quantized and unquantized poles and zeros associated with this quantized filter.

```
[b,a] = ellip(4,.5,20,.6);  
Hq = dfilt.df2(b/2 a/2);  
Hq.arithmetic = 'fixed';  
zplane(Hq);
```



## See Also

`freqz`, `impz`

# Bibliography

---

Advanced Filters (p. A-2)

Suggested reading and sources for advanced filter design topics

Adaptive Filters (p. A-2)

Suggested reading and sources for adaptive filter topics

Multirate Filters (p. A-3)

Suggested reading and sources about multirate filters

Frequency Transformations (p. A-3)

Suggested reading and sources for information about filter frequency transformations

## **Advanced Filters**

[1] Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc., 1993.

[2] Chirlian, P.M., *Signals and Filters*, Van Nostrand Reinhold, 1994.

[3] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.

[4] Jackson, L., *Digital Filtering and Signal Processing with MATLAB Exercises*, Third edition, Kluwer Academic Publishers, 1996.

[5] Lapsley, P., J. Bier, A. Sholam, and E.A. Lee, *DSP Processor Fundamentals: Architectures and Features*, IEEE Press, 1997.

[6] McClellan, J.H., C.S. Burrus, A.V. Oppenheim, T.W. Parks, R.W. Schafer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB 5*, Prentice-Hall, 1998.

[7] Mayer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, refer to the BiQuad block diagram on pp. 126 and the IIR Butterworth example on pp. 140.

[8] Moler, C., "Floating points: IEEE Standard unifies arithmetic model," *Cleve's Corner*, The MathWorks, Inc., 1996. See <http://www.mathworks.com/company/newsletter/pdf/Fall196Cleve.pdf>.

[9] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

[10] Shajaan, M., and J. Sorensen, "Time-Area Efficient Multiplier-Free Recursive Filter Architectures for FPGA Implementation," IEEE International Conference on Acoustics, Speech, and Signal Processing, 1996, pp. 3269-3272.

## **Adaptive Filters**

[11] Hayes, M.H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, 1996.

[12] Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

---

## Multirate Filters

- [13] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.
- [14] harris, fredric j, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.
- [15] 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.
- [16] Lyons, Richard G., *Understanding Digital Signal Processing*, Prentice Hall PTR, 2004
- [17] Mitra, S.K., *Digital Signal Processing*, McGraw-Hill, 1998.
- [18] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Inc., 1996.

## Frequency Transformations

- [19] Constantinides, A.G., "Spectral Transformations for Digital Filters," *IEEE Proceedings*, Vol. 117, No. 8, pp. 1585-1590, August 1970.
- [20] 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.
- [21] Feyh, G., J.C. Franchitti, and C.T. Mullis, "Allpass Filter Interpolation and Frequency Transformation Problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.
- [22] 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.





## A

- AccumFracLength 7-20
- AccumWordLength 7-20
- adaptfilt
  - about 8-22
  - copying 8-30
- adaptfilt object
  - apply to data 4-25
- adaptfilt object properties
  - Algorithm 7-110
  - AvgFactor 7-110
  - BkwdPredErrorPower 7-110
  - BkwdPrediction 7-110
  - Blocklength 7-112
  - Coefficients 7-112
  - ConversionFactor 7-112
  - Delay 7-112
  - DesiredSignalStates 7-113
  - EpsilonStates 7-113
  - ErrorStates 7-113
  - FFTCoefficients 7-113
  - FFTStates 7-113
  - FilteredInputStates 7-113
  - FilterLength 7-113
  - ForgettingFactor 7-113
  - FwdPredErrorPower 7-114
  - FwdPrediction 7-114
  - InitFactor 7-114
  - InvCov 7-114
  - KalmanGain 7-114
  - KalmanGainStates 7-114
  - Leakage 7-114
  - Offset 7-114
  - OffsetCov 7-114
  - Power 7-115
  - ProjectionOrder 7-115
  - ReflectionCoeffsStep 7-115

- ResetBeforeFiltering 7-115
- SecondaryPathCoeffs 7-115
- SecondaryPathEstimate 7-115
- SecondaryPathStates 7-116
- SqrtInvCov 7-116
- States 7-116
- StepSize 7-116
- SwBlockLength 7-116
- adaptive filter object
  - See* adaptfilt object
- adaptive filter properties
  - SqrtCov 7-116
- addstages method 8-292
- Algorithm 7-110
- antisymmetricfir 7-54
- arithmetic
  - about fixed-point 7-20
- arithmetic property
  - double 7-21
  - fixed 7-23
  - single 7-22
- AvgFactor 7-110

## B

- binary point 2-46
  - interpretation 2-46
- bits
  - definition 2-45
- BkwdPredErrorPower 7-110
- BkwdPrediction 7-110
- block method 8-292
- Blocklength 7-112
- Bmax
  - See* CIC filter 3-28

**C**

- cascade method 8-292
- CastBeforeSum 2-42
- changing quantized filter properties in FDATool 6-22
- CIC filter
  - Bmax 3-28
  - MSB 3-28
- CoeffAutoScale 7-36
- CoeffFracLength 7-40
- Coefficients 7-112
- coefficients method 8-292
- CoeffWordLength 7-41
- context-sensitive help 6-89
- controls
  - FDATool 6-10
- ConversionFactor 7-112
- convert filters 7-68
- convert method 8-292
- converting filter structures in FDATool 6-28

**D**

- data format
  - about 2-46
- Delay 7-112
- DenAccumFracLength 7-41
- DenFracLength 7-41
- Denominator 7-42
- DenProdFracLength 7-42
- DenStateFracLength 7-42
- DenStateWordLength 7-43
- designing fixed-point multirate filters 6-81
- designing multirate filters 6-81
- DesiredSignalStates 7-113
- df1 7-47
- df1t 7-48

- df2 7-49
- df2t 7-52
- dfilt
  - cascade 8-311
  - df1 8-321
  - df1sos 8-331
  - df1t 8-6, 8-343
  - df1tsos 8-6, 8-354
  - df2 8-6, 8-366
  - df2sos 8-6, 8-376
  - df2t 8-7, 8-389
  - df2tsos 8-7, 8-400
  - direct-form antisymmetric FIR 8-7, 8-413
  - direct-form FIR transposed 8-7, 8-432
  - direct-form II transposed (df2t) 8-7, 8-389
  - direct-form IIR 8-7, 8-423
  - direct-form symmetric FIR 8-7, 8-442
  - lattice allpass 8-7, 8-453
  - lattice autoregressive 8-7, 8-463
  - lattice moving-average maximum 8-7, 8-483
  - lattice moving-average minimum 8-7, 8-492
  - parallel 8-7, 8-502
  - scalar 8-7, 8-503
  - See* Signal Processing Toolbox documentation
- dfilt function 8-286
  - convert structures 8-299
  - copying 8-299
  - methods 8-291
  - structures 8-286
- dfilt objects
  - See also* quantized filters
- dfilt properties
  - arithmetic 7-20
- dfilt.cascade 8-311
- dfilt.df1 8-321
- dfilt.df1sos 8-331
- dfilt.df1t 8-343

- dfilt.df1tsos 8-354
- dfilt.df2 8-366
- dfilt.df2sos 8-376
- dfilt.df2t 8-389
- dfilt.df2tsos 8-400
- dfilt.dffir 8-423
- dfilt.dffirt 8-432
- dfilt.dfsymfir 8-442
- dfilt.latticeallpass 8-453
- dfilt.latticear 8-463
- dfilt.latticemax 8-483
- dfilt.latticemin 8-492
- dfilt.parallel 8-502
- dfilt.scalar 8-503
- direct-form I 7-48
  - transposed 7-48
- direct-form II 7-49
  - transposed 7-52
- double
  - property value 7-21
- dynamic properties 7-6
- dynamic range
  - fixed-point 2-49
  
- E**
- EpsilonStates 7-113
- ErrorStates 7-113
- exporting quantized filters in FDATool 6-55
  
- F**
- fcfwrite method 8-293
- FDATool
  - about 6-3
  - about importing and exporting filters 6-53
  - about quantization mode 6-8
    - apply option 6-11
    - changing quantized filter properties 6-22
    - context-sensitive help 6-89
    - controls 6-10
    - convert structure option 6-28
    - converting filter structures 6-28
    - exporting quantized filters 6-55
    - frequency point to transform 6-63
    - getting help 6-89
    - import filter dialog 6-54
    - importable filter structures 6-53
    - importing filters 6-54
    - original filter type 6-60
    - quantized filter properties 6-12
    - quantizing filters 6-12
    - quantizing reference filters 6-21
    - set quantization parameters dialog 6-12
    - setting properties 6-12
    - specify desired frequency location 6-64
    - switching to quantization mode 6-8
    - transform filters in FDATool 6-64
    - transformed filter type 6-64
    - user options 6-10
- FFTCoefficients 7-113
- fftc coeffs method 8-293
- FFTStates 7-113
- filter
  - initial conditions 8-30
  - states 8-30
- filter conversions 7-69
- filter design
  - adaptive 4-1
  - multirate 8-10
- Filter Design and Analysis Tool
  - See FDATool
- filter design GUI
  - context-sensitive help 6-89

- help about 6-89
- filter method 8-293
- filter sections
  - specifying 7-69
- filter structures
  - about 7-43
  - all-pass lattice 7-60
  - direct-form antisymmetric FIR 7-54
  - direct-form FIR 7-57
  - direct-form I 7-47
  - direct-form I SOS IIR 7-48
  - direct-form I transposed 7-48
  - direct-form I transposed IIR 7-48
  - direct-form II 7-49
  - direct-form II IIR 7-49
  - direct-form II SOS IIR 7-51
  - direct-form II transposed 7-52
  - direct-form II transposed IIR 7-52
  - direct-form symmetric FIR 7-66
  - direct-form transposed FIR 7-58
  - FIR transposed 7-58
  - fixed-point 7-46
  - lattice allpass 7-60
  - lattice AR 7-62
  - lattice ARMA 7-64
  - lattice autoregressive moving average 7-64
  - lattice moving average maximum phase 7-61
  - lattice moving average minimum phase 7-63
- filter, fixed-point 2-29
- filter, quantized 2-29
- FilteredInputStates 7-113
- filterinternals
  - fixed-point filter 7-43
  - multirate filter 7-122
- FilterLength 7-113
- filters
  - converting 7-68
  - direct-form 2-32
  - exporting as MAT-file 6-57
  - exporting as text file 6-56
  - exporting from FDATool 6-55
  - FIR 7-43
  - getting filter coefficients after exporting 6-56
  - importing and exporting 6-53
  - importing into FDATool 6-54
  - impulse response 8-795
  - initial conditions using dfilt 8-299
  - lattice 7-43
  - objects 8-286
  - states 8-299
  - state-space 7-43
  - test if filter coefficients are real 8-17
  - testing for allpass structure 8-17
  - testing for FIR structure 8-17
  - testing for linear phase sections 8-17
  - testing for maximum phase design 8-17
  - testing for minimum phase design 8-17
  - testing for purely real coefficients 8-17
  - testing for second-order sections 8-18
  - testing for stability 8-18
- FilterStructure property 7-43
- finite impulse response
  - antisymmetric 7-54
  - symmetric 7-66
- fir 7-57
- FIR filters 7-43
- firt 7-58
- firtype method 8-293
- fixed
  - arithmetic property value 7-23
- fixed-point 2-45
  - sign bit 2-45
- fixed-point filter 2-29
- fixed-point filter properties

- AccumFracLength 7-20
- AccumWordLength 7-20
- Arithmetic 7-20
- CastBeforeSum 7-33
- CoeffAutoScale 7-36
- CoeffFracLength 7-40
- CoeffWordLength 7-41
- DenAccumFracLength 7-41
- DenFracLength 7-41
- Denominator 7-42
- DenProdFracLength 7-42
- DenStateFracLength 7-42
- DenStateWordLength 7-43
- FilterStructure 7-43
- fixed-point filter states 7-96
- fixed-point filter structures 7-46
- fixed-point filters
  - dynamic properties 7-6
- fixed-point format 2-46
- fixed-point multirate filters 6-81
- fixed-point numbers
  - scaling 2-50
- ForgettingFactor 7-113
- format 2-46
- format for numeric data 2-46
- fraction length 2-47
  - about 7-30
  - negative number of bits 7-30
- frequency point to transform 6-63
- frequency response 8-13, 8-708
- freqz 8-13, 8-708
- freqz method 8-293
- function for opening FDATool 6-8
- FwdPredErrorPower 7-114
- FwdPrediction 7-114

## G

- getting filter coefficients after exporting 6-56
- getting started 1-4
- getting started example 1-4
- grpdelay method 8-293

## I

- import filter dialog in FDATool 6-54
- import filter dialog options 6-54
  - discrete-time filter 6-54
  - frequency units 6-54
- import/export filters in FDATool 6-53
- importing filters 6-54
- importing quantized filters in FDATool 6-54
- impz method 8-293
- impzlength method 8-293
- info method
  - dfilt function 8-293
- InitFactor 7-114
- initial conditions 8-30
  - using dfilt states 8-299
- InvCov 7-114
- isallpass 8-17
- isallpass method 8-293
- iscascade method 8-294
- isfir 8-17
- isfir method 8-294
- islinphase 8-17
- islinphase method 8-294
- ismaxphase 8-17
- ismaxphase method 8-294
- isminphase 8-17
- isminphase method 8-294
- isparallel method 8-294
- isreal 8-17
- isreal method 8-294

isscalar method 8-294  
issos 8-18  
issos method 8-294  
isstable 8-18  
isstable method 8-294

## K

KalmanGain 7-114  
KalmanGainStates 7-114

## L

latcallpass 7-60  
latcmax 7-61  
lattice filters  
    allpass 7-60  
    AR 7-62  
    ARMA 7-64  
    autoregressive 7-62  
    MA 7-63  
    moving average maximum phase 7-61  
    moving average minimum phase 7-63  
latticear 7-62  
latticearma 7-64  
latticeca 7-61  
latticema 7-63  
Leakage 7-114  
least significant bit 2-46  
LSB 2-46

## M

mfilt object 8-838  
mfilt objects 8-10  
most significant bit 2-45  
MSB 2-45

multiple sections  
    specifying 7-69  
multirate filter functions 8-10  
multirate filter states 7-131  
multirate filters  
    designing 6-81  
multirate object  
    *See* mfilt

## N

negative fraction length  
    interpret 7-30  
normalize 2-50  
nsections method 8-294  
nstages method 8-294  
nstate method 8-294

## O

object  
    adaptfilt 8-22  
    changing properties 8-30, 8-299  
    filter 8-286  
    mfilt 8-838  
    viewing parameters 8-29  
    viewing properties 8-298  
object properties  
    AccumWordLength 7-20  
Offset 7-114  
OffsetCov 7-114  
opening FDATool  
    function for 6-8  
options  
    FDATool 6-10  
order method 8-295  
original filter type 6-60

**P**

- parallel method 8-295
- phasez method 8-295
- plots
  - zero-pole, command for 8-1093
- pole-zero plots 8-1093
- polyphase filters
  - See multirate filter functions 8-10
- Power 7-115
- precision 7-31
  - fixed-point 2-49
  - See fraction length 2-46
- ProjectionOrder 7-115
- properties
  - dynamic 7-6
  - FilterStructure 7-43
  - ScaleValues 7-86

**Q**

- quantization 2-29
- quantization mode in FDATool 6-8
- quantized 2-29
- quantized filter 2-29
- quantized filter properties
  - changing in FDATool 6-22
  - FilterStructure 2-32
- quantized filters
  - architecture 7-43
  - constructing 2-28
  - direct-form FIR 7-57
  - direct-form FIR transposed 7-58
  - direct-form symmetric FIR 7-66
  - filtering data 8-649, 8-651
  - finite impulse response 7-58
  - frequency response 8-13, 8-708
  - lattice allpass 7-60

- lattice AR 7-62
- lattice ARMA 7-64
- lattice coupled-allpass 7-60
- lattice MA maximum phase 7-61
- lattice MA minimum phase 7-63
- reference filter 7-67
- scaling 7-86
- specifying 7-67
- specifying coefficients for multiple sections 7-69
- structures 7-43
- symmetric FIR 7-54
- zero-pole plots 8-1093

quantized filters properties

- ScaleValues 7-86

quantizing filters in FDATool 6-21

**R**

- range
  - fixed-point 2-49
- realizemdl method 8-296
- reference coefficients
  - specifying 7-67
- ReflectionCoeffs 7-115
- ReflectionCoeffsStep 7-115
- removestage method 8-297
- represent numeric data 7-30
- ResetBeforeFiltering 7-115

**S**

- ScaleValues property 7-86
  - interpreting 7-87
- scaling
  - implementing for quantized filters 7-87
  - quantized filters 7-86

- SecondaryPathCoeffs 7-115
- SecondaryPathEstimate 7-115
- SecondaryPathStates 7-116
- second-order sections
  - normalizing 7-69
- set quantization parameters dialog 6-12
- setstage method 8-297
- setting filter properties in FDATool 6-12
- single
  - property value 7-22
- sos method 8-297
- specifying desired frequency location 6-64
- SqrtCov 7-116
- SqrtInvCov 7-116
- ss method 8-297
- starting FDATool 6-8
- States 7-116
- states, fixed-point filter 7-96
- states, multirate filter 7-131
- StepSize 7-116
- stepz method 8-298
- SwBlockLength 7-116
- symmetricfir 7-66

## T

- tf method 8-298
- toolbox
  - getting started 1-4
- transform filter
  - frequency point to transform 6-63
  - original filter type 6-60
  - specify desired frequency location 6-64
  - transformed filter type 6-64
- transformed filter type 6-64
- two's complement arithmetic 2-45

## U

- using adaptfilt objects 4-25
- using FDATool 6-54

## W

- word length
  - about 7-30

## Z

- zerophase method 8-298
- zero-pole plots 8-1093
- zpk method 8-298
- zplane 8-1093
  - plotting options 8-1093
- zplane method 8-298