Filter Design Toolbox™ 4
Getting Started Guide

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science
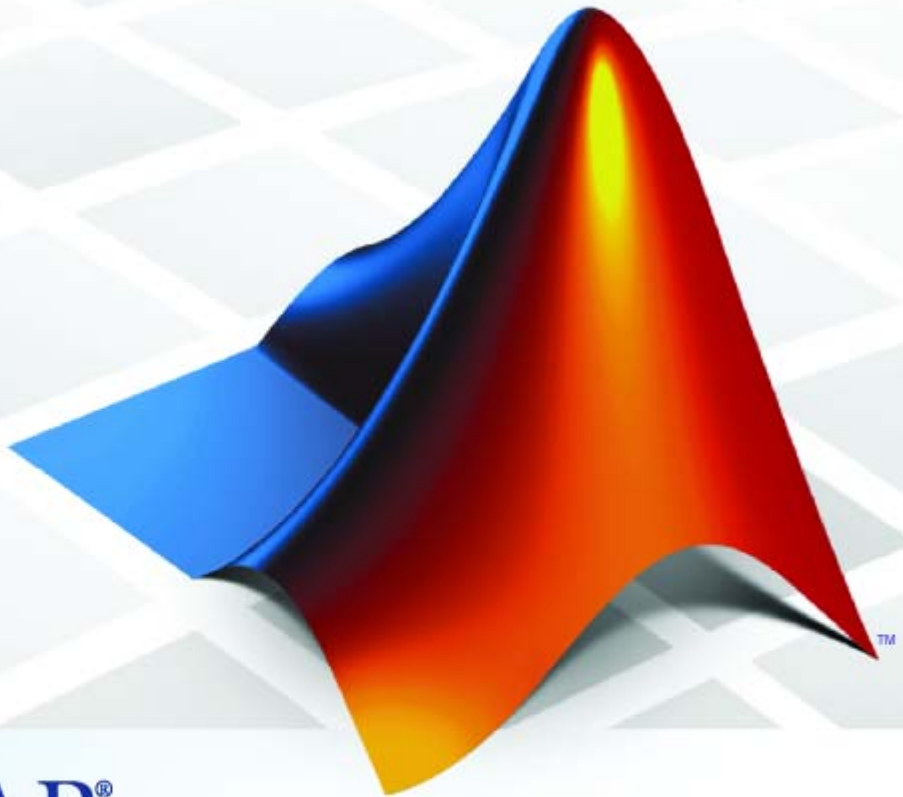
How to Contact The MathWorks

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™ Getting Started Guide

© COPYRIGHT 2000–2009 by The MathWorks, Inc.

Trademarks

Patents

Revision History

Contents

Converting from Floating-Point to Fixed-Point

4

Designing Adaptive Filters

5

Examples

A

Index

Product Overview

- "Introduction" on page 1-2
- "Uses with Other MathWorks Products" on page 1-3
- "Key Features" on page 1-3

Introduction

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

Uses with Other MathWorks Products

Used with Fixed-Point Toolbox™, Filter Design Toolbox software provides functions that simplify the design of fixed-point filters and the analysis of quantization effects. When used with Filter Design HDL Coder™ product, Filter Design Toolbox software lets you generate VHDL and Verilog code for fixed-point filters. Filter Design Toolbox software also brings advanced filter design capabilities to Simulink® and the Signal Processing Blockset™ software.

Key Features

- Advanced FIR filter design methods, including minimum-order, minimum-phase, shaped-stopband, halfband, complexity-optimized multistage, Farrow, and interpolated FIR

- Advanced IIR design methods, including arbitrary magnitude, group-delay equalizers, parametric equalizers, octave, halfband, quasi-linear phase, and comb filters

- Multirate filter design methods, including cascaded integrator-comb (CIC), CIC compensator, polyphase FIR and IIR, and multistage Nyquist filters

- Support for efficient IIR filter implementations, including second-order sections and lattice wave digital filters

- Adaptive filter design, analysis, and implementation, including LMS-based, RLS-based, lattice-based, frequency-domain, fast transversal, and affine projection

2

Filter Design with Fdesign and Filterbuilder

Filter Design Process Overview

> **Note** You must have the Signal Processing Toolbox installed to use and filterbuilder. Some of the features described and displayed in this tutorial may be unavailable if your installation does not additionally include the Filter Design Toolbox license. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

Filter design through user-defined specifications is the core of the `fdesign` approach. This specification-centric approach places less emphasis on the choice of specific filter algorithms, and more emphasis on performance during the design a good working filter. For example, you can take a given set of design parameters for the filter, such as a stopband frequency, a passband frequency, and a stopband attenuation, and— using these parameters— design a specification object for the filter. You can then implement the filter using this specification object. Using this approach, it is also possible to compare different algorithms as applied to a set of specifications.

There are two distinct objects involved in filter design:

• Specification Object — Captures the required design parameters of a filter

• Implementation Object — Describes the designed filter; includes the array of coefficients and the filter structure

The distinction between these two objects is at the core of the filter design methodology. The basic attributes of each of these objects are outlined in the following table.

Specification Object	Implementation Object
High-level specification	Filter coefficients
Algorithmic properties	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 theSignal Processing Toolbox. If you wish to access the full functionality of `fdesign` and `filterbuilder`, you should additionally obtain the Filter Design Toolbox software. You can verify the presence of these products by typing `ver` at the command prompt.

Basic Filter Design Process

Use the following two steps to design a simple filter.

1 Create a filter specification object.

2 Design your filter.

Example — Design a Filter in Two Steps

Assume that you want to design a bandpass filter. Typically a bandpass filter is defined as shown in the following figure.
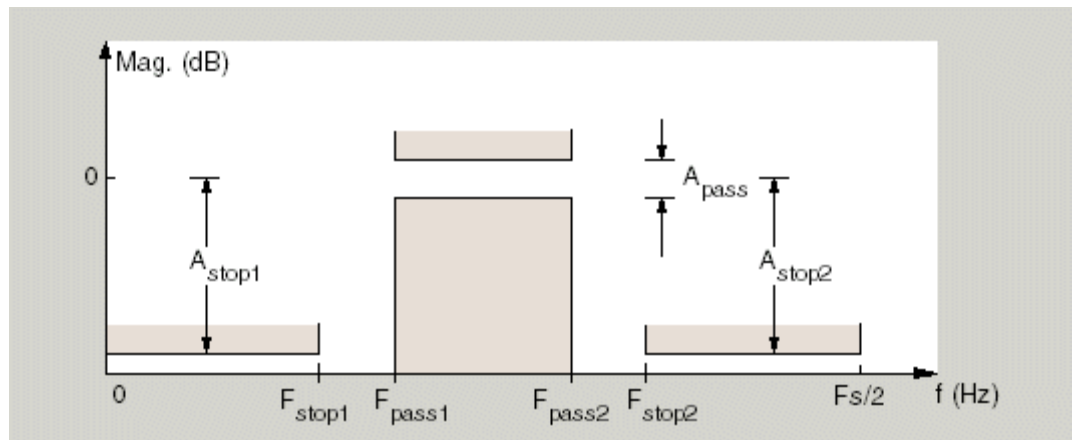
In this example, a sampling frequency of Fs = 48 kHz is used. This bandpass filter has the following specifications, specified here using MATLAB code:

```
A_stop1 = 60;  % Attenuation in the first stopband = 60 dB
F_stop1 = 8400;  % Edge of the stopband = 8400 Hz
F_pass1 = 10800; % Edge of the passband = 10800 Hz
F_pass2 = 15600; % Closing edge of the passband = 15600 Hz
F_stop2 = 18000; % Edge of the second stopband = 18000 Hz
A_stop2 = 60;  % Attenuation in the second stopband = 60 dB
A_pass = 1;  % Amount of ripple allowed in the passband = 1 dB
```

In the following two steps, these specifications are passed to the fdesign.bandpass method as parameters. For more information about

passing specifications to the `fdesign` filter specification object, refer to the reference.

Step 1

To create a filter specification object, evaluate the following code at the MATLAB prompt:

```
d = fdesign.bandpass
```

Now, pass the filter specifications that correspond to the default Specification — `fst1,fp1,fp2,fst2,ast1,ap,ast2`. This example adds `fs` as the final input argument to specify the sampling frequency of 48 kHz.

```
>> BandPassSpecObj = ...
   fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2', ...
  F_stop1, F_pass1, F_pass2, F_stop2, A_stop1, A_pass, ...
  A_stop2, 48000)

BandPassSpecObj =

               Response: 'Bandpass'
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
                 Fstop1: 8400
                 Fpass1: 10800
                 Fpass2: 15600
                 Fstop2: 18000
                 Astop1: 60
                  Apass: 1
                 Astop2: 60
```

Note The order of the filter is not specified, allowing a degree of freedom for the algorithm design in order to achieve the specification. The design will be a minimum order design.

The specification parameters, such as `Fstop1`, are all given default values when none are provided. You can change the values of the specification parameters after the filter specification object has been created. For example, if there are two values that need to be changed, `Fpass2` and `Fstop2`, use the `set` command, which takes the object first, and then the parameter value pairs. Evaluate the following code at the MATLAB prompt:

```
>> set(BandPassSpecObj, 'Fpass2', 15800, 'Fstop2', 18400)
```

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

You may also change parameter values in filter specification objects by accessing them as if they were elements in a `struct` array.

```
>> BandPassSpecObj.Fpass1=15800;
```

Step 2

Design the filter by using the `design` command. You can access the design methods available for you specification object by calling the `designmethods` function. For example, in this case, you can execute the command

```
>> designmethods(BandPassSpecObj)


Design Methods for class
fdesign.bandpass (Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2):


butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

After choosing a design method use, you can evaluate the following at the MATLAB prompt (this example assumes you've chosen 'equiripple'):

```
>> BandPassFilt = design(BandPassSpecObj, 'equiripple')

BandPassFilt =

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

Note If you do not specify a design method, a default method will be used. For example, you can execute the command

```
>> BandPassFilt = design(BandPassSpecObj)

BandPassFilt =

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

and a design method will be selected automatically.

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
```
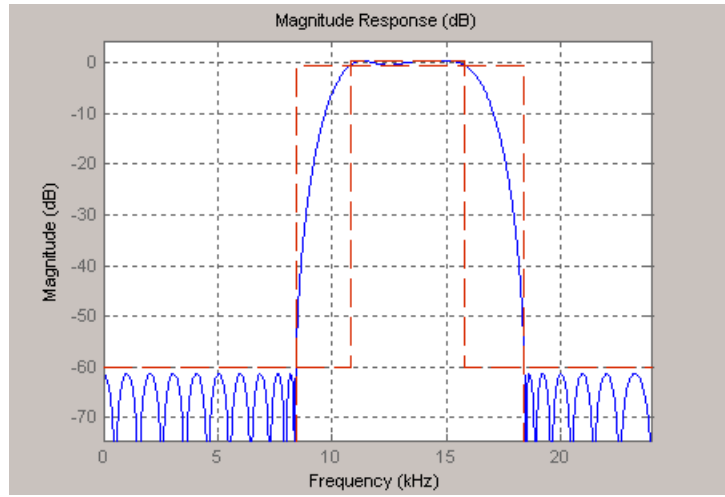
Using Filterbuilder to Design a Filter

Filterbuilder presents the option of designing a filter using a GUI dialog box as opposed to the command line instructions. You can use Filterbuilder to design the same bandpass filter designed in the previous section, "Basic Filter Design Process" on page 2-4

Example — Using Filterbuilder to Design a Simple Filter

To design the filter using FilterBuilder:

1 Type the following at the MATLAB prompt:

```
filterbuilder
```

The following dialog box opens:
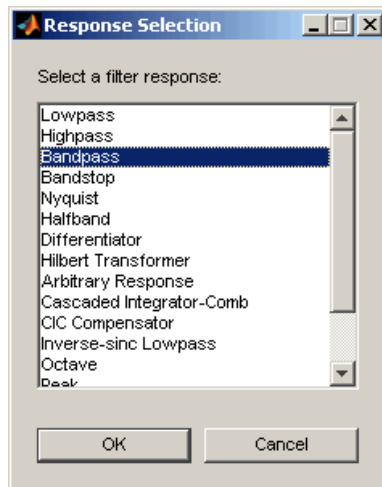
2 Select Bandpass filter response from the list in the dialog box, and hit the **OK** button. The following dialog box opens:
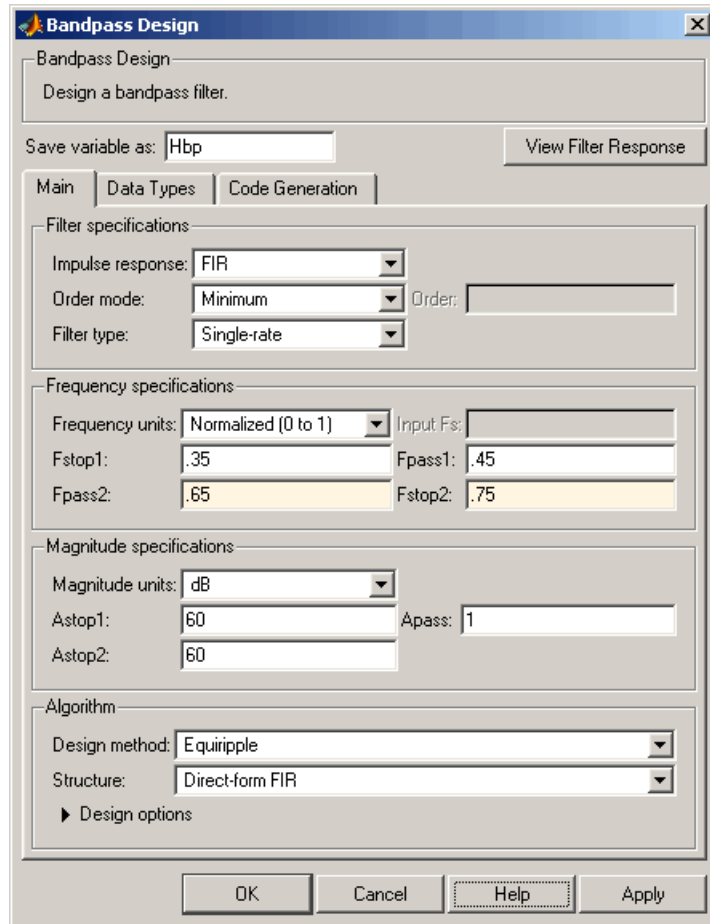
3 Enter the correct frequencies for **Fpass2** and **Fstop2**, as shown in the preceding figure, then click **OK**. Here the specification uses normalized frequency, so that the passband and stopband edges are expressed as a fraction of the Nyquist frequency (in this case, 48/2 kHz). The following message 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.
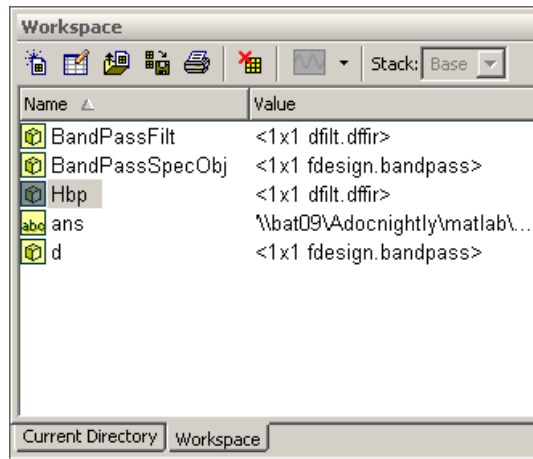
4 To check your work, plot the filter magnitude response using the Filter Visualization tool. Verify that all the design parameters are met:

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

3
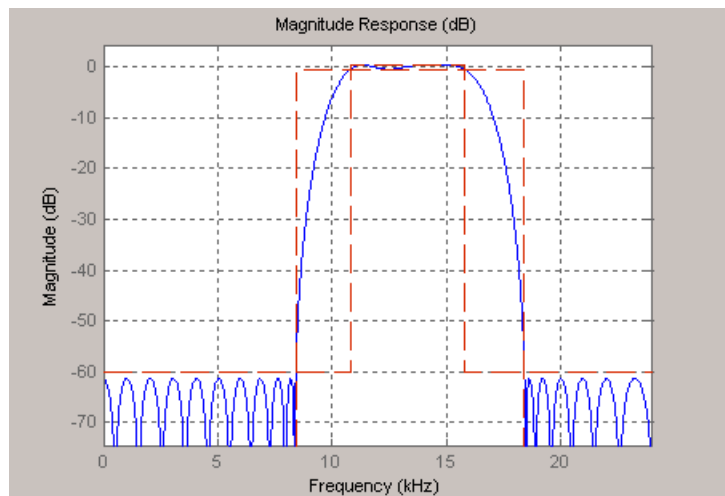
Designing Multirate and Multistage Filters

- "Multirate Filters" on page 3-2
- "Multistage Filters" on page 3-6
- "Example Case for Multirate/Multistage Filters" on page 3-8

Multirate Filters

In this section...
"Why Are Multirate Filters Needed?" on page 3-2
"Overview of Multirate Filters" on page 3-2

Why Are Multirate Filters Needed?

Multirate filters can bring efficiency to a particular filter implementation. In general, multirate filters are filters in which different parts of the filter operate at different rates. The most obvious application of such a filter is when the input sample rate and output sample rate need to differ (decimation or interpolation) — however, multirate filters are also often used in designs where this is not the case. For example you may have a system where the input sample rate and output sample rate are the same, but internally there is decimation and interpolation occurring in a series of filters, such that the final output of the system has the same sample rate as the input. Such a design may exhibit lower cost than could be achieved with a single-rate filter for various reasons. For more information about the relative cost benefit of using multirate filters, refer to [2] Harris, Fredric J., Multirate Signal Processing for Communication Systems, Prentice Hall PTR, 2004.

Overview of Multirate Filters

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, examine the following figure, which illustrates the processes of interpolation and decimation in the time domain.
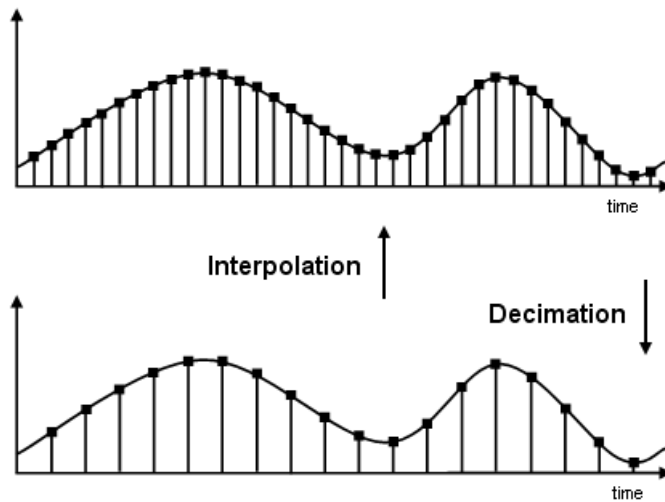
If you start with the top signal, sampled at a frequency `Fs`, then the bottom signal is sampled at `Fs/2` frequency. In this case, the decimation factor, or `M`, is `2`.

The following figure illustrates effect of decimation in the frequency domain.
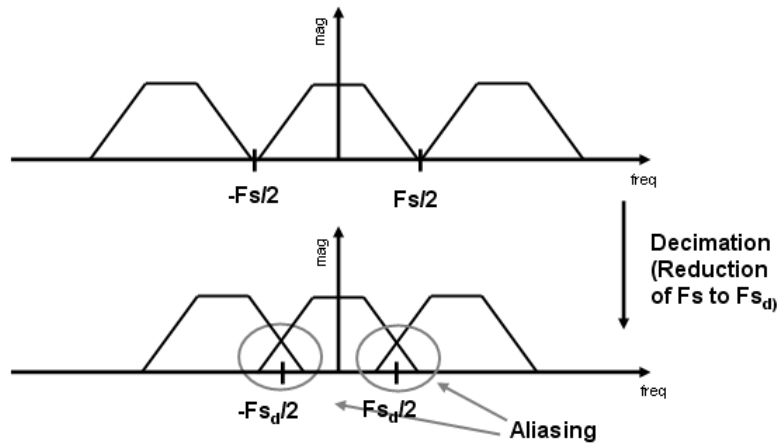
In the first graphic in the figure you can see a signal that is critically sampled, i.e. the sample rate is equal to two times the highest frequency component of the sampled signal. As such the period of the signal in the frequency domain is no greater than the bandwidth of the sampling frequency. When reduce the sampling frequency (decimation), *aliasing* can occur, where the magnitudes at the frequencies near the edges of the original period become indistinguishable, and the information about these values becomes lost. To work around this problem, the signal can be filtered before the decimation process, avoiding overlap of the signal spectra at Fs/2.
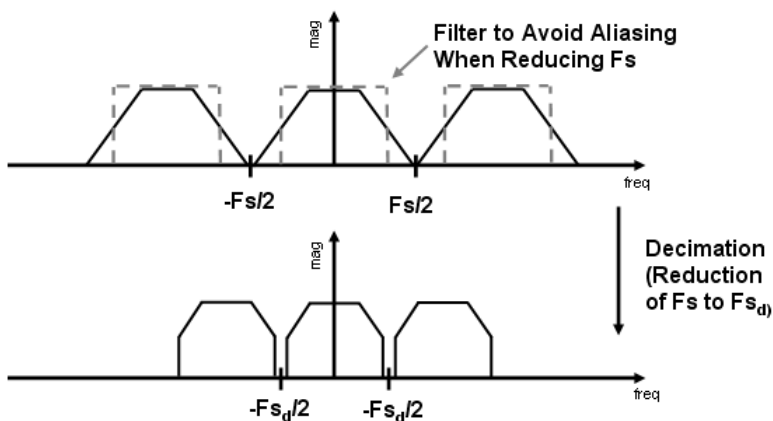
An analogous approach must be taken to avoid *imaging* when performing interpolation on a sampled signal. For more information about the effects of decimation and interpolation on a sampled signal, refer to any one of the references in the "Bibliography" section of the Filter Design Toolbox User Guide.

The following list summarizes some guidelines and general requirements regarding decimation and interpolation:

- By the Nyquist Theorem, for band-limited signals, the sampling frequency must be at least twice the bandwidth of the signal. 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

  - When decimating, you should apply the filter first, then perform the decimation. When interpolating a signal, you should interpolate first, then filter the signal.

- Typically in decimation of a signal a filter is applied first, thereby allowing decimation without aliasing, as shown in the following figure:
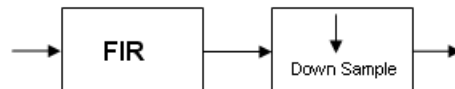
- Conversely, a filter is typically applied after interpolation to avoid imaging:
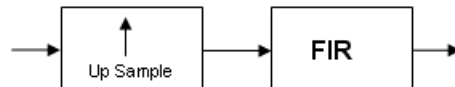
- M must be an integer. Although, if you wish to obtain an M of 4/5, you could interpolate by 4, and then decimate by 5, provided that frequency restrictions are met. This type of multirate filter will be referred to as a *sample rate converter* in the documentation that follows.

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

Multistage Filters

In this section...
"Why Are Multistage Filters Needed?" on page 3-6
"Optimal Multistage Filters in Filter Design Toolbox Software" on page 3-6

Why Are Multistage Filters Needed?

Typically used with multirate filters, *multistage filters* can bring efficiency to a particular filter implementation. Multistage filters are composed of several filters. These different parts of the mulitstage filter, called *stages*, are connected in a cascade or in parallel. However such a design can conserve resources in many cases. 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 mulitrate 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.

Optimal Multistage Filters in Filter Design Toolbox Software

As described in the previous section, within a multirate filter each interconnected filter is called a *stage*. While it is possible to design a multistage filter manually, it is also possible to perform automatic optimization of a multistage filter automatically. When designing a filter manually it can be difficult to guess how many stages would provide an optimal design, optimize each stage, and then optimize all the stages together. Filter Design Toolbox software 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 software determine the optimal number of stages, but it also optimizes the total filter solution.

Example Case for Multirate/Multistage Filters

In this section...

Example Overview

This brief example shows the efficiency gains that are possible when using multirate and multistage filters for certain applications. In this case a distinct advantage is achieved over regular linear-phase equiripple design when a narrow transition-band width is required. A more detailed treatment of the key points made here can be found in the demo entitled "Efficient Narrow Transition-Band FIR Filter Design".

Single-Rate/Single-Stage Equiripple Design

Consider the following design specifications for a lowpass filter (where the ripples are given in linear units):

```
Fpass = 0.13;   % Passband edge
Fstop = 0.14;   % Stopband edge
Rpass = 0.001;  % Passband ripple, 0.0174 dB peak to peak
Rstop = 0.0005; % Stopband ripple, 66.0206 dB minimum attenuation

Hf = fdesign.lowpass(Fpass,Fstop,Rpass,Rstop,'linear');
```

A regular linear-phase equiripple design using these specifications can be designed by evaluating the following:

```
Hd = design(Hf,'equiripple');
```

When you determine the cost of this design, you can see that 694 multipliers are required.

```
cost(Hd)

ans =

Number of Multipliers : 694
Number of Adders      : 693
Number of States      : 693
MultPerInputSample     : 694
AddPerInputSample     : 693
```

Reducing Computational Cost Using Mulitrate/Multistage Design

The number of multipliers required by a filter using a single state, single rate equiripple design is 694. This number can be reduced using multirate/multistage techniques. In any single-rate design, the number of multiplications required by each input sample is equal to the number of non-zero multipliers in the implementation. However, by using a multirate/multistage design, decimation and interpolation can be combined to lessen the computation required. For decimators, the average number of multiplications required per input sample is given by the number of multipliers divided by the decimation factor.

```
Hd_multi = design(Hf,'multistage');
```

You can then view the cost of the filter created using this design step, and you can see that a significant cost advantage has been achieved.

```
>> cost(Hd_multi)

ans =

Number of Multipliers : 392
Number of Adders      : 383
Number of States      : 348
MultPerInputSample     : 72.3333
AddPerInputSample     : 70.1667
```

Comparing the Response

You can compare the responses of the equiripple design and this multirate/multistage design using fvtool:

```
hfvt = fvtool(Hd,Hd_multi);
legend(hfvt,'Equiripple design', 'Multirate/multistage design')
```
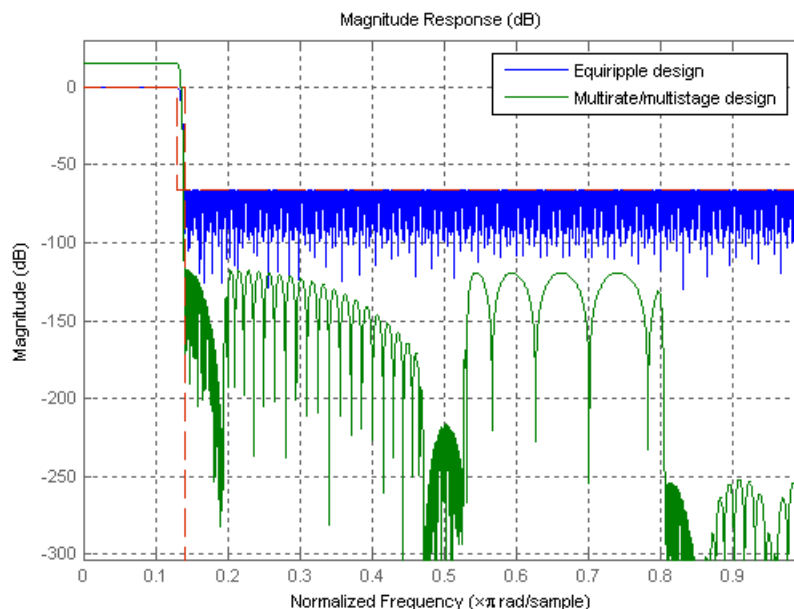

Notice that the stopband attenuation for the multistage design is about twice that of the other designs. This is because the decimators must attenuate out-of-band components by 66 dB in order to avoid aliasing that would violate the specifications. Similarly, the interpolators must attenuate images by 66 dB. You can also see that the passband gain for this design is no longer 0 dB, because each interpolator has a nominal gain (in linear units) equal to its interpolation factor, and the total interpolation factor for the three interpolators is 6.

Further Performance Comparison

You can check the performance of the multirate/multistage design by plotting the power spectral densities of the input and the various outputs, and you can

see that the sinusoid at 0.4π is attenuated comparably by both the equiripple design and the multirate/multistage design.

```
n       = 0:1799;
x       = sin(0.1*pi*n') + 2*sin(0.15*pi*n');
y       = filter(Hd,x);
y_multi = filter(Hd_multi,x);
[Pxx,w] = periodogram(x);
Pyy     = periodogram(y);
Pyy_multi = periodogram(y_multi);
plot(w/pi,10*log10([Pxx,Pyy,Pyy_multi]));
xlabel('Normalized Frequency (x\pi rad/sample)');
ylabel('Power density (dB/rad/sample)');
legend('Input signal PSD','Equiripple output PSD',...
    'Multirate/multistage output PSD')
axis([O 1 -50 30])
grid on
```

4
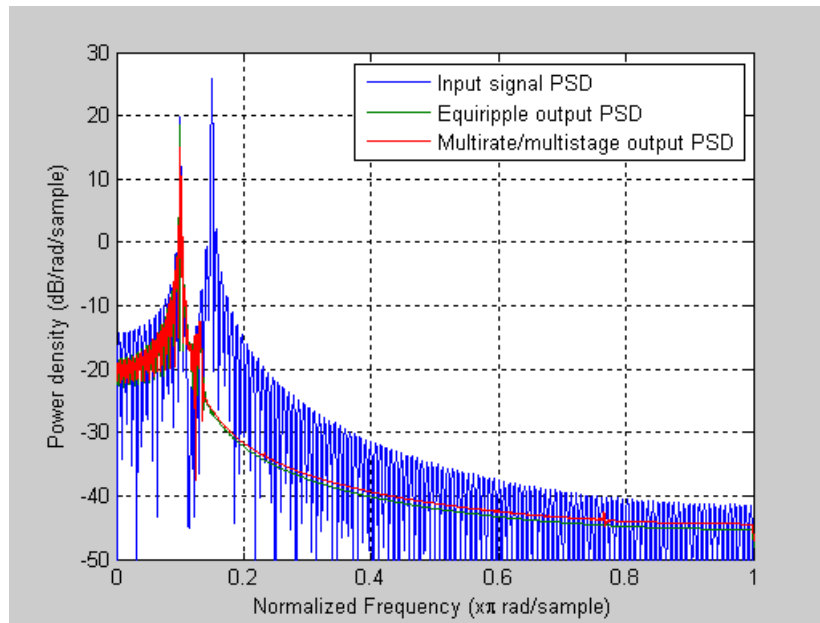
Converting from Floating-Point to Fixed-Point

Overview of Fixed-Point Filters

The most common use of fixed-point filters is in the DSP chips, where the data storage capabilities are limited, or embedded systems and devices where low-power consumption is necessary. 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 software enables you to design the best possible fixed-point filter.

What Is a Fixed-Point Filter?

A *fixed-point filter* uses fixed-point arithmetic and is represented by an equation with fixed-point coefficients. To learn about fixed-point math, see "Fixed-Point Concepts" in "Fixed-Point Toolbox" documentation.

Floating-Point to Fixed-Point Conversion

Process Overview

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.

Designing the Filter

Start by designing a regular, floating-point, equiripple bandpass filter, as shown in the following figure.
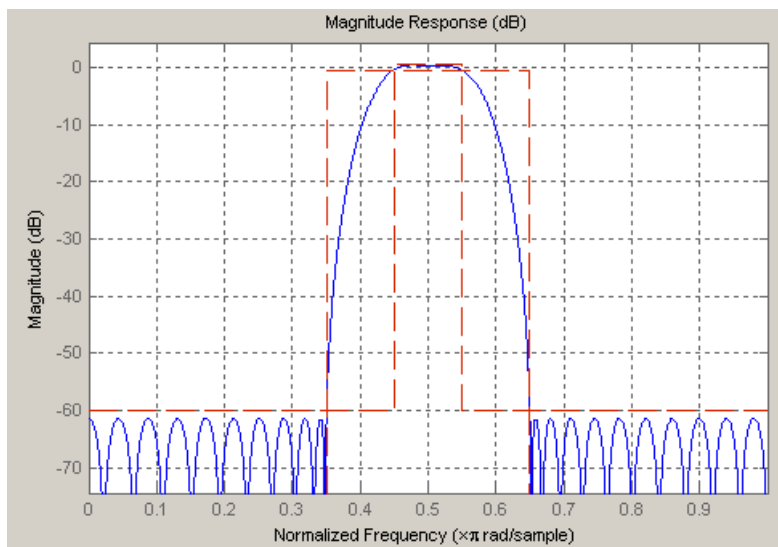
where the passband is from .45 to .55 of normalized frequency, the amount of ripple acceptable in the passband is 1 dB, the first stopband is from 0 to .35 (normalized), the second stopband is from .65 to 1 (normalized), and both stopbands provide 60 dB of attenuation.

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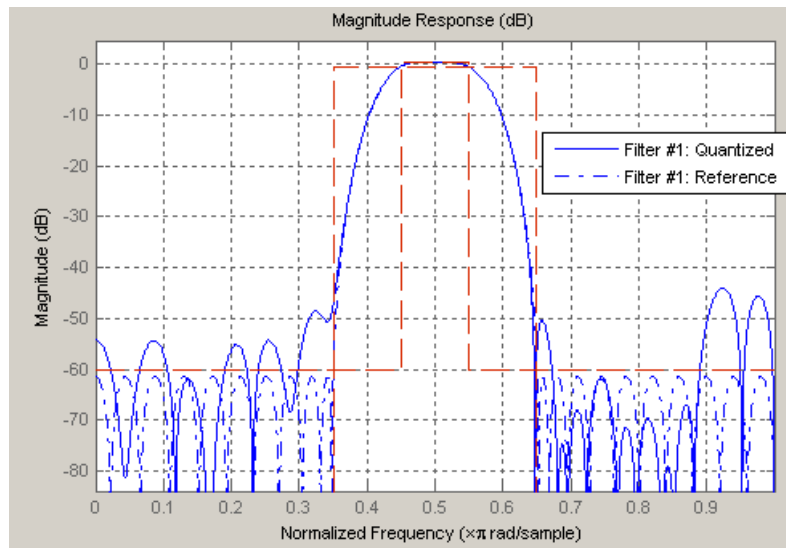
As the figure shows, the filter design constraints are not met. The attenuation is not complete, and there is noise at the edges of the stopbands. You can experiment with different coefficient word lengths if you like. For this example, however, the word length of 12 is sufficient.

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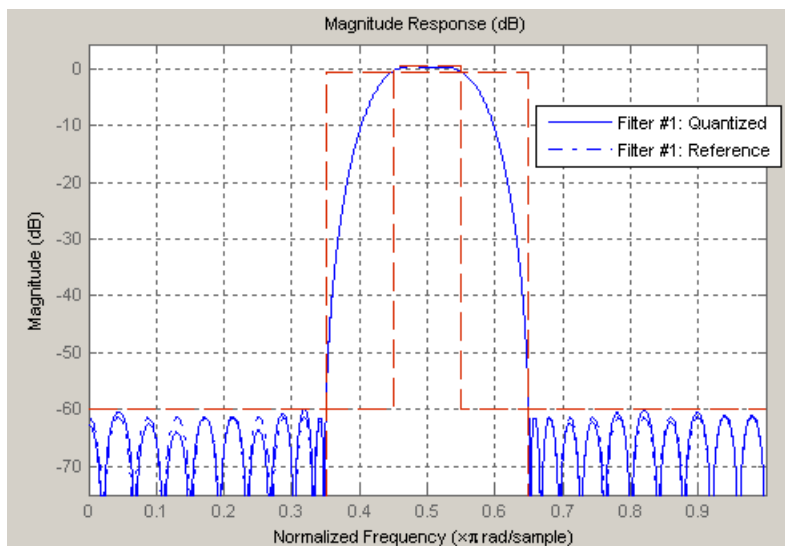
Now that the coefficient word length is set, there are other data width constraints that might require attention. Type the following at the MATLAB command prompt:

```
>> info(Hf)
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 48
Stable            : Yes
Linear Phase      : Yes (Type 2)
Arithmetic        : fixed
Numerator         : s12,14 -> [-1.250000e-001 1.250000e-001)
Input             : s16,15 -> [-1 1)
Filter Internals  : Full Precision
  Output          : s31,29 -> [-2 2)  (auto determined)
  Product         : s27,29 -> [-1.250000e-001 1.250000e-001)...
        (auto determined)
  Accumulator     : s31,29 -> [-2 2)  (auto determined)
  Round Mode      : No rounding
  Overflow Mode   : No overflow
```

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.

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.

Comparing Magnitude Response and Magnitude Response Estimate

You can use the fvtool GUI to do final analysis on your quantized filter, to see the effects of the quantization on stopband attenuation, etc. Two important last checks when analyzing a quantized filter are the Magnitude Response Estimate and the Round-off Noise Power Spectrum. The value of the Magnitude Response Estimate analysis can be seen in the following example.

Viewing Magnitude Response Estimate

Begin by designing a simple lowpass filter using the command.

```
h = design(fdesign.lowpass, 'butter');
```
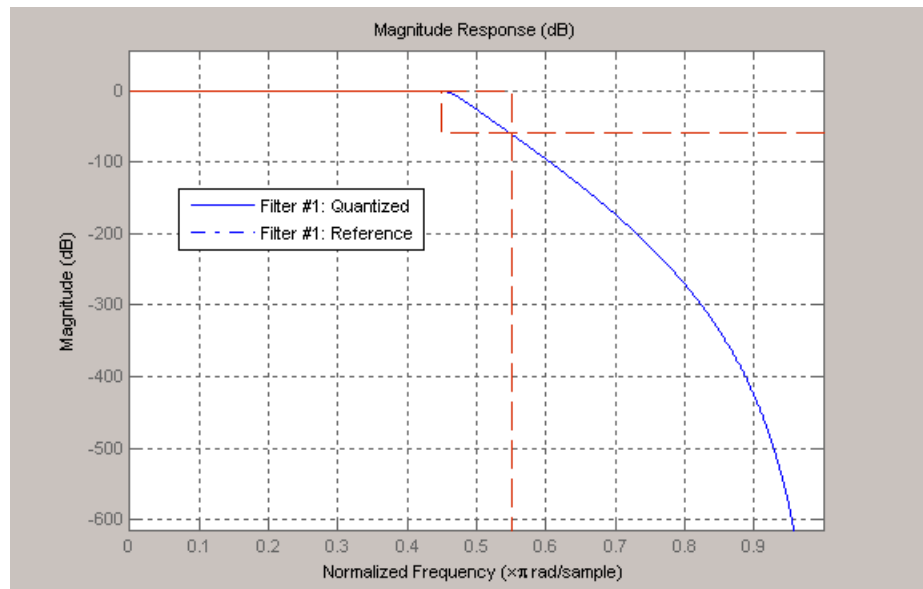
Now set the arithmetic to fixed-point.

```
h.arithmetic = 'fixed';
```

Open the filter using fvtool.

```
fvtool(h)
```

When fvtool displays the filter using the **Magnitude response** view, the quantized filter seems to match the original filter quite well.

However if you look at the **Magnitude Response Estimate** plot from the **Analysis** menu, you will see that the actual filter created may not perform nearly as well as indicated by the **Magnitude Response** plot.

Magnitude Response Estimate
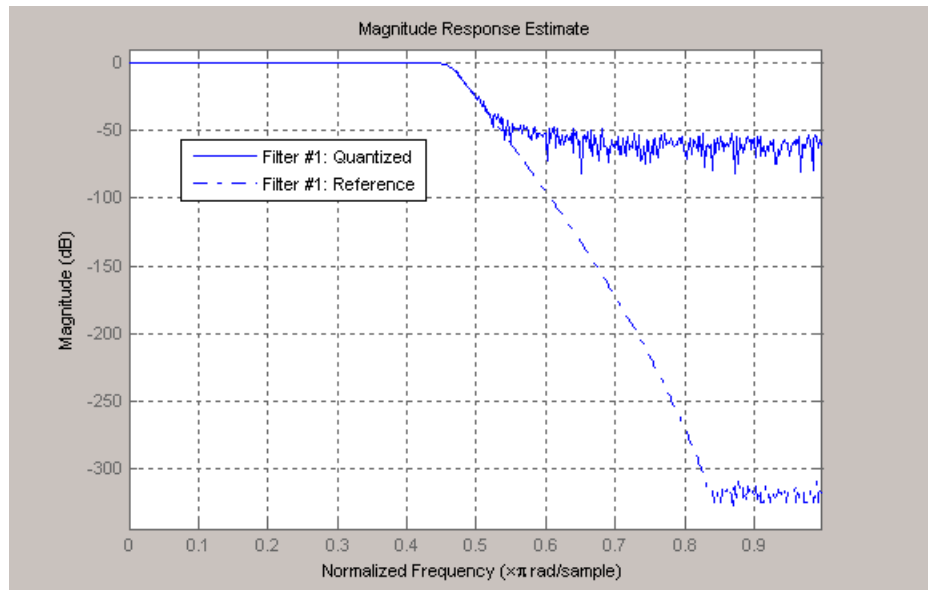
This is because by using the noise-based method of the **Magnitude Response Estimate**, you estimate the complex frequency response for your filter as determined by applying a noise- like signal to the filter input. **Magnitude Response Estimate** uses the Monte Carlo trials to generate a noise signal that contains complete frequency content across the range 0 to Fs. For more information about analyzing filters in this way, refer to the section titled Analyzing Filters with a Noise-Based Method in the User Guide.

For more information, refer to McClellan, et al., Computer-Based Exercises for Signal Processing Using MATLAB 5, Prentice-Hall, 1998. See Project 5: Quantization Noise in Digital Filters, page 231.

Data Types

Data Type Support

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

- Fixed — Requires Fixed Point Toolbox and is supported by packages listed in "Fixed Data Type Support" on page 4-11.
- Double — Double precision, floating point and is the default data type for Filter Design Toolbox software; accepted by all functions
- Single — Single precision, floating point and is supported by specific packages outlined in "Single Data Type Support" on page 4-11.

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';
```

Designing Adaptive Filters

Adaptive Filters Tutorial

Signal Enhancement Example Overview

This demonstration illustrates one way to use a few of the adaptive filter algorithms provided in the toolbox. In this example, a signal enhancement application is used as an illustration. While there are about 30 different adaptive filtering algorithms included with the toolbox, this example demonstrates two algorithms — least means square (LMS), `adaptfilt.lms`, and normalized LMS, `adaptfilt.nlms`, for adaptation.

Create the Signals for Adaptation

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

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

To perform adaptation requires two signals:

- a reference signal

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

Generate the Noise Signal

To create a noise signal, assume that the noise v1 is autoregressive, meaning that the value of the noise at time t depends only on its previous values and on a random disturbance.

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

Corrupt the Desired Signal to Create a Noisy Signal

To generate the noisy signal that contains both the desired signal and the noise, add the noise signal v1 to the desired signal s. The noise-corrupted sinusoid x is

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

where s is the desired signal and the noise is v1. Adaptive filter processing seeks to recover s from x by removing v1. To complete the signals needed to perform adaptive filtering, the adaptation process requires a reference signal.

Create a Reference Signal

Define a moving average signal v2 that is correlated with v1. This v2 is the reference signal for the examples.

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

Construct Two Adaptive Filters

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

```
L = 7;
hlms = adaptfilt.lms(7);
hnlms = adaptfilt.nlms(7);
```

Choose the Step Size

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

- A step size that is too small increases the time for the filter to converge on a set of coefficients. This becomes an issue of speed and accuracy.

- One that is too large may cause the adapting filter to diverge, never reaching convergence. In this case, the issue is stability — the resulting filter might not be stable.

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

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

```
>> [mumaxlms,mumaxmselms]  = maxstep(hlms,x)
[mumaxnlms,mumaxmsenlms] = maxstep(hnlms);
Warning: Step size is not in the range 0 < mu < mumaxmse/2:
Erratic behavior might result.
> In adaptfilt.lms.maxstep at 32

mumaxlms =

    0.2096


mumaxmselms =

    0.1261
```

Set the Adapting Filter Step Size

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

```
hlms.StepSize  = mumaxmselms/30;
% This can also be set graphically: inspect(hlms)
hnlms.StepSize = mumaxmsenlms/20;
% This can also be set graphically: inspect(hnlms)
```

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

```
hlms = adaptfilt.lms(N,step); Adds the step input argument.
```

Filter with the Adaptive Filters

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

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

Since v2 is correlated only with the noise component v1 of x, it can only really emulate v1. The error signal (the desired x), minus the actual output y, constitutes an estimate of the part of x that is not correlated with v2 — s, the signal to extract from x.

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

## Compute the Optimal Solution

For comparison, compute the optimal FIR Wiener filter.

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

## Plot the Results

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

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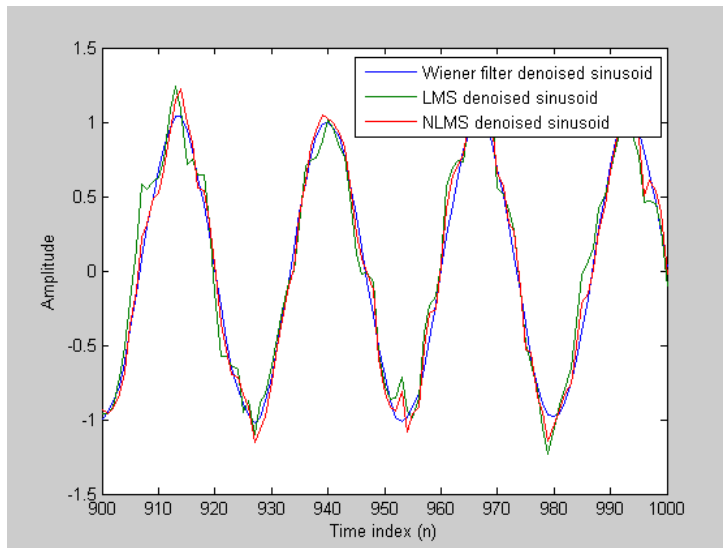

As a reference point, include the noisy signal as a dotted line in the plot.

```
hold on
plot(n(900:end),x(900:end),'k:')
xlabel('Time index (n)');
ylabel('Amplitude');
hold off
```
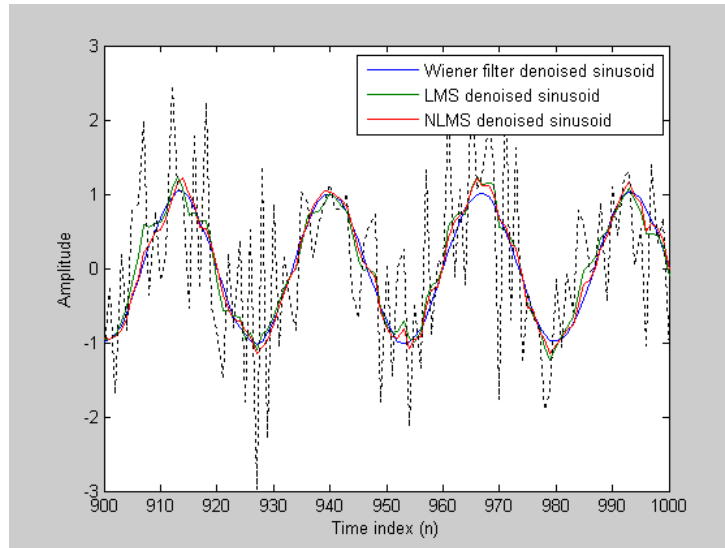
Compare the Final Coefficients

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

```
[bw.' hlms.Coefficients.' hnlms.Coefficients.']

ans =

    1.0317    0.8879    1.0712
    0.3555    0.1359    0.4070
    0.1500    0.0036    0.1539
    0.0848    0.0023    0.0549
    0.1624    0.0810    0.1098
    0.1079    0.0184    0.0521
    0.0492   -0.0001    0.0041
```

Reset the Filter Before Filtering

Adaptive filters have a `PersistentMemory` property that you can use to reproduce experiments exactly. By default, the `PersistentMemory` is `false`.

The states and the coefficients of the filter are reset before filtering and the filter does not remember the results from previous times you use the filter.

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

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

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

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

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

Investigate Convergence Through Learning Curves

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

This demonstration uses 25 sample realizations of the noisy sinusoids.

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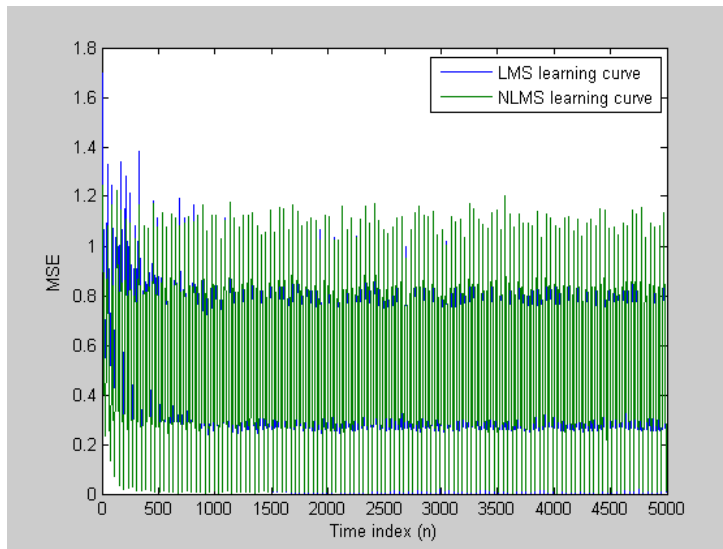
Compute the Learning Curves

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

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

```
reset(hlms);
reset(hnlms);
M = 10; % Decimation factor
mselms = msesim(hlms,v2,x,M);
msenlms = msesim(hnlms,v2,x,M);
plot(1:M:n(end),[mselms,msenlms])
legend('LMS learning curve','NLMS learning curve')
xlabel('Time index (n)');
ylabel('MSE');
```

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

Compute the Theoretical Learning Curves

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

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

```
reset(hlms);
[mmselms,emselms,meanwlms,pmselms] = msepred(hlms,v2,x,M);
plot(1:M:n(end),[mmselms*ones(500,1),emselms*ones(500,1),...
        pmselms,mselms])
legend('MMSE','EMSE','predicted LMS learning curve',...
    'LMS learning curve')
xlabel('Time index (n)');
ylabel('MSE');
```
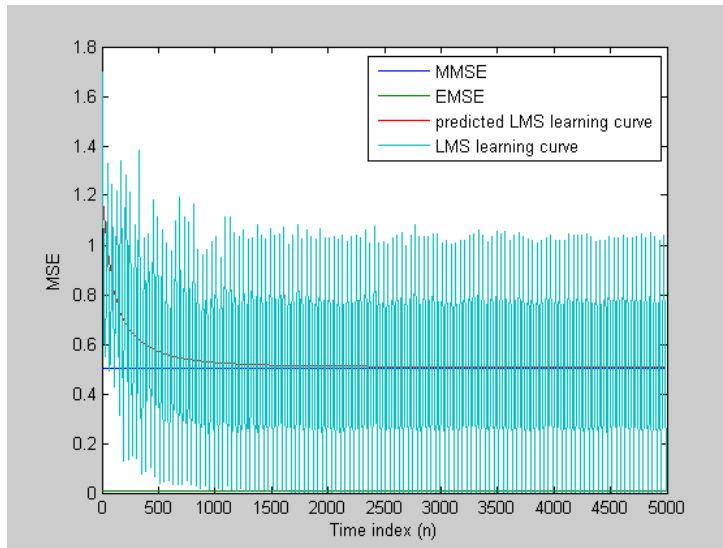
Examples

Use this list to find examples in the documentation.

Getting Started

Using Filterbuilder

Index