Filter Design Toolbox™ 4
User's Guide

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Filter Design Toolbox™ User's Guide

© COPYRIGHT 2000–2008 by The MathWorks, Inc.

Trademarks

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

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

Contents

Designing a Filter — Process Overview

1

Using the Filterbuilder GUI

2

Digital Frequency Transformations

3

Using FDATool with Filter Design Toolbox Software

4

Adaptive Filters

5

Using Integers and FIR Filters with Filter Design Toolbox

6

Reference for the Properties of Filter Objects

7

Bibliography

A

Examples

B

Index

Designing a Filter — Process Overview

Process Flow Diagram and Filter Design Methodology

Exploring the Process Flow Diagram

The process flow diagram shown in the following figure lists the steps and shows the order of the filter design process.
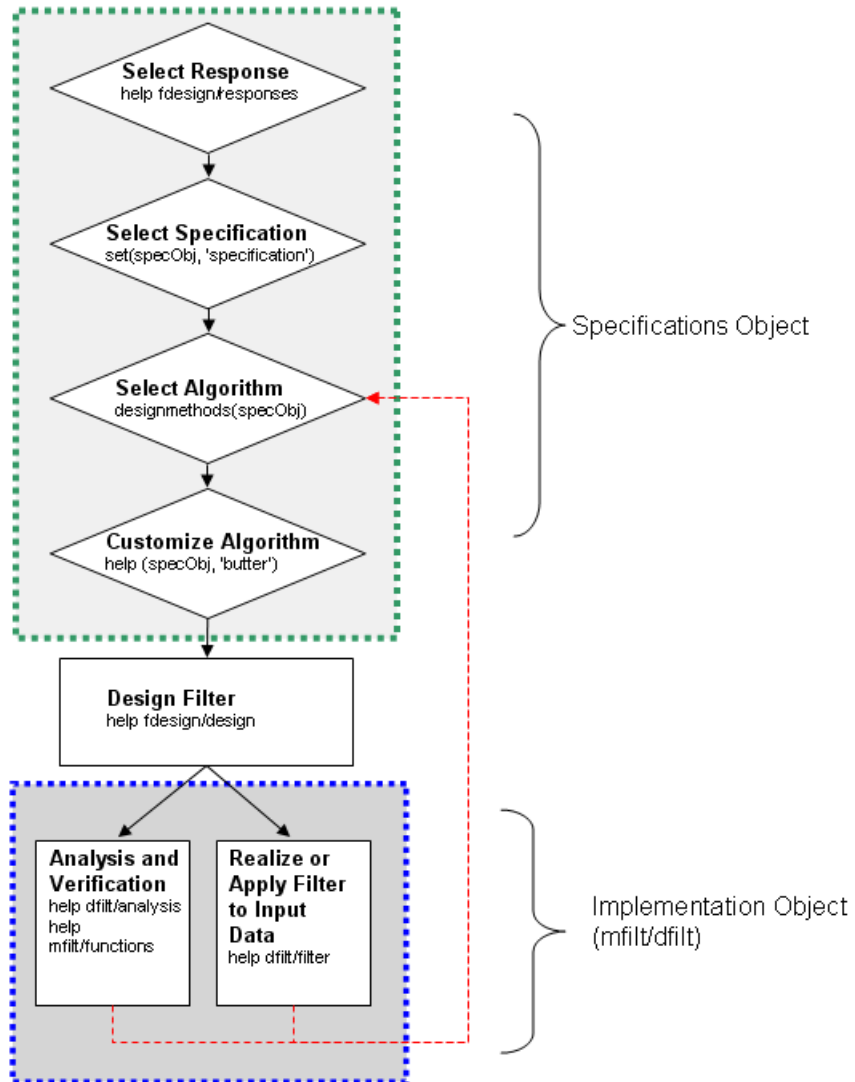
The first four steps of the filter design process relate to the filter Specifications Object, while the last two steps involve the filter Implementation Object. Both of these objects are discussed in more detail in the following sections. Step 5 - the design of the filter, is the transition step from the filter Specifications Object to the Implementation object. The analysis and verification step is

completely optional. It provides methods for the filter designer to ensure that the filter complies with all design criteria. Depending on the results of this verification, you can loop back to steps 3 and 4, to either choose a different algorithm, or to customize the current one. You may also wish to go back to steps 3 or 4 after you filter the input data with the designed filter (step 7), and find that you wish to tweak the filter or change it further.

The diagram shows the help command for each step. Enter the help line at the MATLAB® command prompt to receive instructions and further documentation links for the particular step. Not all of the steps have to be executed explicitly. For example, you could go from step 1 directly to step 5, and the interim three steps are done for you by Filter Design Toolbox™ software.

The following are the details for each of the steps shown above.

Selecting a Response

If you type:

```
help fdesign/responses
```

at the MATLAB command prompt, you see a complete list of all possible filter responses available in Filter Design Toolbox software. After you choose a response, say bandpass, you start the design of the Specifications Object by typing the following:

```
d = fdesign.bandpass
```

This step cannot be skipped, nor is it automatically completed for you by Filter Design Toolbox software. You must select a response to initiate the filter design process.

Selecting a Specification

A *specification* is an array of design parameters for a given filter. The specification itself is a property of the Specifications Object.

> **Note** A specification is not the same as the Specifications Object, rather a Specifications Object contains a specification as one of its properties.

When you select a filter response, there is a number of different specifications available, each containing a different combination of design parameters. In the following example, first set the filter response, then ask for the specifications listing.

```
>> d = fdesign.bandpass; % step 1 - choose the response
>> set (d, 'specification')

ans =

    'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
    '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'

>> d = fdesign.decimator; % step 1 - choose the response
<<% get a list of available specifications
>> set (d, 'specification')
ans =

    'TW,Ast'
    'N'
    'N,Ast'
    'N,TW'
```

After you select the specification that includes all of the given filter's design parameters, you can set it as follows:

```
>> d = fdesign.lowpass; % step 1
>> % step 2: get a list of available specifications
>> 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'

>> %step 2: set the required specification
>> set (d, 'specification', 'N,Fc')
```

If you do not perform this step explicitly, Filter Design Toolbox software selects a default specification for the response you chose in "Selecting a Response" on page 1-4, and even provides default values for all design parameters included in the specification.

Selecting an Algorithm

The availability of algorithms depends on both the chosen filter response and the design parameters. In other words, for the same lowpass filter, changing the specification string also changes the available algorithms. In the following example, for a lowpass filter and a specification of `'N, Fc'`,

only one algorithm is available—window. However, for a specification of 'Fp,Fst,Ap,Ast', a number of algorithms is available.

```
>> %step 2: set the required specification
>> set (d, 'specification', 'N,Fc')
>> designmethods (d) %step3: get available algorithms


Design Methods for class fdesign.lowpass (N,Fc):


window

>> %step2: set a different specification
>> set (d, 'specification', 'Fp,Fst,Ap,Ast')
>> designmethods (d) %step3: get available algorithms


Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

To apply the chosen algorithm, (the Butterworth algorithm in this example), evaluate the following:

```
>> f = design(d, 'butter');
```

The preceding code actually creates the filter, where f is the filter Implementation Object. This concept is discussed further in the next step.

If you do not perform this step explicitly, Filter Design Toolbox software automatically selects the optimum algorithm for the chosen response and specification.

Customizing the Algorithm

The customization options available for any given algorithm depend not only on the algorithm itself, selected in "Selecting an Algorithm" on page 1-6, but also on the specification selected in "Selecting a Specification" on page 1-4. To explore all the available options, type the following at the MATLAB command prompt:

```
help (d, 'algorithm-name')
```

where d is the Specifications Object, and algorithm-name is the name of the algorithm in quotes, such as 'butter' or 'cheby1'.

The application of these customization options takes place while "Designing the Filter" on page 1-8 because these options are the properties of the filter Implementation Object, not the Specification Object.

If you do not perform this step explicitly, Filter Design Toolbox software automatically selects the optimal algorithm structure as well as other options.

Designing the Filter

This next task introduces a new object, the Filter Object, or dfilt. To create a filter, use the design command:

```
>> % design filter w/o specifying the algorithm
>> f = design(d);
```

where f is the Filter Object also referred to sometimes as dfilt, and d is the Specifications Object. This code creates a filter without specifying the algorithm. When the algorithm is not specified, Filter Design Toolbox software selects the best available one.

To apply the algorithm chosen in "Selecting an Algorithm" on page 1-6, use the same design command, but specify the Butterworth algorithm as follows:

```
>> f = design(d, 'butter');
```

where f is the new Filter Object, and d is the Specifications Object.

To obtain help and see all the available options, type:

```
>> help fdesign/design
```

This help command describes not only the options for the `design` command itself, but also options that pertain to the method or the algorithm. If you are customizing the algorithm, you apply these options in this step. In the following example, you design a bandpass filter, and then modify the filter structure:

```
>> f = design(d, 'butter', 'filterstructure', 'df2sos')

f =

   FilterStructure: 'Direct-Form II, Second-Order Sections'
        Arithmetic: 'double'
          sosMatrix: [7x6 double]
        ScaleValues: [8x1 double]
  PersistentMemory: false
```

The filter design step, just like the first task of choosing a response, must be performed explicitly. Filter Design Toolbox software does not create a filter unless you specifically tell it to do so.

Design Analysis

After the filter is designed you may wish to analyze it to determine if the filter satisfies the design criteria. In Filter Design Toolbox software, analysis is broken into three main sections:

- Frequency domain analysis — Includes magnitude response, group delay, and poll zero

- Time domain analysis — Includes impulse and step response

- Implementation analysis — Includes quantization noise and cost

To display help for analysis of a discrete-time filter, type:

```
>> help dfilt/analysis
```

To display help for analysis of a multirate filter, type:

```
>> help mfilt/functions
```

To display help for analysis of a farrow filter, type:

```
>> help farrow/functions
```

To analyze your filter, you must explicitly perform this step.

Realize or Apply the Filter to Input Data

After the filter is designed and optimized, it can be used to filter actual input data. The basic filter command takes input data x, filters it through the Filter Object, and produces output y:

```
>> y = filter (FilterObj, x)
```

This step is never automatically performed for you by Filter Design Toolbox software. To filter your data, you must explicitly execute this step. To understand how the filtering commands work, type:

```
>> help dfilt/filter
```

Note If you have Simulink®, you have the option of exporting this filter to a Simulink block using the realizemdl command. To get help on this command, type:

```
>> help realizemdl
```

2

Using the Filterbuilder GUI

The Graphical Interface to Fdesign

Introduction to Filterbuilder

The `filterbuilder` function provides a graphical interface to the `fdesign` object-object oriented filter design paradigm and is intended to reduce development time during the filter design process. `filterbuilder` uses a specification-centered approach to find the best algorithm for the desired response.

The `filterbuilder` GUI contains many features that are not available in FDATool. For instance:

- Mulitrate/multistage FIR filters for efficient narrow-transition band designs

- Automatically generated optimal multistage interpolators and decimators

- Optimal multistage Nyquist filters

- IIR halfband designs (including IIR filters with approximately linear phase)

- CIC designs (including CIC compensators)

- Farrow filter designs

- Wave digital filter designs

- Arbitrary magnitude and phase designs

Filterbuilder Design Process

The design process when using `filterbuilder` is similar to the process outlined in the section titledChapter 1, "Designing a Filter — Process Overview" in the Getting Started guide. The idea is to choose the constraints and specifications of the filter, and to use those as a starting point in the design. Postponing the choice of algorithm for the filter allows the best design method to be determined automatically, based upon the desired performance criteria. The following are the details of each of the steps for designing a filter with `filterbuilder`.

Select a Response

When you open the `filterbuilder` tool by typing:

```
filterbuilder
```

at the MATLAB command prompt, the **Response Selection** dialog box appears, listing all possible filter responses available in Filter Design Toolbox software.
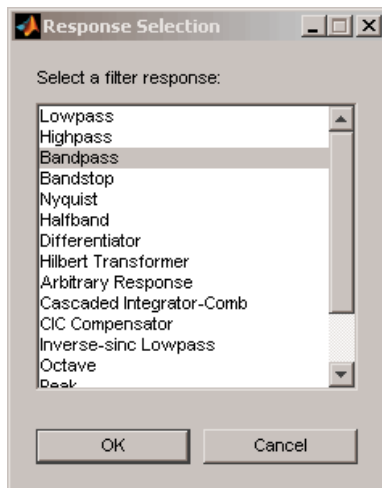
Note This step cannot be skipped because it is not automatically completed for you by Filter Design Toolbox software. You must select a response to initiate the filter design process.

After you choose a response, say bandpass, you start the design of the Specifications Object, and the Bandpass Design dialog box appears. This dialog box contains a **Main** pane, a **Data Types** pane and a **Code Generation** pane. The specifications of your filter are generally set in the **Main** pane of the dialog box.

The **Data Types** pane provides settings for precision and data types, and the **Code Generation** pane contains options for various implementations of the completed filter design. More information about the **Data Types** and **Code Generation** panes can be found in the section titled "Filterbuilder Dialog Box" in the Filter Design Toolbox Reference Guide.

For the initial design of your filter, you will mostly use the **Main** pane.
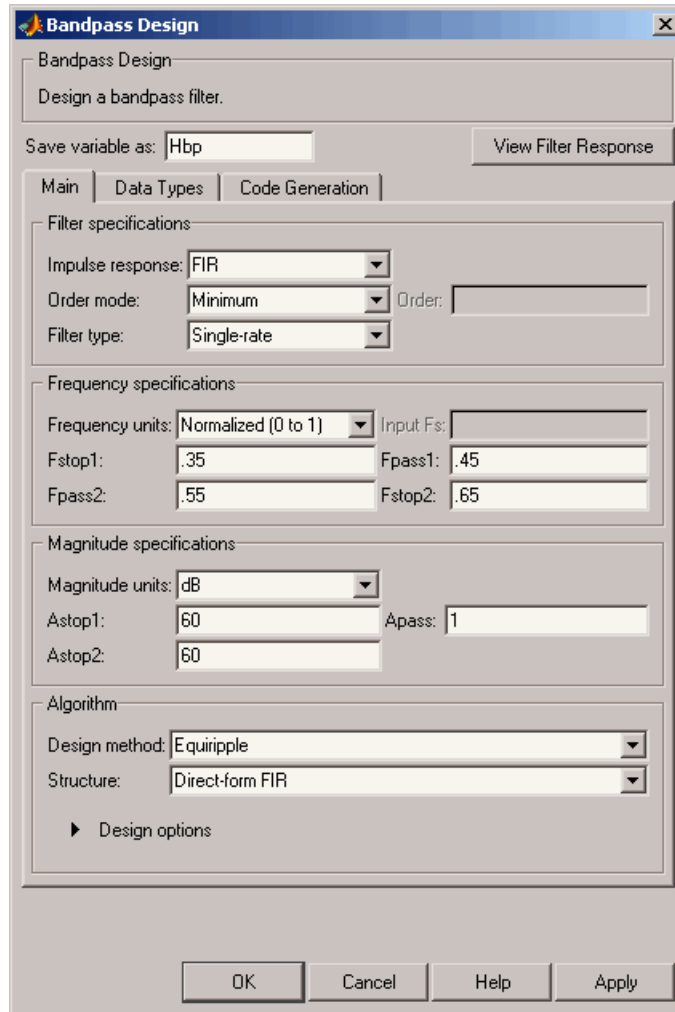
The **Bandpass Design** dialog box contains all the parameters you need to determine the specifications of a bandpass filter. The parameters listed in the **Main** pane depend upon the type of filter you are designing. However, no matter what type of filter you have chosen in the **Response Selection** dialog box, the filter design dialog box contains the **Main**, **Data Types**, and **Code Generation** panes.

Select a Specification

To choose the specification for the bandpass filter, you can begin by selecting an **Impulse Response**, **Order Mode**, and **Filter Type** in the **Filter Specifications** frame of the **Main Pane**. You can further specify the response of your filter by setting frequency and magnitude specifications in the appropriate frames on the **Main Pane**.

Note **Frequency**, **Magnitude**, and **Algorithm** specifications are interdependent and may change based upon your **Filter Specifications** selections. When choosing specifications for your filter, select your Filter Specifications first and work your way down the dialog box- this approach ensures that the best settings for dependent specifications display as available in the dialog box.

Select an Algorithm

The algorithms available for your filter depend upon the filter response and design parameters you have selected in the previous steps. For example, in the case of a bandpass filter, if the impulse response selected is IIR and the **Order Mode** field is set to Minimum, the design methods available is Butterworth, Chebyshev type I or II, or Elliptic, whereas if the **Order Mode** field is set to Specify, the design method available is IIR least p-norm.
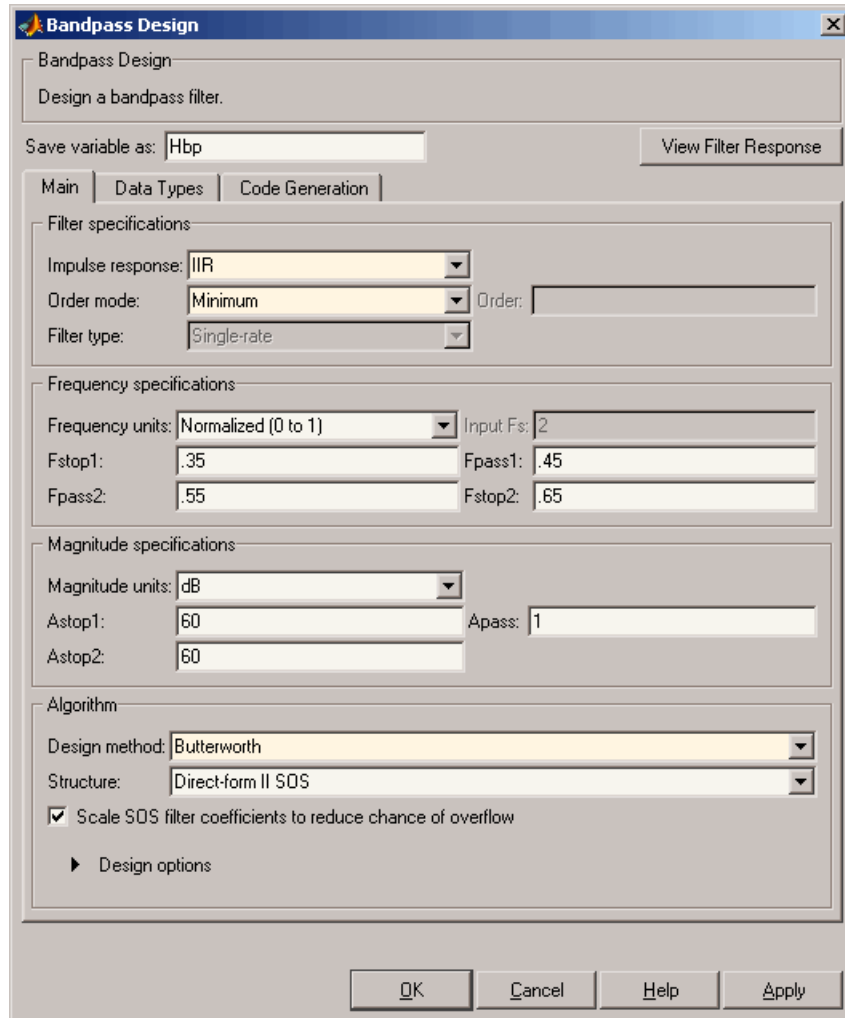
Customize the Algorithm

By expanding the **Design options** section of the **Algorithm** frame, you can further customize the algorithm specified. The options available will depend upon the algorithm and settings that have already been selected in the dialog box. In the case of a bandpass IIR filter using the Butterworth

method, design options such as **Match Exactly** are available, as shown in
the following figure.
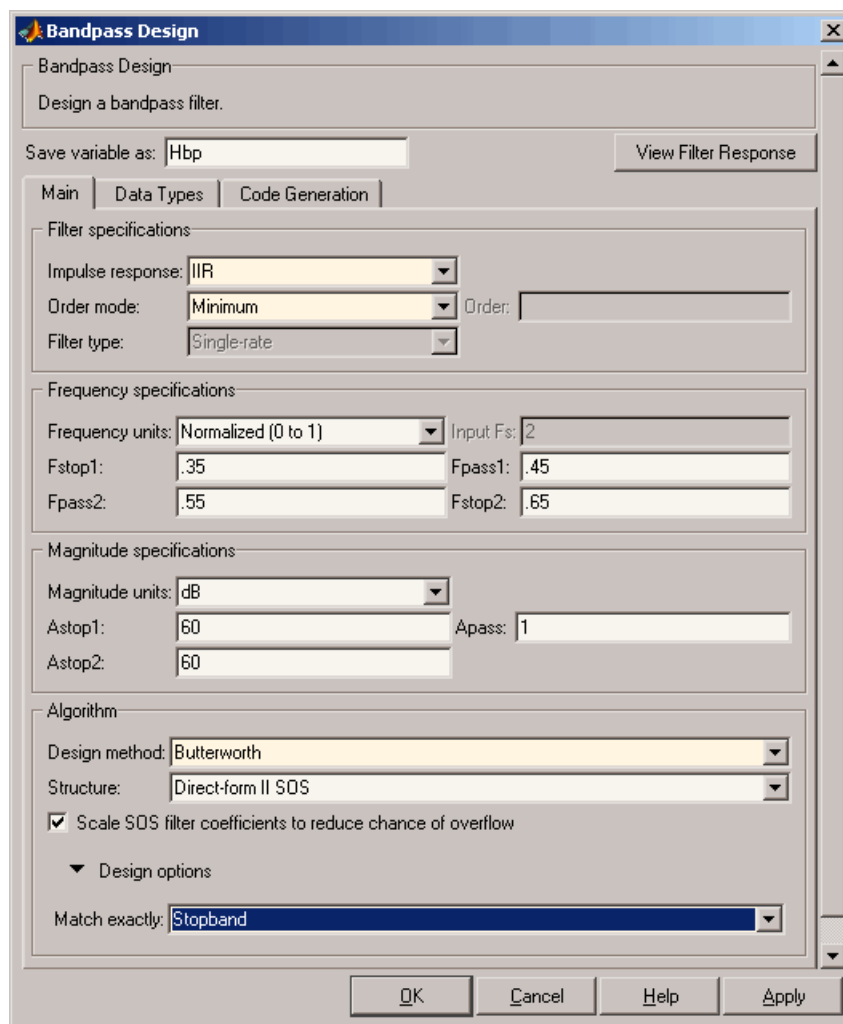
Analyze the Design

To analyze the filter response, click on the View Filter Response button. The Filter Visualization Tool opens displaying the magnitude plot of the filter response.
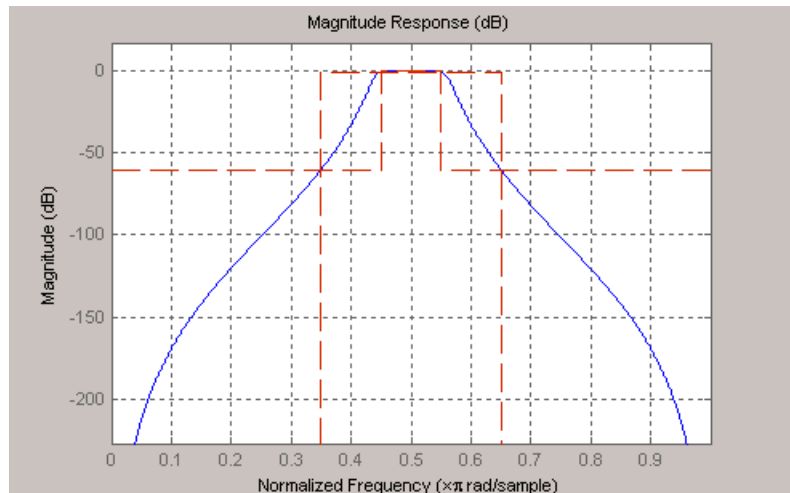
Realize or Apply the Filter to Input Data

When you have achieved the desired filter response through design iterations and analysis using the **Filter Visualization Tool**, apply the filter to the input data. Again, this step is never automatically performed for you by Filter Design Toolbox software. To filter your data, you must explicitly execute this step. In the **Filter Visualization Tool**, click OK and Filter Design Toolbox software creates the filter object with the name specified in the **Save variable as** field and exports it to the MATLAB workspace.

The filter is then ready to be used to filter actual input data. The basic filter command takes input data x, filters it through the Filter Object, and produces output y:

```
>> y = filter (FilterObj, x)
```

To understand how the filtering commands work, type:

```
>> help dfilt/filter
```

Tip If you have Simulink, you have the option of exporting this filter to a Simulink block using the `realizemdl` command. To get help on this command, type:

```
>> help realizemdl
```

Designing a FIR Filter Using filterbuilder

FIR Filter Design

Example – Using Filterbuilder to Design a Finite Impulse Response (FIR) Filter

To design a lowpass FIR filter using filterbuilder:

1 Open the Filterbuilder GUI by typing the following at the MATLAB prompt:

```
filterbuilder
```

The **Response Selection** dialog box appears. In this dialog box, you can select from a list of filter response types. Select Lowpass in the list box.
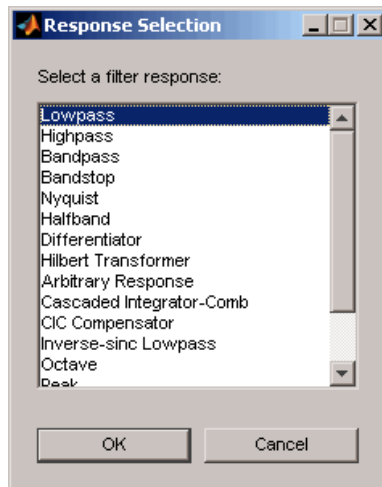
2 Hit the **OK** button. The **Lowpass Design** dialog box opens. Here you can specify the writable parameters of the Lowpass filter object. The components of the **Main** frame of this dialog box are described in the section titled "Lowpass Filter Design Dialog Box — Main Pane". In the dialog box, make the following changes:

- Enter a Fpass value of 0.55.

- Enter a Fstop value of 0.65.
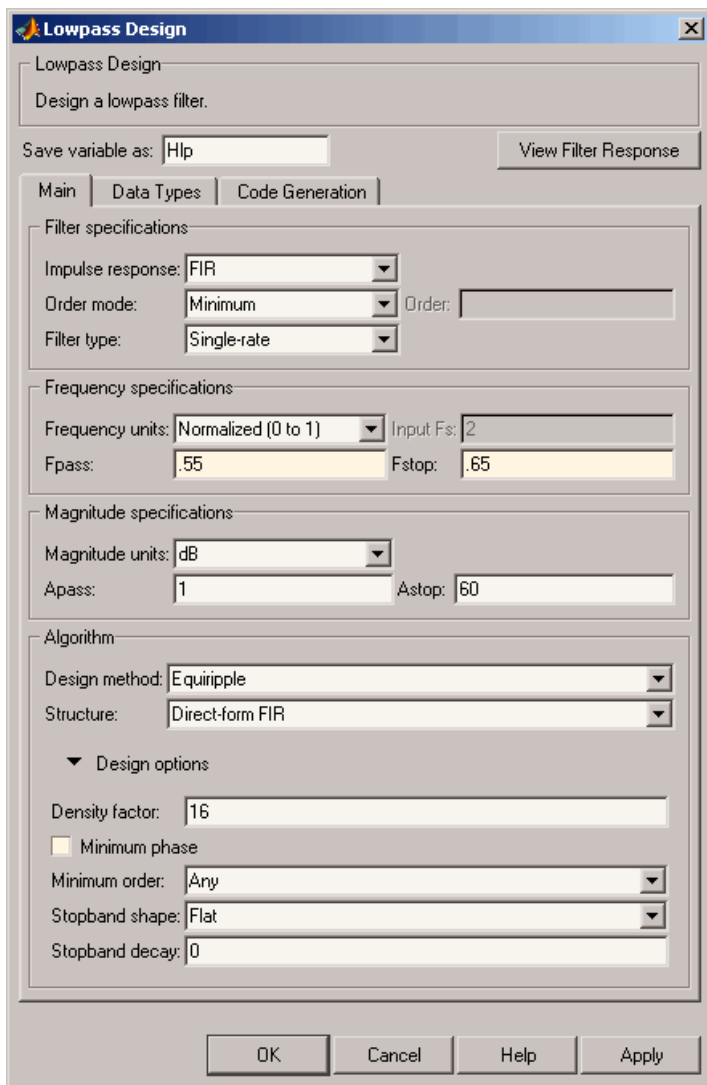
3 Click **Apply**, and the following message appears at the MATLAB prompt:

```
The variable 'Hlp' has been exported to the command window.
```

4 To check your design, click **View Filter Response**. The Filter
Visualization tool appears, showing a plot of the magnitude response of
the filter.
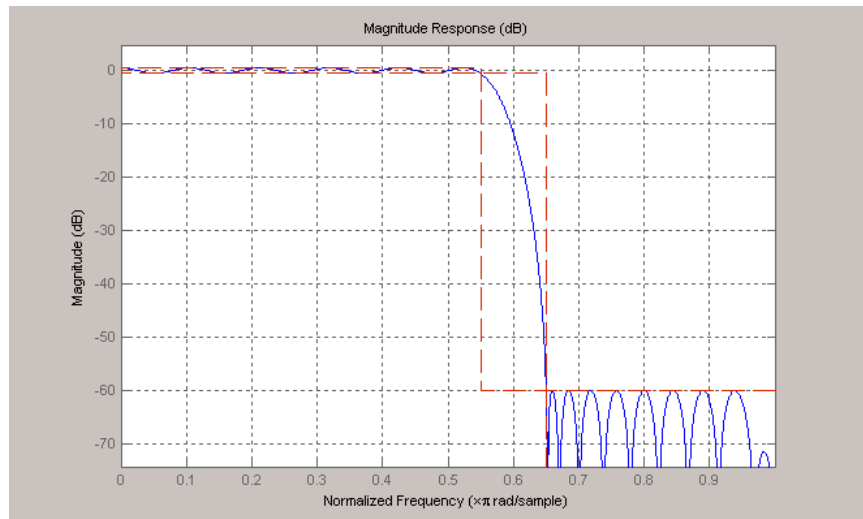
You can change the design and click **Apply**, followed by **View Filter
Response**, as many times as needed until your design specifications are
met.

3

Digital Frequency Transformations

Details and Methodology

In this section...

Overview of Transformations

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.

The basic form of mapping commonly used is

$$H_T(z) = H_o[H_A(z)]$$

The $H_A(z)$ is an Nth-order allpass mapping filter given by

$$H_A(z) = S \frac{\displaystyle\sum_{i=0}^{N} \alpha_i z^{-i}}{\displaystyle\sum_{i=0}^{N} \alpha_i z^{-N+i}} = \frac{N_A(z)}{D_A(z)}$$

$$\alpha_0 = 1$$

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.

Prototype filter Pole–Zero plot   Target filter Pole–Zero plot
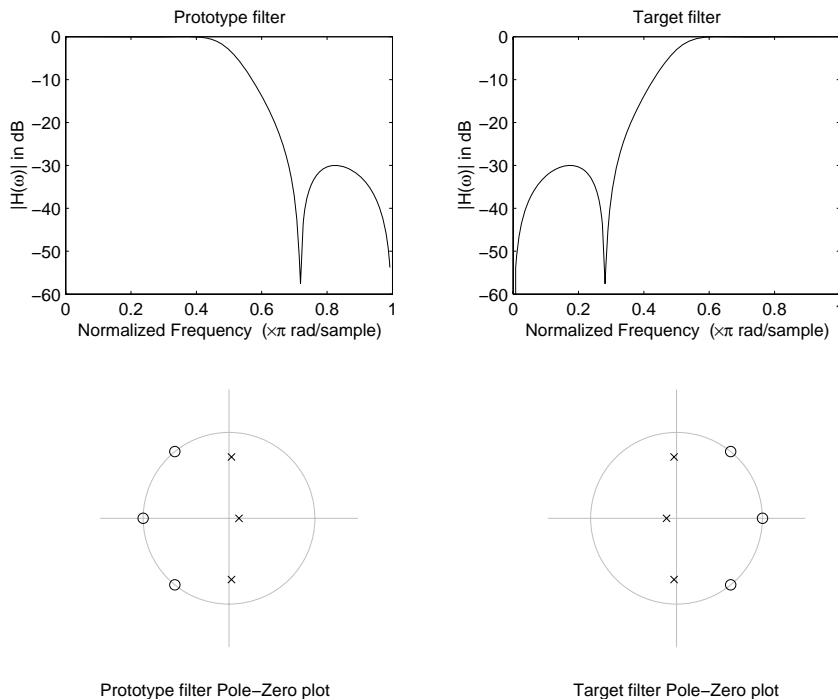
Example of a Simple Mirror Transformation

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 π 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.
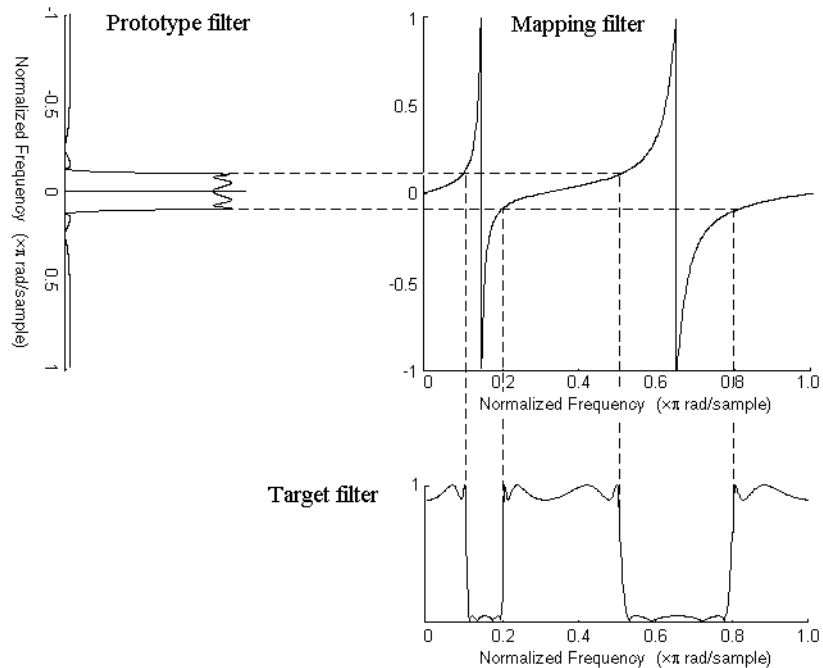
Graphical Interpretation of the Mapping Process

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 α_1 and α_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 α_1 and α_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 [1][2] up to the real multiband filter by Mullis and Franchitti [3], and the complex multiband filter and real/complex multipoint ones by Krukowski, Cain and Kale [4].

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.
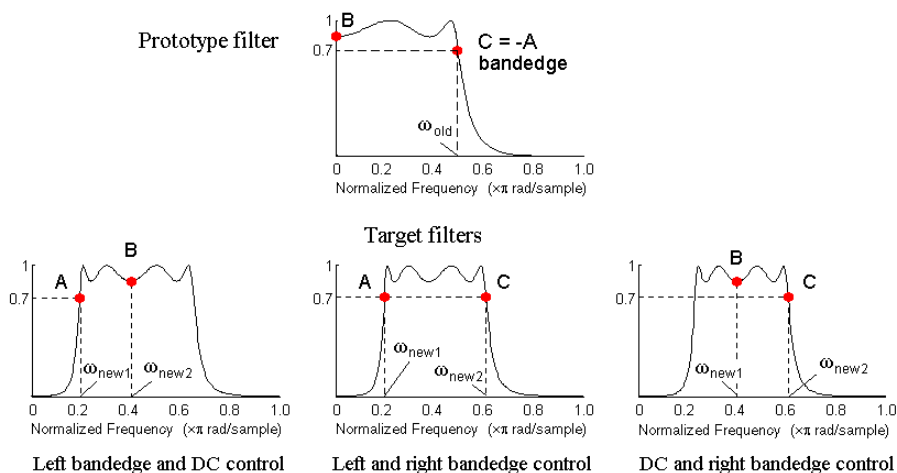
Feature Selection for Real Lowpass to Bandpass Transformation

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.
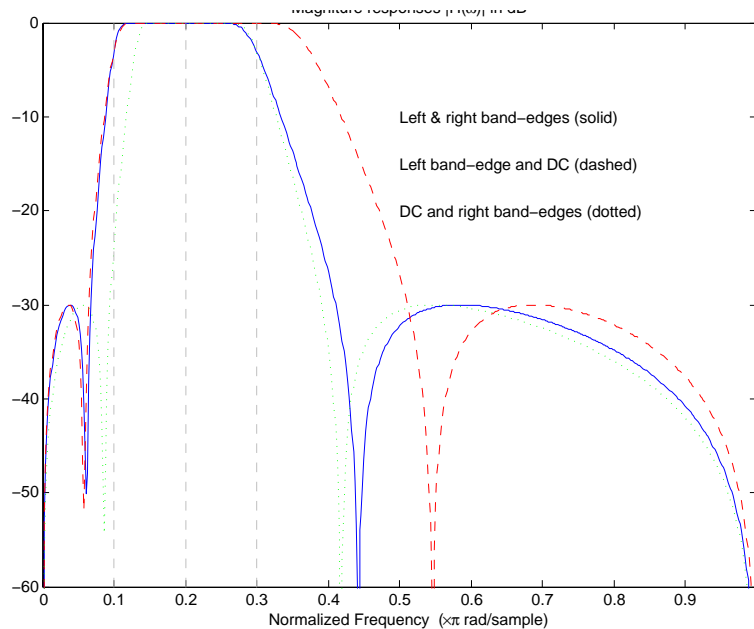
Result of Choosing Different Features

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 passband 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_Ath-order allpass filter, $H_A(z)$. The general form of the allpass mapping filter is given in "Overview of Transformations" on page 3-2. The frequency mapping is a mathematical operation that replaces each delayer 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 Nth-order prototype filter is given with pole-zero form

$$H_o(z) = \frac{\displaystyle\sum_{i=1}^{N}(z - z_i)}{\displaystyle\sum_{i=1}^{N}(z - p_i)}$$

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:

$$H_o(z) = \frac{\displaystyle\sum_{i=1}^{N}\left(S\sum_{k=0}^{N-1}\alpha_k z^k - z_i \cdot \sum_{k=0}^{N-1}\alpha_k z^{N-k}\right)}{\displaystyle\sum_{i=1}^{N}\left(S\sum_{k=0}^{N-1}\alpha_k z^k - p_i \cdot \sum_{k=0}^{N-1}\alpha_k z^{N-k}\right)}$$

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:

$$H_T(z) = \left.\frac{\beta_0 z^N + \beta_1 z^{N-1} + \ldots + \beta_N}{\alpha_0 z^N + \alpha_1 z^{N-1} + \ldots + \alpha_N}\right|_{z = H_A(z)}$$

Then the mapping process will require a number of convolutions in order to calculate the numerator and denominator of the target filter:

$$I_T(z) = \frac{\beta_1 N_A(z)^N + \beta_2 N_A(z)^{N-1} D_A(z) + \ldots + \beta_N D_A(z)^N}{\beta_1 N_A(z)^N + \beta_2 N_A(z)^{N-1} D_A(z) + \ldots + \beta_N D_A(z)^N}$$

For each coefficient α_i and β_i of the prototype filter the N_Ath-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

Overview

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

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 α given by

$$
\alpha = \begin{cases} \dfrac{\cos\dfrac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\cos\dfrac{\pi}{2}\omega_{old}} & \text{for} \qquad \left|\cos\dfrac{\pi}{2}(\omega_{old} - 2\omega_{new})\right| < 1 \\[2em] \dfrac{\sin\dfrac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\sin\dfrac{\pi}{2}\omega_{old}} & otherwise \end{cases}
$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

ω_{new} — Position of the feature originally at ω_{old} in the target filter

The following example 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);
```
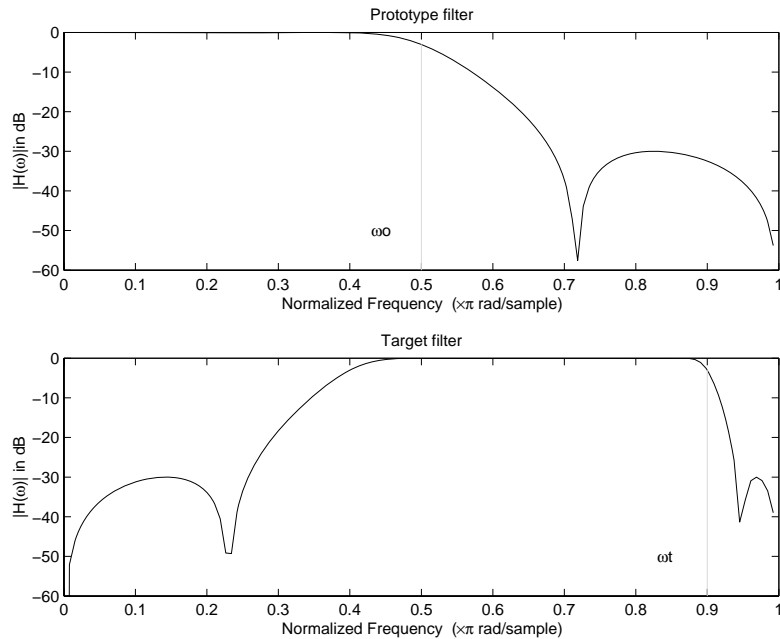
Example of Real Frequency Shift Mapping

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 α given by

$$\alpha = \frac{\sin\frac{\pi}{2}(w_{old} - w_{new})}{\sin\frac{\pi}{2}(w_{old} + w_{new})}$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

ω_{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 following figure.

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] = iirlp2lp(b, a, 0.5, 0.75);
```
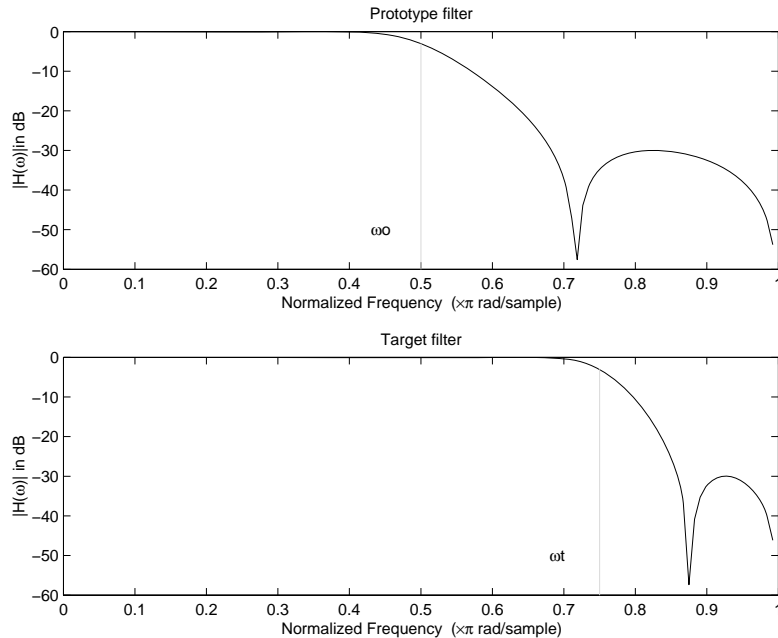
Example of Real Lowpass to Real Lowpass Mapping

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 α given by

3-15

$$\alpha = -\left(\frac{\cos\frac{\pi}{2}(w_{old} + w_{new})}{\cos\frac{\pi}{2}(w_{old} - w_{new})} \right)$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

ω_{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] = iirlp2hp(b, a, 0.5, 0.75);
```
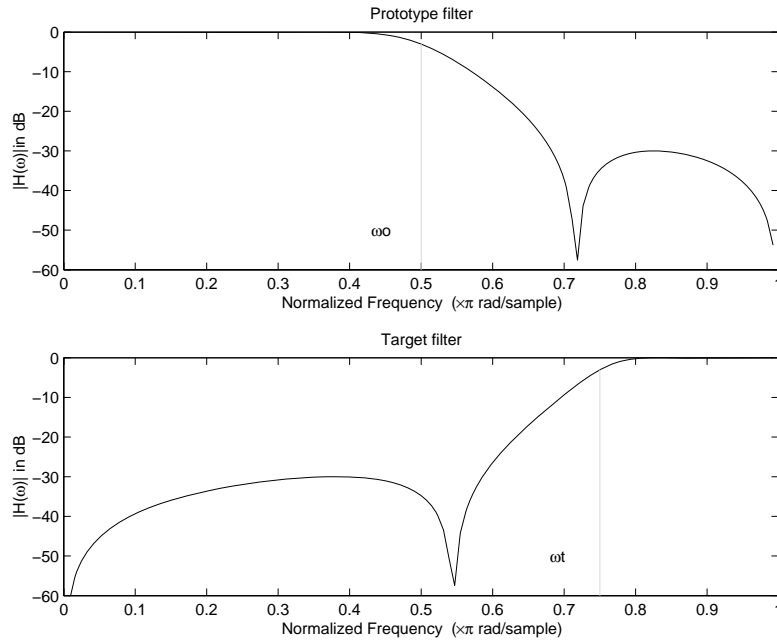
Example of Real Lowpass to Real Highpass Mapping

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 α and β given by

$$\alpha = \frac{\sin\frac{\pi}{4}(2w_{old} - w_{new,2} + w_{new,1})}{\sin\frac{\pi}{4}(2w_{old} + w_{new,2} - w_{new,1})}$$

$$\beta = \cos\frac{\pi}{2}(\omega_{new,1} + \omega_{new,2})$$

where

ω_{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]);
```
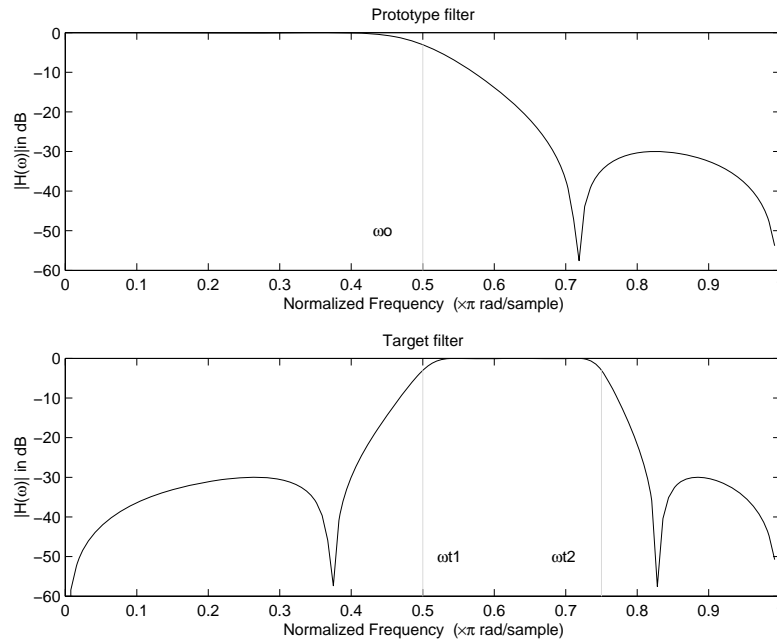
Example of Real Lowpass to Real Bandpass Mapping

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 α and β 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

ω_{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 following example 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]);
```
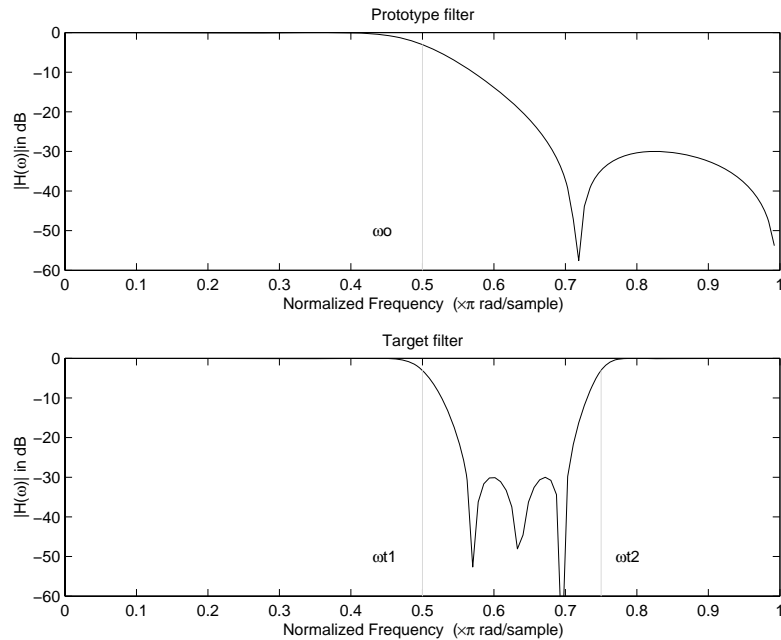
Example of Real Lowpass to Real Bandstop Mapping

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.

$$H_A(z) = S \frac{\displaystyle\sum_{i=0}^{N} \alpha_i z^{-i}}{\displaystyle\sum_{i=0}^{N} \alpha_i z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients α are given by

$$\begin{cases} \alpha_0 = 1 \qquad k = 1, \ldots, N \\[4mm] \alpha_k = -S \dfrac{\sin\dfrac{\pi}{2}(N\omega_{new} + (-1)^k \omega_{old})}{\sin\dfrac{\pi}{2}((N-2k)\omega_{new} + (-1)^k \omega_{old})} \end{cases}$$

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 stopbands, from DC to 0.2, from 0.4 to 0.6 and from 0.8 to Nyquist:

```
[num,den] = iirlp2mb(b, a, 0.5, [0.2, 0.4, 0.6, 0.8], `pass');
```
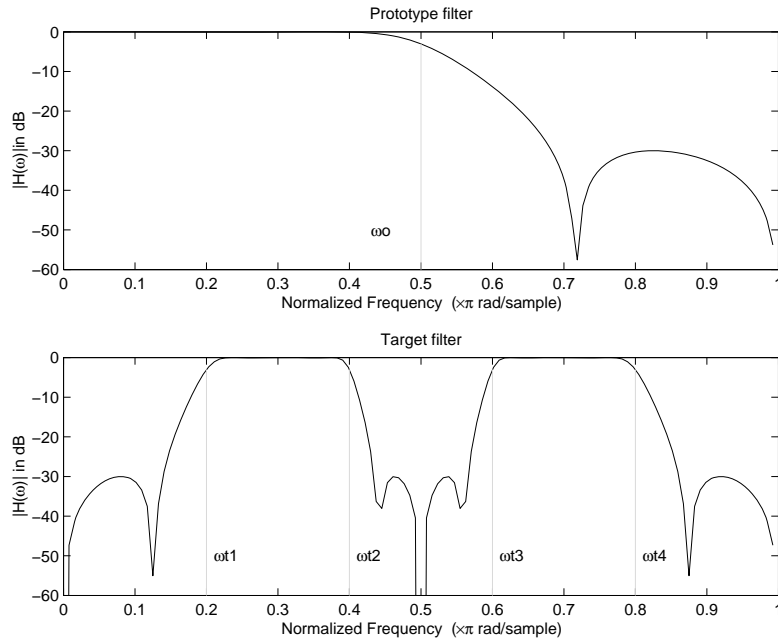
Example of Real Lowpass to Real Multiband Mapping

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.

$$H_A(z) = S \frac{\displaystyle\sum_{i=0}^{N} \alpha_i z^{-i}}{\displaystyle\sum_{i=0}^{N} \alpha_i z^{-N+i}}$$

$$\alpha_0 = 1$$

For the *N*th-order multipoint frequency transformation the coefficients α are

$$
\begin{cases}
\displaystyle\sum_{i=1}^{N} \alpha_{N-i} z_{old,k} \cdot z_{new,k}^{i} - S \cdot z_{new,k}^{N-i} = -z_{old,k} - S \cdot z_{new,k} \\[4mm]
z_{old,k} = e^{j\pi\omega_{old,k}} \\[2mm]
z_{new,k} = e^{j\pi\omega_{new,k}} \\[2mm]
k = 1,\dots,N
\end{cases}
$$

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] = iirlp2xn(b, a, [-0.5, 0.5], [0.5, 0.75], `pass');
```
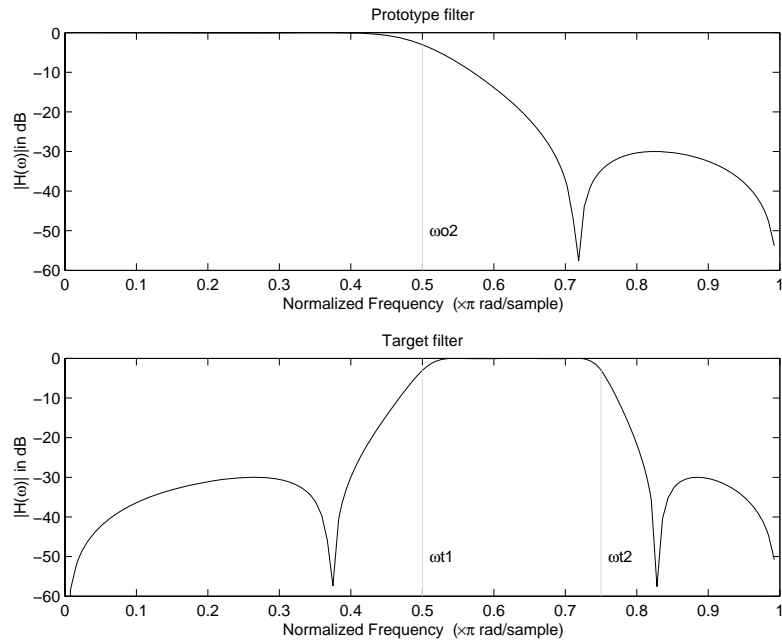
Example of Real Lowpass to Real Multipoint Mapping

Frequency Transformations for Complex Filters

Overview

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

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 α given by

$$\alpha = e^{j2\pi(\nu_{new} - \nu_{old})}$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

ω_{new} — Position of the feature originally at ω_{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

$$\alpha = \begin{cases} 1 & forward \\ -1 & inverse \end{cases}$$

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 0.3:

```
[num,den] = iirshiftc(b, a, 0.5, 0.3);
```
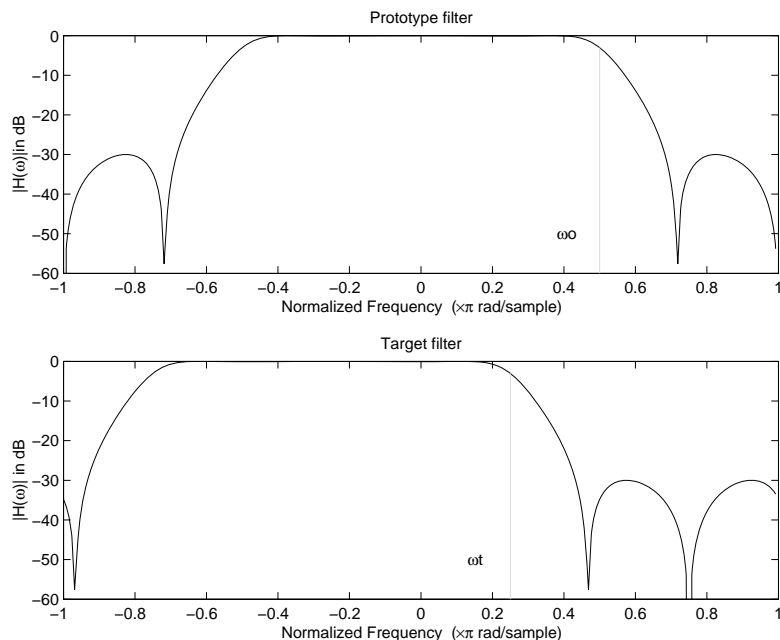
Example of Complex Frequency Shift Mapping

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 α and β 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

ω_{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 following example 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]);
```
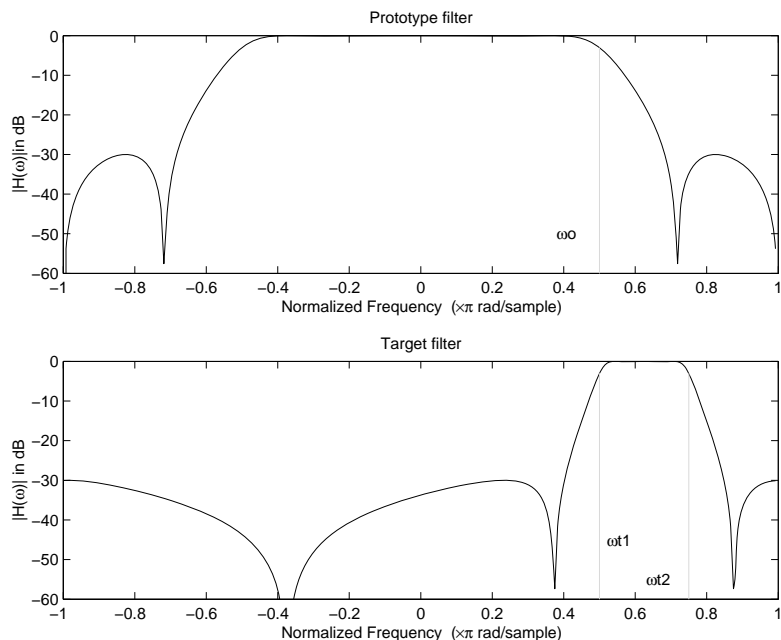
Example of Real Lowpass to Complex Bandpass Mapping

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 α and β 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

ω_{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] = iirlp2bsc(b, a, 0.5, [0.5 0.75]);
```
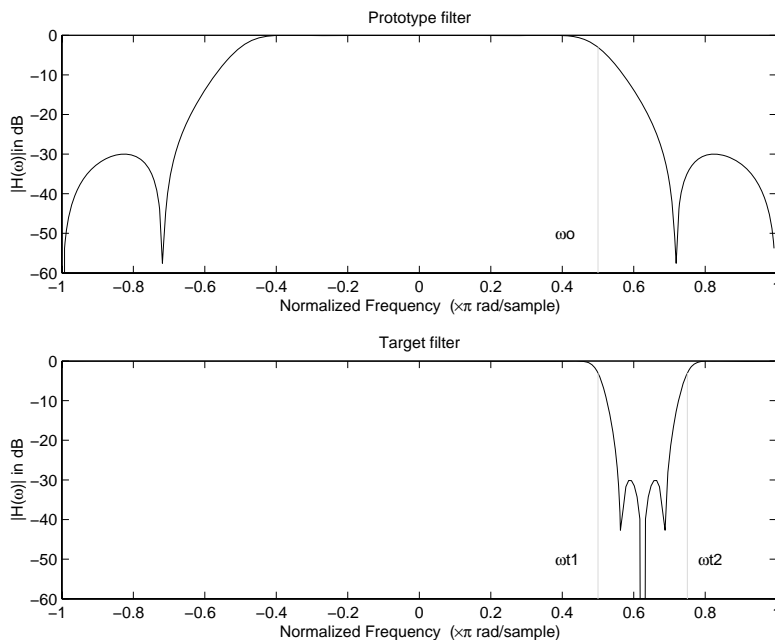
Example of Real Lowpass to Complex Bandstop Mapping

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 following general transfer function form:

$$H_A(z) = S \frac{\displaystyle\sum_{i=0}^{N} \alpha_i z^{-i}}{\displaystyle\sum_{i=0}^{N} \alpha_i{}^* z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients α are calculated from the system of linear equations:

$$\begin{cases} \displaystyle\sum_{i=1}^{N} \Re(\alpha_i) \cdot [\cos\beta_{1,k} - \cos\beta_{2,k}] + \Im(\alpha_i) \cdot [\sin\beta_{1,k} + \sin\beta_{2,k}] = \cos\beta_{3,k} \\[2mm] \displaystyle\sum_{i=1}^{N} \Re(\alpha_i) \cdot [\sin\beta_{1,k} - \sin\beta_{2,k}] - \Im(\alpha_i) \cdot [\cos\beta_{1,k} + \cos\beta_{2,k}] = \sin\beta_{3,k} \\[2mm] \beta_{1,k} = -\pi[\nu_{old} \cdot (-1)^k + \nu_{new,k}(N-k)] \\[1mm] \beta_{2,k} = -\pi[\Delta C + \nu_{new,k} k] \\[1mm] \beta_{3,k} = -\pi[\nu_{old} \cdot (-1)^k + \nu_{new,k} N] \\[1mm] k = 1 \ldots N \end{cases}$$

where

ω_{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 ΔC, 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.
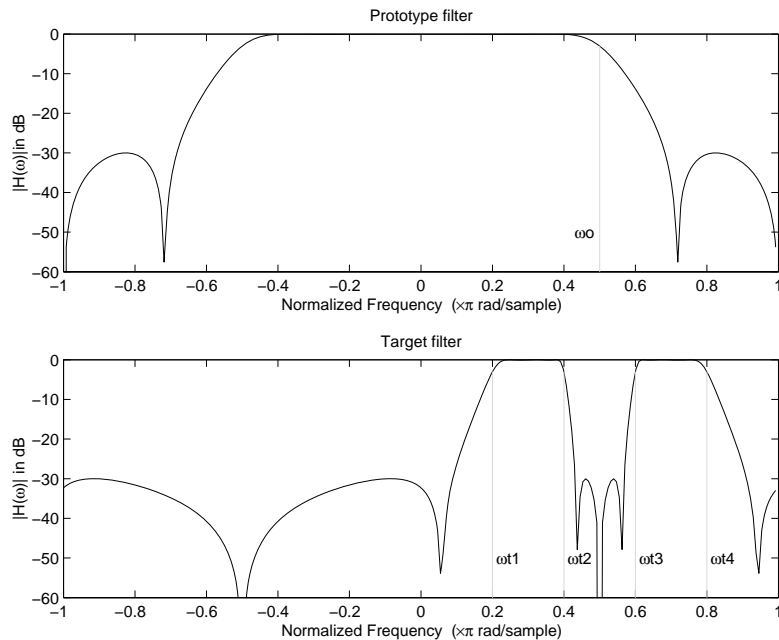
Example of Real Lowpass to Complex Multiband Mapping

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 following general transfer function form.

$$H_A(z) = S \frac{\displaystyle\sum_{i=0}^{N} \alpha_i z^{-i}}{\displaystyle\sum_{i=0}^{N} \alpha_i{}^* z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients α can be calculated from the system of linear equations:

$$\begin{cases} \displaystyle\sum_{i=1}^{N} \Re(\alpha_i) \cdot [\cos\beta_{1,k} - \cos\beta_{2,k}] + \Im(\alpha_i) \cdot [\sin\beta_{1,k} + \sin\beta_{2,k}] = \cos\beta_{3,k} \\[2ex] \displaystyle\sum_{i=1}^{N} \Re(\alpha_i) \cdot [\sin\beta_{1,k} - \sin\beta_{2,k}] - \Im(\alpha_i) \cdot [\cos\beta_{1,k} + \cos\beta_{2,k}] = \sin\beta_{3,k} \\[2ex] \beta_{1,k} = -\frac{\pi}{2}[w_{old,k} + w_{new,k}(N-k)] \\[1ex] \beta_{2,k} = -\frac{\pi}{2}[2\Delta C + w_{new,k}k] \\[1ex] \beta_{3,k} = -\frac{\pi}{2}[w_{old,k} + w_{new,k}N] \\[1ex] k = 1...N \end{cases}$$

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

Parameter S is the additional rotation factor by the frequency distance ΔC, giving the additional flexibility of achieving the required mapping:

$$S = e^{-j\pi\Delta C}$$

The following example 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] = iirlp2xc(b,a,0.5*[-1,1,-1,1], [-0.5,-0.25,0.1,0.3]);
```
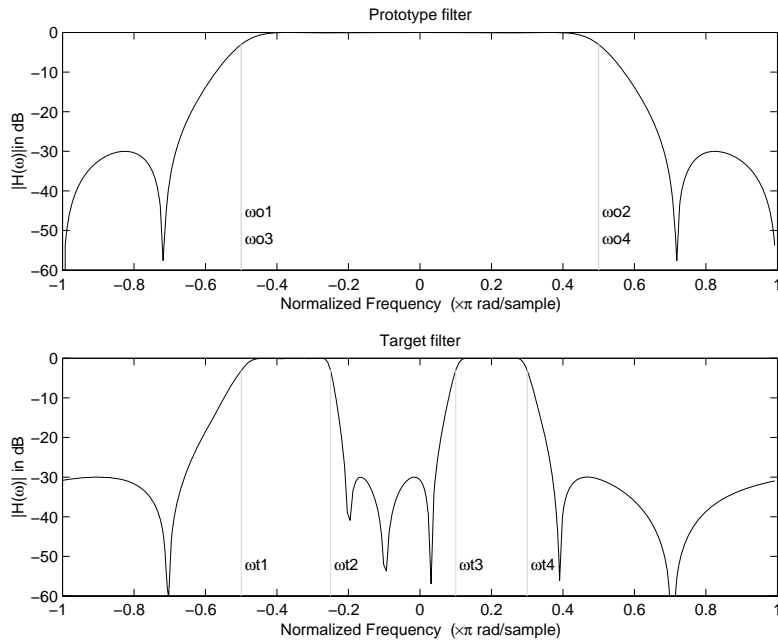

Example of Real Lowpass to Complex Multipoint Mapping

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 α and β 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 following example 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]);
```
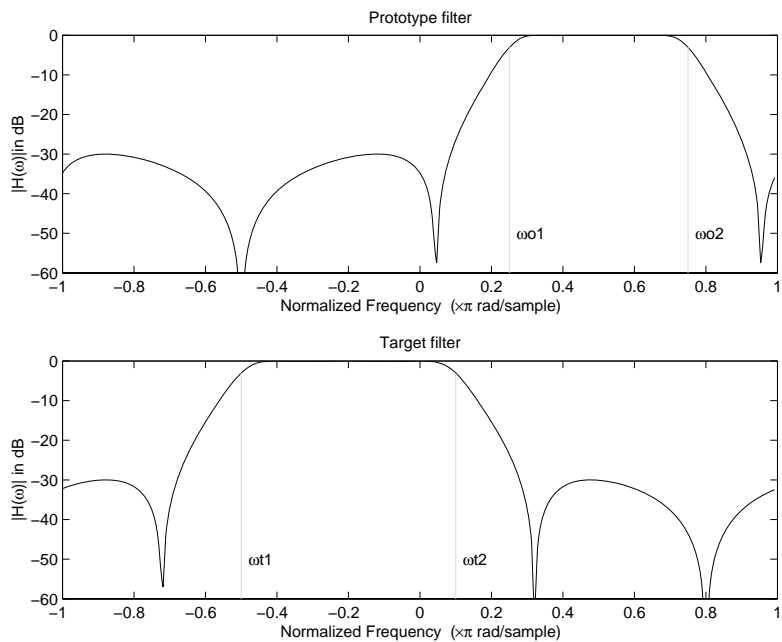
Example of Complex Bandpass to Complex Bandpass Mapping

4

Using FDATool with Filter Design Toolbox Software

Designing Advanced Filters in FDATool

Overview of FDATool Features

Filter Design Toolbox software adds new dialog boxes and operating modes, and new menu selections, to the Filter Design and Analysis Tool (FDATool) provided by Signal Processing Toolbox™ software. From the new dialog boxes, one titled **Set Quantization Parameters** and one titled **Frequency Transformations**, you can:

- Design advanced filters that Signal Processing Toolbox software 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 box let you quantize the filter and investigate the effects of various quantization settings.

Options in the frequency transformations dialog box let you change the frequency response of your filter, keeping various important features while changing the response shape.

Using FDATool with Filter Design Toolbox Software

Adding Filter Design Toolbox software 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 Signal Processing Toolbox software. 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 right panes 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:
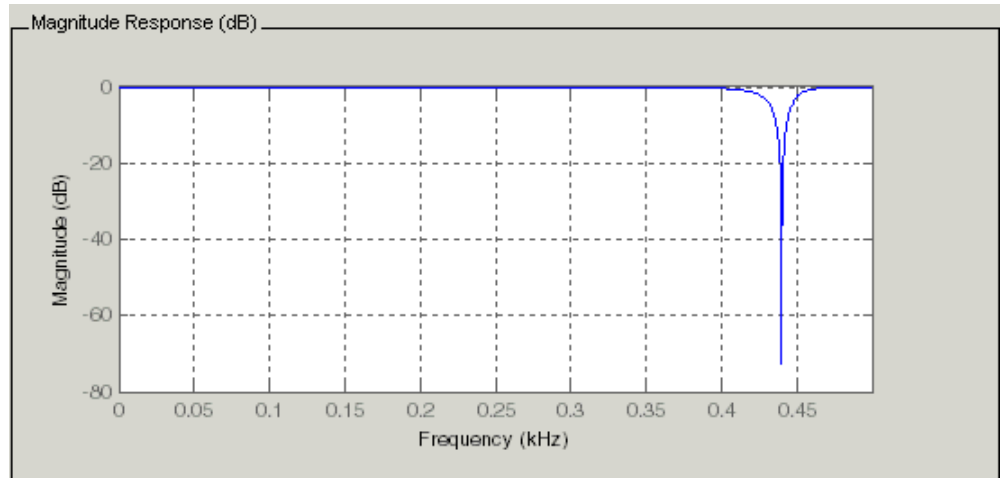
For more information about a design method, refer to the online Help system. For instance, to get further information about the **Q** setting for the notch filter in FDATool, enter

```
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 Signal Processing Toolbox documentation, rather than the page in 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.
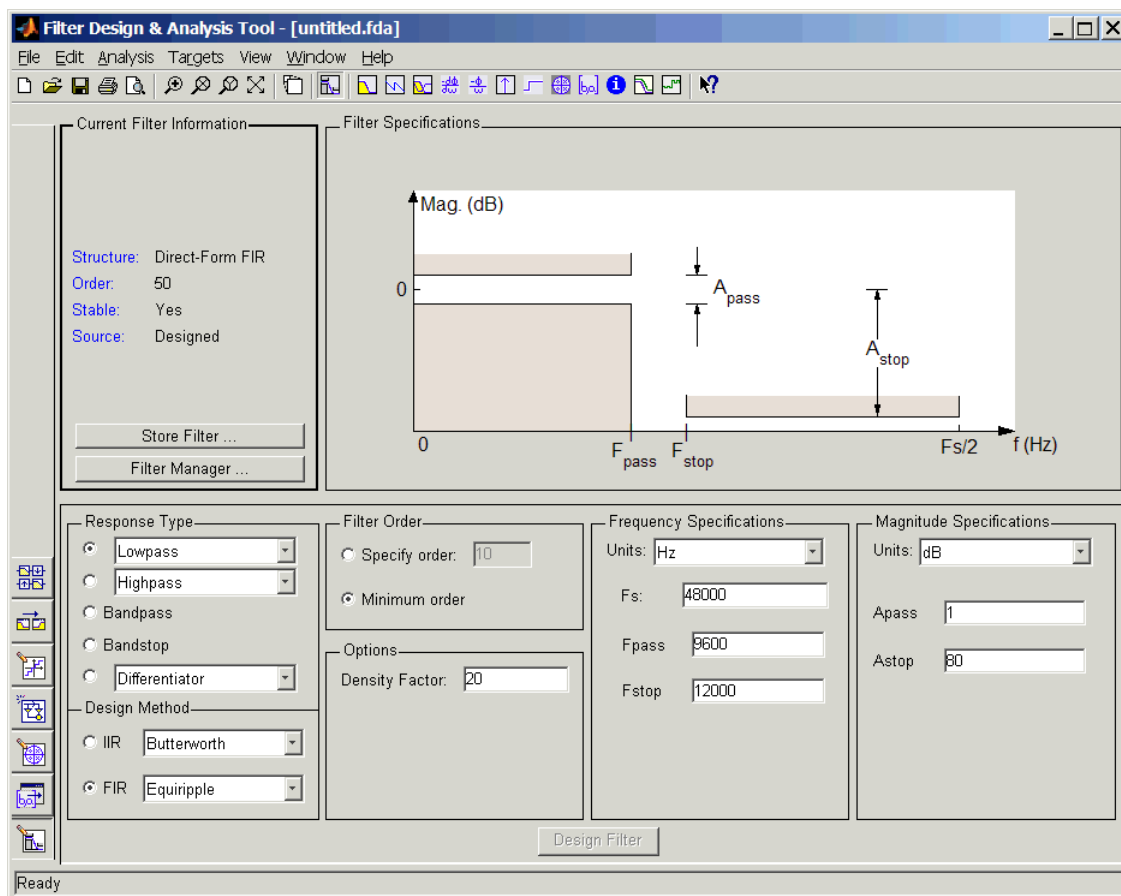
When FDATool opens, click the **Set Quantization Parameters** button on the side bar. FDATool switches to quantization mode and you see the following panel at the bottom of FDATool, with the default double-precision option shown for **Filter Arithmetic**.
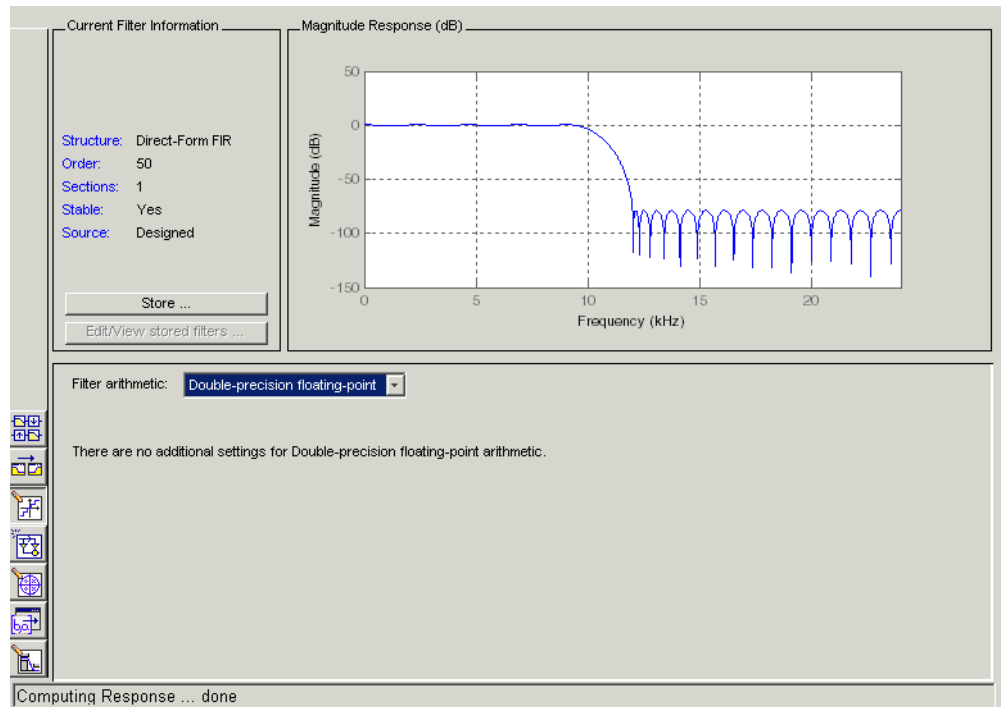
The **Filter Arithmetic** option lets you quantize filters and investigate the effects of changing quantization settings. To enable the quantization settings in FDATool, select Fixed-point from the **Filter Arithmetic**.

The quantization options appear in the lower panel of FDATool. You see tabs that access various sets of options for quantizing your filter.
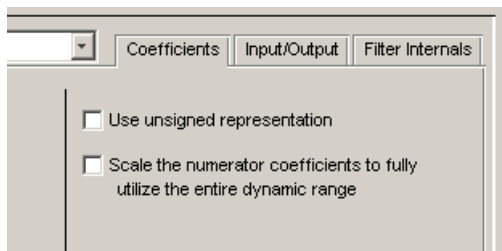
You use the following tabs in the dialog box to perform tasks related to quantizing filters in FDATool:

- **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

Setting Quantization Parameters

Quantized filters have properties that define how they quantize data you filter. Use the **Set Quantization Parameters** dialog box in FDATool to set the properties. Using options in the **Set Quantization Parameters** dialog box, 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.
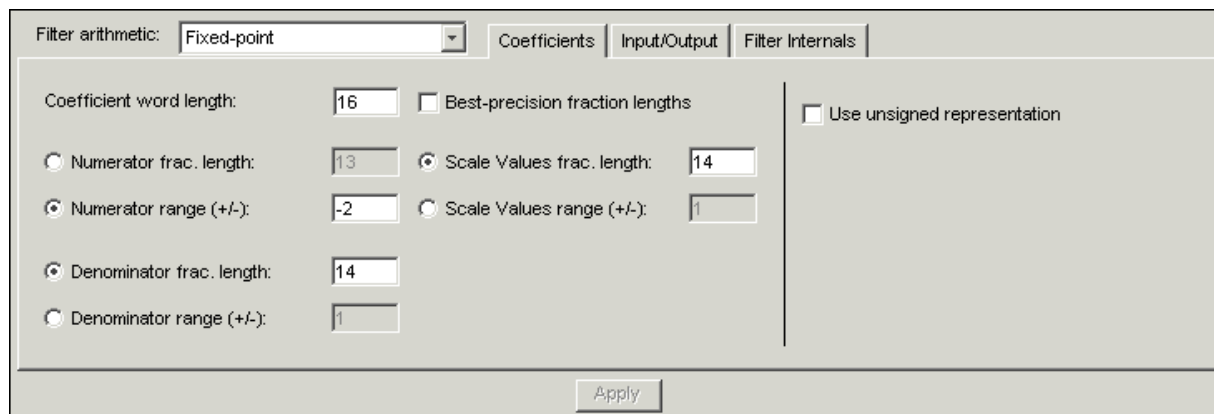
Coefficients Options

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
Numerator Word Length	FIR filters only	Sets the word length used to represent numerator coefficients in FIR filters.
Numerator Frac. Length	FIR/IIR	Sets the fraction length used to interpret numerator coefficients in FIR filters.
Numerator Range (+/-)	FIR/IIR	Lets you set the range the numerators represent. You use this instead of the **Numerator Frac. Length** 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
Coefficient Word Length	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.
Denominator Frac. Length	IIR filters	Sets the fraction length used to interpret denominator coefficients in IIR filters.
Denominator Range (+/-)	IIR filters	Lets you set the range the denominator coefficients represent. You use this instead of the **Denominator Frac. Length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Best-precision fraction lengths	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.
Scale Values frac. length	SOS IIR filters	Sets the fraction length used to interpret the scale values in SOS filters.

Option Name	When Used	Description
Scale Values range (+/-)	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 **Scale Values frac. length** option. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Use unsigned representation	All filters	Tells FDATool to interpret the coefficients as unsigned values.
Scale the numerator coefficients to fully utilize the entire dynamic range	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.

Option Name	When Used	Description
Input Word Length	All filters	Sets the word length used to represent the input to a filter.
Input fraction length	All filters	Sets the fraction length used to interpret input values to filter.
Input range (+/-)	All filters	Lets you set the range the inputs represent. You use this instead of the **Input fraction length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Output word length	All filters	Sets the word length used to represent the output from a filter.
Avoid overflow	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 **Output fraction length**.
Output fraction length	All filters	Sets the fraction length used to represent output values from a filter.
Output range (+/-)	All filters	Lets you set the range the outputs represent. You use this instead of the **Output fraction length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Stage input word length	SOS filters only	Sets the word length used to represent the input to an SOS filter section.
Avoid overflow	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 **Stage input fraction length**.

Option Name	When Used	Description
Stage input fraction length	SOS filters only	Sets the fraction length used to represent input to a section of an SOS filter.
Stage output word length	SOS filters only	Sets the word length used to represent the output from an SOS filter section.
Avoid overflow	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 **Stage output fraction length**.
Stage output fraction length	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.

Option	Equivalent Filter Property (Using Wildcard *)	Description
Round towards	`RoundMode`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). Choose from one of: • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix/zero` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
Overflow Mode	`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.
Filter Product (Multiply) Options		

Option	Equivalent Filter Property (Using Wildcard *)	Description
Product Mode	`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.
Product word length	`*ProdWordLength`	Sets the word length applied to interpret the results of multiply operations.
Num. fraction length	`NumProdFracLength`	Sets the fraction length used to interpret the results of product operations that involve numerator coefficients.
Den. fraction length	`DenProdFracLength`	Sets the fraction length used to interpret the results of product operations that involve denominator coefficients.
Filter Sum Options		
Accum. mode	`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.
Accum. word length	`*AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.

Option	Equivalent Filter Property (Using Wildcard *)	Description
Num. fraction length	`NumAccumFracLength`	Sets the fraction length used to interpret the numerator coefficients.
Den. fraction length	`DenAccumFracLength`	Sets the fraction length the filter uses to interpret denominator coefficients.
Cast signals before sum	`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.
Filter State Options		
State word length	`*StateWordLength`	Sets the word length used to represent the filter states. Applied to both numerator- and denominator-related states
Avoid overflow	None	Prevent overflows in arithmetic calculations by setting the fraction length appropriately.
State fraction length	`*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

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 Fs. 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.

Analysis Parameter	Default Setting	Description
Number of Points	512	Number of equally spaced points around the upper half of the unit circle.
Frequency Range	0 to Fs/2	Frequency range of the plot x-axis.

Analysis Parameter	Default Setting	Description
Frequency Units	Hz	Units for specifying the frequency range.
Sampling Frequency	48000	Inverse of the sampling period.
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 box and adjust your settings appropriately, such as changing the number of trials or number of points.

To open the **Analysis Parameters** dialog box, 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 box as shown in the figure. Notice that the settings for the options reflect the defaults.
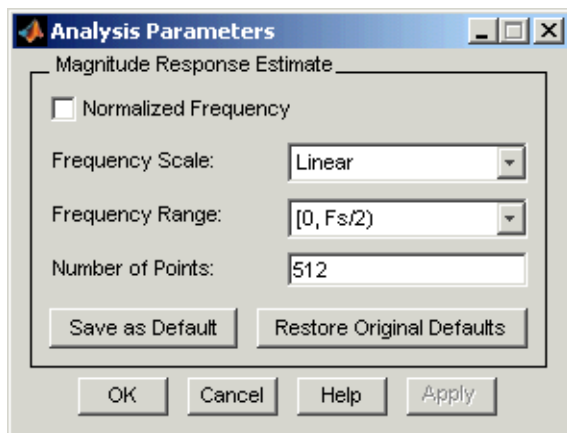
Example — Noise Method Applied to a Filter

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.
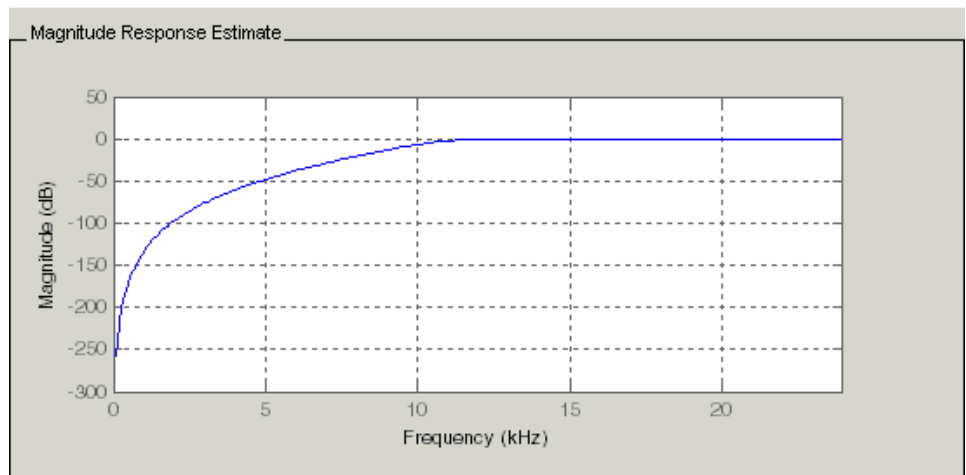
In the above figure you see the magnitude response as estimated by the analysis method.

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 box.

To Change Your Noise Analysis Parameters

In "Example — Noise Method Applied to a Filter" on page 4-22, 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" on page 4-20.

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 box opens as shown here (with default settings).
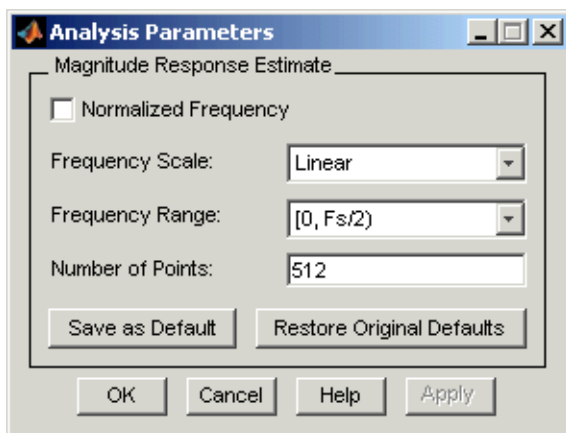
2 To use more points in the spectrum to estimate the magnitude response, change **Number of Points** to 1024 and click **OK** to run the analysis.

FDATool closes the **Analysis Parameters** dialog box and reruns the noise estimate, returning the results in the analysis area.

To rerun the test without closing the dialog box, press **Enter** after you type your new value into a setting, then click **Apply**. Now FDATool runs the test without closing the dialog box. 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 7-49

- "Direct Form II Filter Structure" on page 7-50

- "Direct Form I Filter Structure" on page 7-47

- "Direct Form Finite Impulse Response (FIR) Filter Structure" on page 7-56

- "Direct Form FIR Transposed Filter Structure" on page 7-57

- "Lattice Autoregressive Moving Average (ARMA) Filter Structure" on page 7-63

- dfilt.calattice

- dfilt.calatticepc

- "Direct Form Symmetric FIR Filter Structure (Any Order)" on page 7-64

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

In this section...
"Using the Reordering and Scaling Second-Order Sections Dialog Box" on page 4-28
"Example — Scale an SOS Filter" on page 4-30

Using the Reordering and Scaling Second-Order Sections Dialog Box

FDATool provides the ability to scale SOS filters after you create them. Using options on the Reordering and Scaling Second-Order Sections dialog box, FDATool scales either or both the filter numerators and filter scale values according to your choices for the scaling options.

Parameter	Description and Valid Value
Scale	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.
No Overflow — High SNR slider	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. 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.
Maximum Numerator	Maximum allowed value for numerator coefficients after scaling.
Numerator Constraint	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, except that the coefficients will be clipped if they exceed the **Maximum Numerator**. With `Normalize` the maximum absolute value of the numerator is forced to equal the **Maximum Numerator** value (for all other constraints, the **Maximum Numerator** is only an upper limit, above which coefficients will be clipped). The `power of 2` option forces scaling to use numerator values that are powers of 2, such as 2 or 0.5. With `unit`, the leading coefficient of each numerator is forced to a value of 1.

Parameter	Description and Valid Value
Overflow Mode	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.
Scale Value Constraint	Specify whether to constrain the filter scale values, and how to constrain them. Valid options are unit, power of 2, and none. Choosing unit for the constraint disables the **Max. Scale Value** setting and forces scale values to equal 1. 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, except that they cannot exceed the **Max. Scale Value**.
Max. Scale Value	Sets the maximum allowed scale values. SOS filter scaling applies the **Max. Scale Value** limit only when you set **Scale Value Constraint** to a value other than unit (the default setting). Setting a maximum scale value removes any other limits on the scale values.
Revert to Original Filter	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 Specification**s, 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.
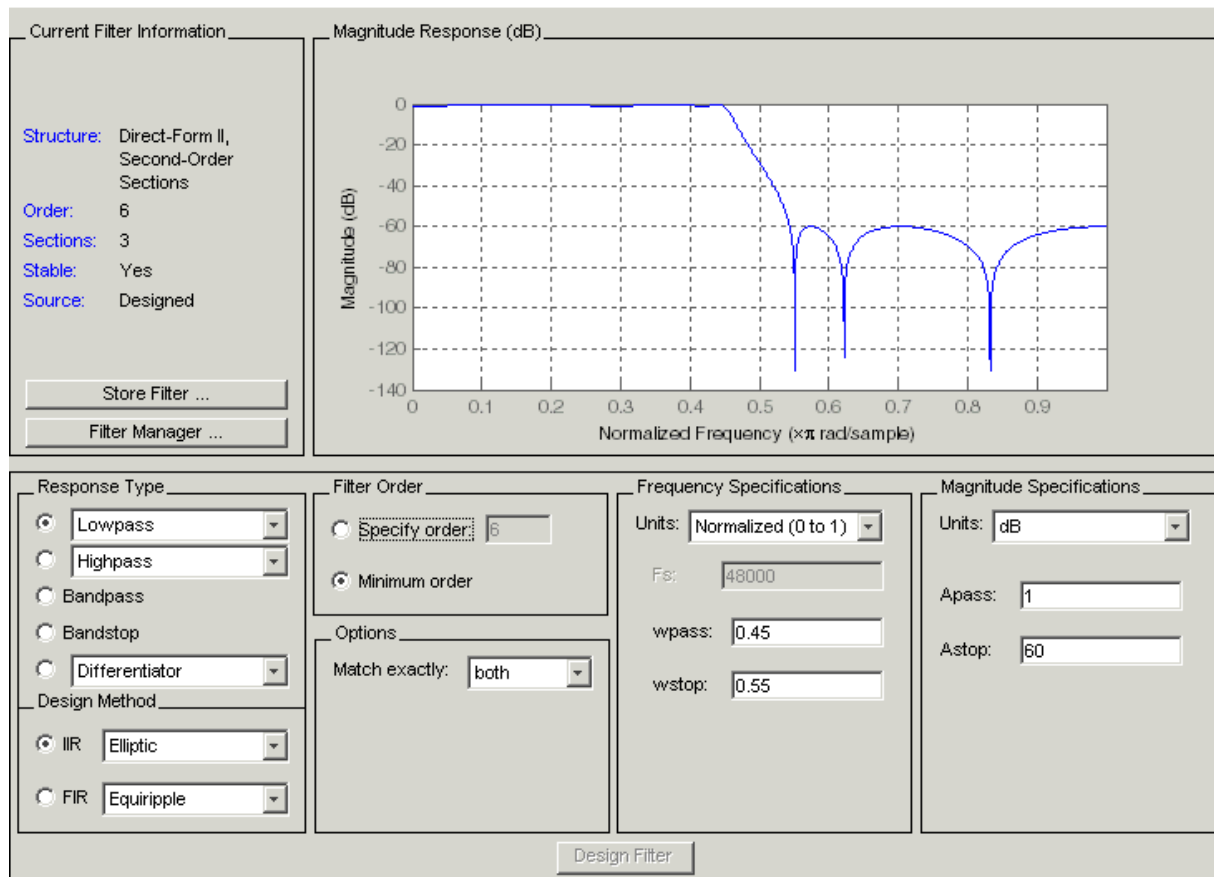
You kept the **Options** setting for **Match exactly** as both, meaning the filter design matches the specification for the passband and the stopband.

8 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 box shown here.
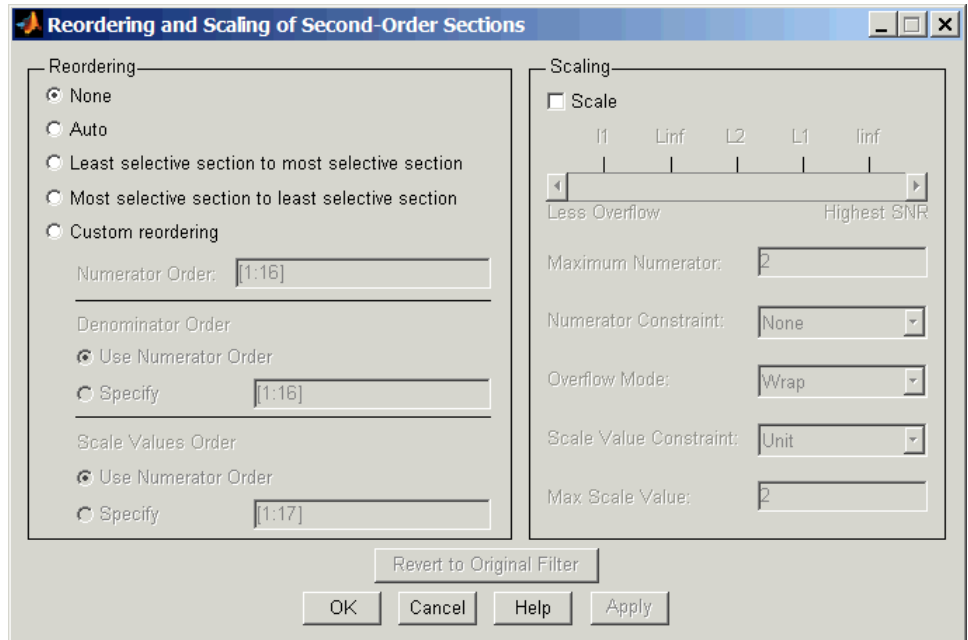
9 To see the filter coefficients, return to FDATool and select **Filter Coefficients** from the **Analysis** menu. FDATool displays the coefficients and scale values in FDATool.

Filter Coefficients

```
SOS matrix:
1  1.729397029770226  1  1  -1.087129242767174  0.418972372994757
1  0.748473366528337  1  1  -0.590169079330449  0.709160377407721
1  0.318456238149725  1  1  -0.304359210053485  0.922024429931248
Scale Factors:
0.079303838869308
0.407131941573331
0.697733773559928
1.000000000000000
```
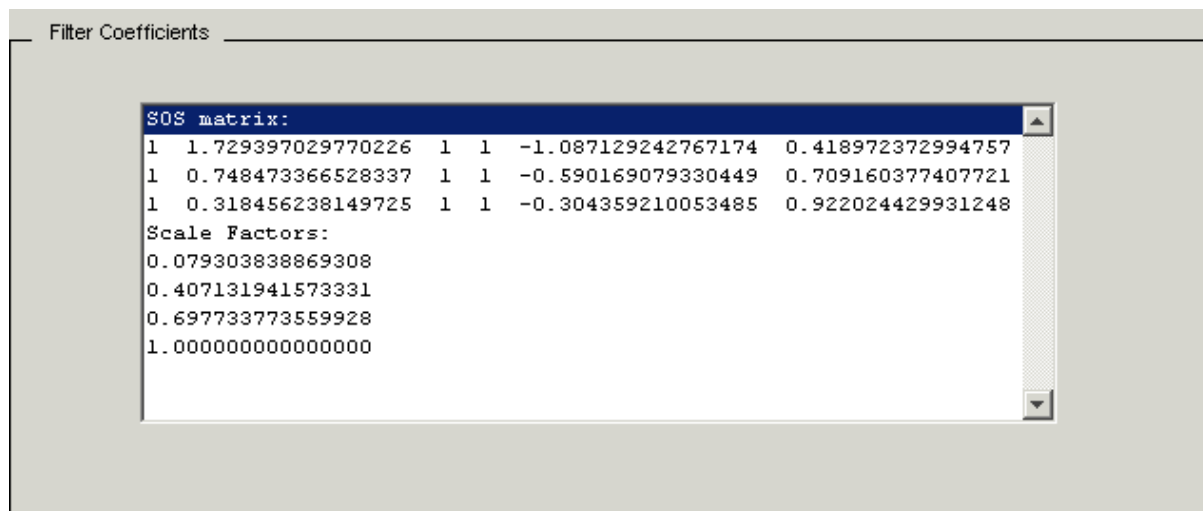
With the coefficients displayed you can see the effects of scaling your filter directly in the scale values and filter coefficients.

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 box 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 box open.

After a few moments, FDATool updates the coefficients displayed so you see the new scaling, as shown in the following figure.

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

Switching FDATool to Reorder 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.

To reorder the sections of a filter, you access the Reorder and Scaling of Second-Order Sections dialog box in FDATool.

With your SOS filter in FDATool, select **Edit > Reorder and Scale** from the menu bar. FDATool returns the reordering dialog box shown here with the default settings.
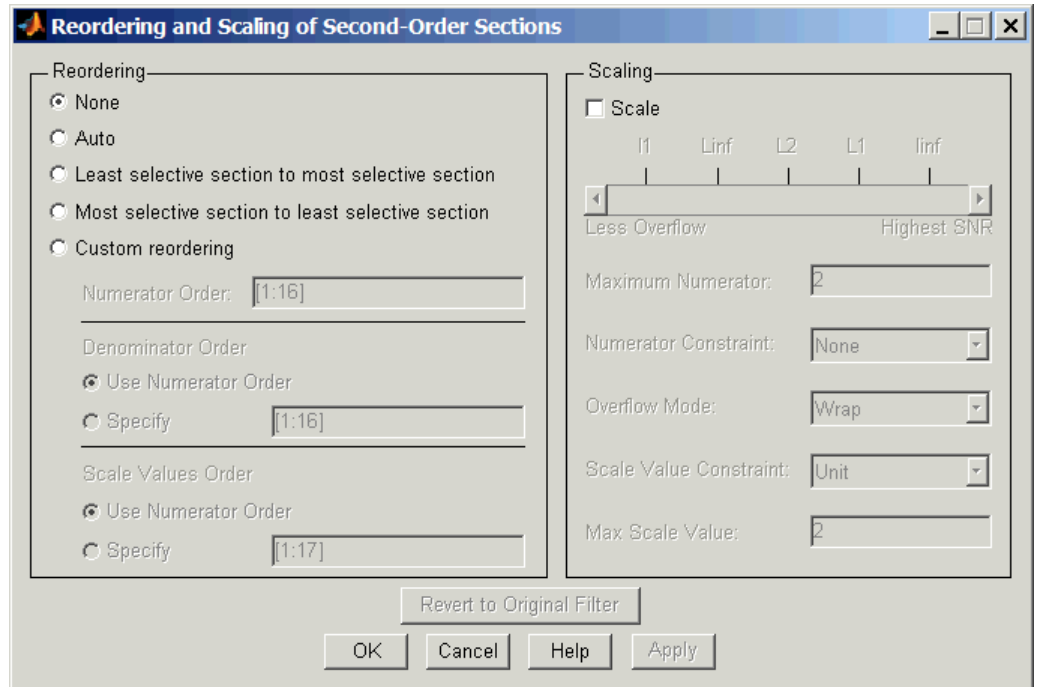
Controls on the Reordering and Scaling of Second-Order Sections dialog box

In this dialog box, 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 4-28 and to scale in the reference section.

Reordering SOS filters involves using the options in the **Reordering and Scaling of Second-Order Sections** dialog box. The following table lists each reorder option and provides a description of what the option does.

4-37

Control Option	Description
Auto	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.
None	Does no reordering on your filter. Selecting **None** lets you scale your filter without applying reordering at the same time. When you access this dialog box with a current filter, this is the default setting — no reordering is applied.
Least selective section to most selective section	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.
Most selective section to least selective section	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.
Custom reordering	Lets you specify the section ordering to use by enabling the Numerator Order and Denominator Order options
Numerator Order	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].
Use Numerator Order	Rearranges the denominators in the order assigned to the numerators.

Control Option	Description
Specify	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].
Use Numerator Order	Reorders the scale values according to the order of the numerators.
Specify	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].
Revert to Original Filter	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 you 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 **Fs** and **Fc** 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 box that controls reordering of the sections of your filter.
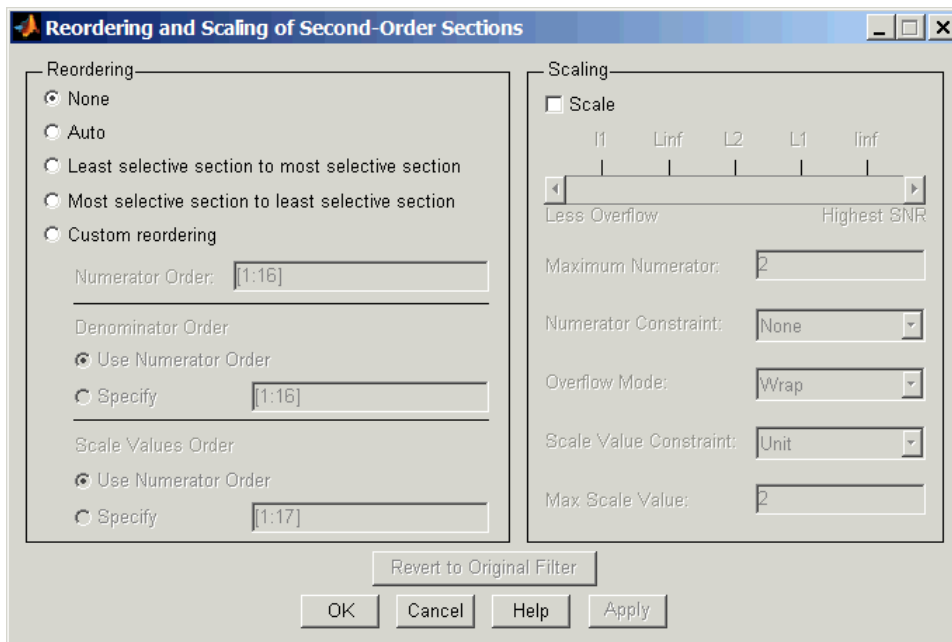
Now you are ready to reorder the sections of your filter. Note that FDATool performs the reordering on the current filter in the session.

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 box.

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 box, 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 box, click **Apply** to reorder your filter according to the Qs of the filter sections, and keep the dialog box 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.
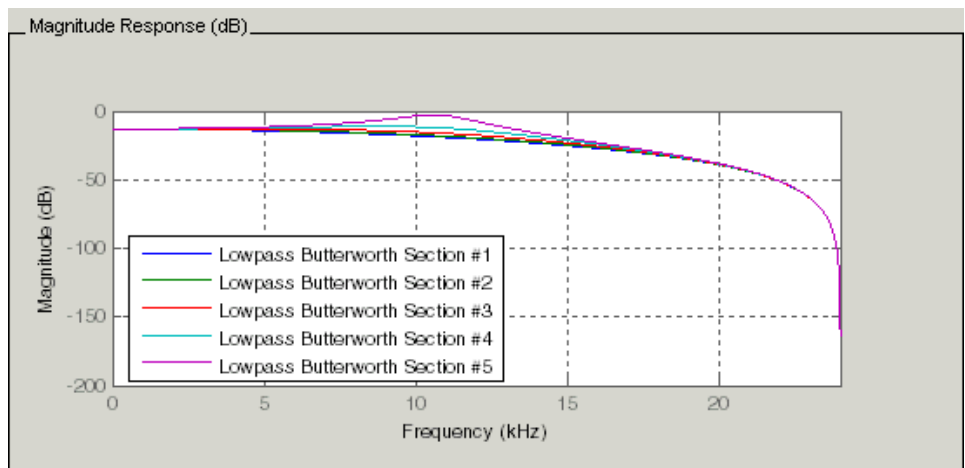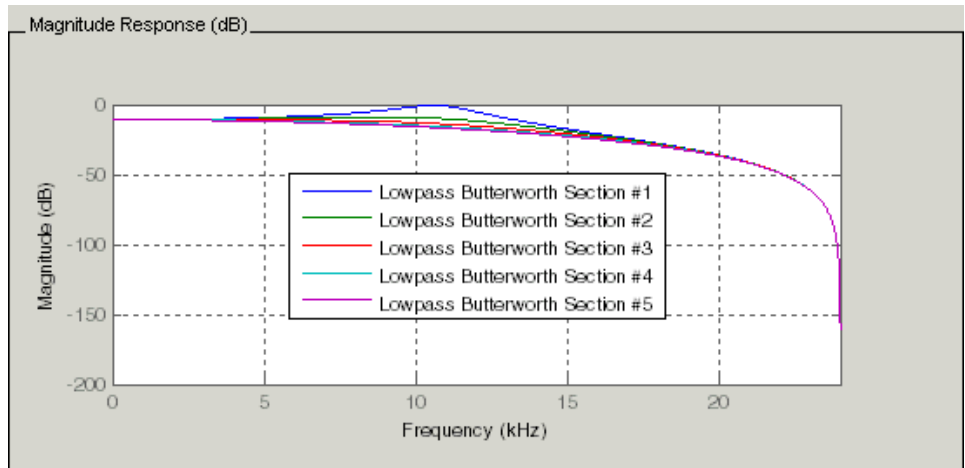
You see what reordering does, although the result is a bit subtle. Now try custom reordering the sections of your filter or using the most selective to least selective reordering option.

Viewing SOS Filter Sections

In this section...
"Using the SOS View Dialog Box" on page 4-43
"Example — View the Sections of SOS Filters" on page 4-46

Using the SOS View Dialog Box

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 Box

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 box containing options to configure SOS View operation.
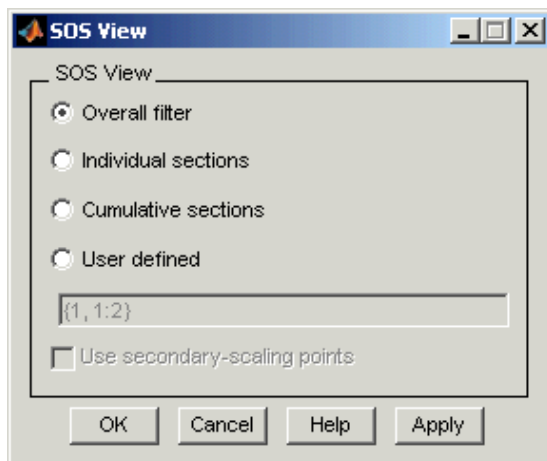
By default, SOS View shows the overall response of SOS filters. Options in the SOS View dialog box let you change the display. This table lists all the options and describes the effects of each.

Option	Description
Overall Filter	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.
Individual sections	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 **Cumulative sections**.

Option	Description
Cumulative sections	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: • The first curve plots the response for the first filter section. • The second curve plots the response for the combined first and second sections. • The third curve plots the response for the first, second, and third sections combined. And so on until all filter sections appear in the display. The final curve represents the overall filter response. Compare to **Cumulative sections** and **Overall Filter**.

Option	Description
User defined	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. 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. Note that after you enter the cell array, you need to click **OK** or **Apply** to update the FDATool analysis area to the new SOS View configuration.
Use secondary-scaling points	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 df2sos or df1tsos 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. You use this with the **Cumulative sections** option only.

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 box in FDATool. Options here let you specify how to display the filter sections.
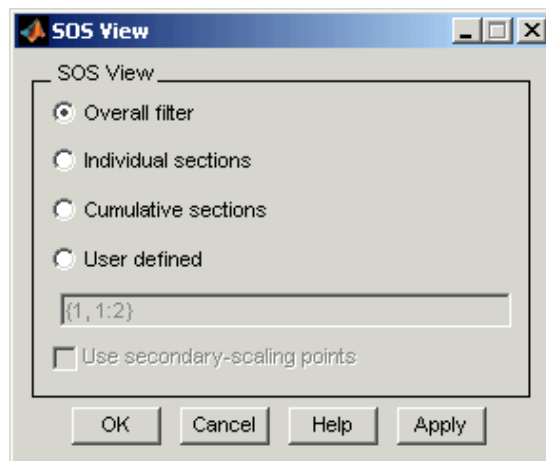
By default the analysis area in FDATool shows the overall filter response, not the individual filter section responses. This dialog box lets you change the display configuration to see the sections.

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.
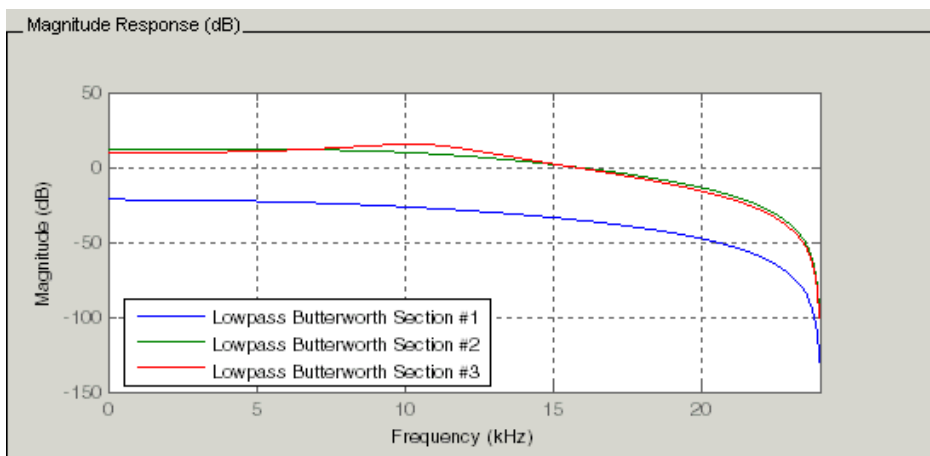
If you switch FDATool to display filter phase responses, you see the phase response for each filter section in the analysis area.
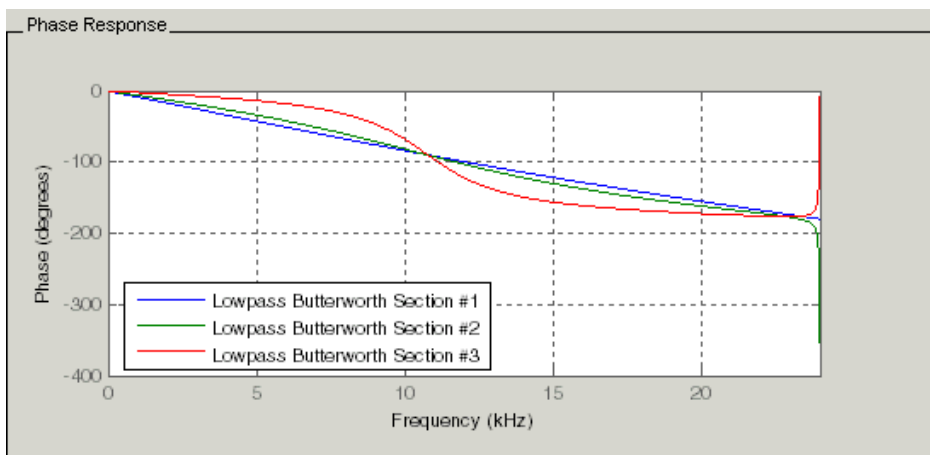
7 To define your own display of the sections, you use the **User defined** option and enter a vector of section indices to display. Now you see a

display of 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 box.

- 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 box).

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.
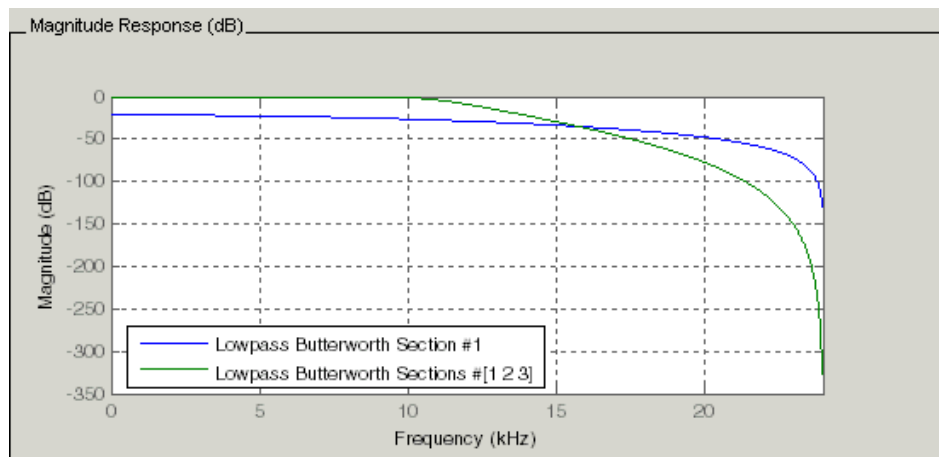
Importing and Exporting Quantized Filters

In this section...
"Overview and Structures" on page 4-50
"Example — Import Quantized Filters" on page 4-51
"To Export Quantized Filters" on page 4-52

Overview and Structures

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.

For general information about importing and exporting filters in FDATool, refer to "FDATool: A Filter Design and Analysis GUI" in the *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.
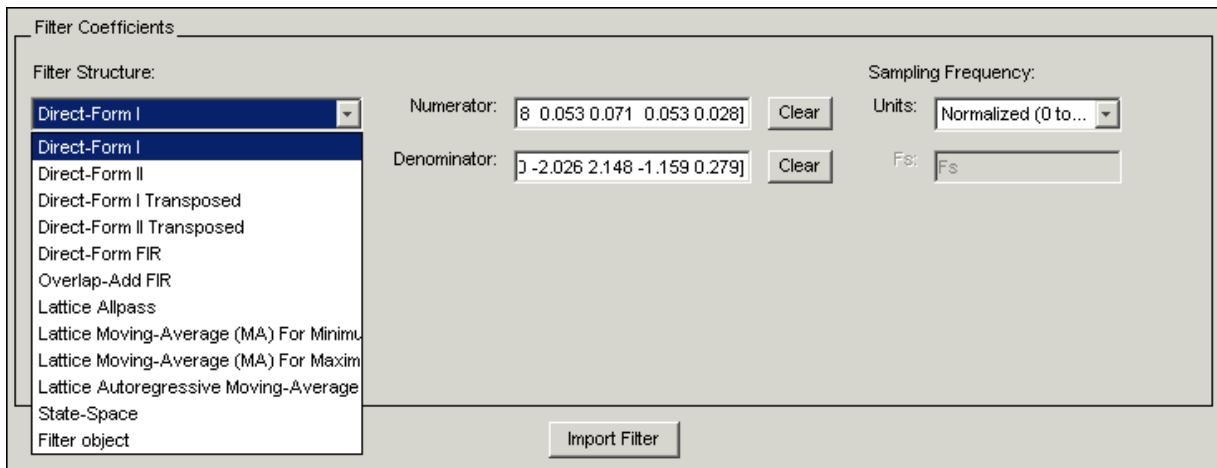
3 From the **Filter Structure** list, select Filter object.

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 **Fs** if needed. Your sampling frequency must correspond to the units you select. For example, when you select Normalized (0 to 1), **Fs** 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 box.

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 box 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 box, 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 box. Click **Apply** to export the filter without closing the **Export** dialog box. Clicking **Apply** lets you export your quantized filter to more than one name without leaving the **Export** dialog box.

The **Export Filter Coefficients to Text-file** dialog box appears. This is the standard Microsoft Windows save file dialog box.

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 box. Click **Apply** to export the filter without closing the **Export** dialog box. Clicking **Apply** lets you export your quantized filter to more than one name without leaving the **Export** dialog box.

The **Export Filter Coefficients to MAT-file** dialog box appears. This is the standard Microsoft Windows save file dialog box.

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

Example — Import XILINX .COE Files

You can import XILINX coefficients (.coe) files into FDATool to create quantized filters directly using the imported filter coefficients.

To use the 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 box, find and select the .coe file to import.

3 Click **Open** to dismiss the dialog box and start the import process.

FDATool imports the coefficient file and creates a quantized, single-section, direct-form FIR filter.

Transforming Filters

FDATool Filter Transformation Capabilities

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.

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.

Original Filter	Available Transformed Filter Types
Lowpass FIR	• **Lowpass** • **Lowpass (FIR)** • **Highpass** • **Highpass (FIR) narrowband** • **Highpass (FIR) wideband** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**
Lowpass IIR	• **Lowpass** • **Highpass** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**

Original Filter	Available Transformed Filter Types
Highpass FIR	• **Lowpass** • **Lowpass (FIR) narrowband** • **Lowpass (FIR) wideband** • **Highpass (FIR)** • **Highpass** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**
Highpass IIR	• **Lowpass** • **Highpass** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**
Bandpass FIR	• **Bandpass** • **Bandpass (FIR)**
Bandpass IIR	**Bandpass**
Bandstop FIR	• **Bandstop** • **Bandstop (FIR)**
Bandstop IIR	**Bandstop**

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 3-11 and "Frequency Transformations for Complex Filters" on page 3-26.

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 Chapter 3, "Digital Frequency Transformations".

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 3-11 and "Frequency Transformations for Complex Filters" on page 3-26.

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 Chapter 3, "Digital Frequency Transformations".

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.
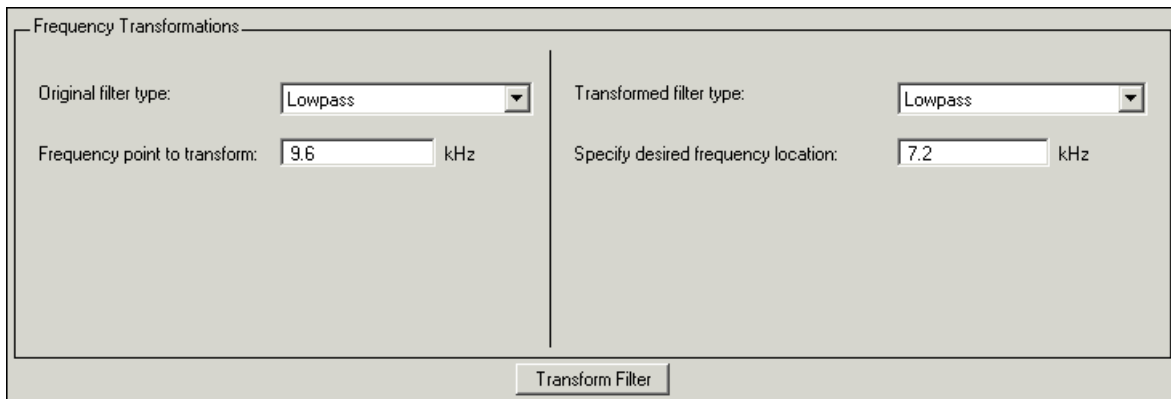
3 From the **Original filter type** list, select the response form of the filter you are transforming.

   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.

4-63

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.
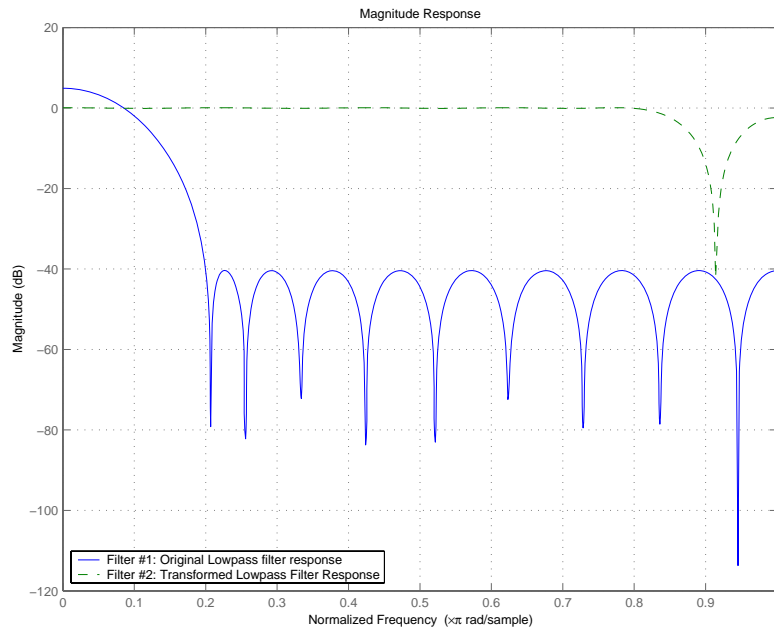
- To transform your lowpass filter to a highpass filter, select **Lowpass to Highpass**.

  When you select **Lowpass to Highpass**, FDATool returns the dialog box shown here. More information about the **Select Transform...** dialog box 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.
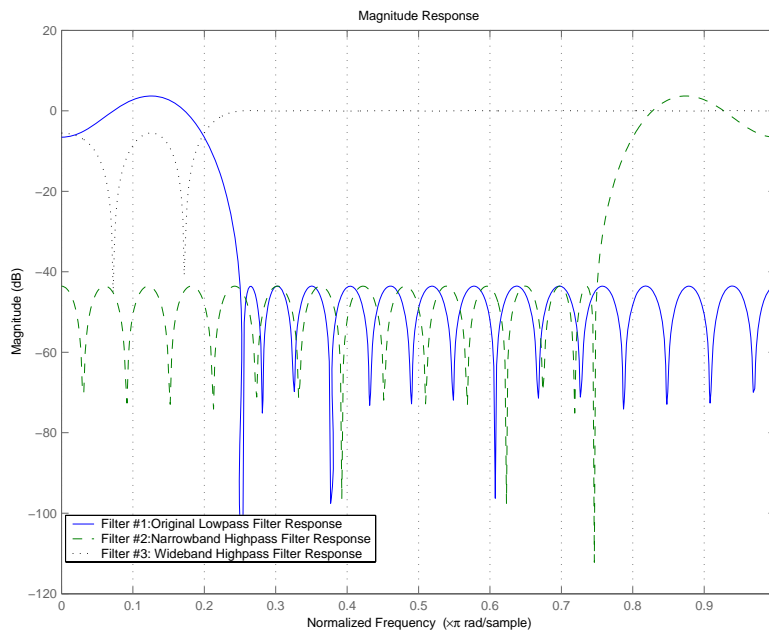
For the narrowband case, the transformation algorithm essentially reverses the magnitude response, like reflecting the curve around the *y*-axis, then translating the curve to the right until the origin lies at 1 on the *x*-axis. After reflecting and translating, the passband at high frequencies is the reverse of the passband of the original filter at low frequencies with the same rolloff and ripple characteristics.

Designing Multirate Filters in FDATool

In this section...

Introduction

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 following figure 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.
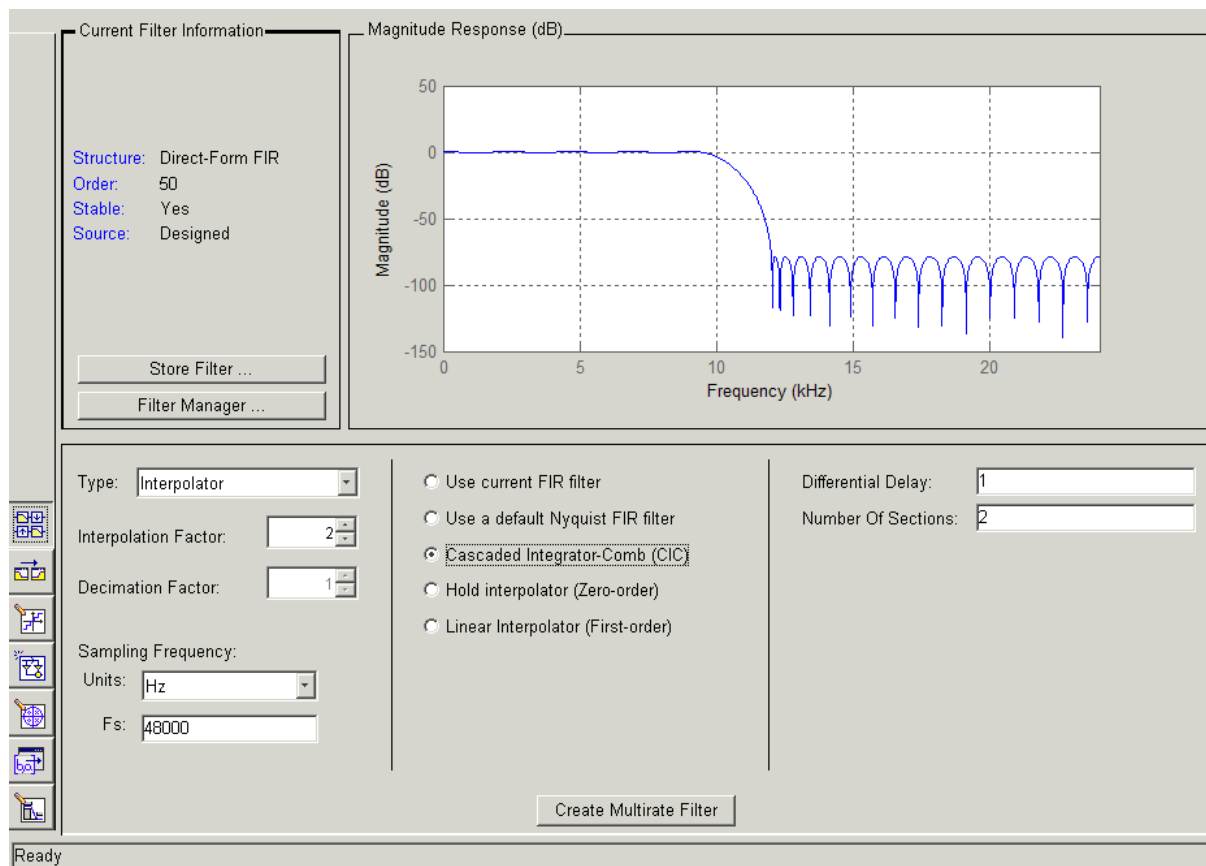
When the current filter in FDATool is not an FIR filter, the multirate filter design panel removes the **Use current FIR filter** option and selects the **Use default Nyquist FIR filter** option instead as the default setting.

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.

Selecting and Configuring Your Filter

Option	Description
Type	Specifies the type of multirate filter to design. Choose from Decimator, Interpolator, or Fractional-rate convertor. • When you choose Decimator, set **Decimation Factor** to specify the decimation to apply. • When you choose Interpolator, set **Interpolation Factor** to specify the interpolation amount applied. • When you choose Fractional-rate convertor, set both **Interpolation Factor** and **Decimation Factor**. FDATool uses both to determine the fractional rate change by dividing **Interpolation Factor** by **Decimation Factor** to determine the fractional rate change in the signal. 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. For example, if the interpolation factor is 6 and the decimation factor is 3, FDATool reduces 6/3 to 2/1 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.
Interpolation Factor	Use the up-down control arrows to specify the amount of interpolation to apply to the signal. Factors range upwards from 2.
Decimation Factor	Use the up-down control arrows to specify the amount of decimation to apply to the signal. Factors range upwards from 2.

Selecting and Configuring Your Filter (Continued)

Option	Description
Sampling Frequency	No settings here. Just **Units** and **Fs** below.
Units	Specify whether **Fs** is specified in Hz, kHz, MHz, GHz, or Normalized (0 to 1) units.
Fs	Set the full scale sampling frequency in the frequency units you specified in **Units**. When you select Normalized for **Units**, you do not enter a value for **Fs**.

Designing Your Filter

Option	Description
Use current FIR filter	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.
Use a default Nyquist Filter	Tells FDATool to use the default Nyquist design method when the current filter in FDATool is not an FIR filter.
Cascaded Integrator-Comb (CIC)	Design CIC filters using the options provided in the right-hand area of the multirate design panel.

Designing Your Filter (Continued)

Option	Description
Hold Interpolator (Zero-order)	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 **Linear Interpolator** option.
Linear Interpolator (First-order)	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 **Linear Interpolator** option.

To see the difference between hold interpolation and linear interpolation, the following figure presents a sine wave signal s1 in three forms:

- The top subplot in the figure presents signal 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.

Uninterpolated Signal s1

First-Order Linear Interpolation By 5 of Signal s1

Zero-Order Hold Interpolation By 5 of Signal s1
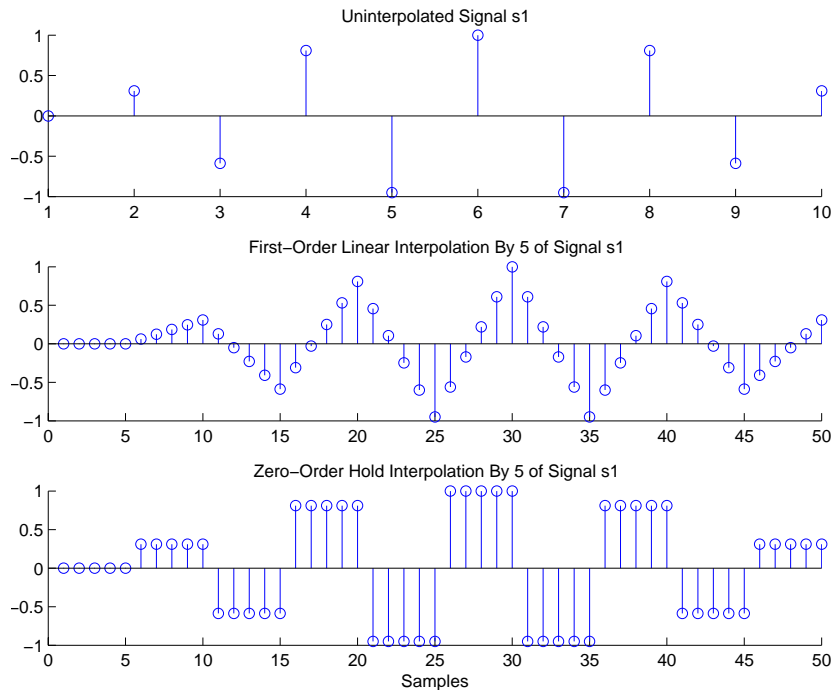
We used FDATool to create interpolators similar to the following code for the figure:

- Linear interpolator — `hm=mfilt.linearinterp(5)`

- Hold interpolator — `hm=mfilt.holdinterp(5)`

Options for Designing CIC Filters	Description
Differential Delay	Sets the differential delay for the CIC filter. Usually a value of one or two is appropriate.
Number of Sections	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 in the following figure.
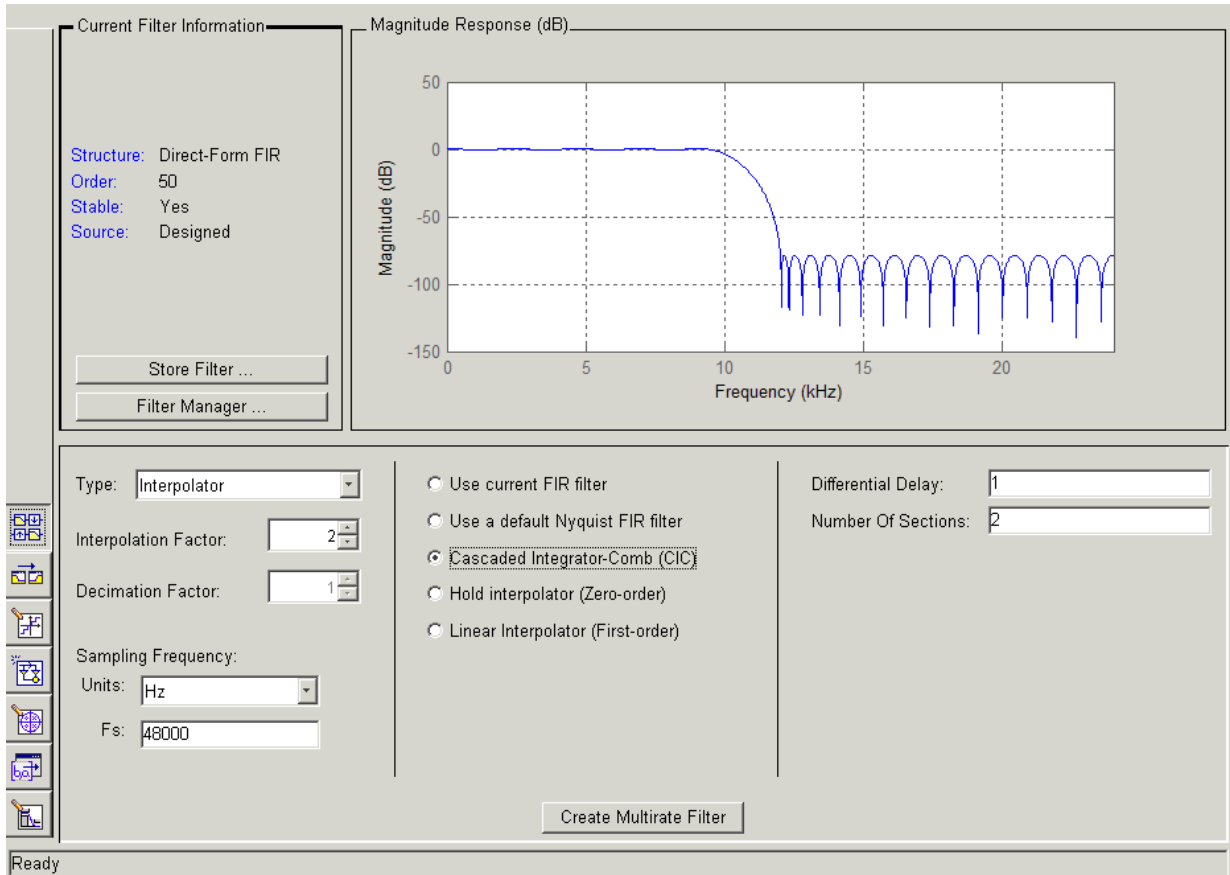
4 To design a fractional rate filter, select `Fractional-rate convertor` from the **Type** list. The **Interpolation Factor** and **Decimation Factor** options become available.

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 **Fs**.

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.
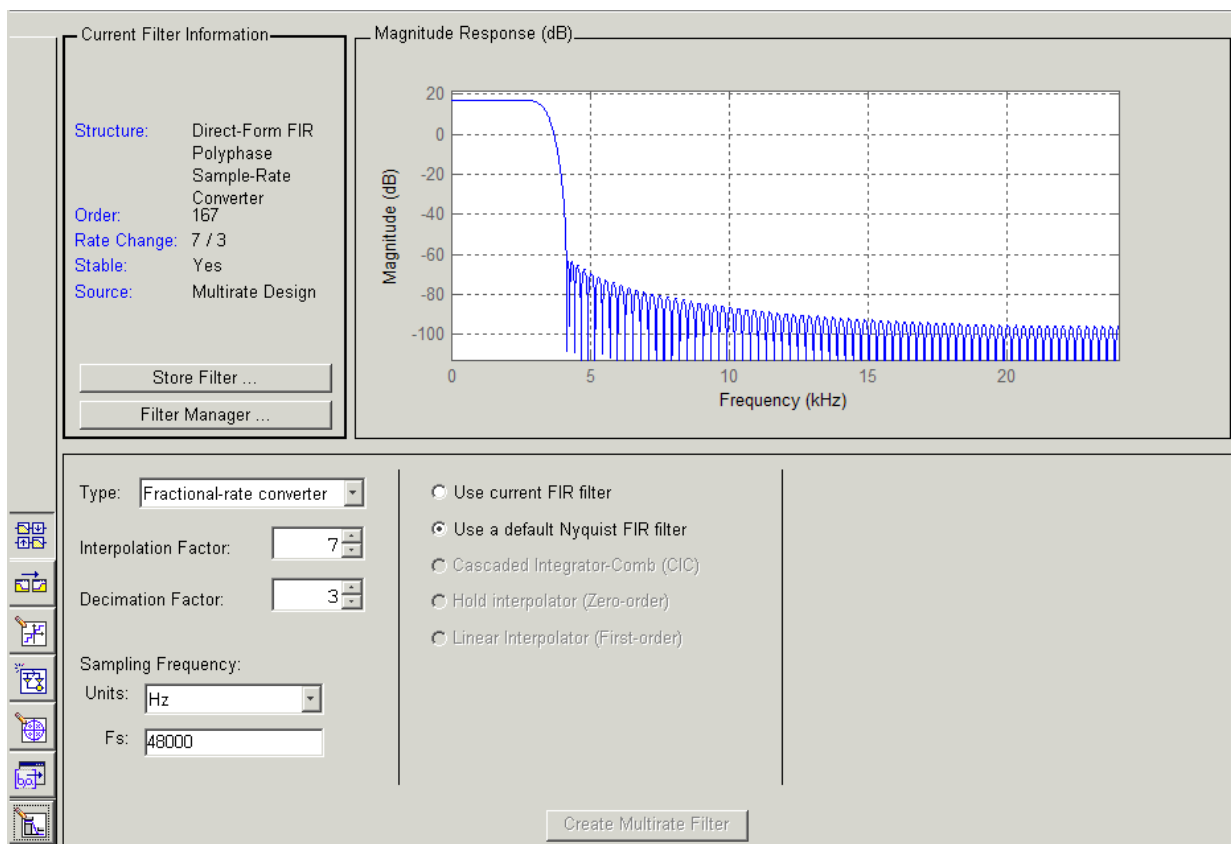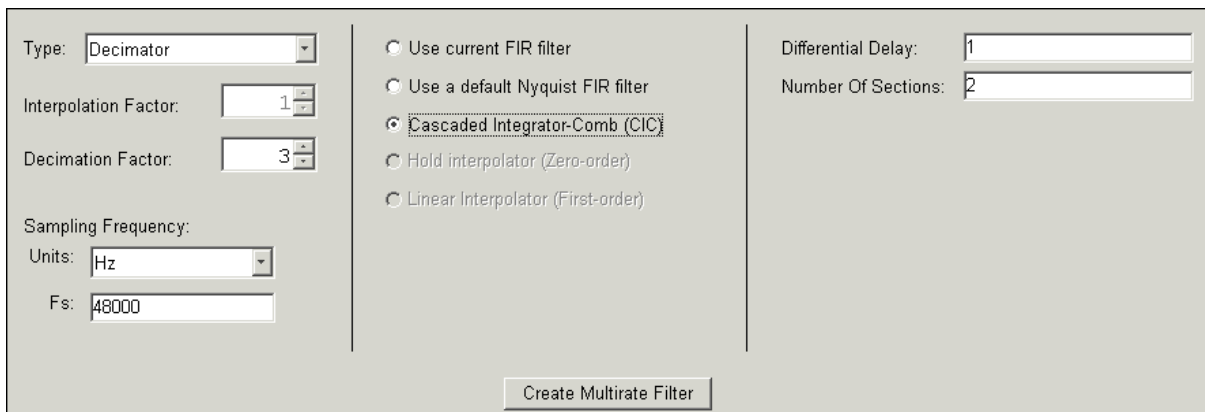
You can test the filter by exporting it to your workspace and using it to filter a signal. For information about exporting filters, refer to "Importing and Exporting Quantized Filters" on page 4-50.

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.

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.
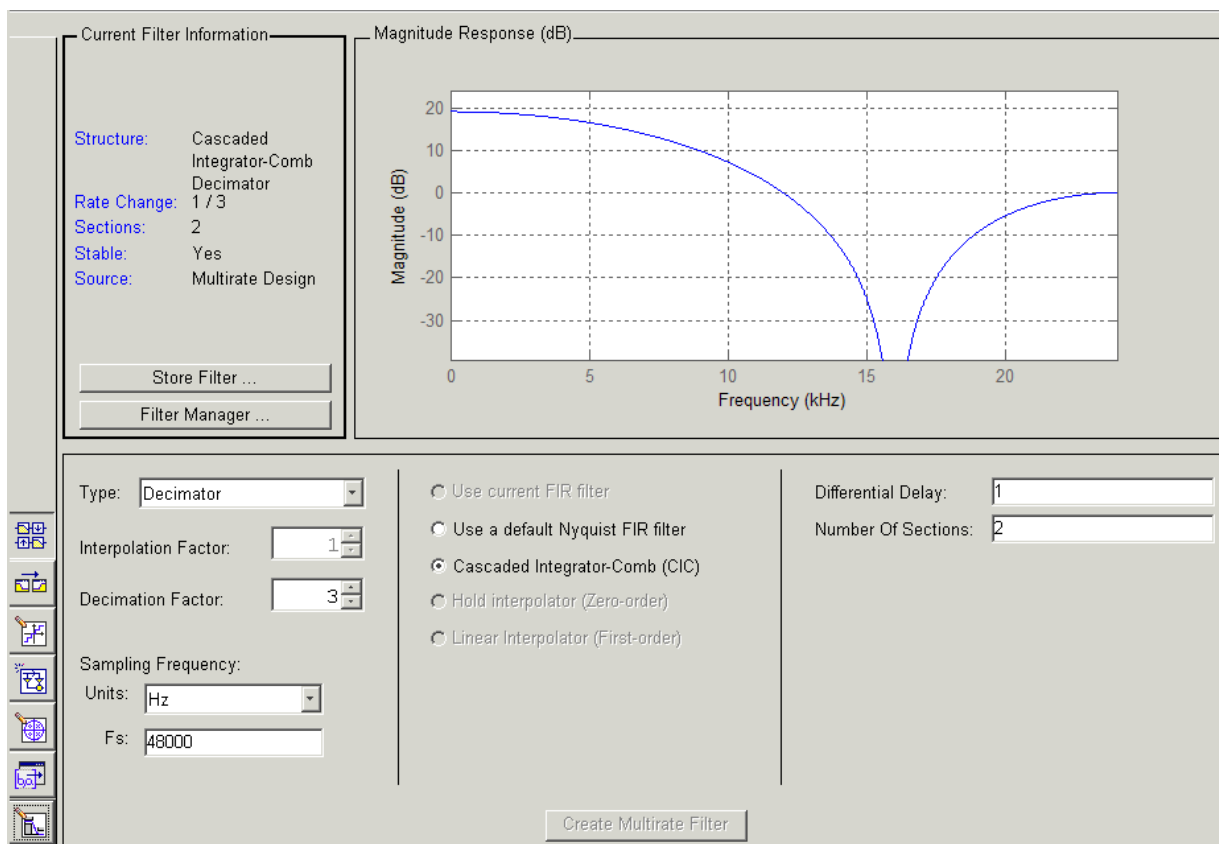
Designing other multirate filters follows the same pattern.

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**.

4-79

5 Click **Apply**.

When you 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
Input Word Length	Sets the word length used to represent the input to a filter.
Input fraction length	Sets the fraction length used to interpret input values to filter.
Input range (+/-)	Lets you set the range the inputs represent. You use this instead of the **Input fraction length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Output word length	Sets the word length used to represent the output from a filter.
Avoid overflow	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 **Output fraction length**.
Output fraction length	Sets the fraction length used to represent output values from a filter.
Output range (+/-)	Lets you set the range the outputs represent. You use this instead of the **Output fraction length** 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_{accum} by Harris in [2]. 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 [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.

Exporting the Individual Phase Coefficients of a Polyphase Filter to the Workspace

After designing a polyphase filter in Filter Design Analysis Tool (FDATool), you can obtain the individual phase coefficients of the filter by:

1 Exporting the filter to an object in the MATLAB workspace.

2 Using the polyphase method to create a matrix of the filter's coefficients.

Exporting the Polyphase Filter to an Object

To export a polyphase filter to an object in the MATLAB workspace, complete the following steps.

1 In FDATool, open the **File** menu and select **Export...**. This opens the dialog box for exporting the filter coefficients.
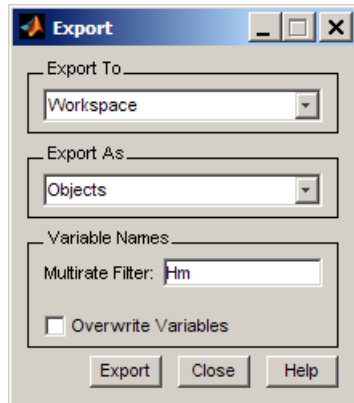
2 In the Export dialog box, for **Export To**, select **Workspace**.

3 For **Export As**, select **Object**.

4 (Optional) For **Variable Names**, enter the name of the **Multirate Filter** object that will be created in the MATLAB workspace.

5 Click the **Export** button. The multirate filter object, Hm in this example, appears in the MATLAB workspace.
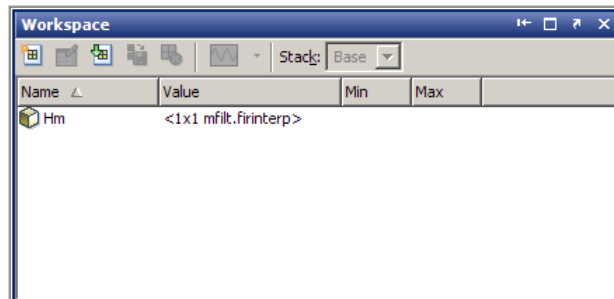
Using polyphase() to Create a Matrix of Coefficients

To create a matrix of the filter's coefficients, enter p=polyphase(Hm) at the command line. The polyphase method creates a matrix, p, of filter coefficients from the filter object, Hm. Each row of p consists of the coefficients of an individual phase subfilter. The first row contains to the coefficients of the first phase subfilter, the second row contains those of the second phase subfilter, and so on.

Realizing Filters as Simulink Subsystem Blocks

In this section...
"Introduction" on page 4-84
"About the Realize Model Panel in FDATool" on page 4-84
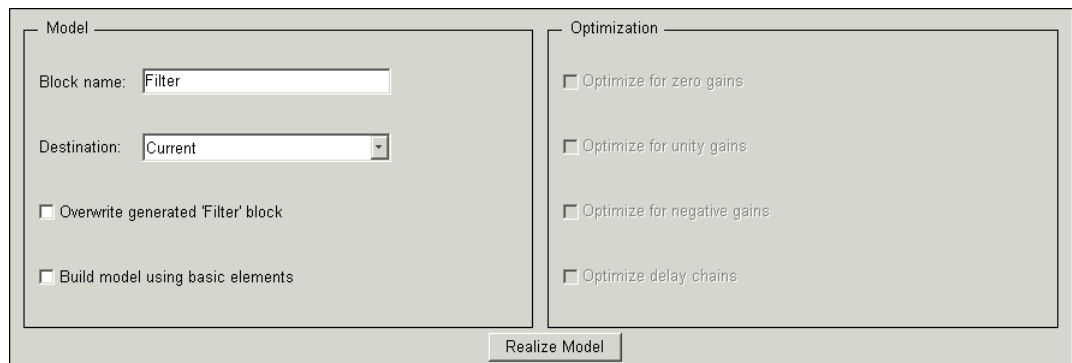
Introduction

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 either the Digital Filter block or the delay, gain, and sum blocks in Simulink. If you do not own Simulink® Fixed Point™software, 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.

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 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.

Build block using basic elements. You can determine how FDATool models the specified filter using this check box. When you select this check box, FDATool creates a subsystem block that implements your filter using Sum, Gain, and Delay blocks. When you clear this check box, FDATool uses a Digital Filter block to implement your filter. Filters that you realize with the Digital Filter block accept sample-based, vector, or frame-based input.

The **Build model using basic elements** check box is available only when your filter can be implemented using a Digital Filter block.

Note Filters that use only basic elements accept individual sample-based input, not input vectors or frames. The mathematics of filtering a frame-based input signal with a filter constructed of basic blocks involves an algebraic loop that Simulink cannot solve. If your input data is in frames, consider unbuffering the input, converting the frames to sample-by-sample input in some other way, or clearing the **Build block using basic elements** option to implement your filter with the Digital Filter block.

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

Structure	Description
`firinterp`	Interpolators based on FIR filters
`multirate polyphase`	Multirate filters
`holdinterp`	Interpolators that use the hold interpolation algorithm
`dfilt.allpass`	Discrete-time filters with allpass structure
`dfilt.cascadeallpass`	
`dfilt.cascadewdfallpass`	
`mfilt.iirdecim`	Decimators based on IIR filters
`mfilt.iirwdfdecim`	
`mfilt.iirinterp`	Interpolators based on IIR filters
`mfilt.iirwdfinterp`	
`dfilt.wdfallpass`	

Getting Help for FDATool

In this section...
"The What's This? Option" on page 4-89
"Additional Help for FDATool" on page 4-89

The What's This? Option

To find information on a particular option or region of the dialog box:

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 "FDATool: A Filter Design and Analysis GUI" in your Signal Processing Toolbox documentation.

5

Adaptive Filters

Introducing Adaptive Filtering

Adaptive filtering involves the changing of filter parameters (coefficients) over time, to adapt to changing signal characteristics. Over the past three decades, digital signal processors have made great advances in increasing speed and complexity, and reducing power consumption. As a result, real-time adaptive filtering algorithms are quickly becoming practical and 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.

- "Overview of Adaptive Filters and Applications" on page 5-4 presents a general discussion of adaptive filters and their applications.

  - "System Identification" on page 5-7 — 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 5-8—Using adaptive filters to develop a filter that has a response that is the inverse of an unknown system.

  - "Noise or Interference Cancellation" on page 5-9— Performing active noise cancellation where the filter adapts in real-time to remove noise by keeping the error small.

  - "Prediction" on page 5-9 — describes using adaptive filters to predict a signal's future values.

- "System Identification" on page 5-7 describes the important considerations for selecting an adaptive filter for an application.

- "Adaptive Filters in Filter Design Toolbox Software" on page 5-11 lists the adaptive filters included in the toolbox.

- "Examples of Adaptive Filters That Use LMS Algorithms" on page 5-15 presents a discussion of using LMS techniques to perform the filter adaptation process.

- "Example of Adaptive Filter That Uses RLS Algorithm" on page 5-36 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 5-41.

Overview of Adaptive Filters and Applications

Adaptive Filtering Methodology

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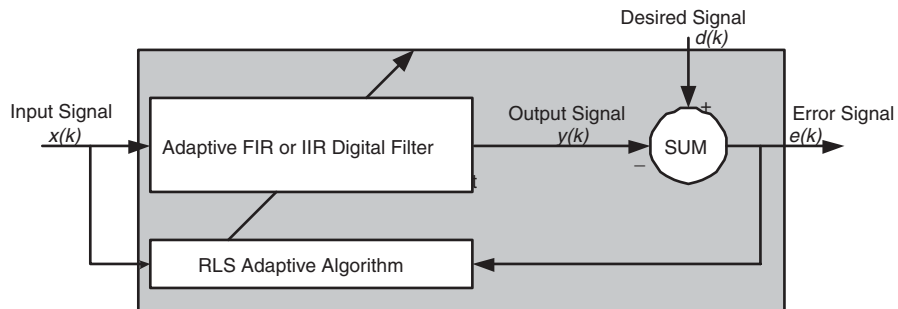
Block Diagram That Defines the Inputs and Output of a Generic RLS Adaptive Filter

The next figure provides the general adaptive filter setup with inputs and outputs.
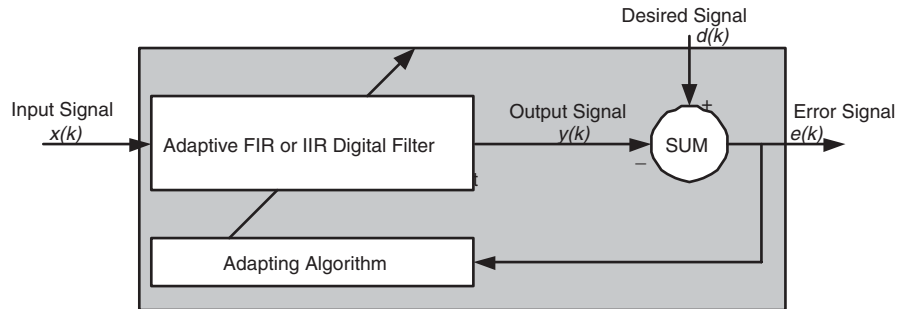
Block Diagram Defining General Adaptive Filter Algorithm Inputs and Outputs

Filter Design Toolbox software 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 Filter Design Toolbox Software" on page 5-11 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 Filter Design Toolbox software 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.
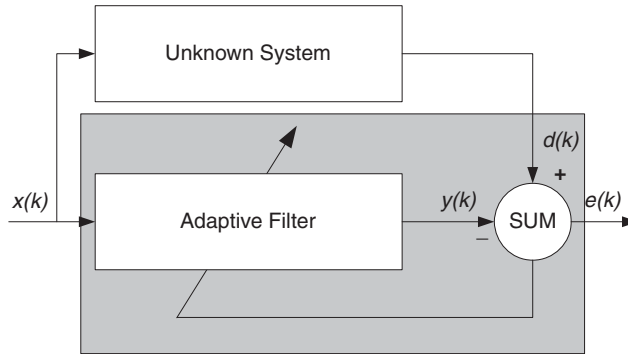
Using an Adaptive Filter to Identify an Unknown System

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.
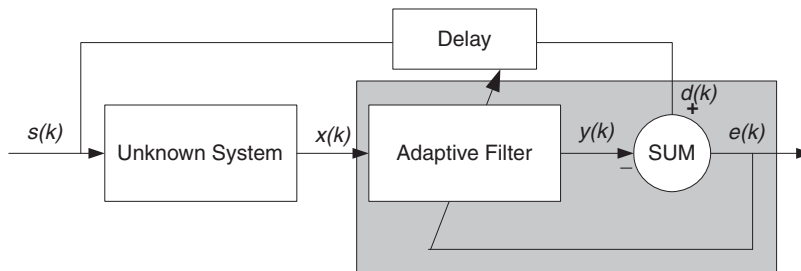
Determining an Inverse Response to an Unknown System

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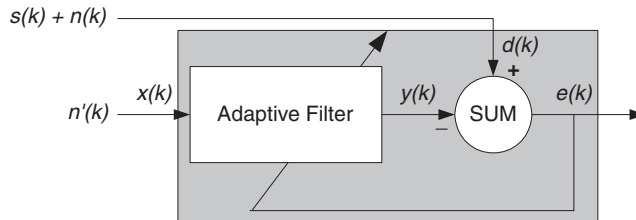
Using an Adaptive Filter to Remove Noise from an Unknown System

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.
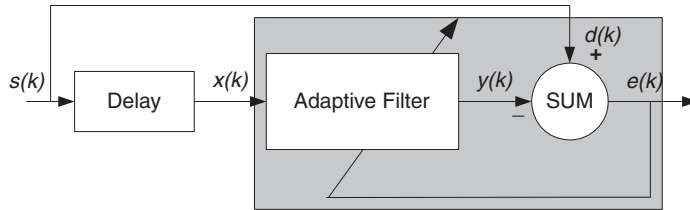
Predicting Future Values of a Periodic Signal

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 Filter Design Toolbox Software

In this section...
"Overview of Adaptive Filtering in Filter Design Toolbox Software" on page 5-11
"Algorithms" on page 5-11
"Using Adaptive Filter Objects" on page 5-14

Overview of Adaptive Filtering in Filter Design Toolbox Software

Filter Design Toolbox software 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

Adaptive Filter Method	Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
adaptfilt.adjlms	Adjoint LMS FIR adaptive filter algorithm
adaptfilt.blms	Block LMS FIR adaptive filter algorithm
adaptfilt.blmsfft	FFT-based Block LMS FIR adaptive filter algorithm
adaptfilt.dlms	Delayed LMS FIR adaptive filter algorithm
adaptfilt.filtxlms	Filtered-x LMS FIR adaptive filter algorithm
adaptfilt.lms	LMS FIR adaptive filter algorithm
adaptfilt.nlms	Normalized LMS FIR adaptive filter algorithm
adaptfilt.sd	Sign-data LMS FIR adaptive filter algorithm
adaptfilt.se	Sign-error LMS FIR adaptive filter algorithm
adaptfilt.ss	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

Adaptive Filter Method	Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
adaptfilt.ftf	Fast transversal least-squares adaptation algorithm
adaptfilt.qrdrls	QR-decomposition RLS adaptation algorithm
adaptfilt.hrls	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

Adaptive Filter Method	Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
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)

Adaptive Filter Method	Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
adaptfilt.fdaf	Frequency domain adaptation algorithm
adaptfilt.pbfdaf	Partition block version of the FDAF algorithm
adaptfilt.pbufdaf	Partition block unconstrained version of the FDAF algorithm
adaptfilt.tdafdct	Transform domain adaptation algorithm using DCT
adaptfilt.tdafdft	Transform domain adaptation algorithm using DFT
adaptfilt.ufdaf	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

Adaptive Filter Method	Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
`adaptfilt.gal`	Gradient adaptive lattice filter adaptation algorithm
`adaptfilt.lsl`	Least squares lattice adaptation algorithm
`adaptfilt.qrdlsl`	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 5-41.

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 5-15
- "Example of Adaptive Filter That Uses RLS Algorithm" on page 5-36

Examples of Adaptive Filters That Use LMS Algorithms

LMS Methods Available in Filter Design Toolbox Software

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 fircband examples.

```
[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 5-7 for the layout.
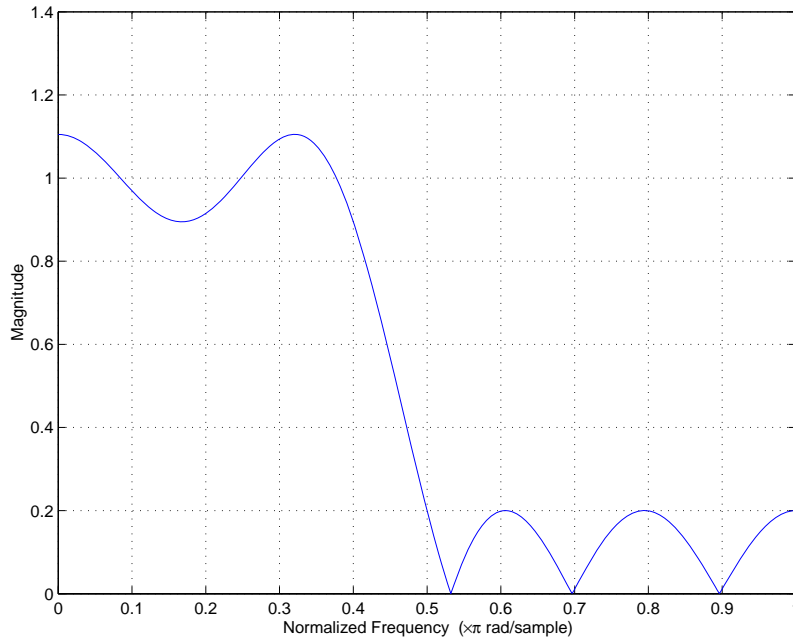
For the sign variations of the LMS algorithm, the examples use noise cancellation as the demonstration application, as opposed to the system identification application used in the LMS examples.

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(12,[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);
```

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.'])
```
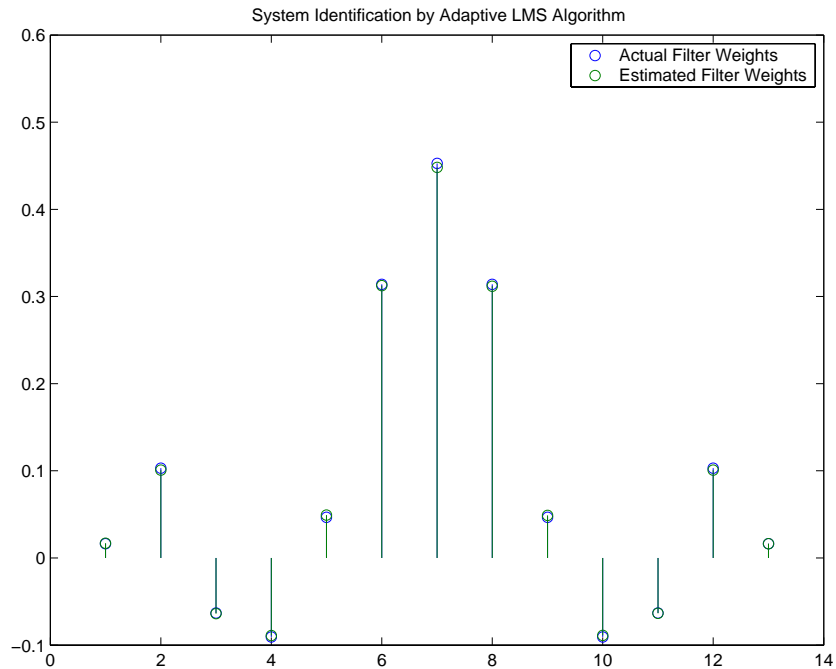

In the stem plot the actual and estimated filter weights are the same. As an experiment, try changing the step size to 0.2. Repeating the example with

`mu = 0.2` results in the following stem plot. The estimated weights fail to approximate the actual weights closely.

System Identification by Adaptive LMS Algorithm
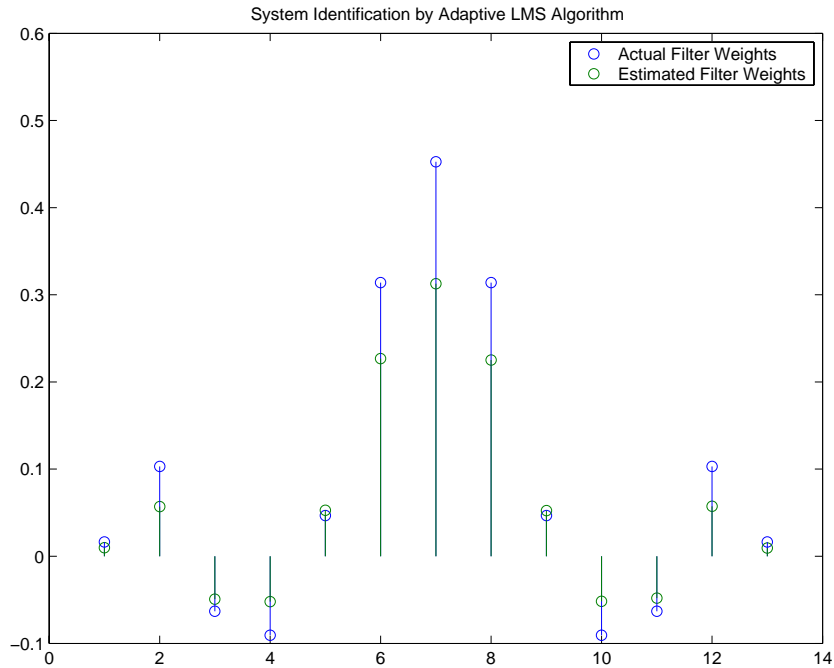
Since this may be because you did not iterate over the LMS algorithm enough times, try using 1000 samples. With 1000 samples, the stem plot, shown in the next figure, looks much better, albeit at the expense of much more computation. Clearly you should take care to select the step size with both the computation required and the fidelity of the estimated filter in mind.

System Identification by Adaptive LMS Algorithm
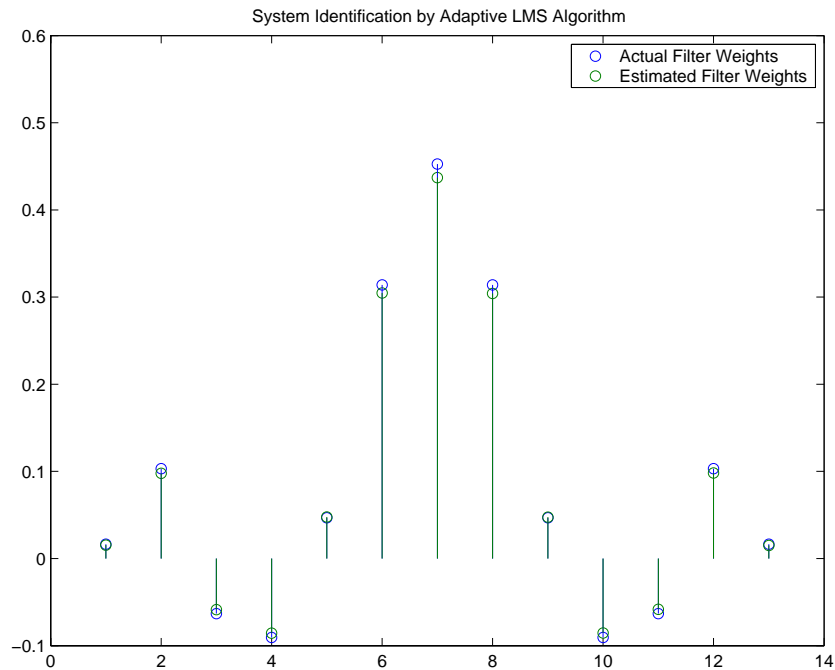
adaptfilt.nlms Example — System Identification

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.
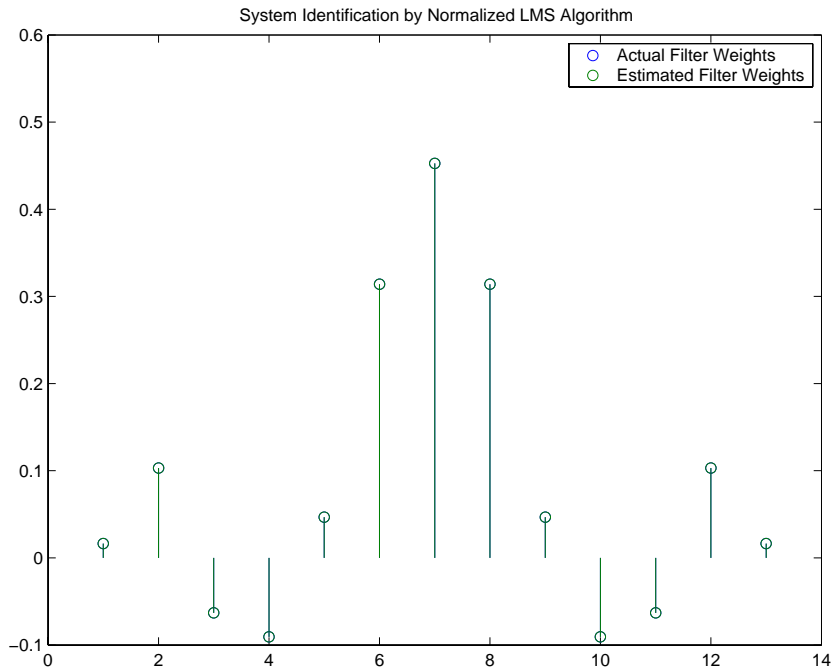
If you compare the convergence performance of the regular LMS algorithm to the normalized LMS variant, you see the normalized version adapts in far fewer iterations to a result almost as good as the nonnormalized version.

Comparing the LMS and NLMS Convergence Performance
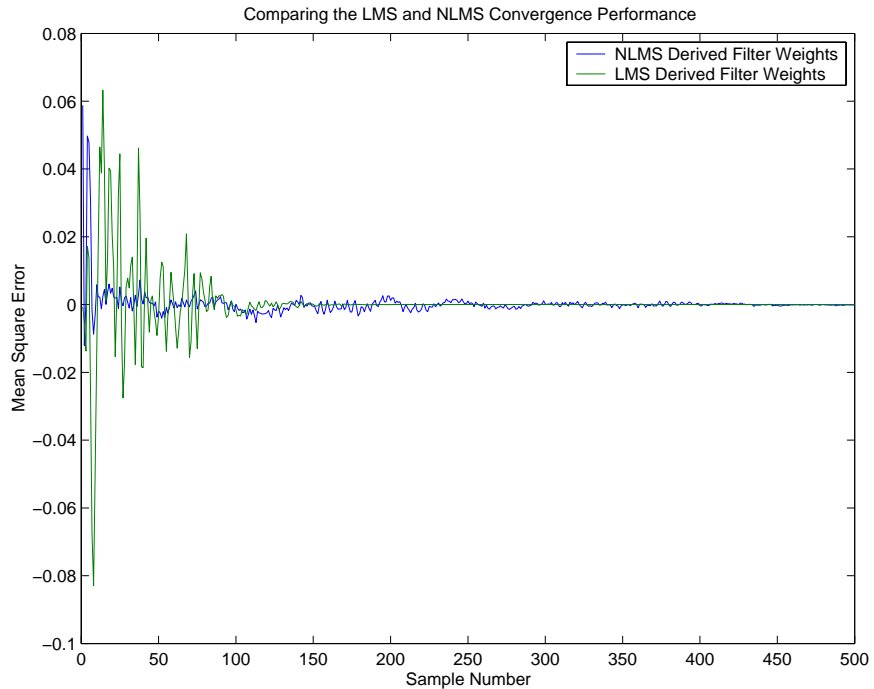
adaptfilt.sd Example — Noise Cancellation

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 μ. If the error is negative, the new coefficients are again the previous coefficients minus the error multiplied by μ — 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

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu e(k) sgn[\mathbf{x}(k)], \qquad sgn[\mathbf{x}(k)] = \begin{cases} 1, & \mathbf{x}(k) > 0 \\ 0, & \mathbf{x}(k) = 0 \\ -1, & \mathbf{x}(k) < 0 \end{cases}$$

with vector **w** containing the weights applied to the filter coefficients and vector **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. μ (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

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

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 (μ<< 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 Using an Adaptive Filter to Remove Noise from an Unknown System on page 5-9, this is *d(k)*, the desired signal. The noise cancellation process removes the noise, leaving the signal.

- Data containing random noise (*x(k)* in Using an Adaptive Filter to Remove Noise from an Unknown System on page 5-9) 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 5-17, 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.

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.
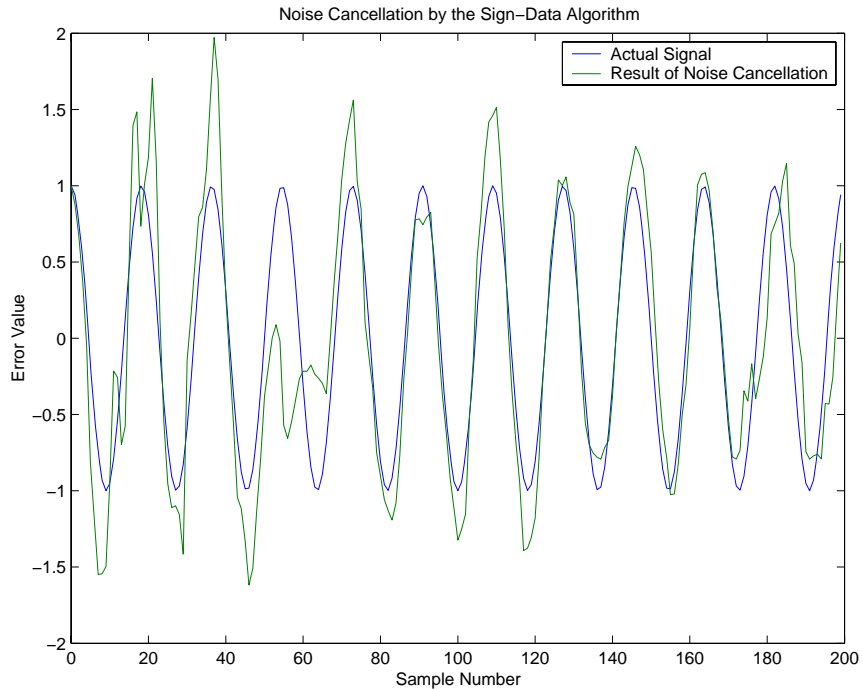
adaptfilt.se Example — Noise Cancellation

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 μ. If the error is negative, the new coefficients are again the previous coefficients minus the error multiplied by μ — 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

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu \, sgn[e(k)][\mathbf{x}(k)] \qquad sgn[e(k)] = \begin{cases} 1, & e(k) > 0 \\ 0, & e(k) = 0 \\ -1, & e(k) < 0 \end{cases}$$

with vector **w** containing the weights applied to the filter coefficients and vector **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. μ (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

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

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 ($\mu << 1$) 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 Using an Adaptive Filter to Remove Noise from an Unknown System on page 5-9, this is *d(k)*, the desired signal. The noise cancellation process removes the noise, leaving the signal.

- Data containing random noise (*x(k)* in Using an Adaptive Filter to Remove Noise from an Unknown System on page 5-9) 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 5-17, 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 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.

5-31
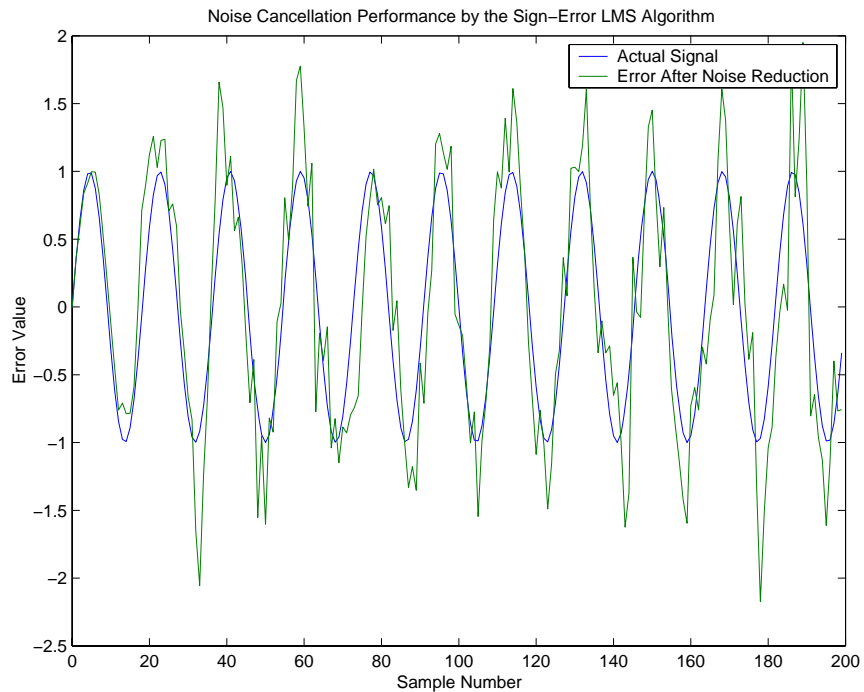
adaptfilt.ss Example — Noise Cancellation

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 5-23.

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 μ.

If the error is negative, the new coefficients are again the previous coefficients minus the error multiplied by μ — 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

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu \operatorname{sgn}[e(k)] \operatorname{sgn}[\mathbf{x}(k)], \qquad \operatorname{sgn}[z(k)] = \begin{cases} 1, & z(k) > 0 \\ 0, & z(k) = 0 \\ -1, & z(k) < 0 \end{cases}$$

where

$$z(k) = [e(k)] \operatorname{sgn}[\mathbf{x}(k)]$$

Vector **w** contains the weights applied to the filter coefficients and vector **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. μ(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 ($\mu <<$ 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 Using an Adaptive Filter to Remove Noise from an Unknown System on page 5-9, 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 Using an Adaptive Filter to Remove Noise from an Unknown System on page 5-9) 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 5-17, 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.
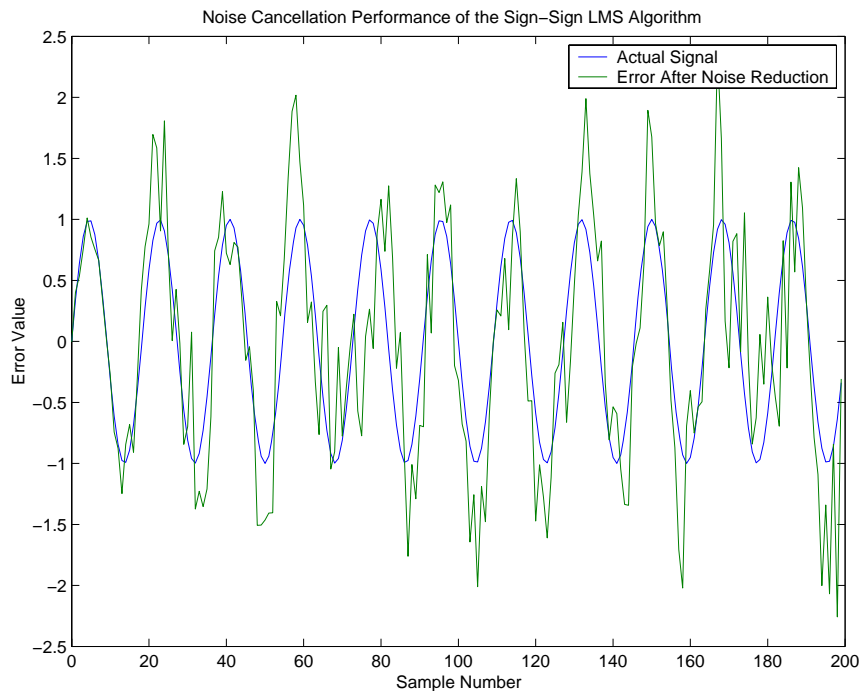
As an aside, the sign-sign LMS algorithm is part of the international CCITT standard for 32 Kb/s ADPCM telephony.

Example of Adaptive Filter That Uses RLS Algorithm

Introduction and Comparison to the LMS 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.

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, λ, 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 λ 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.
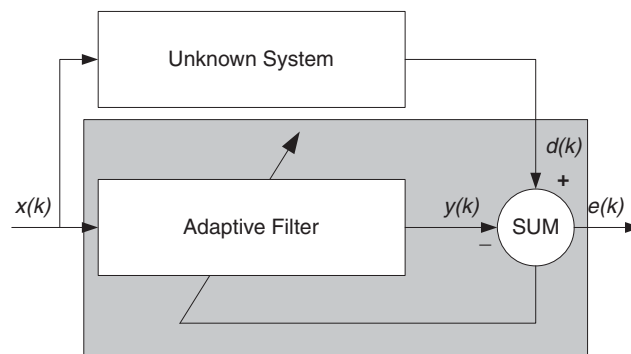
Cascading the adaptive filter with the unknown filter causes the adaptive filter to converge to a solution that is the inverse of the unknown system.

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)*H(z) = 1. For this relation to be true, G(z) must equal 1/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.

Comparing the Inverse Filter to the Unknown System
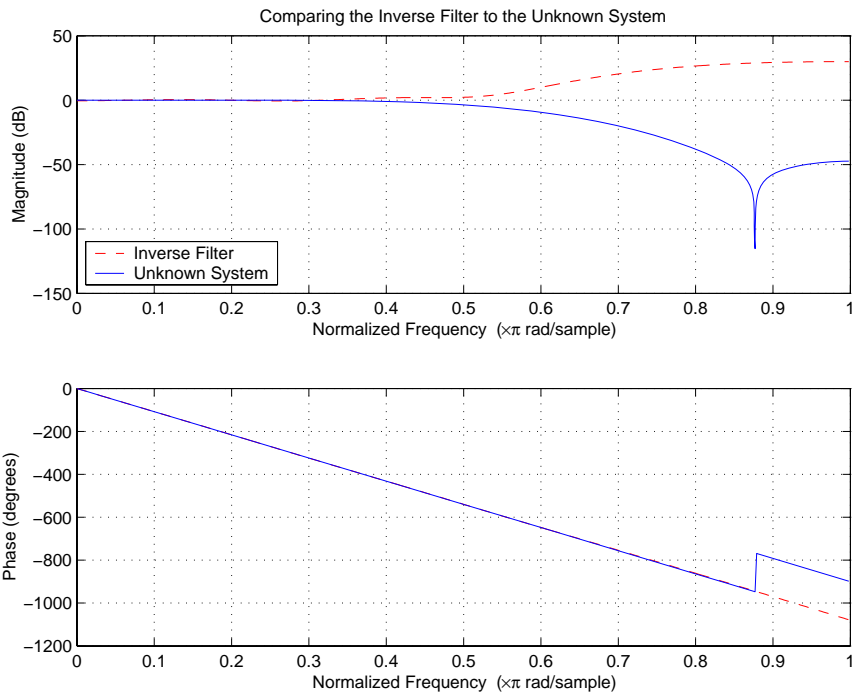
Viewed alone (refer to the following figure), the inverse system looks like a fair compensator for the unknown lowpass filter — a high pass filter with linear phase.

Inverse Filter Resulting from RLS Adaptation
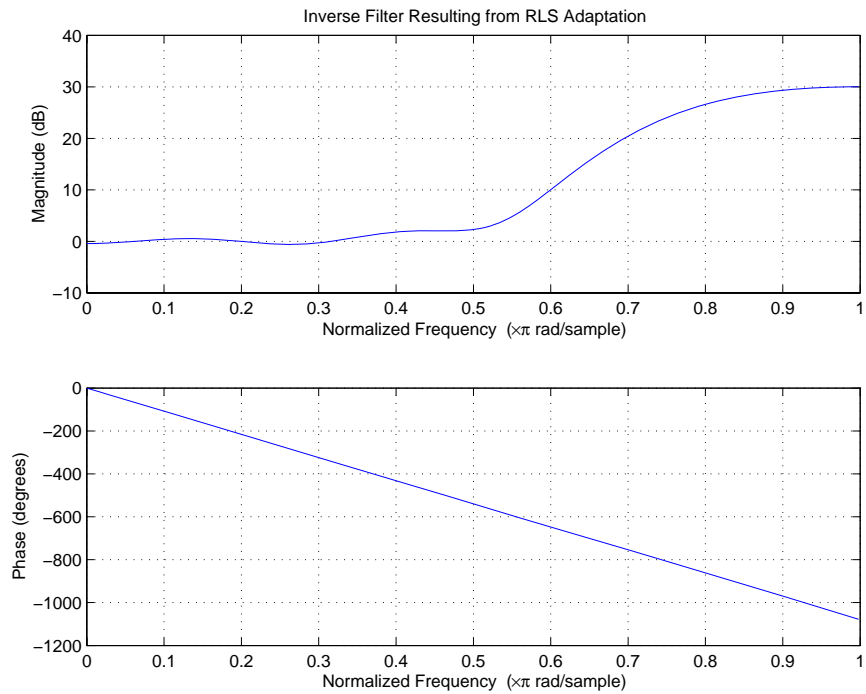
Selected Bibliography

[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

Using Integers and FIR Filters with Filter Design Toolbox

Using integers when filtering with FIR filters can make it easier to work with certain classes of hardware, such as FPGA and ASIC processors. In some cases, it is convenient to work directly with integers when designing your filter. This chapter explores a few of the issues that commonly arise when using integers with FIR filters.

Review of Fixed-Point Numbers

Terminology of Fixed-Point Numbers

Filter Design Toolbox assumes fixed-point quantities are represented in two's complement format, and are described using the WordLength and FracLength parameters. It is common to represent fractional quantities of WordLength 16 with the leftmost bit representing the sign and the remaining bits representing the fraction to the right of the binary point. Often the FracLength is thought of as the number of bits to the right of the binary point. However, there is a problem with this interpretation when the FracLength is larger than the WordLength, or when the FracLength is negative.

To work around these cases, you can use the following interpretation of a fixed-point quantity:

$$\overbrace{\boxed{b_{B-1}} \; \boxed{b_{B-2}} \; \ldots \; \boxed{b_1} \; \boxed{b_0}}^{B-\text{bits}}$$
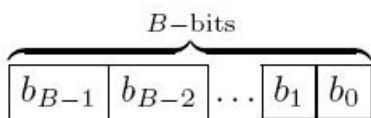
The register has a WordLength of B, or in other words it has B bits. The bits are numbered from left to right from 0 to B-1. The most significant bit (MSB) is the leftmost bit, b_{B-1}. The least significant bit is the right-most bit, b_0. You can think of the FracLength as a quantity specifying how to interpret the bits stored and resolve the value they represent. The value represented by the bits is determined by assigning a weight to each bit:

$$\underset{-2^{B-1-L}}{\boxed{b_{B-1}}} \; \underset{2^{B-2-L}}{\boxed{b_{B-2}}} \; \ldots \; \underset{2^{1-L}}{\boxed{b_1}} \; \underset{2^{-L}}{\boxed{b_0}}$$

In this figure, L is the integer FracLength. It can assume any value, depending on the quantization step size. L is necessary to interpret the value that the bits represent. This value is given by the equation

$$value = -b_{B-1}2^{B-1-L} + \sum_{k=0}^{B-2} b_k 2^{k-L}$$

.

The value 2^{-L} is the smallest possible difference between two numbers represented in this format, otherwise known as the *quantization step*. In this way, it is preferable to think of the FracLength as the negative of the exponent used to weigh the right-most, or least-significant, bit of the fixed-point number.

To reduce the number of bits used to represent a given quantity, you can discard the least-significant bits. This method minimizes the quantization error since the bits you are removing carry the least weight. For instance, the following figure illustrates reducing the number of bits from 4 to 2:

This means that the FracLength has changed from *L* to *L – 2*.

You can think of integers as being represented with a FracLength of *L = 0*, so that the quantization step becomes $2^0 = 1$.

Suppose *B = 16* and *L = 0*. Then the numbers that can be represented are the integers $\{-32768, -32767, ..., -1, 0, 1..., 32766, 32767\}$.

If you need to quantize these numbers to use only 8 bits to represent them, you will want to discard the LSBs as mentioned above, so that B=8 and L = 0–8 = –8. The increments, or quantization step then becomes $2^{-(-8)} = 2^8 = 256$. So you will still have the same range of values, but with less precision, and the numbers that can be represented become $\{-32768, -32512, ..., -256, 0, 256, ...32256, 32512\}$.

With this quantization the largest possible error becomes about 256/2 when rounding to the nearest, with a special case for 32767.

Integers and Fixed-Point Filters

This section provides an example of how you can create a filter with integer coefficients. In this example, a raised-cosine filter with floating-point coefficients is created, and the filter coefficients are then converted to integers.

Example Filter Coefficients

To illustrate the concepts of using integers with fixed-point filters, this example will use a raised-cosine filter:

```
b = firrcos(100, .25, .25, 2, 'rolloff', 'sqrt');
```

The coefficients of b are normalized so that the passband gain is equal to 1, and are all smaller than 1. In order to make them integers, they will need to be scaled. If you wanted to scale them to use 18 bits for each coefficient, the range of possible values for the coefficients becomes:

$$[-2^{-17}, 2^{17} - 1] = [-131072, 131071]$$

Because the largest coefficient of b is positive, it will need to be scaled as close as possible to 131071 (without overflowing) in order to minimize quantization error. You can determine the exponent of the scale factor by executing:

```
B = 18; % Number of bits
L = floor(log2((2^(B-1)-1)/max(b)));  % Round towards zero to avoid overflow
bsc = b*2^L;
```

Alternatively, you can use the fixed-point numbers autoscaling tool as follows:

```
bq = fi(b, true, B);  % signed = true, B = 18 bits
L = bq.FractionLength;
```

It is a coincidence that B and L are both 18 in this case, because of the value of the largest coefficient of b. If, for example, the maximum value of b were 0.124, L would be 20 while B (the number of bits) would remain 18.

Building the FIR Filter

First create the filter using the direct form, tapped delay line structure:

```
h = dfilt.dffir(bsc);
```

In order to set the required parameters, the arithmetic must be set to fixed-point:

```
h.Arithmetic = 'fixed';
h.CoeffWordLength = 18;
```

You can check that the coefficients of h are all integers:

```
all(h.Numerator == round(h.Numerator))

ans =

 1
```

Now you can examine the magnitude response of the filter using `fvtool`:

```
fvtool(h, 'Color', 'white')
```
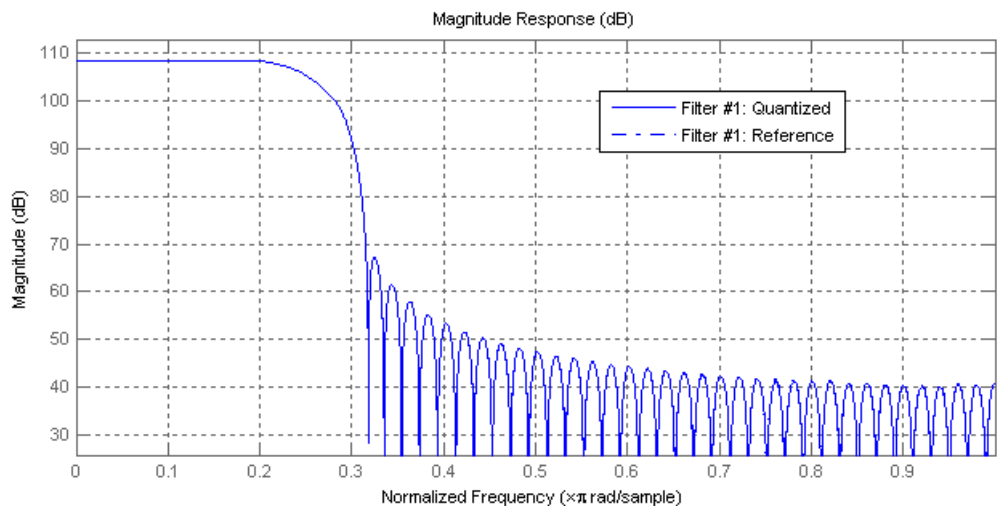

This shows a large gain of 108 dB in the passband, which is due to the large values of the coefficients— this will cause the output of the filter to be much

larger than the input. A method of addressing this will be discussed in the following sections.

Setting Filter Parameters to Work with Integers

You will need to set the input parameters of your filter to appropriate values for working with integers. For example, if the input to the filter is from a A/D converter with 12 bit resolution, you should set the input as follows:

```
h.InputWordLength = 12;
h.InputFracLength = 0;
```

The info method returns a summary of the filter settings.

```
info(h)

Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 101
Stable            : Yes
Linear Phase      : Yes (Type 1)
Arithmetic        : fixed
Numerator         : s18,0 -> [-131072 131072)
Input             : s12,0 -> [-2048 2048)
Filter Internals  : Full Precision
  Output          : s31,0 -> [-1073741824 1073741824)  (auto determined)
  Product         : s29,0 -> [-268435456 268435456)  (auto determined)
  Accumulator     : s31,0 -> [-1073741824 1073741824)  (auto determined)
  Round Mode      : No rounding
  Overflow Mode   : No overflow
```

In this case, all the fractional lengths are now set to zero, meaning that the filter h is set up to handle integers.

Creating a Test Signal for the Filter

You can generate an input signal for the filter by quantizing to 12 bits using the autoscaling feature, or you can follow the same procedure that was used

for the coefficients, discussed previously. In this example, create a signal with two sinusoids:

```
n = 0:999;
f1 = 0.1*pi;  % Normalized frequency of first sinusoid
f2 = 0.8*pi;  % Normalized frequency of second sinusoid
x = 0.9*sin(0.1*pi*n) + 0.9*sin(0.8*pi*n);
xq = fi(x, true, 12);  % signed = true, B = 12
xsc = fi(xq.int, true, 12, 0);
```

Filtering the Test Signal

To filter the input signal generated above, enter the following:

```
ysc = filter(h, xsc);
```

Here `ysc` is a full precision output, meaning that no bits have been discarded in the computation. This makes `ysc` the best possible output you can achieve given the 12–bit input and the 18–bit coefficients. This can be verified by filtering using double-precision floating-point and comparing the results of the two filtering operations:

```
hd = double(h);
xd = double(xsc);
yd = filter(hd, xd);
norm(yd-double(ysc))

ans =

      0
```

Now you can examine the output compared to the input. This example is plotting only the last few samples to minimize the effect of transients:

```
idx = 800:950;
xscext = double(xsc(idx)');
gd = grpdelay(h, [f1 f2]);
yidx = idx + gd(1);
yscext = double(ysc(yidx)');
stem(n(idx)', [xscext, yscext]);
axis([800 950 -2.5e8 2.5e8]);
```

```
legend('input', 'output');
set(gcf, 'color', 'white');
```
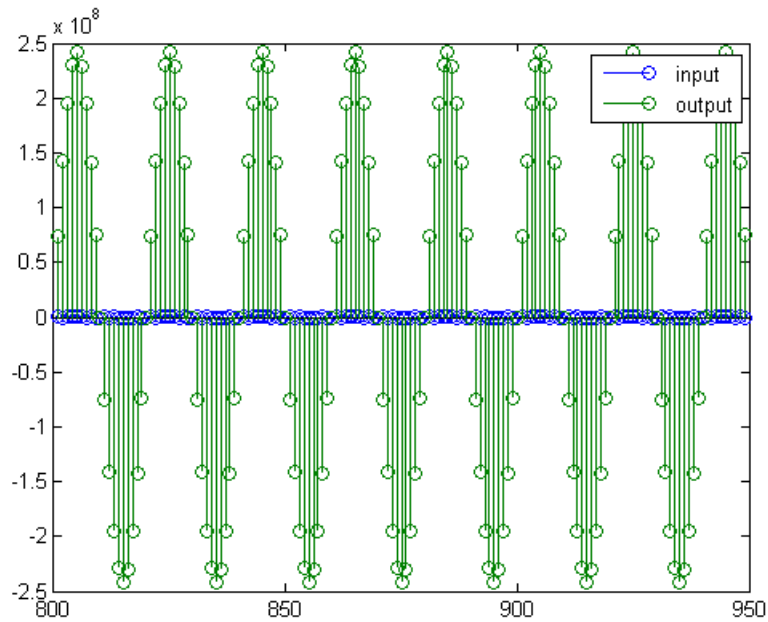

It is difficult to compare the two signals in this figure because of the large difference in scales. This is due to the large gain of the filter, so you will need to compensate for the filter gain:

```
stem(n(idx)', [2^18*xscext, yscext]);
axis([800 950 -5e8 5e8]);
legend('scaled input', 'output');
```
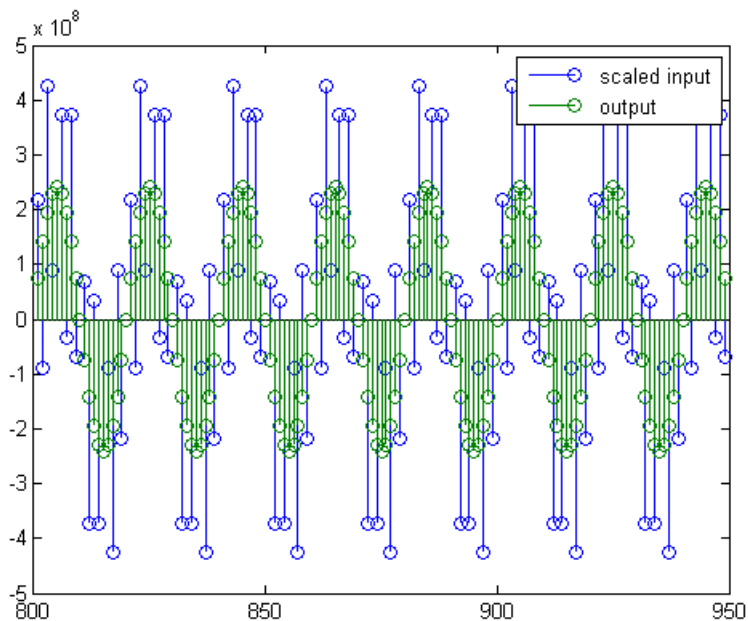
You can see how the signals compare much more easily once the scaling has been done, as seen in the above figure.

Truncating the Output WordLength

If you examine the output wordlength,

```
ysc.WordLength

ans =

    31
```

you will notice that the number of bits in the output is considerably greater than in the input. Because such growth in the number of bits representing the data may not be desirable, you may need to truncate the wordlength of the output. As discussed in "Terminology of Fixed-Point Numbers" on page 6-2the best way to do this is to discard the least significant bits, in order

to minimize error. However, if you know there are *unused* high order bits, you should discard those bits as well.

To determine if there are unused most significant bits (MSBs), you can look at where the growth in WordLength arises in the computation. In this case, the bit growth occurs to accommodate the results of adding products of the input (12 bits) and the coefficients (18 bits). Each of these products is 29 bits long (you can verify this using `info(h)`). The bit growth due to the accumulation of the product depends on the filter length and the coefficient values- however, this is a worst-case determination in the sense that no assumption on the input signal is made besides, and as a result there may be unused MSBs. You will have to be careful though, as MSBs that are deemed unused incorrectly will cause overflows.

Suppose you want to keep 16 bits for the output. In this case, there is no bit-growth due to the additions, so the output bit setting will be 16 for the wordlength and –14 for the fraction length.

Since the filtering has already been done, you can discard some bits from `ysc`:

```
yout = fi(ysc, true, 16, -14);
```

Alternatively, you can set the filter output bit lengths directly (this is useful if you plan on filtering many signals):

```
specifyall(h);
h.OutputWordLength = 16;
h.OutputFracLength = -14;
yout2 = filter(h, xsc);
```

You can verify that the results are the same either way:

```
norm(double(yout) - double(yout2))

ans =

     0
```

However, if you compare this to the full precision output, you will notice that there is rounding error due to the discarded bits:

```
norm(double(yout)-double(ysc))

ans =

    1.446323386867543e+005
```

In this case the differences are hard to spot when plotting the data, as seen below:

```
stem(n(yidx), [double(yout(yidx)'), double(ysc(yidx)')]);
axis([850 950 -2.5e8 2.5e8]);
legend('Scaled Input', 'Output');
set(gcf, 'color', 'white');
```
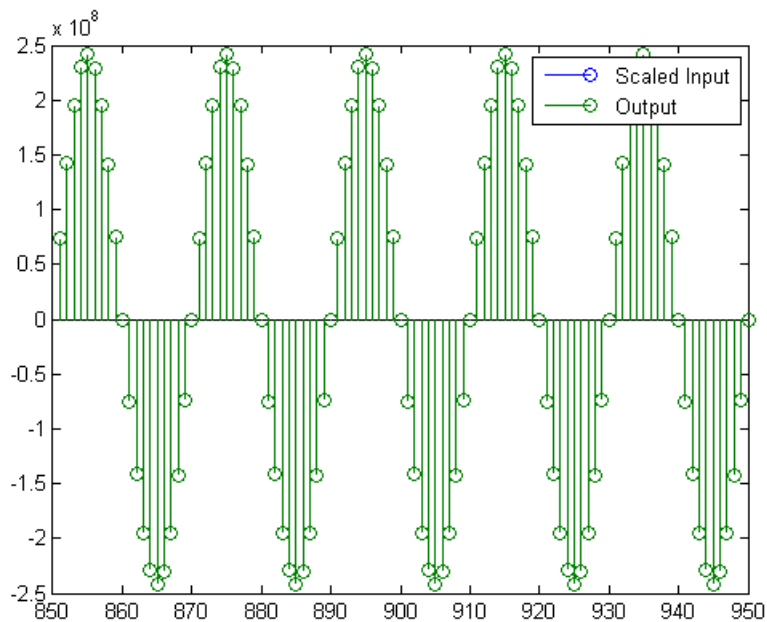

Scaling the Output

Because the filter in this example has such a large gain, the output is at a different scale than the input. This scaling is purely theoretical however,

and you can scale the data however you like. In this case, you have 16 bits for the output, but you can attach whatever scaling you choose. It would be natural to reinterpret the output to have a weight of 2^0 (or L = 0) for the LSB. This is equivalent to scaling the output signal down by a factor of 2^{-14}. However, there is no computation or rounding error involved. You can do this by executing the following:

```
yri = fi(yout.int, true, 16, 0);
stem(n(idx)', [xscext, double(yri(yidx)')]);
axis([800 950 -1.5e4 1.5e4]);
legend('input', 'rescaled output');
```
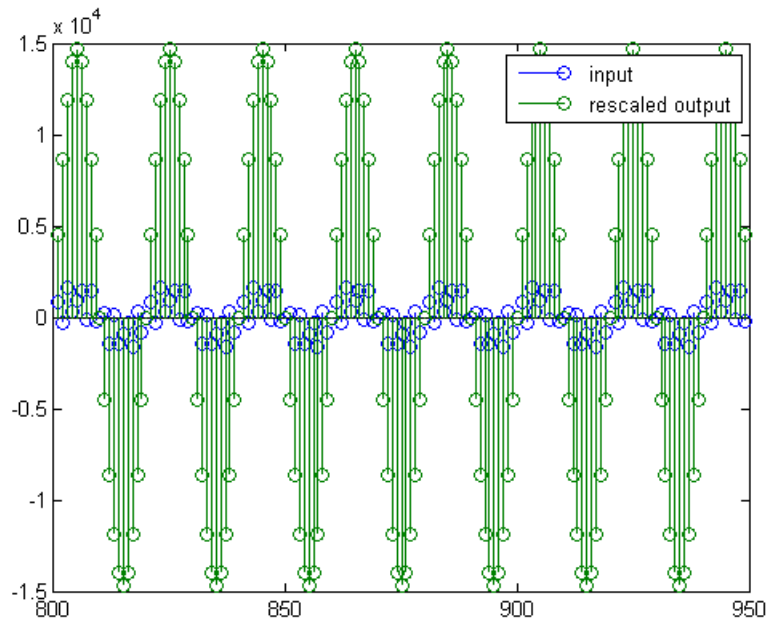

This plot shows that the output is still larger than the input. If you had done the filtering in double-precision floating-point, this would not be the case—because here more bits are being used for the output than for the input, so the MSBs are weighted differently. You can see this another way by looking at the magnitude response of the scaled filter:

```
[H,w] = freqz(h);
plot(w/pi, 20*log10(2^(-14)*abs(H)));
```
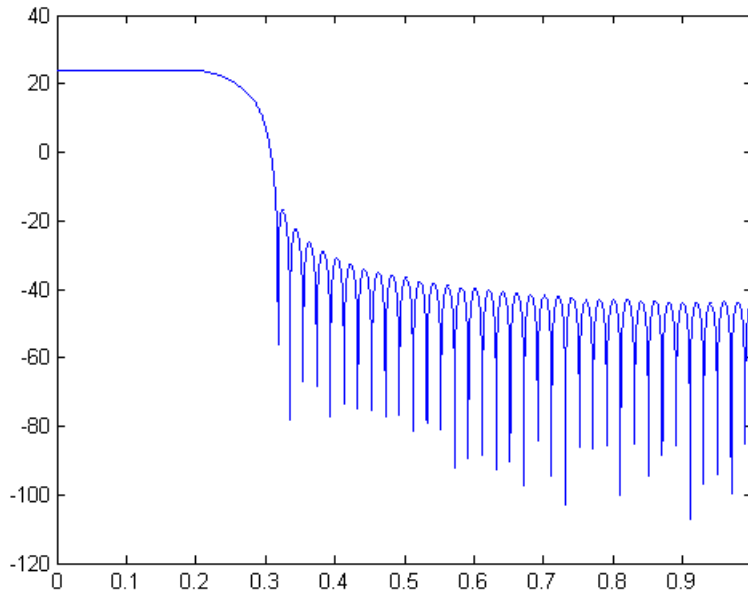

This plot shows that the passband gain is still above 0 dB.

To put the input and output on the same scale, the MSBs must be weighted equally. The input MSB has a weight of 2^{11}, whereas the scaled output MSB has a weight of $2^{(29-14)} = 2^{15}$. You need to give the output MSB a weight of 2^{11} as follows:

```
yf = fi(zeros(size(yri)), true, 16, 4);
yf.bin = yri.bin;
stem(n(idx)', [xscext, double(yf(yidx)')]);
legend('input', 'rescaled output');
```
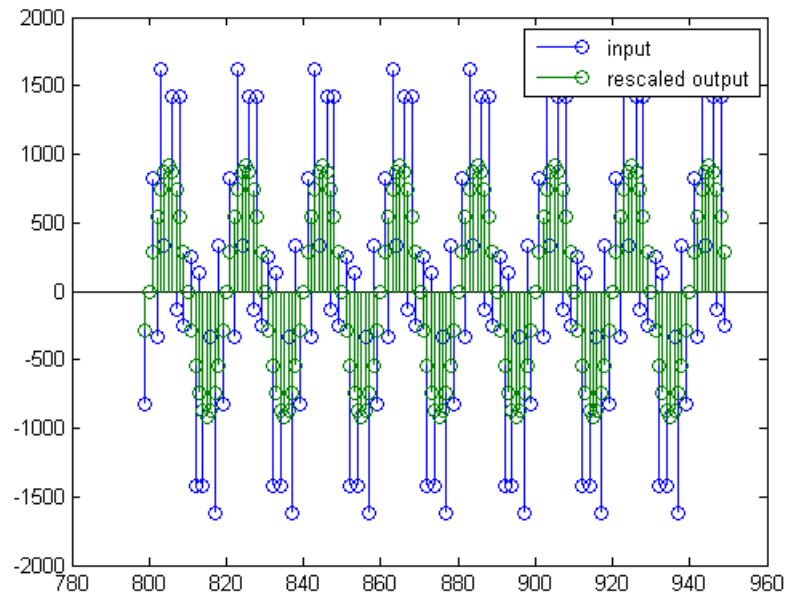
This operation is equivalent to scaling the filter gain down by $2^{\wedge}(-18)$.

```
[H,w] = freqz(h);
plot(w/pi, 20*log10(2^(-18)*abs(H)));
```
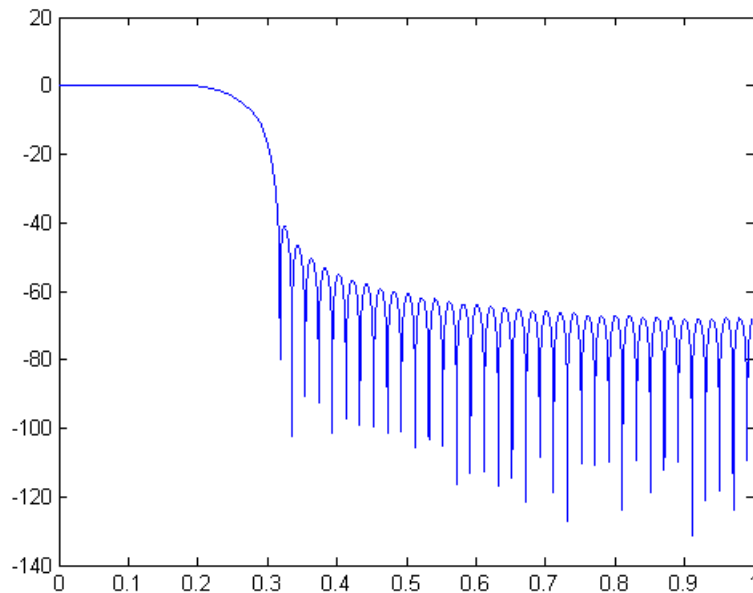
The above plot shows a 0 dB gain in the passband, as desired.

With this final version of the output, yf is no longer an integer. However this is only due to the interpretation- the integers represented by the bits in yf are identical to the ones represented by the bits in yri. You can verify this by comparing them:

```
max(abs(yf.int - yri.int))

ans =

    0
```

Using the set2int Method

Setting Filter Parameters to Work with Integers

The set2int method provides a convenient way of setting filter parameters to work with integers. The method works by scaling the coefficients to integer numbers, and setting the coefficients and input fraction length to zero. This makes it possible for you to use floating-point coefficients directly.

```
h = dfilt.dffir(b);
h.Arithmetic = 'fixed';
```

The coefficients are represented with 18 bits and the input signal is represented with 12 bits:

```
g = set2int(h, 18, 12);
g_dB = 20*log10(g)

g_dB =

    1.083707984390332e+002
```

The set2int method returns the gain of the filter by scaling the coefficients to integers, so the gain is always a power of 2. You can verify that the gain we get here is consistent with the gain of the filter previously. Now you can also check that the filter h is set up properly to work with integers:

```
info(h)
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 101
Stable            : Yes
Linear Phase      : Yes (Type 1)
```

```
Arithmetic       : fixed
Numerator        : s18,0 -> [-131072 131072)
Input            : s12,0 -> [-2048 2048)
Filter Internals : Full Precision
  Output    : s31,0 -> [-1073741824 1073741824) (auto determined)
  Product   : s29,0 -> [-268435456 268435456) (auto determined)
  Accumulator: s31,0 -> [-1073741824 1073741824) (auto determined)
  Round Mode    : No rounding
  Overflow Mode : No overflow
```

Here you can see that all fractional lengths are now set to zero, so this filter is set up properly for working with integers.

Reinterpreting the Output

You can compare the output to the double-precision floating-point reference output, and verify that the computation done by the filter h is done in full precision.

```
yint = filter(h, xsc);
norm(yd - double(yint))

ans =

      0
```

You can then truncate the output to only 16 bits:

```
yout = fi(yint, true, 16);
stem(n(yidx), [xscext, double(yout(yidx)')]);
axis([850 950 -2.5e8 2.5e8]);
legend('input', 'output');
```
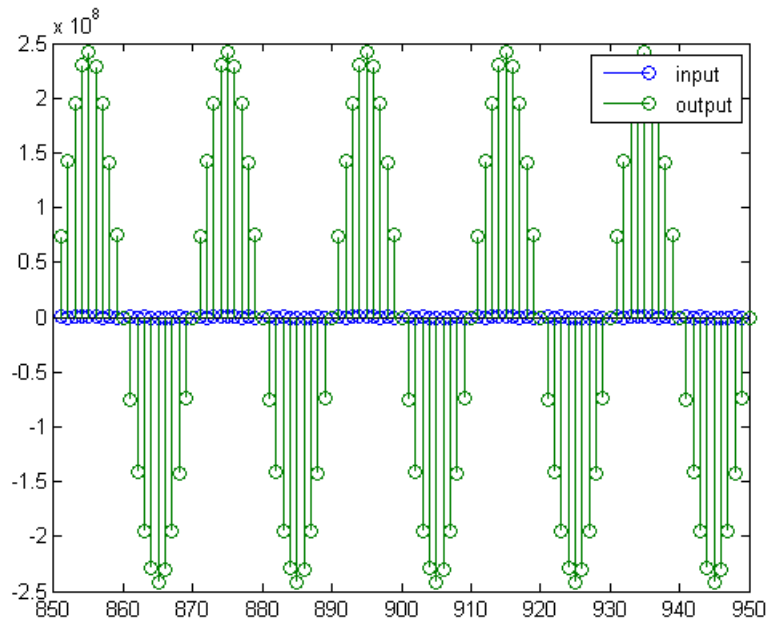
Once again, the plot shows that the input and output are at different scales. In order to scale the output so that the signals can be compared more easily in a plot, you will need to weigh the MSBs appropriately. You can compute the new fraction length using the gain of the filter when the coefficients were integer numbers:

```
WL = yout.WordLength;
FL = yout.FractionLength + log2(g);
yf2 = fi(zeros(size(yout)), true, WL, FL);
yf2.bin = yout.bin;

stem(n(idx)', [xscext, double(yf2(yidx)')]);
axis([800 950 -2e3 2e3]);
legend('input', 'rescaled output');
```
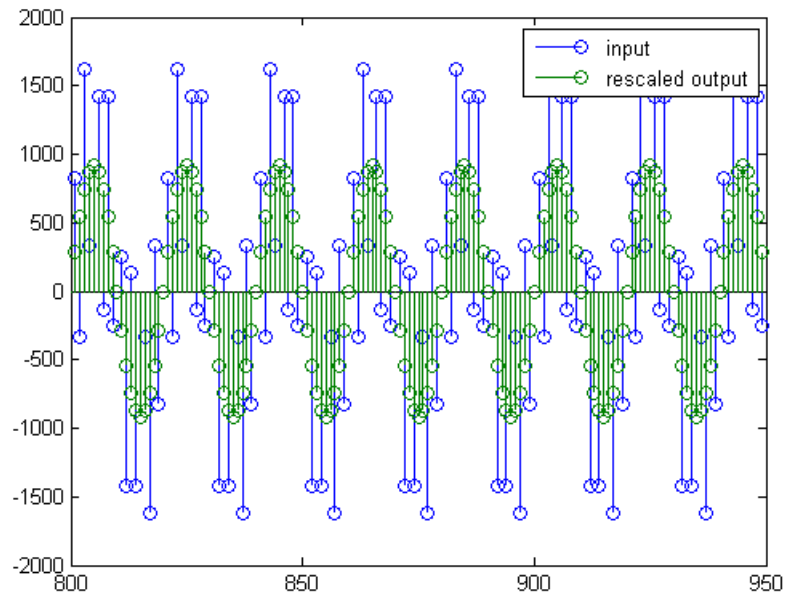
This final plot shows the filtered data re-scaled to match the input scale.

7

Reference for the Properties of Filter Objects

Fixed-Point Filter Properties

In this section...
"Overview of Fixed-Point Filters" on page 7-2
"Fixed-Point Objects and Filters" on page 7-2
"Summary — Fixed-Point Filter Properties" on page 7-5
"Property Details for Fixed-Point Filters" on page 7-19

Overview of Fixed-Point Filters

There is a distinction between fixed-point filters and quantized filters — quantized filters represent a superset that includes fixed-point filters.

When `dfilt` objects have their `Arithmetic` property set to `single` or `fixed`, they are quantized filters. However, after you set the `Arithmetic` property to `fixed`, the resulting filter is both quantized and fixed-point. Fixed-point filters perform arithmetic operations without allowing the binary point to move in response to the calculation — hence the name fixed-point. You can find out more about fixed-point arithmetic in your Fixed-Point Toolbox™ documentation or from the Help system.

With the `Arithmetic` property set to `single`, meaning the filter uses single-precision floating-point arithmetic, the filter allows the binary point to move during mathematical operations, such as sums or products. Therefore these filters cannot be considered fixed-point filters. But they are quantized filters.

The following sections present the properties for fixed-point filters, which includes all the properties for double-precision and single-precision floating-point filters as well.

Fixed-Point Objects and Filters

Fixed-point filters depend in part on fixed-point objects from Fixed-Point Toolbox software. You can see this when you display a fixed-point filter at the command prompt.

```
hd=dfilt.df2t
```

```
hd =

        FilterStructure: 'Direct-Form II Transposed'
             Arithmetic: 'double'
              Numerator: 1
            Denominator: 1
       PersistentMemory: false
                 States: [0x1 double]

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

hd =

        FilterStructure: 'Direct-Form II Transposed'
             Arithmetic: 'fixed'
              Numerator: 1
            Denominator: 1
       PersistentMemory: false
                 States: [1x1 embedded.fi]

        CoeffWordLength: 16
         CoeffAutoScale: true
                 Signed: true

        InputWordLength: 16
        InputFracLength: 15

       OutputWordLength: 16
       OutputFracLength: 15

        StateWordLength: 16
         StateAutoScale: true

            ProductMode: 'FullPrecision'

        AccumWordLength: 40
          CastBeforeSum: true
```

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

Look at the States property, shown here

```
    States: [1x1 embedded.fi]
```

The notation embedded.fi indicates that the states are being represented by fixed-point objects, usually called fi objects. If you take a closer look at the property States, you see how the properties of the fi object represent the values for the filter states.

```
    hd.states

    ans =

    []

                  DataType: Fixed
                   Scaling: BinaryPoint
                    Signed: true
                WordLength: 16
            FractionLength: 15

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

To learn more about fi objects (fixed-point objects) in general, refer to your Fixed-Point Toolbox documentation. Commands like the following can help you get the information you are looking for:

```
    docsearch(fixed-point object)
```

or

```
    docsearch(fi)
```

Either command opens the Help system and searches for information about fixed-point objects in Fixed-Point Toolbox software.

As inputs (data to be filtered), fixed-point filters accept both regular double-precision values and `fi` objects. Which you use depends on your needs. How your filter responds to the input data is determined by the settings of the filter properties, discussed in the next few sections.

Summary — Fixed-Point Filter Properties

Discrete-time filters in this toolbox use objects that perform the filtering and configuration of the filter. As objects, they include properties and methods that are often referred to as functions — not strictly the same as MATLAB functions but mostly so) to provide filtering capability. In discrete-time filters, or `dfilt` objects, many of the properties are dynamic, meaning they become available depending on the settings of other properties in the `dfilt` object or filter.

Dynamic Properties

When you use a `dfilt.`*structure* function to create a filter, MATLAB displays the filter properties in the command window in return (unless you end the command with a semicolon which suppresses the output display). Generally you see six or seven properties, ranging from the property `FilterStructure` to `PersistentMemory`. These first properties are always present in the filter. One of the most important properties is `Arithmetic`. The `Arithmetic` property controls all of the dynamic properties for a filter.

Dynamic properties become available when you change another property in the filter. For example, when you change the `Arithmetic` property value to `fixed`, the display now shows many more properties for the filter, all of them considered dynamic. Here is an example that uses a direct form II filter. First create the default filter:

```
hd=dfilt.df2

hd =

        FilterStructure: 'Direct-Form II'
             Arithmetic: 'double'
```

```
                Numerator: 1
              Denominator: 1
         PersistentMemory: false
                   States: [0x1 double]
```

With the filter `hd` in the workspace, convert the arithmetic to fixed-point. Do this by setting the property `Arithmetic` to `fixed`. Notice the display. Instead of a few properties, the filter now has many more, each one related to a particular part of the filter and its operation. Each of the now-visible properties is dynamic.

```
   hd.arithmetic='fixed'

   hd =

           FilterStructure: 'Direct-Form II'
                 Arithmetic: 'fixed'
                  Numerator: 1
                Denominator: 1
           PersistentMemory: false
                     States: [1x1 embedded.fi]

           CoeffWordLength: 16
            CoeffAutoScale: true
                    Signed: true

           InputWordLength: 16
           InputFracLength: 15

          OutputWordLength: 16
                 OutputMode: 'AvoidOverflow'

           StateWordLength: 16
           StateFracLength: 15

                ProductMode: 'FullPrecision'

          AccumWordLength: 40
              CastBeforeSum: true
```

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

Even this list of properties is not yet complete. Changing the value of other properties such as the `ProductMode` or `CoeffAutoScale` properties may reveal even more properties that control how the filter works. Remember this feature about `dfilt` objects and dynamic properties as you review the rest of this section about properties of fixed-point filters.

An important distinction is you cannot change the value of a property unless you see the property listed in the default display for the filter. Entering the filter name at the MATLAB prompt generates the default property display for the named filter. Using `get(filtername)` does not generate the default display — it lists all of the filter properties, both those that you can change and those that are not available yet.

The following table summarizes the properties, static and dynamic, of fixed-point filters and provides a brief description of each. Full descriptions of each property, in alphabetical order, follow the table.

Property Name	Valid Values [Default Value]	Brief Description
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.

Property Name	Valid Values [Default Value]	Brief Description
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.
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.

Property Name	Valid Values [Default Value]	Brief Description
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.
FracDelay	Any decimal value between 0 and 1 samples	Specifies the fractional delay provided by the filter, in decimal fractions of a sample.
FDAutoScale	[True] or false	Specifies whether the filter automatically chooses the proper scaling to represent the fractional delay value without overflowing. Turning this off by setting the value to false enables you to change the FDWordLength and FDFracLength properties to specify the data format applied.
FDFracLength	Any positive or negative integer number of bits [5]	Specifies the fraction length to represent the fractional delay.
FDProdFracLength	Any positive or negative integer number of bits [34]	Specifies the fraction length to represent the result of multiplying the coefficients with the fractional delay.
FDProdWordLength	Any positive or negative integer number of bits [39]	Specifies the word length to represent result of multiplying the coefficients with the fractional delay.
FDWordLength	Any positive or negative integer number of bits [6]	Specifies the word length to represent the fractional delay.

Property Name	Valid Values [Default Value]	Brief Description
DenStateWordLength	Any positive integer number of bits [16]	Specifies the word length used to represent the states associated with denominator coefficients in the filter.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter sets the output word and fraction lengths, and the accumulator word and fraction lengths automatically to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.
FilterStructure	Not applicable.	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret data to be processed by the filter.
InputWordLength	Any positive integer number of bits [16]	Specifies the word length applied to represent input data.
Ladder	Any ladder coefficients in double-precision data type [1]	latticearma filters include this property to store the ladder coefficients.
LadderAccumFrac Length	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.

Property Name	Valid Values [Default Value]	Brief Description
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.
LatticeAccumFrac Length	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.
MultiplicandFrac Length	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.
MultiplicandWord Length	Any positive integer number of bits [16]	Sets the word length applied to the values input to a multiply operation (the multiplicands). The filter structure df1t includes this property.
NumAccumFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients.
Numerator	Any double-precision filter coefficients [1]	Holds the numerator coefficient values for the filter.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.

Property Name	Valid Values [Default Value]	Brief Description
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	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: • AvoidOverflow — directs the filter to set the output data fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the fraction length used by the filtered data.
OutputWordLength	Any positive integer number of bits [16]	Determines the word length used for the filtered data.
OverflowMode	Saturate or [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — hey maintain full precision.

Property Name	Valid Values [Default Value]	Brief Description
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	Valid Values [Default Value]	Brief Description
RoundMode	[Convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Valid Values [Default Value]	Brief Description
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.
SectionInputAuto Scale	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to prevent overflow by data entering a section of an SOS filter. Setting this property to `false` enables you to change the `SectionInputFracLength` property to specify the precision used. Available only for SOS filters.

Property Name	Valid Values [Default Value]	Brief Description
SectionInputFrac Length	Any positive or negative integer number of bits [29]	Section values work with SOS filters. Setting this property controls how your filter interprets the section values between sections of the filter by setting the fraction length. This applies to data entering a section. Compare to SectionOutputFracLength. Available only when you disable SectionInputAutoScale by setting it to false.
SectionInputWord Length	Any positive or negative integer number of bits [29]	Sets the word length used to represent the data moving into a section of an SOS filter.
SectionOutputAuto Scale	[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.
SectionOutputFrac Length	Any positive or negative integer number of bits [29]	Section values work with SOS filters. Setting this property controls how your filter interprets the section values between sections of the filter by setting the fraction length. This applies to data leaving a section. Compare to SectionInputFracLength. Available after you disable SectionOutputAutoScale by setting it to false.
SectionOutputWord Length	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.

Property Name	Valid Values [Default Value]	Brief Description
StateFracLength	Any positive or negative integer number of bits [15]	Lets you set the fraction length applied to interpret the filter states.
States	[1x1 embedded fi]	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Notice that the states use 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	Any positive integer number of bits [16]	Sets the word length used to represent the filter states.
TapSumFracLength	Any positive or negative integer number of bits [15]	Sets the fraction length used to represent the filter tap values in addition operations. This is available after you set TapSumMode to false. Symmetric and antisymmetric FIR filters include this property.

Property Name	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 (FullPrecision) to prevent overflows, or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when outputting results from the accumulator. To let you set the precision (the fraction length) used by the output from the accumulator, set FilterInternals to SpecifyPrecision. Symmetric and antisymmetric FIR filters include this property.
TapSumWordLength	Any positive number of bits [17]	Sets the word length used to represent the filter tap weights during addition. Symmetric and antisymmetric FIR filters include this property.

Property Details for Fixed-Point Filters

When you create a fixed-point filter, you are creating a filter object (a dfilt object). In this manual, the terms filter, dfilt object, and filter object are used interchangeably. To filter data, you apply the filter object to your data set. The output of the operation is the data filtered by the filter and the filter property values.

Filter objects have properties to which you assign property values. You use these property values to assign various characteristics to the filters you create, including

- The type of arithmetic to use in filtering operations

- The structure of the filter used to implement the filter (not a property you can set or change — you select it by the dfilt.*structure* function you choose)

- The locations of quantizations and cast operations in the filter

- The data formats used in quantizing, casting, and filtering operations

Details of the properties associated with fixed-point filters are described in alphabetical order on the following pages.

AccumFracLength

Except for state-space filters, all `dfilt` objects that use fixed arithmetic have this property that defines the fraction length applied to data in the accumulator. Combined with `AccumWordLength`, `AccumFracLength` helps fully specify how the accumulator outputs data after processing addition operations. As with all fraction length properties, `AccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers.

AccumWordLength

You use `AccumWordLength` to define the data word length used in the accumulator. Set this property to a value that matches your intended hardware. For example, many digital signal processors use 40-bit accumulators, so set `AccumWordLength` to 40 in your fixed-point filter:

```
set(hq,'arithmetic','fixed');
set(hq,'AccumWordLength',40);
```

Note that `AccumWordLength` only applies to filters whose `Arithmetic` property value is `fixed`.

Arithmetic

Perhaps the most important property when you are working with `dfilt` objects, `Arithmetic` determines the type of arithmetic the filter uses, and the properties or quantizers that compose the fixed-point or quantized filter. You use strings to set the `Arithmetic` property value.

The next table shows the valid strings for the Arithmetic property. Following the table, each property string appears with more detailed information about what happens when you select the string as the value for `Arithmetic` in your `dfilt`.

Arithmetic Property String	Brief Description of Effect on the Filter
double	All filtering operations and coefficients use double-precision floating-point representations and math. When you use dfilt.*structure* to create a filter object, double is the default value for the Arithmetic property.
single	All filtering operations and coefficients use single-precision floating-point representations and math.
fixed	This string applies selected default values for the properties in the fixed-point filter object, including such properties as coefficient word lengths, fraction lengths, and various operating modes. Generally, the default values match those you use on many digital signal processors. Allows signed fixed data types only. Fixed-point arithmetic filters are available only when you install Fixed-Point Toolbox software with this toolbox.

double. When you use one of the dfilt.*structure* methods to create a filter, the Arithmetic property value is double by default. Your filter is identical to the same filter without the Arithmetic property, as you would create if you used Signal Processing Toolbox software.

Double means that the filter uses double-precision floating-point arithmetic in all operations while filtering:

• All input to the filter must be double data type. Any other data type returns an error.

• The states and output are doubles as well.

• All internal calculations are done in double math.

When you use double data type filter coefficients, the reference and quantized (fixed-point) filter coefficients are identical. The filter stores the reference coefficients as double data type.

single. When your filter should use single-precision floating-point arithmetic, set the `Arithmetic` property to `single` so all arithmetic in the filter processing gets restricted to single-precision data type.

- Input data must be single data type. Other data types return errors.
- The filter states and filter output use single data type.

When you choose `single`, you can provide the filter coefficients in either of two ways:

- Double data type coefficients. With `Arithmetic` set to `single`, the filter casts the double data type coefficients to single data type representation.
- Single data type. These remain unchanged by the filter.

Depending on whether you specified single or double data type coefficients, the reference coefficients for the filter are stored in the data type you provided. If you provide coefficients in double data type, the reference coefficients are double as well. Providing single data type coefficients generates single data type reference coefficients. Note that the arithmetic used by the reference filter is always double.

When you use `reffilter` to create a reference filter from the reference coefficients, the resulting filter uses double-precision versions of the reference filter coefficients.

To set the `Arithmetic` property value, create your filter, then use `set` to change the `Arithmetic` setting, as shown in this example using a direct form FIR filter.

```
b=fir1(7,0.45);

hd=dfilt.dffir(b)

hd =

          FilterStructure: 'Direct-Form FIR'
                Arithmetic: 'double'
                 Numerator: [1x8 double]
          PersistentMemory: false
```

```
                        States: [7x1 double]

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

hd =

        FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'single'
               Numerator: [1x8 double]
        PersistentMemory: false
                  States: [7x1 single]
```

fixed. Converting your `dfilt` object to use fixed arithmetic results in a filter structure that uses properties and property values to match how the filter would behave on digital signal processing hardware.

Note The `fixed` option for the property `Arithmetic` is available only when you install Fixed-Point Toolbox software as well as Filter Design Toolbox software.

After you set `Arithmetic` to `fixed`, you are free to change any property value from the default value to a value that more closely matches your needs. You cannot, however, mix floating-point and fixed-point arithmetic in your filter when you select `fixed` as the `Arithmetic` property value. Choosing `fixed` restricts you to using either fixed-point or floating point throughout the filter (the data type must be homogenous). Also, all data types must be signed. `fixed` does not support unsigned data types except for unsigned coefficients when you set the property `Signed` to `false`. Mixing word and fraction lengths within the fixed object is acceptable. In short, using fixed arithmetic assumes

- fixed word length.
- fixed size and dedicated accumulator and product registers.
- the ability to do either saturation or wrap arithmetic.
- that multiple rounding modes are available.

Making these assumptions simplifies your job of creating fixed-point filters by reducing repetition in the filter construction process, such as only requiring you to enter the accumulator word size once, rather than for each step that uses the accumulator.

Default property values are a starting point in tailoring your filter to common hardware, such as choosing 40-bit word length for the accumulator, or 16-bit words for data and coefficients.

In this `dfilt` object example, `get` returns the default values for `dfilt.df1t` structures.

```
[b,a]=butter(6,0.45);
hd=dfilt.df1(b,a)

hd =

          FilterStructure: 'Direct-Form I'
                Arithmetic: 'double'
                 Numerator: [1x7 double]
               Denominator: [1x7 double]
          PersistentMemory: false
                    States: Numerator:  [6x1 double]
                            Denominator:[6x1 double]


set(hd,'arithmetic','fixed')
get(hd)
          PersistentMemory: false
           FilterStructure: 'Direct-Form I'
                    States: [1x1 filtstates.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 you can use [word length fraction length]. For example, for the format [16 16], the second 16 (the fraction length) is the number of fractional bits or bits to the right of the binary point. In this example, all 16 bits are to the right of the binary point.

But it is also possible to have fixed-point formats of [16 18] or [16 -45]. In these cases the fraction length can no longer be the number of bits to the right of the binary point since the format says the word length is 16 — there cannot be 18 fraction length bits on the right. And how can there be a negative number of bits for the fraction length, such as [16 -45]?

A better way to think about fixed-point format [word length fraction length] and what it means is that the representation of a fixed-point number is a weighted sum of powers of two driven by the fraction length, or the two's complement representation of the fixed-point number.

Consider the format [B L], where the fraction length L can be positive, negative, 0, greater than B (the word length) or less than B. (B and L are always integers and B is always positive.)

Given a binary string b(1) b(2) b(3) ... 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

```
value =-b(1)*2^(B-L-1)+b(2)*2^(B-L-2)+b(3)*2^(B-L-3)+...+ 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 x 2^{16} (the rightmost term in the formula above), which is 65536. Note the precision does not depend on the word length.

Take a look at another example. When the word length set to 8 bits, the decimal value 12 is represented in binary by 00001100. That 12 is the decimal equivalent of 00001100 tells you that you are using [8 0] data format representation — the word length is 8 bits and fraction length 0 bits, and the step size or precision (the smallest difference between two adjacent values in the format [8,0], is 2^0=1.

Suppose you plan to keep only the upper 5 bits and discard the other three. The resulting precision after removing the right-most three bits comes from the weight of the lowest remaining bit, the fifth bit from the left, which is $2^3=8$, so the format would be [5,-3].

Note that in this format the step size is 8, I cannot represent numbers that are between multiples of 8.

In MATLAB, with Fixed-Point Toolbox software installed:

```
x=8;
q=quantizer([8,0]); % Word length = 8, fraction length = 0
xq=quantize(q,x);
binxq=num2bin(q,xq);
q1=quantizer([5 -3]); % Word length = 5, fraction length = -3
xq1 = quantize(q1,xq);
binxq1=num2bin(q1,xq1);
binxq

binxq =

00001000

binxq1

binxq1 =

00001
```

But notice that in [5,-3] format, 00001 is the two's complement representation for 8, not for 1; q = quantizer([8 0]) and q1 = quantizer([5 -3]) are not the same. They cover the about the same range — range(q)>range(q1) — but their quantization step is different — eps(q)= 8, and 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) = eps(q)`.

Now construct two quantizers, `q1` and `q2`. Let `q1` use the format [32,0] and let `q2` use the format [16, -16].

```
q1 = quantizer([32,0])
q2 = quantizer([16,-16])
```

Quantizers `q1` and `q2` cover the same range, but `q2` has less precision. It covers the range in steps of 2^{16}, while `q` covers the range in steps of 1.

This lost precision is due to (or can be used to model) throwing out 16 least-significant bits.

An important point to understand is that in `dfilt` objects and filtering you control which bits are carried from the sum and product operations in the filter to the filter output by setting the format for the output from the sum or product operation.

For instance, if you use [16 0] as the output format for a 32-bit result from a sum operation when the original format is [32 0], you take the lower 16 bits from the result. If you use [16 -16], you take the higher 16 bits of the original 32 bits. You could even take 16 bits somewhere in between the 32 bits by choosing something like [16 -8], but you probably do not want to do that.

Filter scaling is directly implicated in the format and precision for a filter. When you know the filter input and output formats, as well as the filter internal formats, you can scale the inputs or outputs to stay within the format ranges. For more information about scaling filters, refer to "Converting from Floating-Point to Fixed-Point".

Notice that overflows or saturation might occur at the filter input, filter output, or within the filter itself, such as during add or multiply or accumulate operations. Improper scaling at any point in the filter can result in numerical errors that dramatically change the performance of your fixed-point filter implementation.

CastBeforeSum

Setting the `CastBeforeSum` property determines how the filter handles the input values to sum operations in the filter. After you set your filter `Arithmetic` property value to `fixed`, you have the option of using `CastBeforeSum` to control the data type of some inputs (addends) to summations in your filter. To determine which addends reflect the `CastBeforeSum` property setting, refer to the reference page for the signal flow diagram for the filter structure.

`CastBeforeSum` specifies whether to cast selected addends to summations in the filter to the output format from the addition operation before performing the addition. When you specify `true` for the property value, the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

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

Notice that the output format for every sum operation reflects the value of the output property specified in the filter structure diagram. Which input property is referenced by `CastBeforeSum` depends on the structure.

Property Value	Description
`false`	Configures filter summation operations to retain the addends in the format carried from the previous operation.
`true`	Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually this generates results from the summation that more closely match those found from digital signal processors

Another point — with `CastBeforeSum` set to `false`, the filter realization process inserts an intermediate data type format to hold temporarily the full precision sum of the inputs. A separate Convert block performs the process of casting the addition result to the accumulator format. This intermediate data format occurs because the Sum block in Simulink always casts input (addends) to the output data type.

Diagrams of CastBeforeSum Settings. When `CastBeforeSum` is `false`, sum elements in filter signal flow diagrams look like this:
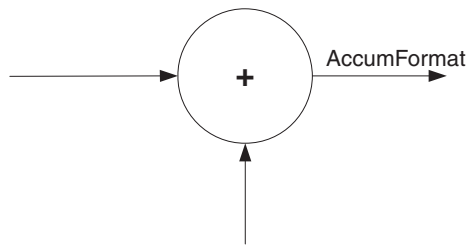
showing that the input data to the sum operations (the addends) retain their format word length and fraction length from previous operations. The addition process uses the existing input formats and then casts the output to the format defined by `AccumFormat`. Thus the output data has the word length and fraction length defined by `AccumWordLength` and `AccumFracLength`.

When `CastBeforeSum` is `true`, sum elements in filter signal flow diagrams look like this:
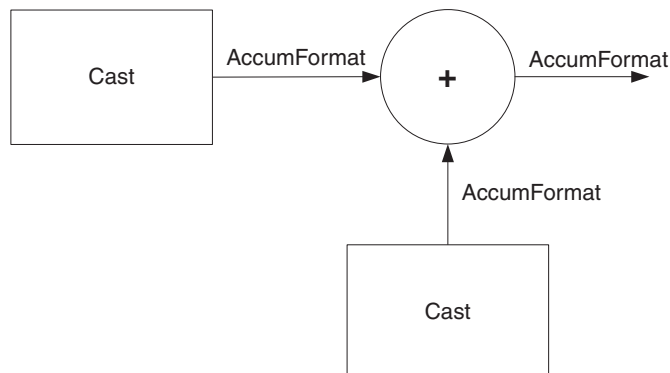
showing that the input data gets recast to the accumulator format word length and fraction length (AccumFormat) before the sum operation occurs. The data output by the addition operation has the word length and fraction length defined by `AccumWordLength` and `AccumFracLength`.

CoeffAutoScale

How the filter represents the filter coefficients depends on the property value of `CoeffAutoScale`. When you create a `dfilt` object, you use coefficients in double-precision format. Converting the `dfilt` object to fixed-point arithmetic forces the coefficients into a fixed-point representation. The representation the filter uses depends on whether the value of `CoeffAutoScale` is `true` or `false`.

- `CoeffAutoScale = true` means the filter chooses the fraction length to maintain the value of the coefficients as close to the double-precision values as possible. When you change the word length applied to the coefficients, the filter object changes the fraction length to try to accommodate the change. `true` is the default setting.

- `CoeffAutoScale = false` removes the automatic scaling of the fraction length for the coefficients and exposes the property that controls the coefficient fraction length so you can change it. For example, if the filter is a direct form FIR filter, setting `CoeffAutoScale = false` exposes the `NumFracLength` property that specifies the fraction length applied to numerator coefficients. If the filter is an IIR filter, setting `CoeffAutoScale = false` exposes both the `NumFracLength` and `DenFracLength` properties.

Here is an example of using `CoeffAutoScale` with a direct form filter.

```
hd2=dfilt.dffir([0.3 0.6 0.3])

hd2 =

        FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'double'
                Numerator: [0.3000 0.6000 0.3000]
          PersistentMemory: false
                  States: [2x1 double]


hd2.arithmetic='fixed'
```

```
hd2 =

             FilterStructure: 'Direct-Form FIR'
                  Arithmetic: 'fixed'
                   Numerator: [0.3000 0.6000 0.3000]
            PersistentMemory: false
                      States: [1x1 embedded.fi]

            CoeffWordLength: 16
             CoeffAutoScale: true
                     Signed: true

            InputWordLength: 16
            InputFracLength: 15

           OutputWordLength: 16
                 OutputMode: 'AvoidOverflow'

                  ProductMode: 'FullPrecision'

           AccumWordLength: 40
              CastBeforeSum: true

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

To this point, the filter coefficients retain the original values from when
you created the filter as shown in the Numerator property. Now change the
CoeffAutoScale property value from true to false.

```
hd2.coeffautoScale=false

hd2 =

             FilterStructure: 'Direct-Form FIR'
                  Arithmetic: 'fixed'
                   Numerator: [0.3000 0.6000 0.3000]
            PersistentMemory: false
                      States: [1x1 embedded.fi]
```

```
               CoeffWordLength: 16
               CoeffAutoScale: false
                 NumFracLength: 15
                        Signed: true

               InputWordLength: 16
               InputFracLength: 15

              OutputWordLength: 16
                    OutputMode: 'AvoidOverflow'

                   ProductMode: 'FullPrecision'

               AccumWordLength: 40
                 CastBeforeSum: true

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

With the `NumFracLength` property now available, change the word length
to 5 bits.

Notice the coefficient values. Setting `CoeffAutoScale` to `false` removes the
automatic fraction length adjustment and the filter coefficients cannot be
represented by the current format of [5 15] — a word length of 5 bits, fraction
length of 15 bits.

```
  hd2.coeffwordlength=5

  hd2 =

             FilterStructure: 'Direct-Form FIR'
                   Arithmetic: 'fixed'
                    Numerator: [4.5776e-004 4.5776e-004 4.5776e-004]
             PersistentMemory: false
                       States: [1x1 embedded.fi]

               CoeffWordLength: 5
               CoeffAutoScale: false
```

```
          NumFracLength: 15
                 Signed: true

      InputWordLength: 16
      InputFracLength: 15

     OutputWordLength: 16
           OutputMode: 'AvoidOverflow'

            ProductMode: 'FullPrecision'

      AccumWordLength: 40
        CastBeforeSum: true

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

Restoring CoeffAutoScale to true goes some way to fixing the coefficient
values. Automatically scaling the coefficient fraction length results in setting
the fraction length to 4 bits. You can check this with get(hd2) as shown
below.

```
hd2.coeffautoScale=true

hd2 =

          FilterStructure: 'Direct-Form FIR'
                Arithmetic: 'fixed'
                 Numerator: [0.3125 0.6250 0.3125]
          PersistentMemory: false
                    States: [1x1 embedded.fi]

          CoeffWordLength: 5
           CoeffAutoScale: true
                   Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
```

```
                    OutputMode: 'AvoidOverflow'

                   ProductMode: 'FullPrecision'

              AccumWordLength: 40
                CastBeforeSum: true

                    RoundMode: 'convergent'
                 OverflowMode: 'wrap'

    get(hd2)
            PersistentMemory: false
    FilterStructure: 'Direct-Form FIR'
                       States: [1x1 embedded.fi]
                    Numerator: [0.3125 0.6250 0.3125]
                   Arithmetic: 'fixed'
             CoeffWordLength: 5
              CoeffAutoScale: 1
                      Signed: 1
                   RoundMode: 'convergent'
                OverflowMode: 'wrap'
             InputWordLength: 16
             InputFracLength: 15
            OutputWordLength: 16
                  OutputMode: 'AvoidOverflow'
                 ProductMode: 'FullPrecision'
               NumFracLength: 4
             OutputFracLength: 12
           ProductWordLength: 21
           ProductFracLength: 19
             AccumWordLength: 40
             AccumFracLength: 19
                CastBeforeSum: 1
```

Clearly five bits is not enough to represent the coefficients accurately.

CoeffFracLength

Fixed-point scalar filters that you create using dfilt.scalar use this
property to define the fraction length applied to the scalar filter coefficients.

Like the coefficient-fraction-length-related properties for the FIR, lattice, and IIR filters, `CoeffFracLength` is not displayed for scalar filters until you set `CoeffAutoScale` to `false`. Once you change the automatic scaling you can set the fraction length for the coefficients to any value you require.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 14 bits, with the `CoeffWordlength` of 16 bits.

CoeffWordLength

One primary consideration in developing filters for hardware is the length of a data word. `CoeffWordLength` defines the word length for these data storage and arithmetic locations:

- Numerator and denominator filter coefficients
- Tap sum in `dfilt.dfsymfir` and `dfilt.dfasymfir` filter objects
- Section input, multiplicand, and state values in direct-form SOS filter objects such as `dfilt.df1t` and `dfilt.df2`
- Scale values in second-order filters
- Lattice and ladder coefficients in lattice filter objects, such as `dfilt.latticearma` and `dfilt.latticemamax`
- Gain in `dfilt.scalar`

Setting this property value controls the word length for the data listed. In most cases, the data words in this list have separate fraction length properties to define the associated fraction lengths.

Any positive, integer word length works here, limited by the machine you use to develop your filter and the hardware you use to deploy your filter.

DenAccumFracLength

Filter structures `df1`, `df1t`, `df2`, and `df2t` that use `fixed` arithmetic have this property that defines the fraction length applied to denominator coefficients in the accumulator. In combination with `AccumWordLength`, the properties fully specify how the accumulator outputs data stored there.

As with all fraction length properties, `DenAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. To be able to change the property value for this property, you set `FilterInternals` to `SpecifyPrecision`.

DenFracLength

Property `DenFracLength` contains the value that specifies the fraction length for the denominator coefficients for your filter. `DenFracLength` specifies the fraction length used to interpret the data stored in `C`. Used in combination with `CoeffWordLength`, these two properties define the interpretation of the coefficients stored in the vector that contains the denominator coefficients.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `CoeffWordLength` of 16 bits.

Denominator

The denominator coefficients for your IIR filter, taken from the prototype you start with, are stored in this property. Generally this is a 1-by-N array of data in double format, where N is the length of the filter.

All IIR filter objects include `Denominator`, except the lattice-based filters which store their coefficients in the `Lattice` property, and second-order section filters, such as `dfilt.df1tsos`, which use the `SosMatrix` property to hold the coefficients for the sections.

DenProdFracLength

A property of all of the direct form IIR `dfilt` objects, except the ones that implement second-order sections, `DenProdFracLength` specifies the fraction length applied to data output from product operations that the filter performs on denominator coefficients.

Looking at the signal flow diagram for the `dfilt.df1t` filter, for example, you see that denominators and numerators are handled separately. When you set `ProductMode` to `SpecifyPrecision`, you can change the `DenProdFracLength` setting manually. Otherwise, for multiplication operations that use the

denominator coefficients, the filter sets the fraction length as defined by the `ProductMode` setting.

DenStateFracLength

When you look at the flow diagram for the `dfilt.df1sos` filter object, the states associated with denominator coefficient operations take the fraction length from this property. In combination with the `DenStateWordLength` property, these properties fully specify how the filter interprets the states.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `DenStateWordLength` of 16 bits.

DenStateWordLength

When you look at the flow diagram for the `dfilt.df1sos` filter object, the states associated with the denominator coefficient operations take the data format from this property and the `DenStateFracLength` property. In combination, these properties fully specify how the filter interprets the state it uses.

By default, the value is 16 bits, with the `DenStateFracLength` of 15 bits.

FilterInternals

Similar to the FilterInternals pane in FDATool, this property controls whether the filter sets the output word and fraction lengths automatically, and the accumulator word and fraction lengths automatically as well, to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. Setting `FilterInternals` to `SpecifyPrecision` exposes the output and accumulator related properties so you can set your own word and fraction lengths for them. Note that

FilterStructure

Every `dfilt` object has a `FilterStructure` property. This is a read-only property containing a string that declares the structure of the filter object you created.

When you construct filter objects, the `FilterStructure` property value is returned containing one of the strings shown in the following table. Property `FilterStructure` indicates the filter architecture and comes from the constructor you use to create the filter.

After you create a filter object, you cannot change the `FilterStructure` property value. To make filters that use different structures, you construct new filters using the appropriate methods, or use `convert` to switch to a new structure.

Default value. Since this depends on the constructor you use and the constructor includes the filter structure definition, there is no default value. When you try to create a filter without specifying a structure, MATLAB returns an error.

Filter Constructor Name	FilterStructure Property String and Filter Type
`'dfilt.df1'`	Direct form I
`'dfilt.df1sos'`	Direct form I filter implemented using second-order sections
`'dfilt.df1t'`	Direct form I transposed
`'dfilt.df2'`	Direct form II
`'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

Filter Constructor Name	FilterStructure Property String and Filter Type
`'dfilt.latticearma'`	Lattice ARMA
`'dfilt.dfsymfir'`	Symmetric FIR. Even and odd forms
`'dfilt.scalar'`	Scalar

Filter Structures with Quantizations Shown in Place. To help you understand how and where the quantizations occur in filter structures in this toolbox, the figure below shows the structure for a Direct Form II filter, including the quantizations (fixed-point formats) that compose part of the fixed-point filter. You see that one or more quantization processes, specified by the *format label, accompany each filter element, such as a delay, product, or summation element. The input to or output from each element reflects the result of applying the associated quantization as defined by the word length and fraction length format. Wherever a particular filter element appears in a filter structure, recall the quantization process that accompanies the element as it appears in this figure. Each filter reference page, such as the `dfilt.df2` reference page, includes the signal flow diagram showing the formatting elements that define the quantizations that occur throughout the filter flow.

For example, a product quantization, either numerator or denominator, follows every product (gain) element and a sum quantization, also either numerator or denominator, follows each sum element. The figure shows the `Arithmetic` property value set to `fixed`.

df2 IIR Filter Structure Including the Formatting Objects, with Arithmetic Property Value fixed
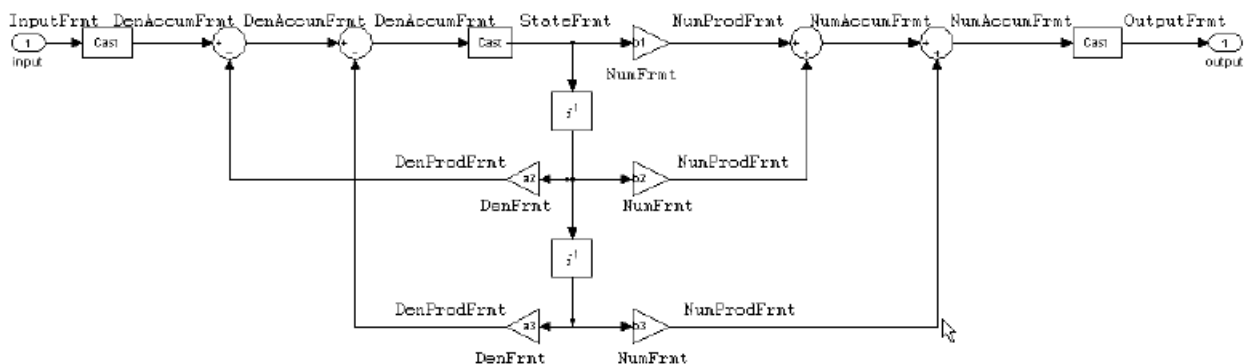
When your `df2` filter uses the `Arithmetic` property set to `fixed`, the filter structure contains the formatting features shown in the diagram. The formats included in the structure are fixed-point objects that include properties to set various word and fraction length formats. For example, the `NumFormat` or `DenFormat` in the fixed-point arithmetic filter set the properties for quantizing numerator or denominator coefficients according to word and fraction length settings.

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 above, the entries `InputFormat` and `OutputFormat` are the formats applied, that is the word length and fraction length, to the filter input and output data. For example, filter properties like `OutputWordLength` and `InputWordLength` specify values that control filter operations at the input and output points in the structure and are represented by the formatting objects `InputFormat` and `OutputFormat` shown in the filter structure diagrams.

Direct Form I Filter Structure. The following figure depicts the *direct form I* filter structure that directly realizes a transfer function with a second-order numerator and denominator. The numerator coefficients are numbered $b(i)$, i=1, 2, 3; the denominator coefficients are numbered $a(i)$, $i = 1, 2, 3$; and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, the `Arithmetic` property is set to `fixed`.

Example — Specifying a Direct Form I Filter. You can specify a second-order direct form I structure for a quantized filter hq with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1(b,a);
```

To create the fixed-point filter, set the Arithmetic property to fixed as shown here.

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

Direct Form I Filter Structure With Second-Order Sections. The following figure depicts a *direct form I* filter structure that directly realizes a transfer function with a second-order numerator and denominator and second-order sections. The numerator coefficients are numbered $b(i)$, i=1, 2, 3; the denominator coefficients are numbered $a(i)$, $i = 1, 2, 3$; and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, the Arithmetic property is set to fixed to place the filter in fixed-point mode.

Example — Specifying a Direct Form I Filter with Second-Order Sections. You can specify an eighth-order direct form I structure for a quantized filter hq with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1sos(b,a);
```

To create the fixed-point filter, set the Arithmetic property to fixed, as shown here.

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

Direct Form I Transposed Filter Structure. The next signal flow diagram depicts a *direct form I transposed* filter structure that directly realizes a transfer function with a second-order numerator and denominator. The numerator coefficients are $b(i)$, i = 1, 2, 3; the denominator coefficients are $a(i)$, i = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$. With the Arithmetic property value set to fixed, the figure shows the filter with the properties indicated.

Example — Specifying a Direct Form I Transposed Filter. You can specify a second-order direct form I transposed filter structure for a quantized filter hq with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1t(b,a);
set(hq,'arithmetic','fixed');
```

Direct Form II Filter Structure. The following graphic depicts a *direct form II* filter structure that directly realizes a transfer function with a second-order numerator and denominator. In the figure, the Arithmetic property value is fixed. Numerator coefficients are named $b(i)$; denominator coefficients are named $a(i)$, $i = 1, 2, 3$; and the states (used for initial and final state values in filtering) are named $z(i)$.

Use the method `dfilt.df2` to construct a quantized filter whose `FilterStructure` property is `Direct-Form II`.

Example — Specifying a Direct Form II Filter. You can specify a second-order direct form II filter structure for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df2(b,a);
hq.arithmetic = 'fixed'
```

To convert your initial double-precision filter `hq` to a quantized or fixed-point filter, set the `Arithmetic` property to `fixed`, as shown.

Direct Form II Filter Structure With Second-Order Sections

The following figure depicts *direct form II* filter structure using second-order sections that directly realizes a transfer function with a second-order numerator and denominator sections. In the figure, the `Arithmetic` property value is `fixed`. Numerator coefficients are labeled $b(i)$; denominator coefficients are labeled $a(i)$, i = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$.

Use the method `dfilt.df2sos` to construct a quantized filter whose
`FilterStructure` property is `Direct-Form II`.

**Example — Specifying a Direct Form II Filter with Second-Order
Sections.** You can specify a tenth-order direct form II filter structure that
uses second-order sections for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df2sos(b,a);
hq.arithmetic = 'fixed'
```

To convert your prototype double-precision filter `hq` to a 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.
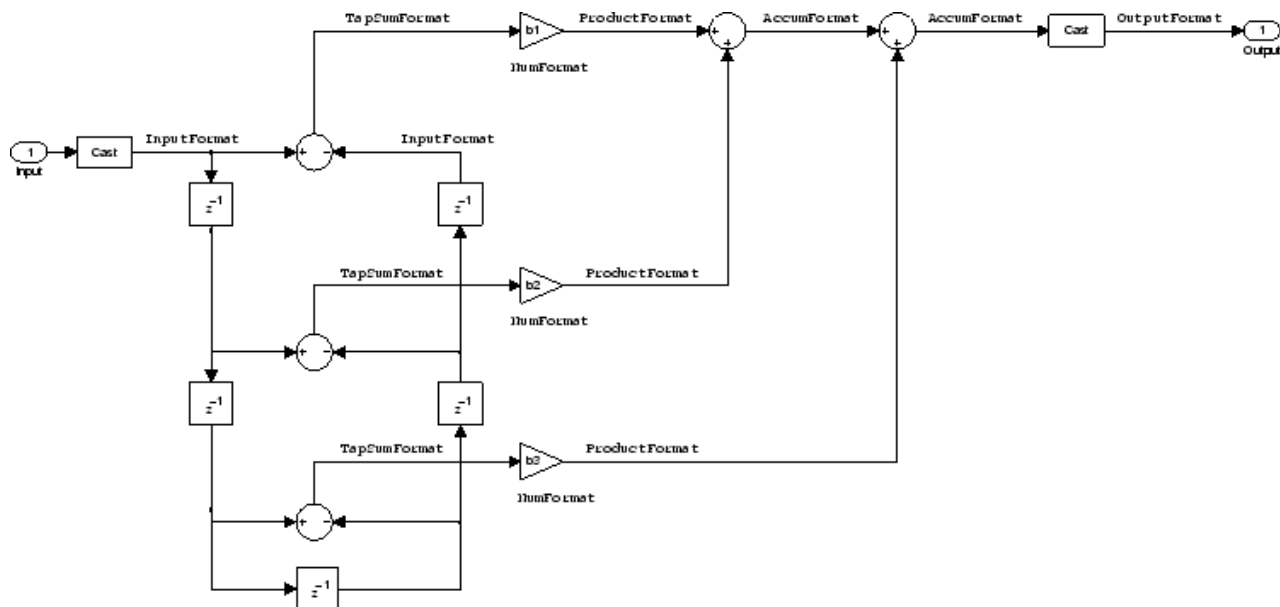
Use the method dfilt.dfasymfir to construct the filter, and then set the
Arithmetic property to fixed to convert to a fixed-point filter with this
structure.

**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.
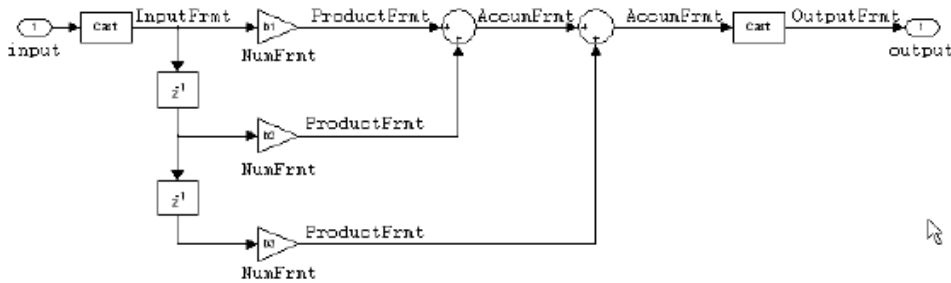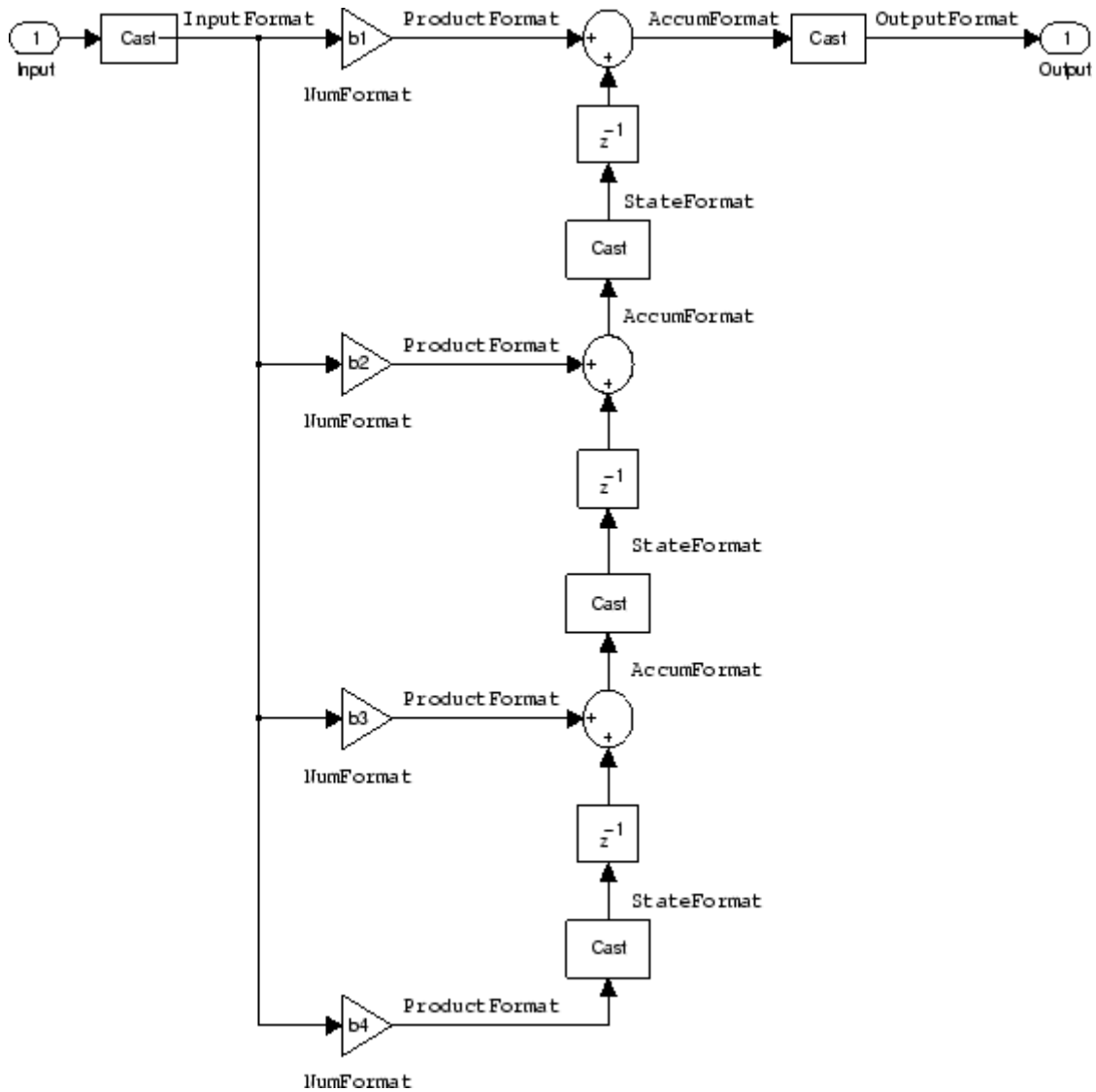
Use the `dfilt.dffir` method to generate a filter that uses this structure.

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.
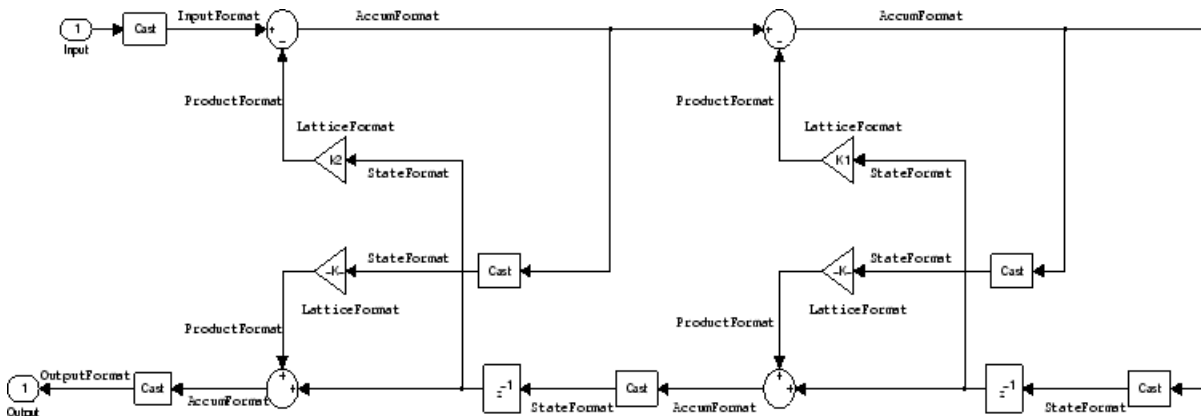
With the `Arithmetic` property set to `fixed`, your filter matches the figure. Using the method `dfilt.dffirt` returns a double-precision filter that you convert to a fixed-point filter.

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 Allpass Filter Structure. The following figure depicts the *lattice allpass* filter structure. The pictured structure directly realizes third-order lattice allpass filters using fixed-point arithmetic. The filter reflection coefficients are labeled $k1(i)$, $i = 1, 2, 3$. The states (used for initial and final state values in filtering) are labeled $z(i)$.

To create a quantized filter that uses the lattice allpass structure shown in the figure, use the `dfilt.latticeallpass` method and set the `Arithmetic` property to `fixed`.

Example — Specifying a Lattice Allpass Filter. You can create a third-order lattice allpass filter structure for a quantized filter `hq` with the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticeallpass(k);
set(hq,'arithmetic','fixed');
```

Lattice Moving Average Maximum Phase Filter Structure. In the next figure you see a *lattice moving average maximum phase* filter structure. This signal flow diagram directly realizes a third-order lattice moving average (MA) filter with the following phase form depending on the initial transfer function:

- When you start with a minimum phase transfer function, the upper branch of the resulting lattice structure returns a minimum phase filter. The lower branch returns a maximum phase filter.

- When your transfer function is neither minimum phase nor maximum phase, the lattice moving average maximum phase structure will not be maximum phase.

- When you start with a maximum phase filter, the resulting lattice filter is maximum phase also.
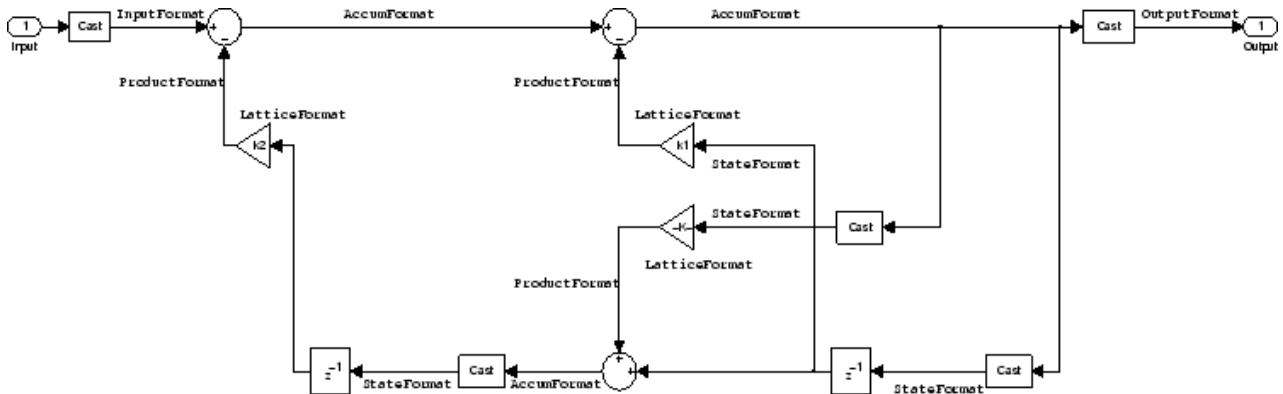
The filter reflection coefficients are labeled $k(i)$, $i = 1, 2, 3$. The states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, we set the `Arithmetic` property to `fixed` to reveal the fixed-point arithmetic format features that control such options as word length and fraction length.
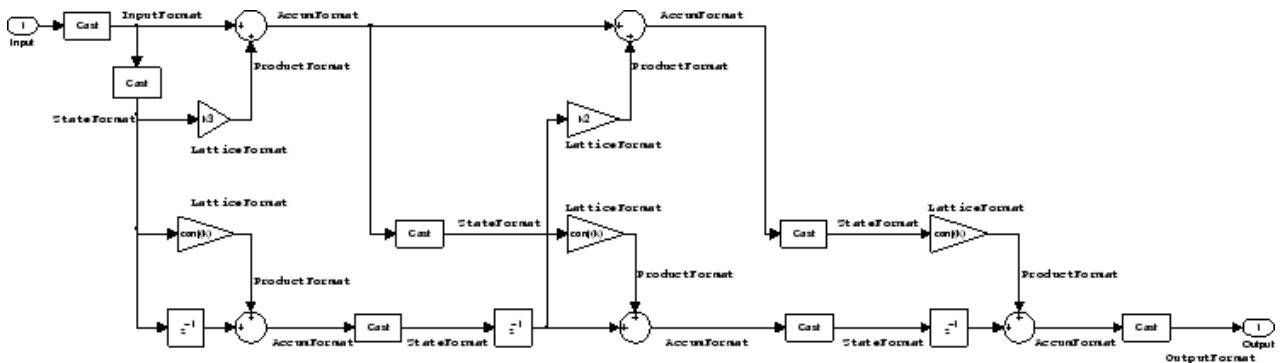
Example — Constructing a Lattice Moving Average Maximum Phase Filter. Constructing a fourth-order lattice MA maximum phase filter structure for a quantized filter hq begins with the following code.

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

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.arithmetic = 'custom';
```
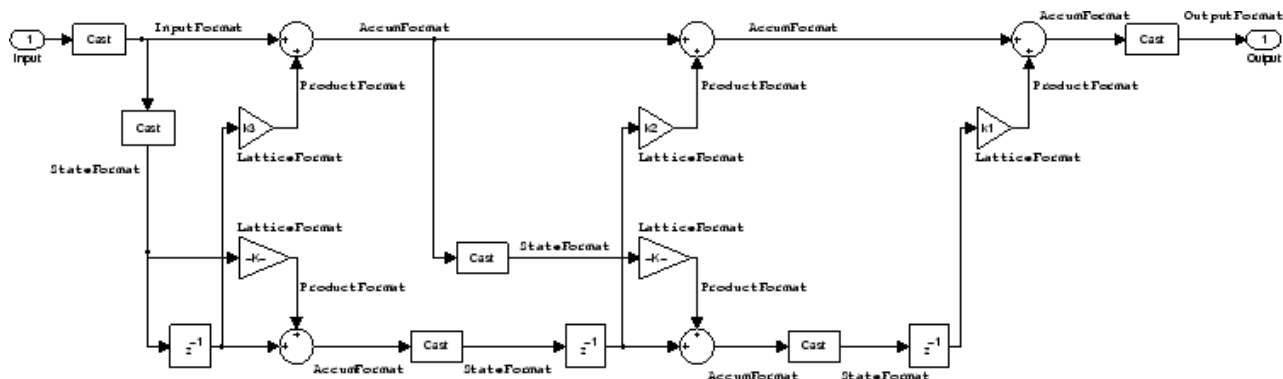
Lattice Moving Average (MA) Filter Structure for Minimum Phase.
The following figures depict *lattice moving average (MA)* filter structures
that directly realize third-order lattice MA filters for minimum phase. The
filter reflection coefficients are labeled $k(i)$, $i = 1, 2, 3$, and the states (used
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.
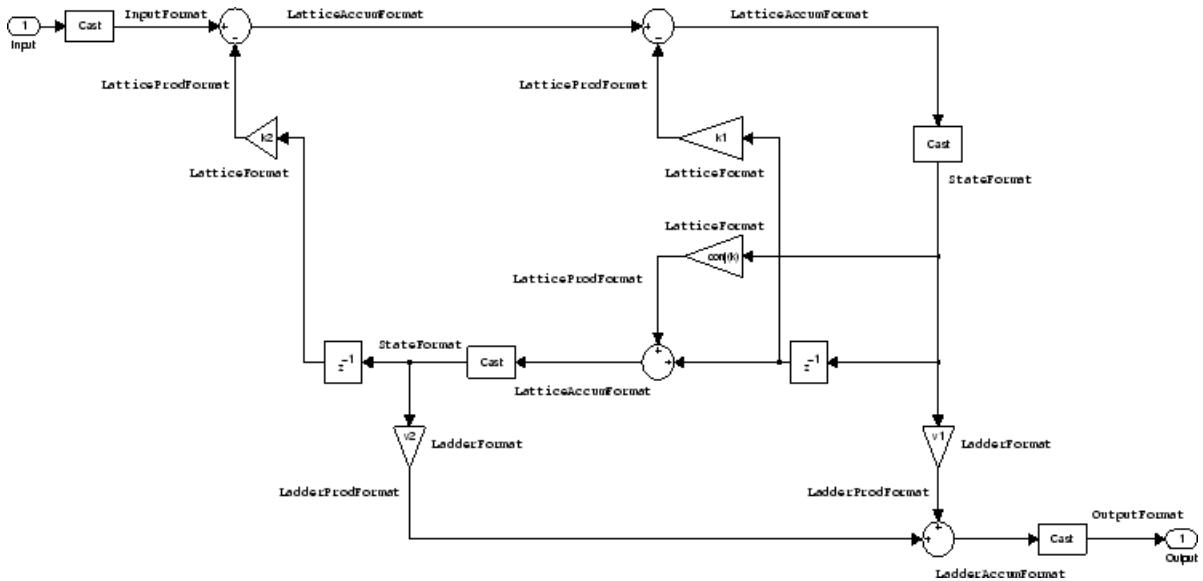
The filter reflection coefficients are labeled $k(i)$, i = 1, 2, 3. The states (used for initial and final state values in filtering) are labeled $z(i)$. This figure shows the filter structure when the Arithmetic property is set to fixed to reveal the fixed-point arithmetic format features that control such options as word length and fraction length.

Example — Specifying a Minimum Phase Lattice MA Filter. You can specify a third-order lattice MA filter structure for minimum phase applications using variations of the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticemamin(k);
set(hq,'arithmetic','fixed');
```

Lattice Autoregressive Moving Average (ARMA) Filter Structure. The figure below depicts a *lattice autoregressive moving average (ARMA)* filter structure that directly realizes a fourth-order lattice ARMA filter. The filter reflection coefficients are labeled $k(i)$, i = 1, ..., 4; the ladder coefficients are labeled $v(i)$, i = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$.

Example — Specifying an Lattice ARMA Filter. The following code specifies a fourth-order lattice ARMA filter structure for a quantized filter hq, starting from hd, a floating-point version of the filter.

```
k = [.66 .7 .44 .66];
v = [1 0 0];
hd=dfilt.latticearma(k,v);
hq.arithmetic = 'fixed';
```

Direct Form Symmetric FIR Filter Structure (Any Order). Shown in the next figure, you see signal flow that depicts a *direct form symmetric FIR* filter structure that directly realizes a fifth-order direct form symmetric FIR filter. Filter coefficients are labeled $b(i)$, $i = 1, ..., n$, and states (used for initial and final state values in filtering) are labeled $z(i)$. Showing the filter structure used when you select fixed for the Arithmetic property value, the first figure details the properties in the filter object.
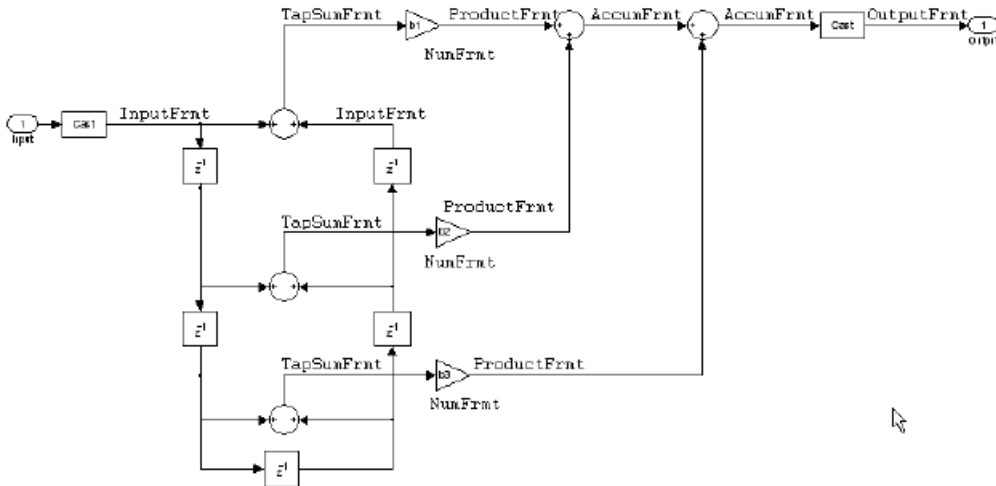
Example — Specifying an Odd-Order Direct Form Symmetric FIR Filter. By using the following code in MATLAB, you can specify a fifth-order direct form symmetric FIR filter for a fixed-point filter hq:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
hd=dfilt.dfsymfir(b);
set(hq,'arithmetic','fixed');
```

Assigning Filter Coefficients. The syntax you use to assign filter coefficients for your floating-point or fixed-point filter depends on the structure you select for your filter.

Converting Filters Between Representations. Filter conversion functions in this toolbox and in Signal Processing Toolbox software let you convert filter transfer functions to other filter forms, and from other filter forms to transfer function form. Relevant conversion functions include the following functions.

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.
tf2ss	Converts from a transfer function to state-space form.
tf2zp	Converts from a rational transfer function to its factored (single section) form (zero-pole-gain form).
zp2sos	Converts a zero-pole-gain form to a second-order section form.
zp2ss	Conversion of zero-pole-gain form to a state-space form.
zp2tf	Conversion of zero-pole-gain form to transfer functions of multiple order sections.

Note that these conversion routines do not apply to dfilt objects.

The function convert is a special case — when you use convert to change the filter structure of a fixed-point filter, you lose all of the filter states and settings. Your new filter has default values for all properties, and it in 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 (lattticearma) 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`, `latticemamax`, and `latticemamin`) use lattice coefficients to define the filter. In combination with `LatticeFracLength` and `CoeffWordLength`, these three properties specify how the accumulator outputs lattice coefficient-related data stored there. As with all fraction length properties, `LatticeAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. By default, the property is set to 31 bits.

LatticeFracLength

To let you control the way your filter interprets the denominator coefficients, `LatticeFracLength` sets the fraction length applied to the lattice coefficients for your lattice filter. When you create the default lattice filter, `LatticeFracLength` is 16 bits.

As with all fraction length properties, `LatticeFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers.

MultiplicandFracLength

Each input data element for a multiply operation has both word length and fraction length to define its representation. `MultiplicandFracLength` sets the fraction length to use when the filter object performs any multiply operation during filtering. For default filters, this is set to 15 bits.

As with all word and fraction length properties, `MultiplicandFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers.

MultiplicandWordLength

Each input data element for a multiply operation has both word length and fraction length to define its representation. `MultiplicandWordLength` sets the word length to use when the filter performs any multiply operation during filtering. For default filters, this is set to 16 bits. Only the `df1t` and `df1tsos` filter objects include the `MultiplicandFracLength` property.

Only the `df1t` and `df1tsos` filter objects include the `MultiplicandWordLength` property.

NumAccumFracLength

Filter structures `df1`, `df1t`, `df2`, and `df2t` that use `fixed` arithmetic have this property that defines the fraction length applied to numerator coefficients in output from the accumulator. In combination with `AccumWordLength`, the `NumAccumFracLength` property fully specifies how the accumulator outputs numerator-related data stored there.

As with all fraction length properties, `NumAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. 30 bits is the default value when you create the filter object. To be able to change the value for this property, set `FilterInternals` for the filter to `SpecifyPrecision`.

Numerator

The numerator coefficients for your filter, taken from the prototype you start with or from the default filter, are stored in this property. Generally this is a 1-by-N array of data in double format, where N is the length of the filter.

All of the filter objects include `Numerator`, except the lattice-based and second-order section filters, such as `dfilt.latticema` and `dfilt.df1tsos`.

NumFracLength

Property `NumFracLength` contains the value that specifies the fraction length for the numerator coefficients for your filter. `NumFracLength` specifies the fraction length used to interpret the numerator coefficients. Used in combination with `CoeffWordLength`, these two properties define the interpretation of the coefficients stored in the vector that contains the numerator coefficients.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `CoeffWordLength` of 16 bits.

NumProdFracLength

A property of all of the direct form IIR `dfilt` objects, except the ones that implement second-order sections, `NumProdFracLength` specifies the fraction length applied to data output from product operations the filter performs on numerator coefficients.

Looking at the signal flow diagram for the `dfilt.df1t` filter, for example, you see that denominators and numerators are handled separately. When you set `ProductMode` to `SpecifyPrecision`, you can change the `NumProdFracLength` setting manually. Otherwise, for multiplication operations that use the numerator coefficients, the filter sets the word length as defined by the `ProductMode` setting.

NumStateFracLength

All the variants of the direct form I structure include the property `NumStateFracLength` to store the fraction length applied to the numerator states for your filter object. By default, this property has the value 15 bits, with the `CoeffWordLength` of 16 bits, which you can change after you create the filter object.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well.

NumStateWordLength

When you look at the flow diagram for the `df1sos` filter object, the states associated with the numerator coefficient operations take the data format from this property and the `NumStateFracLength` property. In combination, these properties fully specify how the filter interprets the state it uses.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 16 bits, with the `NumStateFracLength` of 11 bits.

OutputFracLength

To define the output from your filter object, you need both the word and fraction lengths. `OutputFracLength` determines the fraction length applied to interpret the output data. Combining this with `OutputWordLength` fully specifies the format of the output.

Your fraction length can be any negative or positive integer, or zero. In addition, the fraction length you specify can be larger than the associated word length. Generally, the default value is 11 bits.

OutputMode

Sets the mode the filter uses to scale the filtered (output) data. You have the following choices:

- `AvoidOverflow` — directs the filter to set the property that controls the output data fraction length to avoid causing the data to overflow. In a `df2` filter, this would be the `OutputFracLength` property.

- `BestPrecision` — directs the filter to set the property that controls the output data fraction length to maximize the precision in the output data. For `df1t` filters, this is the `OutputFracLength` property. When you change the word length (`OutputWordLength`), the filter adjusts the fraction length to maintain the best precision for the new word size.

- `SpecifyPrecision` — lets you set the fraction length used by the filtered data. When you select this choice, you can set the output fraction length using the `OutputFracLength` property to define the output precision.

All filters include this property except the direct form I filter which takes the output format from the filter states.

Here is an example that changes the mode setting to bestprecision, and then adjusts the word length for the output.

```
hd=dfilt.df2

hd =

            FilterStructure: 'Direct-Form II'
                 Arithmetic: 'double'
                  Numerator: 1
                Denominator: 1
           PersistentMemory: false
                     States: [0x1 double]

hd.arithmetic='fixed'

hd =

            FilterStructure: 'Direct-Form II'
                 Arithmetic: 'fixed'
                  Numerator: 1
                Denominator: 1
           PersistentMemory: false
                     States: [1x1 embedded.fi]

            CoeffWordLength: 16
             CoeffAutoScale: true
                     Signed: true

            InputWordLength: 16
            InputFracLength: 15

           OutputWordLength: 16
                 OutputMode: 'AvoidOverflow'

            StateWordLength: 16
            StateFracLength: 15
```

```
                   ProductMode: 'FullPrecision'

               AccumWordLength: 40
                 CastBeforeSum: true

                     RoundMode: 'convergent'
                  OverflowMode: 'wrap'

get(hd)
             PersistentMemory: false
FilterStructure: 'Direct-Form II'
                       States: [1x1 embedded.fi]
                    Numerator: 1
                  Denominator: 1
                   Arithmetic: 'fixed'
               CoeffWordLength: 16
                CoeffAutoScale: 1
                       Signed: 1
                    RoundMode: 'convergent'
                  OverflowMode: 'wrap'
              InputWordLength: 16
              InputFracLength: 15
             OutputWordLength: 16
                   OutputMode: 'AvoidOverflow'
                  ProductMode: 'FullPrecision'
              StateWordLength: 16
              StateFracLength: 15
                NumFracLength: 14
                DenFracLength: 14
             OutputFracLength: 13
            ProductWordLength: 32
            NumProdFracLength: 29
            DenProdFracLength: 29
              AccumWordLength: 40
           NumAccumFracLength: 29
           DenAccumFracLength: 29
                CastBeforeSum: 1

hd.outputMode='bestprecision'
```

```
hd =

        FilterStructure: 'Direct-Form II'
              Arithmetic: 'fixed'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: [1x1 embedded.fi]

        CoeffWordLength: 16
         CoeffAutoScale: true
                 Signed: true

        InputWordLength: 16
        InputFracLength: 15

       OutputWordLength: 16
             OutputMode: 'BestPrecision'

        StateWordLength: 16
        StateFracLength: 15

            ProductMode: 'FullPrecision'

        AccumWordLength: 40
          CastBeforeSum: true

              RoundMode: 'convergent'
           OverflowMode: 'wrap'

hd.outputWordLength=8;

get(hd)
        PersistentMemory: false
         FilterStructure: 'Direct-Form II'
                  States: [1x1 embedded.fi]
               Numerator: 1
             Denominator: 1
              Arithmetic: 'fixed'
```

　
```
        CoeffWordLength: 16
         CoeffAutoScale: 1
                 Signed: 1
              RoundMode: 'convergent'
           OverflowMode: 'wrap'
        InputWordLength: 16
        InputFracLength: 15
       OutputWordLength: 8
             OutputMode: 'BestPrecision'
            ProductMode: 'FullPrecision'
        StateWordLength: 16
        StateFracLength: 15
          NumFracLength: 14
          DenFracLength: 14
       OutputFracLength: 5
      ProductWordLength: 32
      NumProdFracLength: 29
      DenProdFracLength: 29
        AccumWordLength: 40
     NumAccumFracLength: 29
     DenAccumFracLength: 29
          CastBeforeSum: 1
```

Changing the `OutputWordLength` to 8 bits caused the filter to change the
`OutputFracLength` to 5 bits to keep the best precision for the output data.

OutputWordLength

Use the property `OutputWordLength` to set the word length used by the output
from your filter. Set this property to a value that matches your intended
hardware. For example, some digital signal processors use 32-bit output so
you would set `OutputWordLength` to 32.

```
[b,a] = butter(6,.5);
hd=dfilt.df1t(b,a);

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

hd

hd =
```

```
            FilterStructure: 'Direct-Form I Transposed'
                 Arithmetic: 'fixed'
                  Numerator: [1x7 double]
                Denominator: [1 0 0.7777 0 0.1142 0 0.0018]
           PersistentMemory: false
                     States: Numerator:  [6x1 fi]
                             Denominator:[6x1 fi]


            CoeffWordLength: 16
             CoeffAutoScale: true
                     Signed: true

            InputWordLength: 16
            InputFracLength: 15

          OutputWordLength: 16
                 OutputMode: 'AvoidOverflow'

    MultiplicandWordLength: 16
    MultiplicandFracLength: 15

            StateWordLength: 16
             StateAutoScale: true

                ProductMode: 'FullPrecision'

            AccumWordLength: 40
              CastBeforeSum: true

                  RoundMode: 'convergent'
               OverflowMode: 'wrap'

hd.outputwordLength=32

hd =

            FilterStructure: 'Direct-Form I Transposed'
                 Arithmetic: 'fixed'
```

```
              Numerator: [1x7 double]
            Denominator: [1 0 0.7777 0 0.1142 0 0.0018]
       PersistentMemory: false
                 States: Numerator:  [6x1 fi]
                         Denominator:[6x1 fi]


        CoeffWordLength: 16
         CoeffAutoScale: true
                 Signed: true

        InputWordLength: 16
        InputFracLength: 15

       OutputWordLength: 32
             OutputMode: 'AvoidOverflow'

 MultiplicandWordLength: 16
 MultiplicandFracLength: 15

        StateWordLength: 16
         StateAutoScale: true

            ProductMode: 'FullPrecision'

        AccumWordLength: 40
          CastBeforeSum: true

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

When you create a filter object, this property starts with the value 16.

OverflowMode

The `OverflowMode` property is specified as one of the following two strings indicating how to respond to overflows in fixed-point arithmetic:

- `'saturate'` — saturate overflows.

When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the applicable word length and fraction length properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest. `saturate` is the default value for `OverflowMode`.

- `'wrap'` — wrap all overflows to the range of representable values.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number. You can learn more about modular arithmetic in Fixed-Point Toolbox documentation.

These rules apply to the `OverflowMode` property.

- Applies to the accumulator and output data only.

- Does not apply to coefficients or input data. These always saturate the results.

- Does not apply to products. Products maintain full precision at all times. Your filters do not lose precision in the products.

Note Numbers in floating-point filters that extend beyond the dynamic range overflow to ±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 you can use `hd` to filter `x` as two separate data sets. The results `ytot` and `ysec` are the same in both cases.

```
xtot=[x,x];
ytot=filter(hd,xtot)
ytot =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1);  % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hd,x) filter(hd,x)]

ysec =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

RoundMode

The `RoundMode` property value specifies the rounding method used for quantizing numerical values. Specify the `RoundMode` property values as one of the following five strings.

RoundMode String	Description of Rounding Algorithm
Ceiling	Round toward positive infinity.
Floor	Round toward negative infinity.
Nearest	Round toward nearest. Ties round toward positive infinity.
Nearest(Convergent)	Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.
Round	Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
Zero	Round toward zero.

The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

ScaleValueFracLength

Filter structures `df1sos`, `df1tsos`, `df2sos`, and `df2tsos` that use `fixed` arithmetic have this property that defines the fraction length applied to the scale values the filter uses between sections. In combination with `CoeffWordLength`, these two properties fully specify how the filter interprets and uses the scale values stored in the property `ScaleValues`. As with fraction length properties, `ScaleValueFracLength` can be any integer,

including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter.

ScaleValues

The `ScaleValues` property values are specified as a scalar (or vector) that introduces scaling for inputs (and the outputs from cascaded sections in the vector case) during filtering:

- When you only have a single section in your filter:
  - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
  - Specify the `ScaleValues` property as a vector of length 2 if you want to specify scaling to the input (scaled with the first entry in the vector) and the output (scaled with the last entry in the vector).

- When you have *L* cascaded sections in your filter:
  - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
  - Specify the value for the `ScaleValues` property as a vector of length *L*+1 if you want to scale the inputs to every section in your filter, along with the output:

    The first entry of your vector specifies the input scaling

    Each successive entry specifies the scaling at the output of the next section

    The final entry specifies the scaling for the filter output.

The default value for `ScaleValues` is 0.

The interpretation of this property is described as follows with diagrams in "Interpreting the ScaleValues Property" on page 7-85.

Note The value of the `ScaleValues` property is not quantized. Data affected by the presence of a scaling factor in the filter is quantized according to the appropriate data format.

When you apply `normalize` to a fixed-point filter, the value for the `ScaleValues` property is changed accordingly.

It is good practice to choose values for this property that are either positive or negative powers of two.

Interpreting the ScaleValues Property. When you specify the values of the `ScaleValues` property of a quantized filter, the values are entered as a vector, the length of which is determined by the number of cascaded sections in your filter:

- When you have only one section, the value of the `Scalevalues` property can be a scalar or a two-element vector.

- When you have L cascaded sections in your filter, the value of the `ScaleValues` property can be a scalar or an $L+1$-element vector.

The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with only one section.

**Application of ScaleValues
to a Single Section**
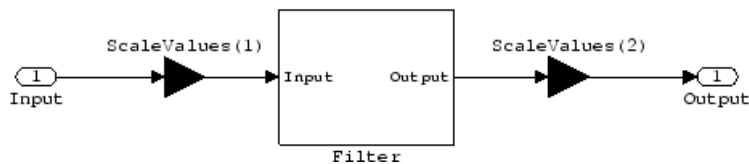
The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with two sections.

**Application of ScaleValues
to Multiple Sections**
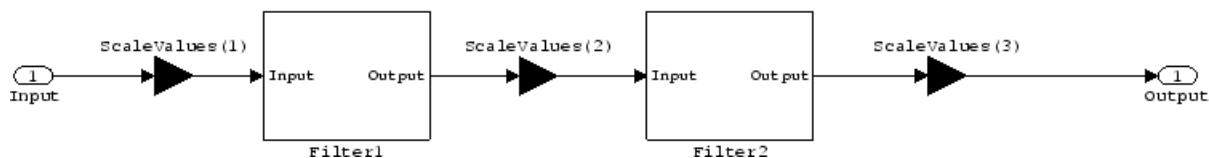
Signed

When you create a dfilt object for fixed-point filtering (you set the property
Arithmetic to fixed, the property Signed specifies whether the filter
interprets coefficients as signed or unsigned. This setting applies only to the
coefficients. While the default setting is true, meaning that all coefficients
are assumed to be signed, you can change the setting to false after you
create the fixed-point filter.

For example, create a fixed-point direct-form II transposed filter with both
negative and positive coefficients, and then change the property value for
Signed from true to false to see what happens to the negative coefficient
values.

```
hd=dfilt.df2t(-5:5)

hd =

            FilterStructure: 'Direct-Form II Transposed'
                  Arithmetic: 'double'
                    Numerator: [-5 -4 -3 -2 -1 0 1 2 3 4 5]
                  Denominator: 1
            PersistentMemory: false
                      States: [10x1 double]

set(hd,'arithmetic','fixed')
hd.numerator

ans =

      -5      -4      -3      -2      -1       0
```

```
      1     2     3     4     5

set(hd,'signed',false)
hd.numerator

ans =

      0     0     0     0     0     0
      1     2     3     4     5
```

Using unsigned coefficients limits you to using only positive coefficients in your filter. Signed is a dynamic property — you cannot set or change it until you switch the setting for the Arithmetic property to fixed.

SosMatrix

When you convert a dfilt object to second-order section form, or create a second-order section filter, sosMatrix holds the filter coefficients as property values. Using the double data type by default, the matrix is in [sections coefficients per section] form, displayed as [15-x-6] for filters with 6 coefficients per section and 15 sections, [15 6].

To demonstrate, the following code creates an order 30 filter using second-order sections in the direct-form II transposed configuration. Notice the sosMatrix property contains the coefficients for all the sections.

```
d = fdesign.lowpass('n,fc',30,0.5);
hd = butter(d);

hd =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
       Arithmetic: 'double'
        sosMatrix: [15x6 double]
      ScaleValues: [16x1 double]
PersistentMemory: false
           States: [2x15 double]

hd.arithmetic='fixed'

hd =
```

```
       FilterStructure: 'Direct-Form II, Second-Order Sections'
            Arithmetic: 'fixed'
              sosMatrix: [15x6 double]
           ScaleValues: [16x1 double]
      PersistentMemory: false
                 States: [1x1 embedded.fi]

      CoeffWordLength: 16
       CoeffAutoScale: true
                Signed: true

      InputWordLength: 16
      InputFracLength: 15

        SectionInputWordLength: 16
         SectionInputAutoScale: true

     SectionOutputWordLength: 16
      SectionOutputAutoScale: true

             OutputWordLength: 16
                   OutputMode: 'AvoidOverflow'

               StateWordLength: 16
               StateFracLength: 15

                   ProductMode: 'FullPrecision'

               AccumWordLength: 40
                 CastBeforeSum: true

                     RoundMode: 'convergent'
                  OverflowMode: 'wrap'

hd.sosMatrix

ans =

    1.0000    2.0000    1.0000    1.0000         0    0.9005
```

1.0000	2.0000	1.0000	1.0000	0	0.7294
1.0000	2.0000	1.0000	1.0000	0	0.5888
1.0000	2.0000	1.0000	1.0000	0	0.4724
1.0000	2.0000	1.0000	1.0000	0	0.3755
1.0000	2.0000	1.0000	1.0000	0	0.2948
1.0000	2.0000	1.0000	1.0000	0	0.2275
1.0000	2.0000	1.0000	1.0000	0	0.1716
1.0000	2.0000	1.0000	1.0000	0	0.1254
1.0000	2.0000	1.0000	1.0000	0	0.0878
1.0000	2.0000	1.0000	1.0000	0	0.0576
1.0000	2.0000	1.0000	1.0000	0	0.0344
1.0000	2.0000	1.0000	1.0000	0	0.0173
1.0000	2.0000	1.0000	1.0000	0	0.0062
1.0000	2.0000	1.0000	1.0000	0	0.0007

The SOS matrix is an M-by-6 matrix, where M is the number of sections in the second-order section filter. Filter hd has M equal to 15 as shown above (15 rows). Each row of the SOS matrix contains the numerator and denominator coefficients (b's and a's) and the scale factors of the corresponding section in the filter.

SectionInputAutoScale

Second-order section filters include this property that determines who the filter handles data in the transitions from one section to the next in the filter.

How the filter represents the data passing from one section to the next depends on the property value of SectionInputAutoScale. The representation the filter uses between the filter sections depends on whether the value of SectionInputAutoScale is true or false.

- SectionInputAutoScale = true means the filter chooses the fraction length to maintain the value of the data between sections as close to the output values from the previous section as possible. true is the default setting.

- SectionInputAutoScale = false removes the automatic scaling of the fraction length for the intersection data and exposes the property that controls the coefficient fraction length (SectionInputFracLength) so you can change it. For example, if the filter is a second-order, direct form FIR filter, setting SectionInputAutoScale to false exposes the

SectionInputFracLength property that specifies the fraction length applied to data between the sections.

SectionInputFracLength

Second-order section filters use quantizers at the input to each section of the filter. The quantizers apply to the input data entering each filter section. Note that the quantizers for each section are the same. To set the fraction length for interpreting the input values, use the property value in SectionInputFracLength.

In combination with CoeffWordLength, SectionInputFracLength fully determines how the filter interprets and uses the state values stored in the property States. As with all word and fraction length properties, SectionInputFracLength can be any integer, including integers larger than CoeffWordLength, and positive or negative integers. 15 bits is the default value when you create the filter object.

SectionInputWordLength

SOS filters are composed of sections, each one a second-order filter. Filtering data input to the filter involves passing the data through each filter section. SectionInputWordLength specifies the word length applied to data as it enters one filter section from the previous section. Only second-order implementations of direct-form I transposed and direct-form II transposed filters include this property.

The following diagram shows an SOS filter composed of sections (the bottom part of the diagram) and a possible internal structure of each Section (the top portion of the diagram), in this case — a direct form I transposed second order sections filter structure. Note that the output of each section is fed through a multiplier. If the gain of the multiplier =1, then the last Cast block of the Section is ignored, and the format of the output is NumSumQ.

* : only if CastBeforeSum is True
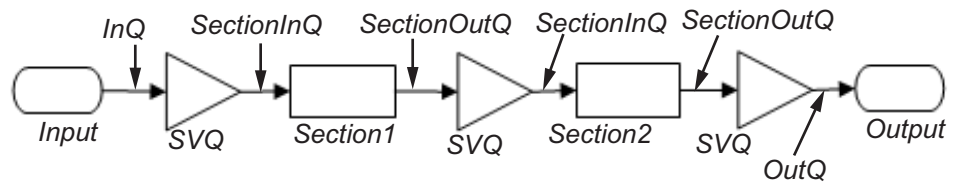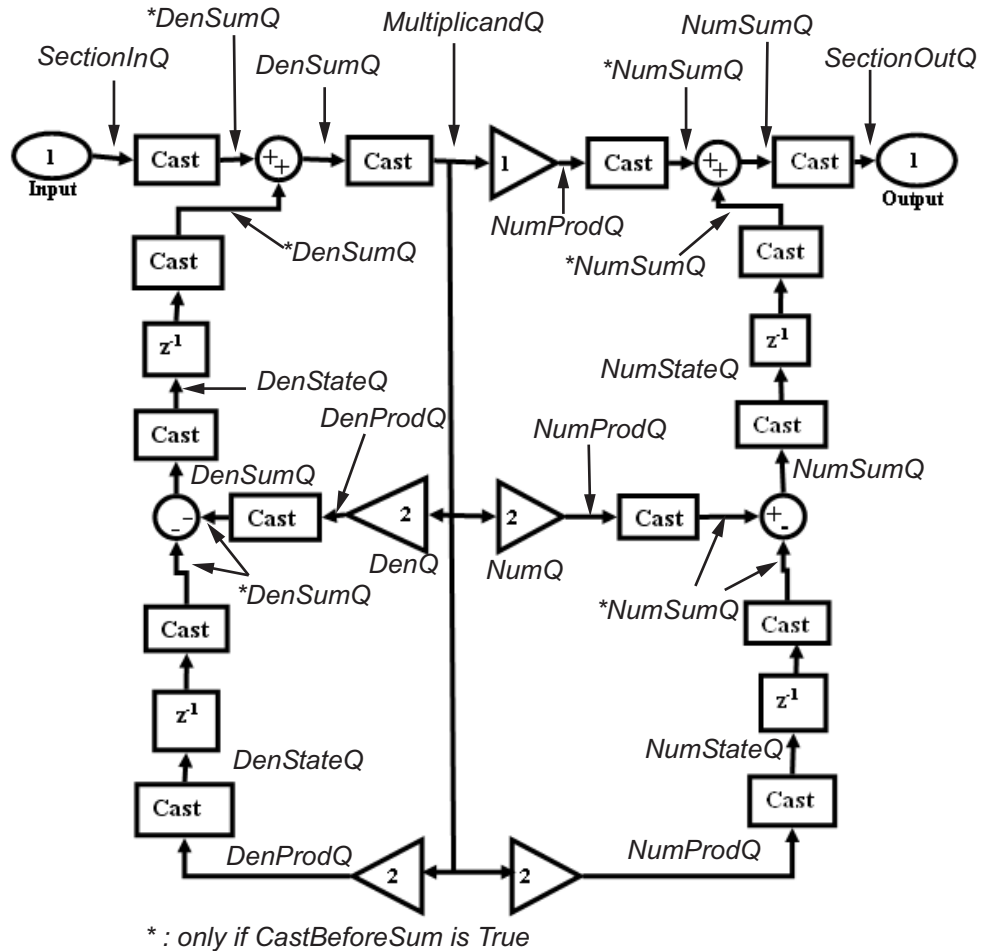
SectionInputWordLength defaults to 16 bits.

SectionOutputAutoScale

Second-order section filters include this property that determines who the filter handles data in the transitions from one section to the next in the filter.

How the filter represents the data passing from one section to the next depends on the property value of SectionOutputAutoScale. The representation the filter uses between the filter sections depends on whether the value of SectionOutputAutoScale is true or false.

- SectionOutputAutoScale = true means the filter chooses the fraction length to maintain the value of the data between sections as close to the output values from the previous section as possible. true is the default setting.

- SectionOutputAutoScale = false removes the automatic scaling of the fraction length for the intersection data and exposes the property that controls the coefficient fraction length (SectionOutputFracLength) so you can change it. For example, if the filter is a second-order, direct form FIR filter, setting SectionOutputAutoScale = false exposes the SectionOutputFracLength property that specifies the fraction length applied to data between the sections.

SectionOutputFracLength

Second-order section filters use quantizers at the output from each section of the filter. The quantizers apply to the output data leaving each filter section. Note that the quantizers for each section are the same. To set the fraction length for interpreting the output values, use the property value in SectionOutputFracLength.

In combination with CoeffWordLength, SectionOutputFracLength determines how the filter interprets and uses the state values stored in the property States. As with all fraction length properties, SectionOutputFracLength can be any integer, including integers larger than CoeffWordLength, and positive or negative integers. 15 bits is the default value when you create the filter object.

SectionOutputWordLength

SOS filters are composed of sections, each one a second-order filter. Filtering data input to the filter involves passing the data through each filter section. `SectionOutputWordLength` specifies the word length applied to data as it leaves one filter section to go to the next. Only second-order implementations direct-form I transposed and direct-form II transposed filters include this property.

The following diagram shows an SOS filter composed of sections (the bottom part of the diagram) and a possible internal structure of each Section (the top portion of the diagram), in this case — a direct form I transposed second order sections filter structure. Note that the output of each section is fed through a multiplier. If the gain of the multiplier =1, then the last Cast block of the Section is ignored, and the format of the output is NumSumQ.

* : only if CastBeforeSum is True
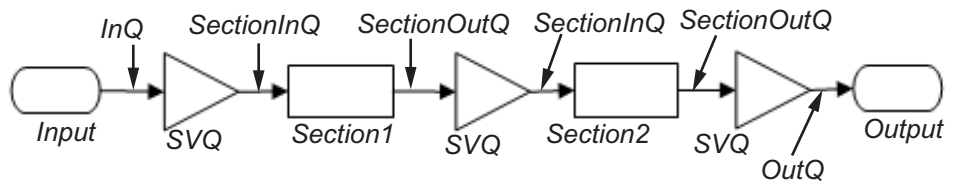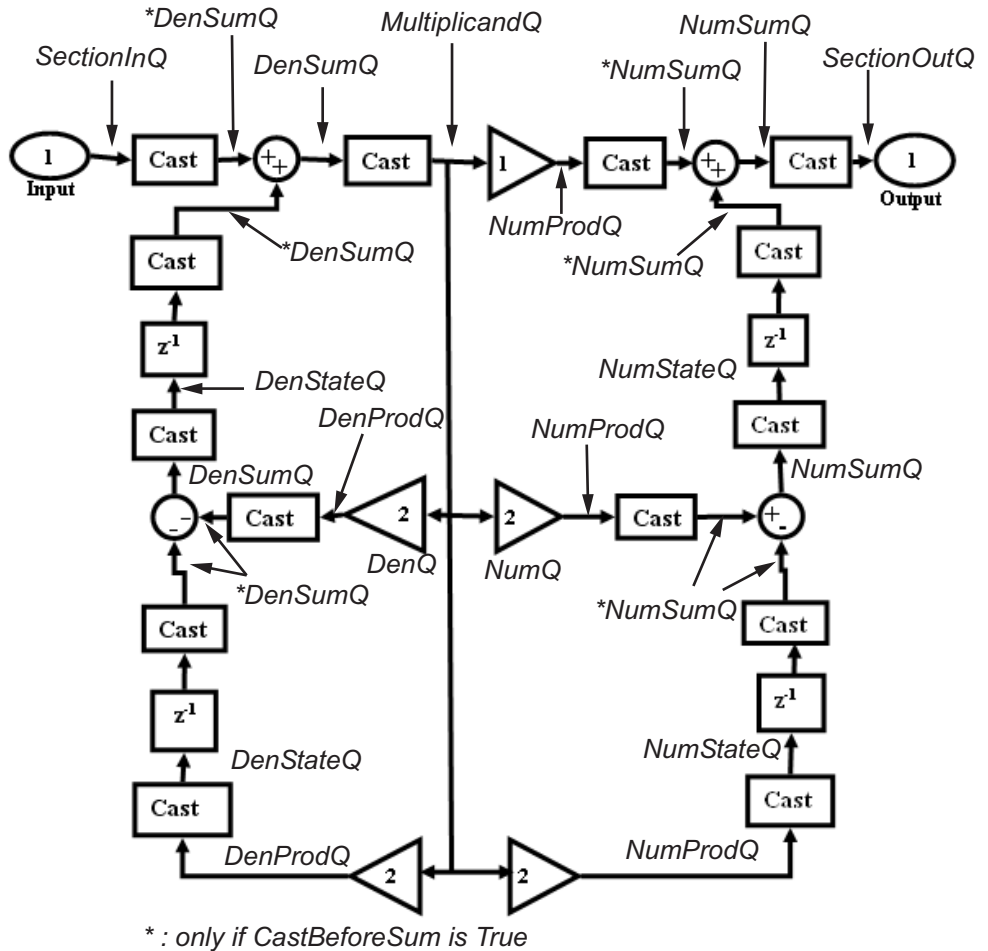
`SectionOutputWordLength` defaults to 16 bits.

StateAutoScale

Although all filters use states, some do not allow you to choose whether the filter automatically scales the state values to prevent overruns or bad arithmetic errors. You select either of the following settings:

- `StateAutoScale = true` means the filter chooses the fraction length to maintain the value of the states as close to the double-precision values as possible. When you change the word length applied to the states (where allowed by the filter structure), the filter object changes the fraction length to try to accommodate the change. `true` is the default setting.

- `StateAutoScale = false` removes the automatic scaling of the fraction length for the states and exposes the property that controls the coefficient fraction length so you can change it. For example, in a direct form I transposed SOS FIR filter, setting `StateAutoScale = false` exposes the `NumStateFracLength` and `DenStateFracLength` properties that specify the fraction length applied to states.

Each of the following filter structures provides the `StateAutoScale` property:

- `df1t`
- `df1tsos`
- `df2t`
- `df2tsos`
- `dffirt`

Other filter structures do not include this property.

StateFracLength

Filter states stored in the property `States` have both word length and fraction length. To set the fraction length for interpreting the stored filter object state values, use the property value in `StateFracLength`.

In combination with `CoeffWordLength`, `StateFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`.

As with all fraction length properties, `StateFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

States

Digital filters are dynamic systems. The behavior of dynamic systems (their response) depends on the input (stimulus) to the system and the current or previous *state* of the system. You can say the system has memory or inertia. All fixed- or floating-point digital filters (as well as analog filters) have states.

Filters use the states to compute the filter output for each input sample, as well using them while filtering in loops to maintain the filter state between loop iterations. This toolbox assumes zero-valued initial conditions (the dynamic system is at rest) by default when you filter the first input sample. Assuming the states are zero initially does not mean the states are not used; they are, but arithmetically they do not have any effect.

Filter objects store the state values in the property `States`. The number of stored states depends on the filter implementation, since the states represent the delays in the filter implementation.

When you review the display for a filter object with fixed arithmetic, notice that the states return an embedded `fi` object, as you see here.

```
b = ellip(6,3,50,300/500);
hd=dfilt.dffir(b)

hd =

        FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'double'
               Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858 0.2938 0.0773]
          PersistentMemory: false
                  States: [6x1 double]
```

```
hd.arithmetic='fixed'

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858 0.2938 0.0773]
         PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: 'on'
                  Signed: 'on'

          InputWordLength: 16
          InputFracLength: 15

         OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

               ProductMode: 'FullPrecision'

          AccumWordLength: 40
            CastBeforeSum: 'on'

                RoundMode: 'convergent'
             OverflowMode: 'wrap'

          InheritSettings: 'off'
```

`fi` objects provide fixed-point support for the filters. To learn more about the details about `fi` objects, refer to your Fixed-Point Toolbox documentation.

The property `States` lets you use a `fi` object to define how the filter interprets the filter states. For example, you can create a `fi` object in MATLAB, then assign the object to States, as follows:

```
statefi=fi([],16,12)
```

```
statefi =

[]
   DataTypeMode = Fixed-point: binary point scaling
         Signed = true
     Wordlength = 16
Fractionlength = 12
```

This fi object does not have a value associated (notice the [] input argument to fi for the value), and it has word length of 16 bits and fraction length of 12 bit. Now you can apply statefi to the States property of the filter hd.

```
set(hd,'States',statefi);
Warning: The 'States' property will be reset to the value
specified at construction before filtering.
Set the 'PersistentMemory' flag to 'True'
to avoid changing this property value.
hd

hd =

          FilterStructure: 'Direct-Form FIR'
               Arithmetic: 'fixed'
                Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858
                           0.2938 0.0773]
         PersistentMemory: false
                   States: [1x1 embedded.fi]

          CoeffWordLength: 16
           CoeffAutoScale: 'on'
                   Signed: 'on'

          InputWordLength: 16
          InputFracLength: 15

         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'

              ProductMode: 'FullPrecision'
          AccumWordLength: 40
```

```
        CastBeforeSum: 'on'

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

StateWordLength

While all filters use states, some do not allow you to directly change the state representation — the word length and fraction lengths — independently. For the others, `StateWordLength` specifies the word length, in bits, the filter uses to represent the states. Filters that do not provide direct state word length control include:

- `df1`

- `dfasymfir`

- `dffir`

- `dfsymfir`

For these structures, the filter derives the state format from the input format you choose for the filter — except for the `df1` IIR filter. In this case, the numerator state format comes from the input format and the denominator state format comes from the output format. All other filter structures provide control of the state format directly.

TapSumFracLength

Direct-form FIR filter objects, both symmetric and antisymmetric, use this property. To set the fraction length for output from the sum operations that involve the filter tap weights, use the property value in `TapSumFracLength`. To enable this property, set the `TapSumMode` to `SpecifyPrecision` in your filter.

As you can see in this code example that creates a fixed-point asymmetric FIR filter, the `TapSumFracLength` property becomes available after you change the `TapSumMode` property value.

```
hd=dfilt.dfasymfir

hd =
```

```
                    FilterStructure: 'Direct-Form Antisymmetric FIR'
                         Arithmetic: 'double'
                          Numerator: 1
                   PersistentMemory: false
                             States: [0x1 double]

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

         hd =

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

                   CoeffWordLength: 16
                    CoeffAutoScale: true
                            Signed: true

                   InputWordLength: 16
                   InputFracLength: 15

                  OutputWordLength: 16
                        OutputMode: 'AvoidOverflow'

                        TapSumMode: 'KeepMSB'
                  TapSumWordLength: 17

                       ProductMode: 'FullPrecision'

                   AccumWordLength: 40

                     CastBeforeSum: true
                         RoundMode: 'convergent'
                      OverflowMode: 'wrap'
```

With the filter now in fixed-point mode, you can change the `TapSumMode` property value to `SpecifyPrecision`, which gives you access to the `TapSumFracLength` property.

```
set(hd,'TapSumMode','SpecifyPrecision');
hd

hd =

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

          CoeffWordLength: 16
           CoeffAutoScale: true
                    Signed: true

          InputWordLength: 16
          InputFracLength: 15

         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'

               TapSumMode: 'SpecifyPrecision'
        TapSumWordLength: 17
        TapSumFracLength: 15

              ProductMode: 'FullPrecision'

          AccumWordLength: 40

            CastBeforeSum: true
                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

In combination with `TapSumWordLength`, `TapSumFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`.

As with all fraction length properties, `TapSumFracLength` can be any integer, including integers larger than `TapSumWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

TapSumMode

This property, available only after your filter is in fixed-point mode, specifies how the filter outputs the results of summation operations that involve the filter tap weights. Only symmetric (`dfilt.dfsymfir`) and antisymmetric (`dfilt.dfasymfir`) FIR filters use this property.

When available, you select from one of the following values:

- `FullPrecision` — means the filter automatically chooses the word length and fraction length to represent the results of the sum operation so they retain all of the precision provided by the inputs (addends).

- `KeepMSB` — means you specify the word length for representing tap sum summation results to keep the higher order bits in the data. The filter sets the fraction length to discard the LSBs from the sum operation. This is the default property value.

- `KeepLSB` — means you specify the word length for representing tap sum summation results to keep the lower order bits in the data. The filter sets the fraction length to discard the MSBs from the sum operation. Compare to the `KeepMSB` option.

- `SpecifyPrecision` — means you specify the word and fraction lengths to apply to data output from the tap sum operations.

TapSumWordLength

Specifies the word length the filter uses to represent the output from tap sum operations. The default value is 17 bits. Only `dfasymfir` and `dfsymfir` filters include this property.

Adaptive Filter Properties

In this section...
"Property Summaries" on page 7-103
"Property Details for Adaptive Filter Properties" on page 7-108

Property Summaries

The following table summarizes the adaptive filter properties and provides a brief description of each. Full descriptions of each property, in alphabetical order, are given in the subsequent section.

Property	Description
Algorithm	Reports the algorithm the object uses for adaptation. When you construct your adaptive filter object, this property is set automatically by the constructor, such as adaptfilt.nlms 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 - step). lambda is the input argument that represents AvgFactor
BkwdPredErrorPower	Returns the minimum mean-squared prediction error. Refer to [2] in the bibliography for details about linear prediction.
BkwdPrediction	Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.

Property	Description
Blocklength	Block length for the coefficient updates. This must be a positive integer such that (1/blocklength) is also an integer. For faster execution, blocklength should be a power of two. blocklength 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 (blocklen - 1) or (swblocklen - 1) depending on the algorithm.
EpsilonStates	Vector of the epsilon values of the adaptive filter. EpsilonStates defaults to a vector of zeros with (projectord - 1) elements.
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 l - 1.

Property	Description
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 [2] in the bibliography for details about linear prediction.
FwdPrediction	Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
InitFactor	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. Called delta 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 l-by-l, where l is the filter length.
KalmanGain	Empty when you construct the object, this gets populated after you run the filter.
KalmanGainStates	Contains the states of the Kalman gain updates during adaptation.

Property	Description
Leakage	Contains the setting for leakage in the adaptive filter algorithm. Using a leakage factor that is not 1 forces the weights to adapt even when they have found the minimum error solution. Forcing the adaptation can improve the numerical performance of the LMS algorithm.
OffsetCov	Contains the offset covariance matrix.
Offset	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors.
Power	A vector of 2*l 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.
SecondaryPathCoeffs	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.
SecondaryPathEstimate	An estimate of the secondary path filter model.

Property	Description
SecondaryPathStates	The states of the secondary path filter, the unknown system.
SqrtCov	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
SqrtlnvCov	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. swblocklen defaults to 16.

Like dfilt objects, adaptfilt objects have properties that govern their behavior and store some of the results of filtering operations. The following pages list, in alphabetical order, the name of every property associated with adaptfilt objects. Note that not all adaptfilt objects have all of these properties. To view the properties of a particular adaptive filter, such as an adaptfilt.bap filter, use get with the object handle, like this:

```
ha = adaptfilt.bap(32,0.5,4,1.0);
get(ha)
        PersistentMemory: false
               Algorithm: 'Block Affine Projection FIR Adaptive Filter'
            FilterLength: 32
            Coefficients: [1x32 double]
```

```
             States: [35x1 double]
           StepSize: 0.5000
    ProjectionOrder: 4
          OffsetCov: [4x4 double]
```

`get` shows you the properties for `ha` and the values for the properties. Entering the object handle returns the same values and properties without the formatting of the list and the more familiar property names.

Property Details for Adaptive Filter Properties

Algorithm

Reports the algorithm the object uses for adaptation. When you construct you adaptive filter object, this property is set automatically. You cannot change the value—it is read only.

AvgFactor

Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. `AvgFactor` should lie between zero and one. For default filter objects, `AvgFactor` equals (1 - `step`). `lambda` is the input argument that represent `AvgFactor`

BkwdPredErrorPower

Returns the minimum mean-squared prediction error in the backward direction. Refer to [2] in the bibliography for details about linear prediction.

BkwdPrediction

When you use an adaptive filter that does backward prediction, such as `adaptfilt.ftf`, one property of the filter contains the backward prediction coefficients for the adapted filter. With these coefficient, the forward coefficients, and the system under test, you have the full set of knowledge of how the adaptation occurred. Two values stored in properties compose the `BkwdPrediction` property:

- Coefficients, which contains the coefficients of the system under test, as determined using backward predictions process.

- Error, which is the difference between the filter coefficients determined by backward prediction and the actual coefficients of the sample filter. In this example, `adaptfilt.ftf` identifies the coefficients of an unknown FIR system.

```
x   = randn(1,500);     % Input to the filter
b   = fir1(31,0.5);     % FIR system to be identified
n   = 0.1*randn(1,500); % Observation noise signal
d   = filter(b,1,x)+n;  % Desired signal
N   = 31;               % Adaptive filter order
lam = 0.99;             % RLS forgetting factor
del = 0.1;              % Soft-constrained initialization factor
ha = adaptfilt.ftf(32,lam,del);
[y,e] = filter(ha,x,d);


ha


ha =


              Algorithm: 'Fast Transversal Least-Squares Adaptive Filter'
           FilterLength: 32
           Coefficients: [1x32 double]
                 States: [31x1 double]
        ForgettingFactor: 0.9900
             InitFactor: 0.1000
           FwdPrediction: [1x1 struct]
          BkwdPrediction: [1x1 struct]
              KalmanGain: [32x1 double]
        ConversionFactor: 0.7338
        KalmanGainStates: [32x1 double]
        PersistentMemory: false


ha.coefficients


ans =


  Columns 1 through 8


   -0.0055    0.0048    0.0045    0.0146   -0.0009    0.0002   -0.0019    0.0008
```

7-109

```
      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 (`l/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 (`blocklen - 1`) or (`swblocklen - 1`) depending on the algorithm.

EpsilonStates

Vector of the epsilon values of the adaptive filter. `EpsilonStates` defaults to a vector of zeros with (`projectord - 1`) elements.

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 `l` - 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 [2] 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 l-by-l, where l 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*l 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

Property Summaries

The following table summarizes the multirate filter properties and provides a brief description of each. Full descriptions of each property are given in the subsequent section.

Name	Values	Default	Description
BlockLength	Positive integers	100	Length of each block of data input to the FFT used in the filtering. `fftfirinterp` 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.

Name	Values	Default	Description
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, 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	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.

Name	Values	Default	Description
InterpolationFactor	Positive integers	2	Interpolation factor for the filter. l 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.
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.

Name	Values	Default	Description
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.
PersistentMemory	false or true	false	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.
RateChangeFactors	[l,m]	[2,3] or [3,2]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal. The default changes depending on whether the filter decimates or interpolates.

Name	Values	Default	Description
States	Any m+1-by-n matrix of double values	2-by-2 matrix, int32	Stored conditions for the filter, including values for the integrator and comb sections. n is the number of filter sections and m is the differential delay. Stored in a filtstates object.
SectionWordLengthMode	MinWordLengths or SpecifyWordLengths	MinWordLength	Determines whether the filter object sets the section word lengths or you provide the word lengths explicitly. By default, the filter uses the input and output word lengths in the command to determine the proper word lengths for each section, according to the information in "Constraints and Word Length Considerations". 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
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. The following pages list the mfilt object properties in alphabetical order.

Property Details for Multirate Filter Properties

BitsPerSection
Any integer or a vector of length 2*n.

Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using wrap arithmetic). Enter bps as a scalar or vector of length 2*n, where n is the number of sections. When bps is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.

BlockLength

Length of each block of input data used in the filtering.

`mfilt.fftfirinterp` objects process data in blocks whose length is determined by the value you set for the `BlockLength` property. By default the property value is 100. When you set the `BlockLength` value, try choosing a value so that [`BlockLength` + 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 Mode. There are four usage modes for this that you set using the `FilterInternals` property in multirate filters.

- `FullPrecision` — All word and fraction lengths set to B_{max} + 1, called B_{accum} by fred harris in [2]. Full precision is the default setting.

- `MinWordLengths` — Minimum Word Lengths

- `SpecifyWordLengths` — Specify Word Lengths

- `SpecifyPrecision` — Specify Precision

Full Precision

In full precision mode, the word lengths of all sections and the output are set to B_{accum} as defined by

$$B_{accum} = ceil(N_{secs}(Log_2(D \times M)) + InputWordLength)$$

where N_{secs} is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display looks for this mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

Minimum Word Lengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than B_{accum}, roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [3]) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than B_{accum} as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits

computed using eq. 21 in Hogenauer's paper from B_{accum} to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the fraction length of that section by one bit compared to the input fraction length. Setting the output word length for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'MinWordLengths'

OutputWordLength: 16
```

Specify Word Lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length 2*(NumberOfSections), where each vector element represents the word length for a section. If you specify a calar, such as B_{accum}, the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display looks like for the `SpecifyWordLengths` mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyWordLengths'

SectionWordLengths: [19 18 18 17]

OutputWordLength: 16
```

Specify Precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the `SpecifyPrecision` mode, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the word length for each section. When you enter a scalar such as B_{accum}, the CIC algorithm expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length 2*(NumberOfSections) with elements that represent the fraction length for each section as well. When you enter a calar such as B_{accum}, the design applies scalar expansion as done for the word lengths.

Here is the SpecifyPrecision display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11
```

FilterStructure

Reports the type of filter object, such as a decimator or fractional integrator. You cannot set this property — it is always read only and results from your choice of mfilt object. Because of the length of the names of multirate filters, FilterStructure often returns a vector specification for the string. For example, when you use mfilt.firfracinterp to design a filter, FilterStructure returns as [1x49 char].

```
hm=mfilt.firfracinterp

hm =

          FilterStructure: [1x49 char]
                 Numerator: [1x72 double]
        RateChangeFactors: [3 2]
```

```
     PersistentMemory: false
               States: [24x1 double]
```

InputOffset

When you decimate signals whose length is not a multiple of the decimation factor M, the last samples — (nM +1) to [(n+1)(M) -1], where n is an integer — are processed and used to track where the filter stopped processing input data and when to expect the next output sample. If you think of the filtering process as generating an output for a block of input data, `InputOffset` contains a count of the number of samples in the last incomplete block of input data.

Note `InputOffset` applies only when you set `PersistentMemory` to `true`. Otherwise, `InputOffset` is not available for you to use.

Two different cases can arise when you decimate a signal:

1 The input signal is a multiple of the filter decimation factor. In this case, the filter processes the input samples and generates output samples for all inputs as determined by the decimation factor. For example, processing 99 input samples with a filter that decimates by three returns 33 output samples.

2 The input signal is not a multiple of the decimation factor. When this occurs, the filter processes all of the input samples, generates output samples as determined by the decimation factor, and has one or more input samples that were processed but did not generate an output sample.

For example, when you filter 100 input samples with a filter which has decimation factor of 3, you get 33 output samples, and 1 sample that did not generate an output. In this case, `InputOffset` stores the value 1 after the filter run.

`InputOffset` equal to 1 indicates that, if you divide your input signal into blocks of data with length equal to your filter decimation factor, the filter processed one sample from a new (incomplete) block of data. Subsequent inputs to the filter are concatenated with this single sample to form the next block of length m.

One way to define the value stored in `InputOffset` is

```
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 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 ±`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)

Bibliography

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/Fall96Cleve.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

[1] Hayes, M.H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, 1996.

[2] Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

Multirate Filters

[1] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.

[2] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[3] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 2, April 1981, pp. 155-162.

[4] Lyons, Richard G., *Understanding Digital Signal Processing*, Prentice Hall PTR, 2004

[5] Mitra, S.K., *Digital Signal Processing*, McGraw-Hill, 1998.

[6] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Inc., 1996.

Frequency Transformations

[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] 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.

[4] 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.

Examples

Use this list to find examples in the documentation.

Using FDATool

Adaptive Filters

Index