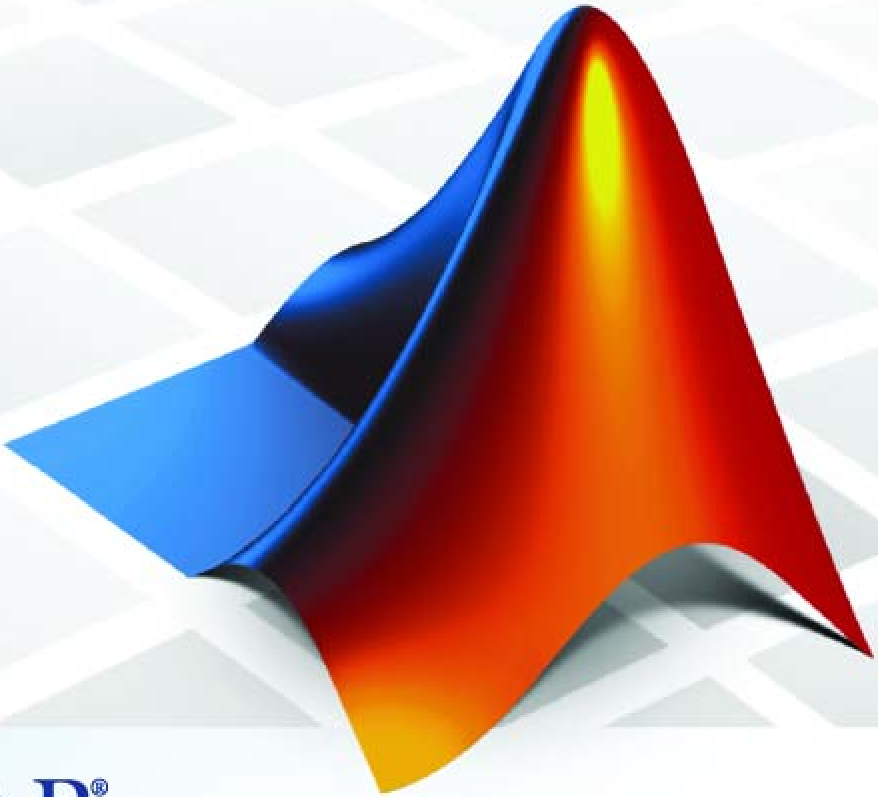
Getting Started with **Filter Design Toolbox 4**



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.

Getting Started with Filter Design Toolbox

Revision History

March 2007	Online Only	Revised for Version 4.1 (Release 2007a)

Contents

4 Designing Multirate and Multistage Filters

5 Converting from Floating-Point to Fixed-Point

6 Data Types

A Examples

Index

What Is Filter Design Toolbox?

Introducing Filter Design Toolbox

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

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

Key Features

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

- Adaptive filter design, analysis, and implementation, including LMS-based, RLS-based, lattice-based, frequency-domain, fast transversal, and affine projection adaptive filters
- VHDL and Verilog code generation for fixed-point filters with the Filter Design HDL Coder

Designing a Filter in Two Steps

How Filter Design Toolbox Works

The unique feature of Filter Design Toolbox is that you do not need to know any specific filter algorithms to design a good working filter. You take a given set of design parameters for the filter, such as a stopband frequency, a passband frequency, and a stopband attenuation, for example, and-using these parameters-design a specification object for the filter. Then, using this specification object, you design the filter.

There are two distinct objects involved in filter design:

- Specification Object — Captures the required design parameters of a filter
- Filter Object — Describes the designed filter; includes the array of coefficients and the filter structure

You can run the code in the following examples from the Help browser (select the code, right-click the selection, and choose **Evaluate Selection** from the context menu), or you can enter the code on the MATLAB® command line. Before you begin this example, start MATLAB and verify that you have installed Signal Processing Toolbox and Filter Design Toolbox (enter `ver` at the command prompt). You should see Filter Design Toolbox, Signal Processing Toolbox, and Fixed-Point Toolbox in the list of installed products.

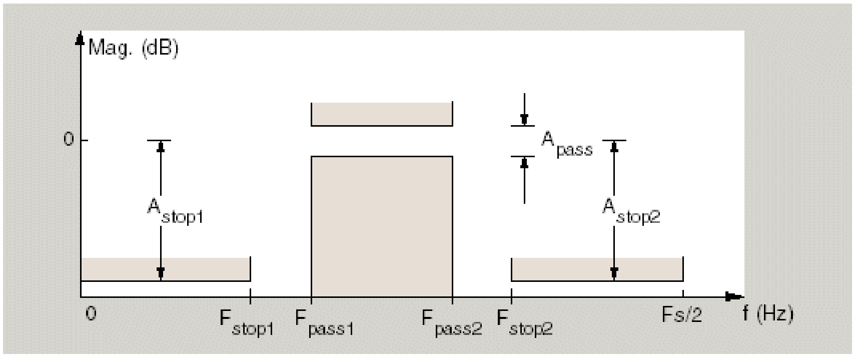
Basic Filter Design Process

Use the following two steps to design a simple filter.

1. Create a filter specification object.

2. Design your filter.

Assume that you want to design a bandpass filter. Typically a bandpass filter is defined as shown in the following figure.



This bandpass filter has the following specifications:

- `ast1 or Astop1` — Attenuation in the first stopband = 60
- `fst1 or Fstop1` — Edge of the stopband = .35
- `fp1 or Fpass1` — Edge of the passband = .45
- `fp2 of Fpass2` — Closing edge of the passband = .65
- `fst2` — Edge of the second stopband = .75
- `ast2 or Astop2` — Attenuation in the second stopband = 60
- `ap or Apass` — Amount of ripple allowed in the passband = 1

Example — Design a Filter in Two Steps

1. To create a filter specification object type or evaluate the following code at the MATLAB prompt:

```
>> BandPassSpecObj = fdesign.bandpass

BandPassSpecObj =

               Response: 'Bandpass'
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7x1 cell}
    NormalizedFrequency: true
                 Fstop1: 0.35
                 Fpass1: 0.45
                 Fpass2: 0.55
                 Fstop2: 0.65
                 Astop1: 60
                  Apass: 1
                 Astop2: 60
```

Note that the specification parameters, such as `Fstop1`, are all given default values when none are provided. There are only two values that need to be changed `Fpass2` and `Fstop2`. To set the correct values, use the `set` command, which takes the object first, and then the parameter value pairs. Type or evaluate the following code at the MATLAB prompt:

```
>> set(BandPassSpecObj, 'Fpass2', 0.65, 'Fstop2', 0.75)
```

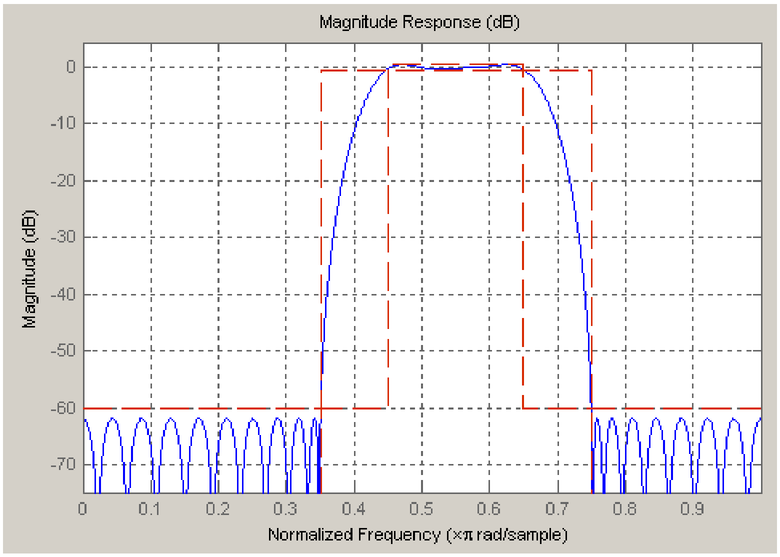
`BandPassSpecObj` is the new filter specification object which contains all the required design parameters, including the filter type.

2 Design the filter by using the `design` command. Evaluate or type the following at the MATLAB prompt:

```
>> BandPassFilt = design(BandPassSpecObj)

BandPassFilt =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x47 double]
    PersistentMemory: false
```

To check your work, you can plot the filter magnitude response using the Filter Visualization tool. Verify that all the design parameters are met:

```
fvtool(BandPassFilt) %plot the filter magnitude response
```



Using FilterBuilder to Design a Filter

The FilterBuilder presents the option of designing a filter using a GUI dialog as opposed to the command line instructions. You use the FilterBuilder to design the same bandpass filter designed in the previous section, "Basic Filter Design Process" on page 2-3

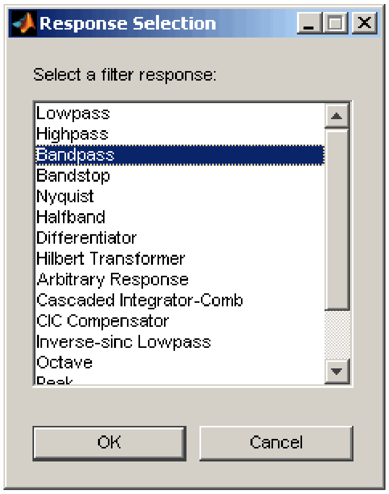
Example — Using Filterbuilder to Design a Simple Filter

To design the filter using the FilterBuilder:

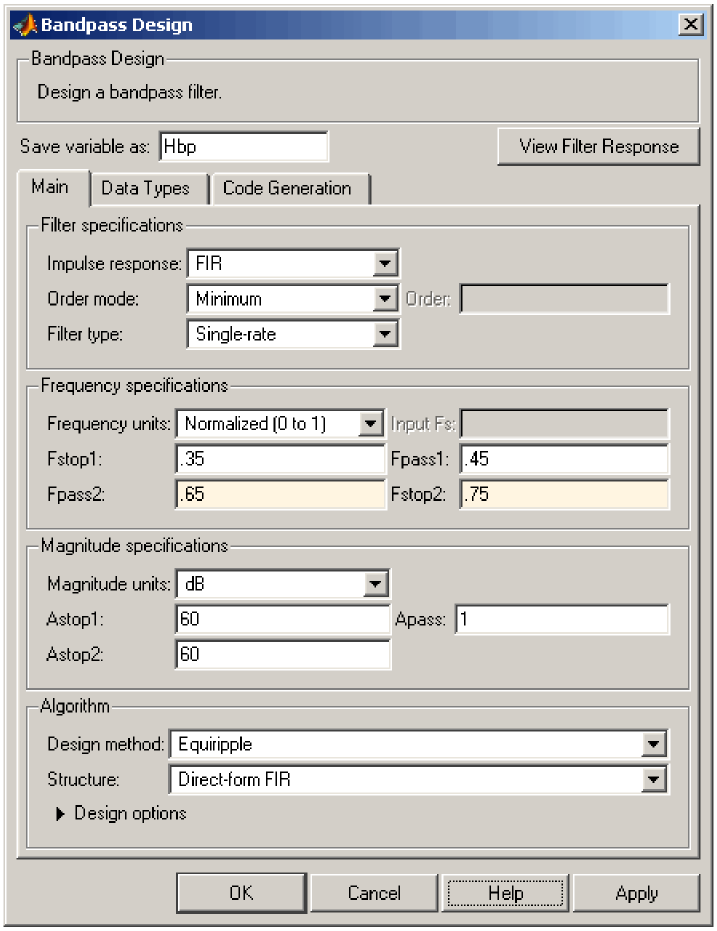
1 Type or evaluate the following at the MATLAB prompt:

```
filterbuilder
```

The following dialog box opens:



2 Select `Bandpass` filter response from the list in the dialog box, and hit the **OK** button. The following dialog box opens:



Bandpass Design

Design a bandpass filter.

Save variable as: Hbp

Filter specifications

Impulse response: FIR

Order mode: Minimum

Order:

Filter type: Single-rate

Frequency specifications

Frequency units: Normalized (0 to 1)

Input Fs:

Fstop1: .35

Fpass1: .45

Fpass2: .65

Fstop2: .75

Magnitude specifications

Magnitude units: dB

Astop1: 60

Apass: 1

Astop2: 60

Algorithm

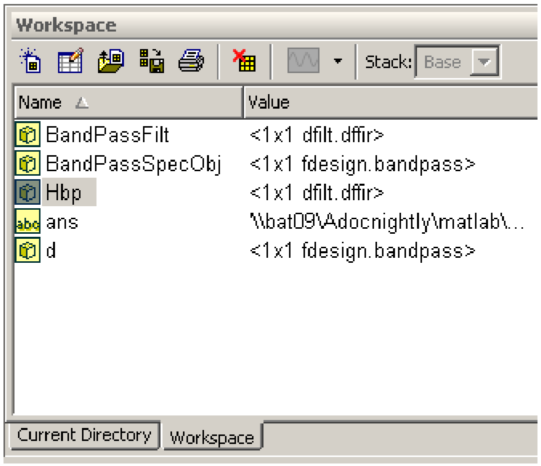
Design method: Equiripple

Structure: Direct-form FIR

3 Enter the correct frequencies for **Fpass2** and **Fstop2**, as shown in the preceding figure, then click **OK**. The following message appears at the MATLAB prompt:

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

If you display the Workspace tab, as shown in the following figure, you see the object `Hbp` has been placed on your workspace.



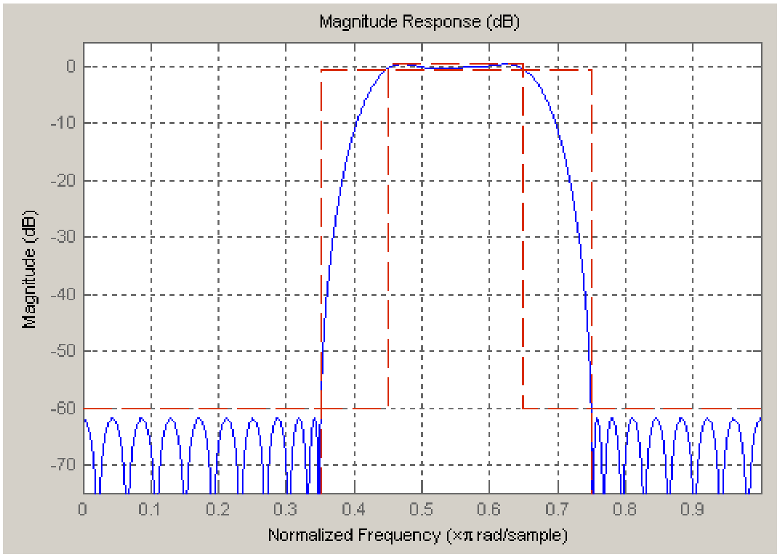
Workspace

Stack: Base

Name	Value
BandPassFilt	<1x1 dfilt.dffir>
BandPassSpecObj	<1x1 fdesign.bandpass>
Hbp	<1x1 dfilt.dffir>
ans	'\\bat09\Adocnightly\matlab\...
d	<1x1 fdesign.bandpass>

4 To check your work, plot the filter magnitude response using the Filter Visualization tool. Verify that all the design parameters are met:

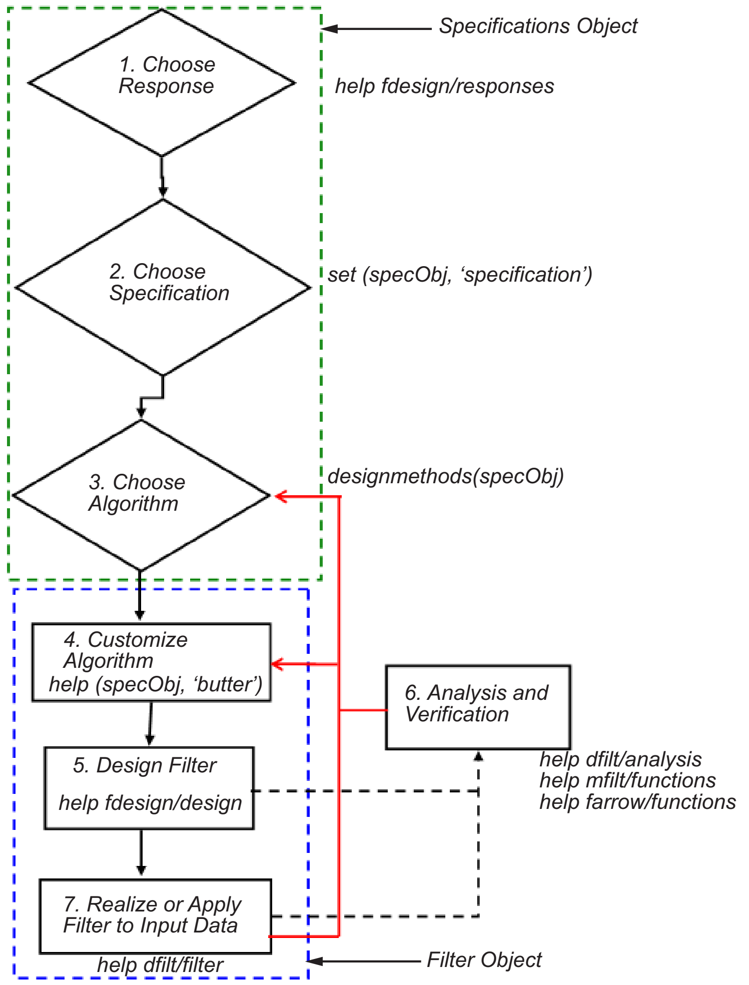
```
fvtool(Hbp) %plot the filter magnitude response
```



Designing a Filter — a High Level Overview

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 three steps of the filter design process relate to the filter Specifications Object, while the last four steps involve the Filter Object. Both of these objects are discussed in more detail in the following sections. Step

6–the analysis and verification of the designed filter, 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.

Here are the details for each of the steps shown above :

- "Step 1: Choose a Response" on page 3-3
- "Step 2: Choose a Specification" on page 3-4
- "Step 3: Choose an Algorithm" on page 3-6
- "Step 4: Customize the Algorithm" on page 3-7
- "Step 5: Design the Filter" on page 3-7
- "Step 6: Design Analysis" on page 3-8
- "Step 7: Realize or Apply the Filter to Input Data" on page 3-9

Step 1: Choose 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. 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. You must select a response to initiate the filter design process.

Step 2: Choose 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 selects a default specification for the response you chose in "Step 1: Choose a Response" on page 3-3, and even provides default values for all design parameters included in the specification.

Step 3: Choose 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), type or evaluate the following:

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

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

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

Step 4: Customize the Algorithm

The customization options available for any given algorithm depend not only on the algorithm itself, selected in "Step 3: Choose an Algorithm" on page 3-6, but also on the specification selected in "Step 2: Choose a Specification" on page 3-4. 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 during the "Step 5: Design the Filter" on page 3-7, because these options are the properties of the Filter Object, not the Specification Object.

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

Step 5: Design 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 selects the best available one.

To apply the algorithm chosen in "Step 3: Choose an Algorithm" on page 3-6, 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 does not create a filter unless you specifically tell it to do so.

Step 6: 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, 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.

Step 7: 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)
```

To understand how the filtering commands work, type:

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

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:

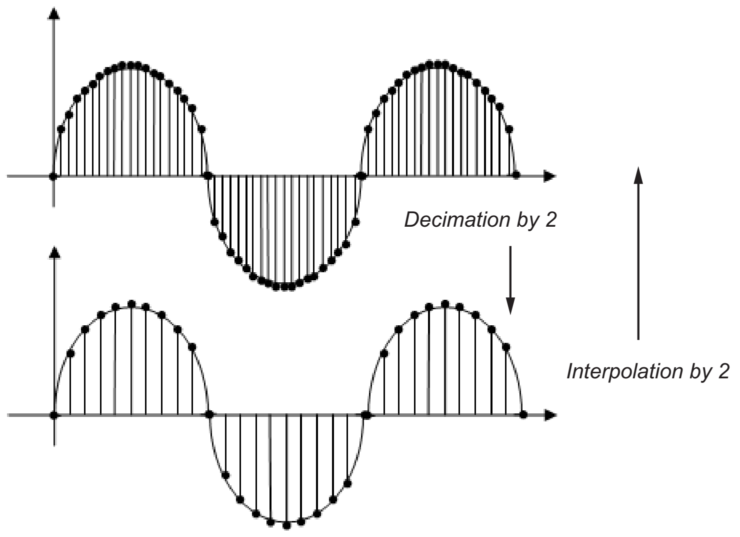
```
>> help realizemdl
```

Again, this step is never automatically performed for you by Filter Design Toolbox. To filter your data, you must explicitly execute this step.

Designing Multirate and Multistage Filters

What Is a Multirate Filter?

A *multirate filter* reduces or increases the input sample rate, resulting in an output rate different from the input rate. A high input frequency, while necessary in some cases, may be rather costly. The higher the frequency, the more samples need to be evaluated per unit time. Increased sampling causes a higher the load on the filter, and therefore, results in the higher the cost. Sometimes, systems are categorized by the maximum input frequency allowed. Hence, a typical use and reason for a multirate filter is to reduce the output frequency of one system to an acceptable value for input by another system. A filter that reduces the input rate is called a *decimator*. A filter that increases the input rate is called an *interpolator*. To visualize this process, review the following figure.



If you start with the top signal, sampled at `x` frequency, then the bottom signal is sampled at `x/2` frequency. In this case, the decimation factor, or `M`, is `2`. Interestingly enough, in decimation, the cost of the filter is also reduced by M. If decimation has these advantages, why not use it all the time? The following list describes restrictions and requirements for use with decimation:

- The sampling frequency divided by 2 must be greater than the system's highest frequency. For example, if you have a lowpass filter with the highest frequency of 10 MHz, and a sampling frequency of 60 MHz, the highest frequency that can be handled by the system without aliasing is $60/2 = 30$, which is greater than 10. You could safely set $M = 2$ in this case, since $(60/2)/2 = 15$, which is still greater than 10.

- If you wish to decimate a signal which does not meet the frequency criteria, you can either:
  - Interpolate first, and then decimate
  - Use a lowpass filter first to reduce the highest signal frequency, thereby allowing decimation without aliasing
- M must be an integer. Although, if you wish to obtain an M of 4/5, you could interpolate by 4, and then decimate by 5, provided that frequency restrictions are met.
- If you decimate in multiple stages, use the highest factors first. For example, if you want to decimate by a factor of 24 in 3 stages, use 4, then 3, then 2. This order brings the highest cost savings.

Multirate filters are most often used in stages. This technique is introduced in the following section.

What Is a Multistage Filter?

A *multistage filter* consists of several filters connected in series. Each filter in this series is called a *stage*.

There are many different uses for a multistage filter. One of these is a filter requirement that includes a very narrow transition width. For example, you need to design a lowpass filter where the difference between the pass frequency and the stop frequency is .01 (normalized). For such a requirement it is possible to design a single filter, but it will be very long (containing many coefficients) and very costly (having many multiplications and additions per input sample). Thus, this single filter may be so costly and require so much memory, that it may be impractical to implement in certain applications where there are strict hardware requirements. In such cases, a multistage filter is a great solution. Another application of a multistage filter is for a *multirate system*, where there is a decimator or an interpolator with a large factor. In these cases, it is usually wise to break up the filter into several multirate stages, each comprising a multiple of the total decimation/interpolation factor.

Typically a multistage filter is not easy to design by hand. It's difficult to guess how many stages would provide an optimal design, to optimize each stage, and then optimize all the stages together. Filter Design Toolbox enables you to create a Specifications Object, and then design a filter using multistage as an option. The rest of the work is done automatically. Not only does Filter Design Toolbox determine the optimal number of stages, but it also optimizes the total filter solution.

Designing a Multirate, Multistage Filter

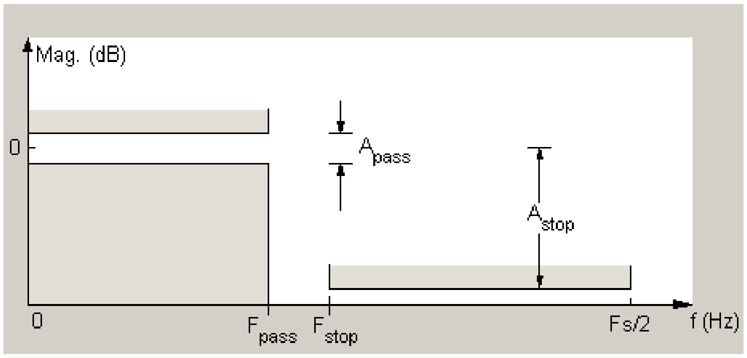
This example consists of the following steps:

1 "Design a Lowpass Filter to Use as a Baseline" on page 4-6

2 "Design a Multirate Filter to Improve Cost" on page 4-6

3 "Design a Multistage Filter to Improve Cost and Performance" on page 4-8

Typically, a lowpass filter is described as shown in the following diagram.



For this lowpass filter, the design parameters are:

- Fpass — The closing frequency of the passband
- Fstop — The opening frequency of the stopband
- Apass — The ripple allowed in the passband
- Astop — The minimum attenuation required in the stopband

In this example, you design a filter where the difference between Fpass and Fstop is very small, i.e., the filter transition width is very narrow. For such filters the multistage design is clearly beneficial.

Design a Lowpass Filter to Use as a Baseline

Start with a simple lowpass filter; this will later serve as the baseline for further comparisons. To design a lowpass filter, type or evaluate the following code:

```
>> Fpass = 0.11;
>> Fstop = 0.12;
>> Apass = 0.02;
>> Astop = 60;
>> SOlowpass = fdesign.lowpass(Fpass, Fstop, Apass, Astop);
>> Flowpass = design (SOlowpass, 'equiripple');
>> cost (Flowpass)

ans =

Number of Multipliers : 649
Number of Adders      : 648
Number of States      : 648
MultPerInputSample    : 649
AddPerInputSample     : 648
```

As shown in the code, this design requires 649 multiplications per input sample (MPIS). Depending on the application where this filter is to be used, this may not be feasible.

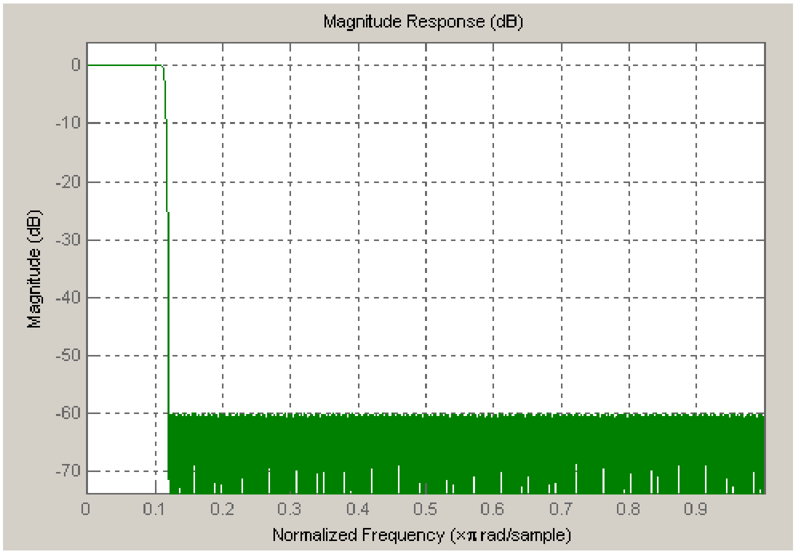
Design a Multirate Filter to Improve Cost

Looking at the filter specifications shows that the Fstop is very low, 0.12. This is the highest frequency of the system, normalized by Fs/2. According to the guidelines for decimation discussed earlier, decimation can be used here to reduce cost. The first step is to find the largest possible decimation factor, M. From decimation restrictions, you know that: $Fstop * M < Fs/2$. Since the system is already normalized by Fs/2, we can rewrite this formula as: $Fstop * M < 1$, where $\mathrm{Fstop} = 0.12$. So, the largest possible integer value of M would be 8, which would make $Fstop * M = .96$, which is still less than 1.

Using the design parameters for the baseline lowpass filter, design a multirate filter with M=8:

```
>> M = 8;
>> SOmultirate = fdesign.decimator(M, 'lowpass',
        Fpass, Fstop, Apass, Astop);
>> Fmultirate = design (SOmultirate, 'equiripple')
>> cost (Fmultirate)

ans =

Number of Multipliers : 649
Number of Adders      : 648
Number of States      : 648
MultPerInputSample    : 81.125
AddPerInputSample     : 81
```

The code shows that the cost has decreased from 649 MPIS to 81.125 MPIS. Plotting the response of the multirate filter together with the lowpass filter shows that they completely overlap each other.



Thus, for exactly the same performance, the cost of the filter is reduced by 8. Now, if you break up the decimation into several stages, the cost can be reduced even further. The next section shows how to further improve cost and performance.

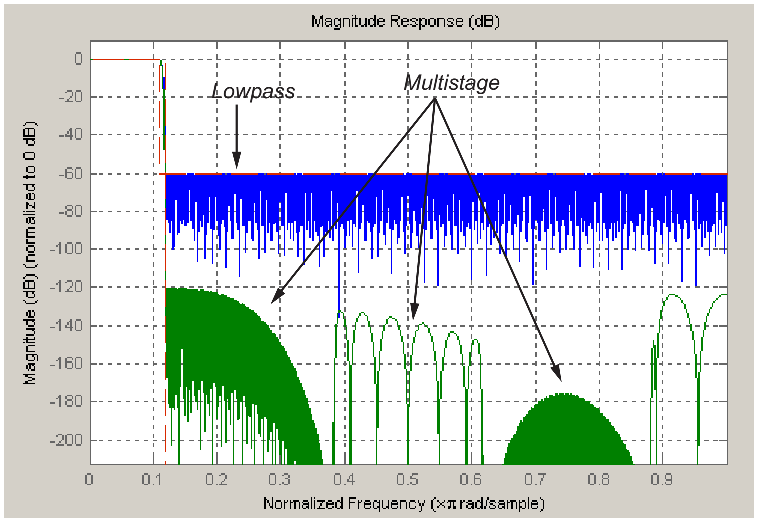
Design a Multistage Filter to Improve Cost and Performance

So far this example has shown that solving a narrow transition width problem with a multirate filter drastically reduces the cost of the filter. The decimation factor of 8 can be easily broken into several stages, thereby even further lowering the cost.

The typical problem at this point is to find the optimal structure, or number of stages. Is it better to have two stages, one with M=4, and the other with M=2, or is it better to have three stages, each with M=2? Only one line of code is necessary for Filter Design Toolbox to find the most optimal solution for the specified problem. To design this multirate, multistage filter, you continue working with the example from the preceding sections, using the filter Specifications Object created for the multirate filter:

```
>> Fmultistage = design(SOmultirate, 'multistage')

Fmultistage =

     FilterStructure: Cascade
            Stage(1): Direct-Form FIR Polyphase Decimator
            Stage(2): Direct-Form FIR Polyphase Decimator
            Stage(3): Direct-Form FIR Polyphase Decimator
    PersistentMemory: false

>> cost (Fmultistage)

ans =

Number of Multipliers : 195
Number of Adders      : 192
Number of States      : 190
MultPerInputSample    : 29.5
AddPerInputSample     : 28.625
```

The optimal filter now has three stages, each with an M=2, and the cost has been reduced from 649 MPIS to 29.5 MPIS — a significant savings! The following figure shows the performance of the regular lowpass filter and the multistage, multirate filter. In this figure, it is clear that not only the cost, but also the performance is improved.


Magnitude Response (dB)
Lowpass
Multistage
Magnitude (dB) (normalized to 0 dB)
0
-20
-40
-60
-80
-100
-120
-140
-160
-180
-200
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)

Converting from Floating-Point to Fixed-Point

What Is a Fixed-Point Filter?

A *fixed-point filter* is represented by an equation with fixed-point coefficients. To learn about fixed-point math, see “Fixed-Point Concepts” in “Fixed-Point Toolbox” documentation. The most common use of fixed-point filters is in the DSP chips, where the data storage capabilities are limited. For example, the data input may come from a 12 bit ADC, the data bus may be 16 bit, and the multiplier may have 24 bits. Within these space constraints, Filter Design Toolbox enables you to design the best possible fixed-point filter.

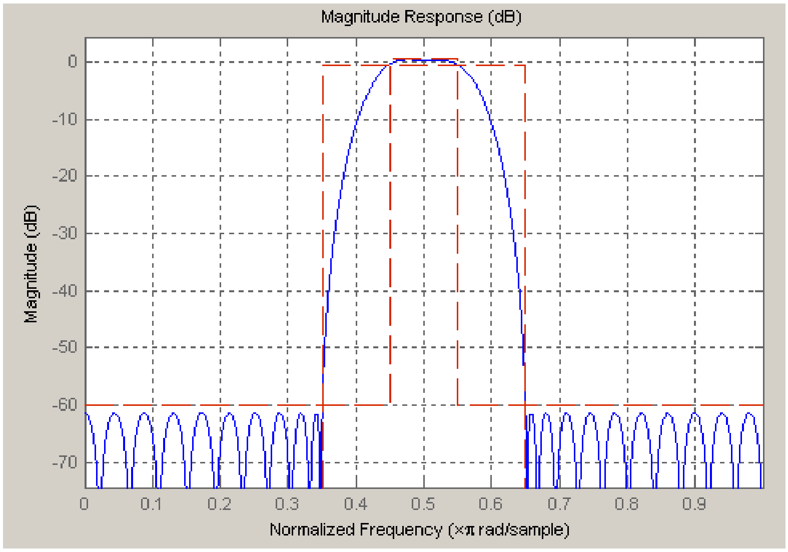
Floating-Point to Fixed-Point Conversion

The conversion from floating-point to fixed-point consists of two main parts: quantizing the coefficients and performing the dynamic range analysis. Quantizing the coefficients is a process of converting the coefficients to fixed-point numbers. The dynamic range analysis is a process of fine tuning the scaling of each node to ensure that the fraction lengths are set for full input range coverage and maximum precision. The following steps describe this conversion process. In the first step you design the floating-point filter to be converted.

1. "Designing the Filter" on page 5-3
2. "Quantizing the Coefficients" on page 5-4
3. "Performing Dynamic Range Analysis" on page 5-7

Designing the Filter

Start by designing a regular, floating-point, equiripple bandpass filter, as shown in the following figure.



where the passband is from .45 to .55 of normalized frequency, the amount of ripple acceptable in the passband is 1 dB, the first stopband is from 0 to .35 (normalized), the second stopband is from .65 to 1 (normalized), and both stopbands provide 60 dB of attenuation.

To design this filter, evaluate the following code, or type it at the MATLAB command prompt:

```
>> f = fdesign.bandpass(.35,.45,.55,.65,60,1,60);
>> Hd = design(f, 'equiripple');
>> fvtool(Hd)
```

The last line of code invokes the Filter Visualization Tool, which displays the designed filter. You use `Hd`, which is a double, floating-point filter, both as the baseline and a starting point for the conversion.

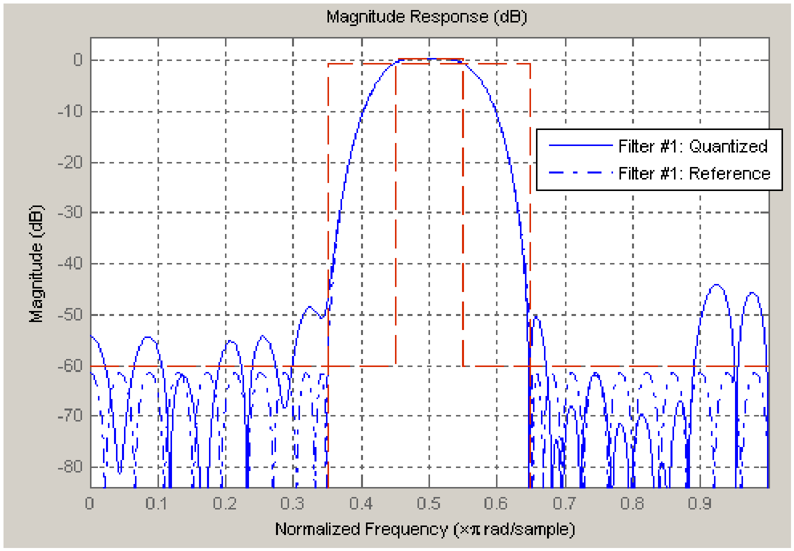
Quantizing the Coefficients

The first step in quantizing the coefficients is to find the valid word length for the coefficients. Here again, the hardware usually dictates the maximum allowable setting. However, if this constraint is large enough, there is room for some trial and error. Start with the coefficient word length of 8 and determine if the resulting filter is sufficient for your needs.

To set the coefficient word length of `8`, evaluate or type the following code at the MATLAB command prompt:

```
>> Hf = Hd;
>> Hf.Arithmetic = 'fixed';
>> set(Hf, 'CoeffWordLength', 8);
>> fvtool(Hf)
```

The resulting filter is shown in the following figure.

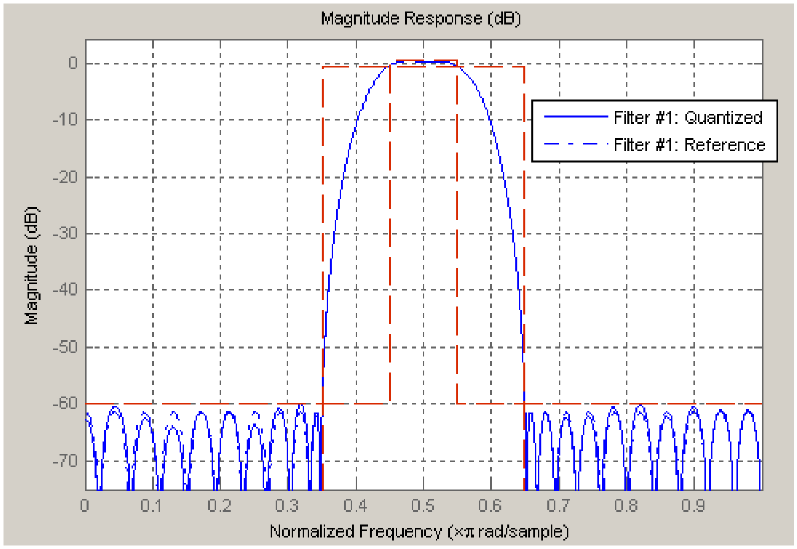


As the figure shows, the filter design constraints are not met. The attenuation is not complete, and there is noise at the edges of the stopbands. You can experiment with different coefficient word lengths if you like. For this example, however, the word length of 12 is sufficient.

To set the coefficient word length of 12, evaluate or type the following code at the MATLAB command prompt:

```
>> set(Hf, 'CoeffWordLength', 12);
>> fvtool(Hf)
```

The resulting filter satisfies the design constraints, as shown in the following figure.



Now that the coefficient word length is set, there are other data width constraints that might require attention. Type the following at the MATLAB command prompt:

```
>> get (Hf)
       PersistentMemory: 0
    NumSamplesProcessed: 0
        FilterStructure: 'Direct-Form FIR'
                 States: [47x1 embedded.fi]
              Numerator: [1x48 double]
             Arithmetic: 'fixed'
        CoeffWordLength: 12
         CoeffAutoScale: 1
                 Signed: 1
              RoundMode: 'convergent'
           OverflowMode: 'wrap'
        InputWordLength: 16
        InputFracLength: 15
          NumFracLength: 14
        FilterInternals: 'FullPrecision'
       OutputWordLength: 31
       OutputFracLength: 29
    ProductWordLength: 27
    ProductFracLength: 29
      AccumWordLength: 31
      AccumFracLength: 29
```

You see the output is 31 bits, the accumulator requires 31 bits and the multiplier requires 27 bits. A typical piece of hardware might have a 16 bit data bus, a 24 bit multiplier, and an accumulator with 4 guard bits. Another reasonable assumption is that the data comes from a 12 bit ADC. To reflect these constraints type or evaluate the following code:

```
>> set (Hf, 'InputWordLength', 12);
>> set (Hf, 'FilterInternals', 'SpecifyPrecision')
>> set (Hf, 'ProductWordLength', 24)
>> set (Hf, 'AccumWordLength', 28)
>> set (Hf, 'OutputWordLength', 16)
```

Although the filter is basically done, if you try to filter some data with it at this stage, you may get erroneous results due to overflows. Such overflows occur because you have defined the constraints, but you have not tuned the filter coefficients to handle properly the range of input data where the filter is designed to operate. Next, the dynamic range analysis is necessary to ensure no overflows.

Performing Dynamic Range Analysis

The purpose of the dynamic range analysis is to fine tune the scaling of the coefficients. The ideal set of coefficients is valid for the full range of input data, while the fraction lengths maximize precision. Consider carefully the range of input data to use for this step. If you provide data that covers the largest dynamic range in the filter, the resulting scaling is more conservative, and some precision is lost. If you provide data that covers a very narrow input range, the precision can be much greater, but an input out of the design range may produce an overflow. In this example, you use the worst-case input signal, covering a full dynamic range, in order to ensure that no overflow ever occurs. This worst-case input signal is a scaled version of the sign of the flipped impulse response.

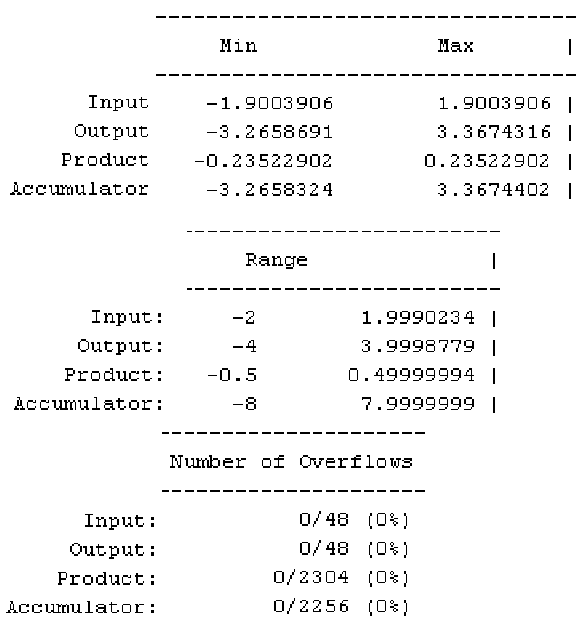
To scale the coefficients based on the full dynamic range, type or evaluate the following code:

```
>> x = 1.9*sign(fliplr(impz(Hf)));
>> Hf = autoscale(Hf, x);
```

To check that the coefficients are in range (no overflows) and have maximum possible precision, type or evaluate the following code:

```
>> fipref('LoggingMode', 'on', 'DataTypeOverride', 'ForceOff');
>> y = filter(Hf, x);
>> fipref('LoggingMode', 'off');
>> R = qreport(Hf)
```

Where `R` is shown in the following figure:



```
                ---------------------------------
                     Min              Max       |
                ---------------------------------
       Input       -1.9003906        1.9003906 |
      Output       -3.2658691        3.3674316 |
     Product      -0.23522902       0.23522902 |
 Accumulator       -3.2658324        3.3674402 |
                  -------------------------
                       Range                  |
                  -------------------------
       Input:       -2        1.9990234 |
      Output:       -4        3.9998779 |
     Product:     -0.5       0.49999994 |
 Accumulator:       -8        7.9999999 |
                 ---------------------
                  Number of Overflows
                 ---------------------
      Input:             0/48  (0%)
     Output:             0/48  (0%)
    Product:           0/2304  (0%)
Accumulator:           0/2256  (0%)
```

The report shows no overflows, and all data falls within the designed range. The conversion has completed successfully.

Data Types

Data Type Support

There are three different data types supported in Filter Design Toolbox:

- Fixed — Requires Fixed Point Toolbox and is supported by packages listed in "Fixed Data Type Support" on page 6-3.
- Double — Double precision, floating point and is the default data type for Filter Design Toolbox; accepted by all functions
- Single — Single precision, floating point and is supported by specific packages outlined in "Single Data Type Support" on page 6-4.

Fixed Data Type Support

To use fixed data type, you must have Fixed Point Toolbox. Type `ver` at the MATLAB command prompt to get a listing of all installed products.

The fixed data type is reserved for any filter whose property `arithmetic` is set to `fixed`. Furthermore all functions that work with this filter, whether in analysis or design, also accept and support the fixed data types.

To set the filter's arithmetic property:

```
>> f = fdesign.bandpass(.35,.45,.55,.65,60,1,60);
>> Hf = design(f, 'equiripple');
>> Hf.Arithmetic = 'fixed';
```

Single Data Type Support

The support of the single data types comes in two varieties. First, input data of type single can be fed into a double filter, where it is immediately converted to double. Thus, while the filter still operates in the double mode, the single data type input does not break it. The second variety is where the filter itself is set to single precision. In this case, it accepts only single data type input, performs all calculations, and outputs data in single precision. Furthermore, such analyses as `noisepsd` and `freqrespest` also operate in single precision.

To set the filter to single precision:

```
>> f = fdesign.bandpass(.35,.45,.55,.65,60,1,60);
>> Hf = design(f, 'equiripple');
>> Hf.Arithmetic = 'single';
```

Examples

Use this list to find examples in the documentation.

Getting Started

Using Filterbuilder

Index