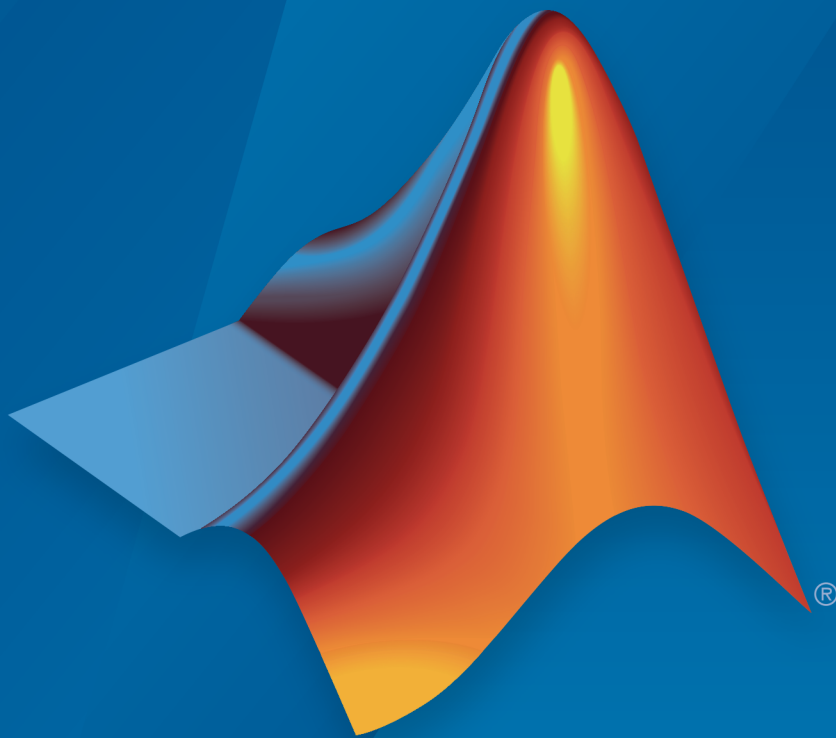# DSP System Toolbox™

User's Guide

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Revision History

April 2011	First printing	Revised for Version 8.0 (R2011a)
September 2011	Online only	Revised for Version 8.1 (R2011b)
March 2012	Online only	Revised for Version 8.2 (R2012a)
September 2012	Online only	Revised for Version 8.3 (R2012b)
March 2013	Online only	Revised for Version 8.4 (R2013a)
September 2013	Online only	Revised for Version 8.5 (R2013b)
March 2014	Online only	Revised for Version 8.6 (R2014a)
October 2014	Online only	Revised for Version 8.7 (R2014b)
March 2015	Online only	Revised for Version 9.0 (R2015a)
September 2015	Online only	Revised for Version 9.1 (R2015b)
March 2016	Online only	Revised for Version 9.2 (R2016a)

Contents

Input, Output, and Display

2

Filter Analysis, Design, and Implementation

4

Adaptive Filters

5

Multirate and Multistage Filters

6

Transforms, Estimation, and Spectral Analysis

7

Mathematics

8

Fixed-Point Design

9

C Code Generation

10

Define New System Objects

11

HDL Code Generation

12

Links to Category Pages

13

Designing Lowpass FIR Filters

14

FDATool: A Filter Design and Analysis GUI

15

Designing a Filter in the Filterbuilder GUI

16

17

Bibliography

DSP Tutorials

Introduction to Streaming Signal Processing in MATLAB

This example shows how to use System objects to do streaming signal processing in MATLAB. The signals are read in and processed frame by frame (or block by block) in each processing loop. You can control the size of each frame.

In this example, frames of 1024 samples are filtered using a notch-peak filter in each processing loop. The input is a sine wave signal that is streamed frame by frame from a `dsp.SineWave` object. The filter is a notch-peak filter created using a `dsp.NotchPeakFilter` object. To ensure smooth processing as each frame is filtered, the System objects maintain the state of the filter from one frame to the next automatically.

Initialize Streaming Components

Initialize the sine wave source to generate the sine wave, the notch-peak filter to filter the sine wave, and the spectrum analer to show the filtered signal. The input sine wave has two frequencies: one at 100 Hz, and the other at 1000 Hz. Create two `dsp.SineWave` objects, one to generate the 100 Hz sine wave, and the other to generate the 1000 Hz sine wave.

```
Fs = 2500;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,...
                    'SampleRate',Fs,'Frequency',100);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
                    'SampleRate',Fs,'Frequency',1000);

SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'NumInputPorts',2,...
    'PlotAsTwoSidedSpectrum',false,...
    'ChannelNames',{'SinewaveInput','NotchOutput'},'ShowLegend',true);
```

Create Notch-Peak Filter

Create a second-order IIR notch-peak filter to filter the sine wave signal. The filter has a notch at 750 Hz and a Q-factor of 35. A higher Q-factor results in a narrower 3-dB bandwidth of the notch. If you tune the filter parameters during streaming, you can see the effect immediately in the spectrum analyzer output.

```
Wo = 750;
Q  = 35;
BW = Wo/Q;
NotchFilter = dsp.NotchPeakFilter('Bandwidth',BW,...
    'CenterFrequency',Wo, 'SampleRate',Fs);
```

```
fvtool(NotchFilter);
```

Magnitude Response (dB)
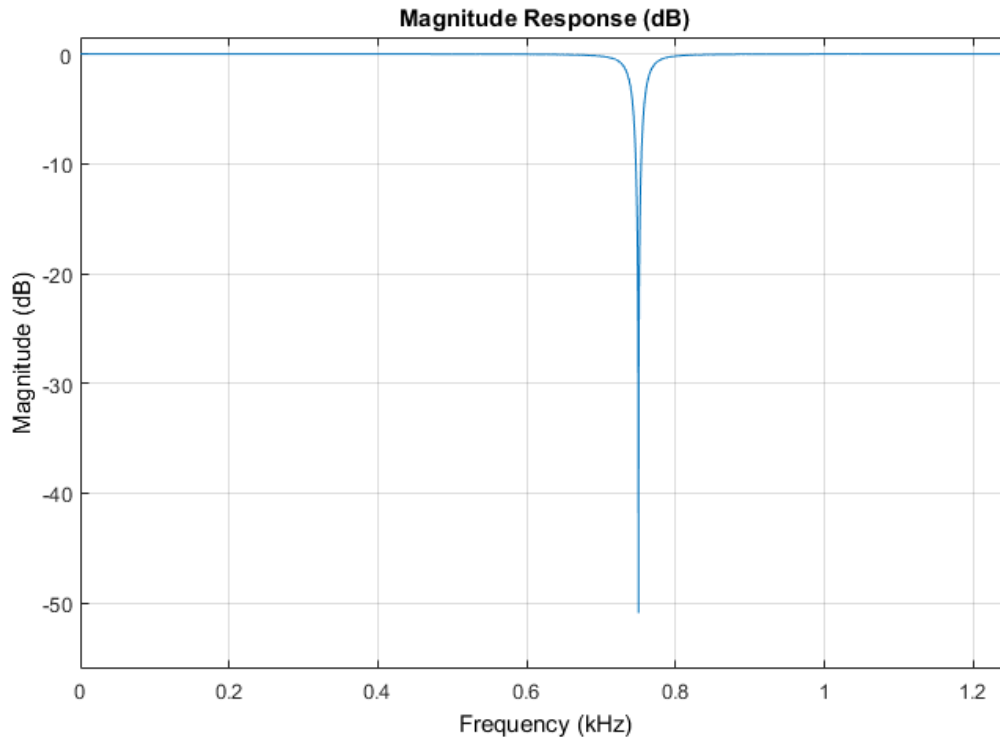
Stream In and Process Signal

Construct a for-loop to run for 3000 iterations. In each iteration, stream in 1024 samples (one frame) of the sinewave and apply a notch filter on each frame of the input signal. To generate the input signal, add the two sine waves. The resultant signal is a sine wave with two frequencies: one at 100 Hz and the other at 1000 Hz. The notch of the filter is tuned to a frequency of 100, 500, 750, or 1000 Hz, based on the value of `VecIndex`. The filter bandwidth changes accordingly. When the filter parameters change during streaming, the output in the spectrum analyzer gets updated accordingly.

```
FreqVec = [100 500 750 1000];
VecIndex = 1;
VecElem = FreqVec(VecIndex);
for Iter = 1:3000
    Sinewave1 = step(Sineobject1);
```

```
        Sinewave2 = step(Sineobject2);
        Input = Sinewave1 + Sinewave2;
        if (mod(Iter,350)==0)
            if VecIndex < 4
                VecIndex = VecIndex+1;
            else
                VecIndex = 1;
            end
            VecElem = FreqVec(VecIndex);
        end
        NotchFilter.CenterFrequency = VecElem;
        NotchFilter.Bandwidth = NotchFilter.CenterFrequency/Q;
        Output = step(NotchFilter,Input);
        step(SA,Input,Output);
    end
    fvtool(NotchFilter)
```
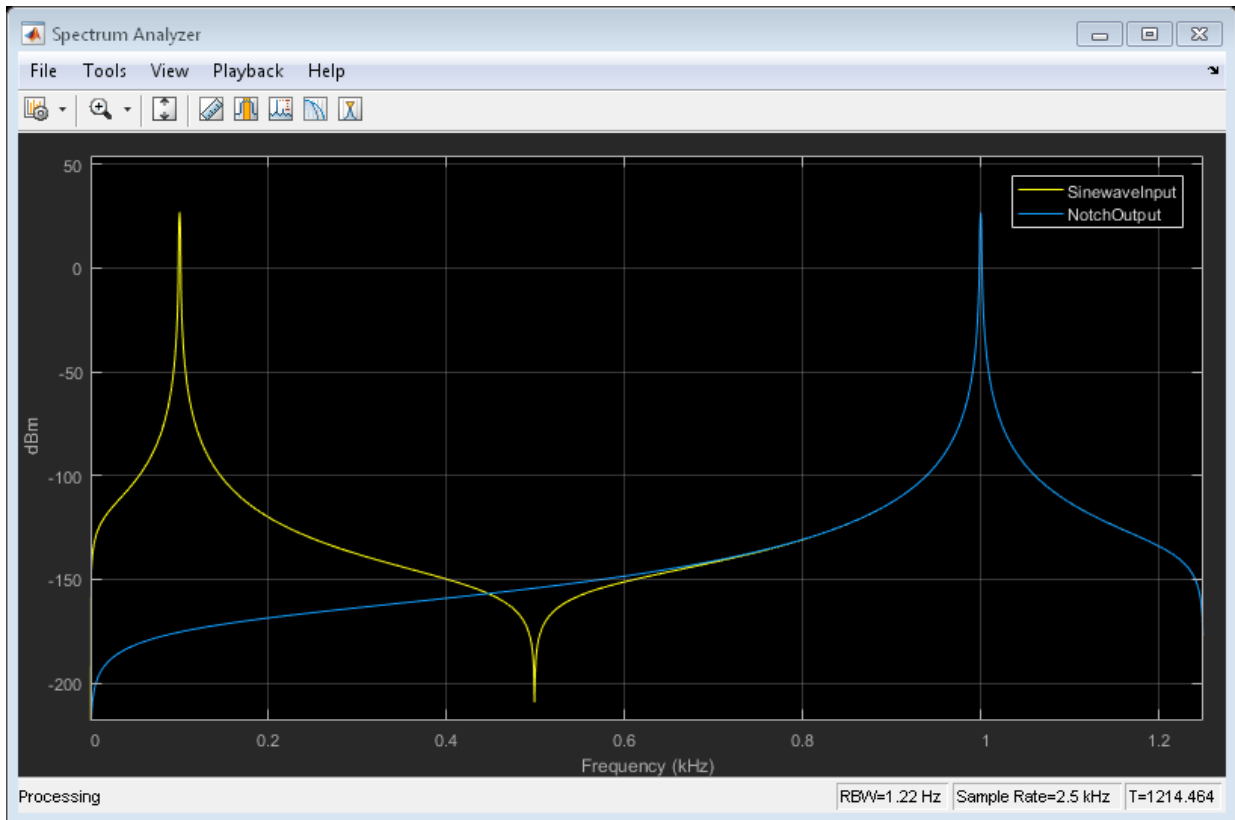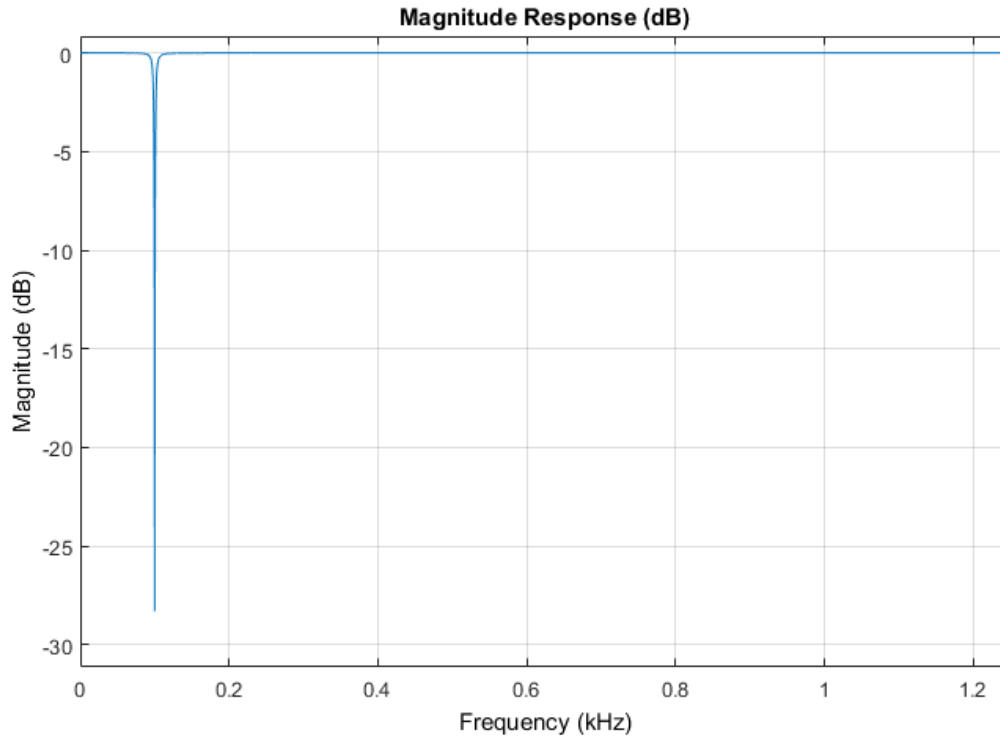
At the end of the processing loop, the `CenterFrequency` is at 100 Hz. In the filter output, the 100 Hz frequency is completely nulled out by the notch filter, while the frequency at 1000 Hz is unaffected.

See Also

"Filter Frames of a Noisy Sine Wave Signal in MATLAB" on page 1-7 | "Filter Frames of a Noisy Sine Wave Signal in Simulink" on page 1-10 | "Lowpass IIR Filter Design in Simulink" on page 1-28 | "Design Multirate Filters" on page 1-46

Filter Frames of a Noisy Sine Wave Signal in MATLAB

This example shows how to lowpass filter a noisy signal in MATLAB and visualize the original and filtered signals using a spectrum analyzer. For a Simulink version of this example, see Filter Frames of a Noisy Sine Wave Signal in Simulink

Specify Signal Source

The input signal is the sum of two sine waves with frequencies of 1 kHz and 10 kHz. The sampling frequency is 44.1 kHz.

```
Sine1 = dsp.SineWave('Frequency',1e3,'SampleRate',44.1e3);
Sine2 = dsp.SineWave('Frequency',10e3,'SampleRate',44.1e3);
```

Create Lowpass Filter

The lowpass FIR filter, `dsp.LowpassFilter`, designs a minimum-order FIR lowpass filter using the generalized Remez FIR filter design algorithm. Set the passband frequency to 5000 Hz and the stopband frequency to 8000 Hz. The passband ripple is 0.1 dB and the stopband attenuation is 80 dB.

```
FIRLowPass = dsp.LowpassFilter('PassbandFrequency',5000,...
    'StopbandFrequency',8000);
```

Create Spectrum Analyzer

Set up the spectrum analyzer to compare the power spectra of the original and filtered signals. The spectrum units are dBm.

```
SpecAna = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum',false, ...
    'SampleRate',Sine1.SampleRate, ...
    'NumInputPorts',2,...
    'ShowLegend',true, ...
    'YLimits',[-145,45]);

SpecAna.ChannelNames = {'Original noisy signal','Low pass filtered signal'};
```

Specify Samples per Frame

This example uses frame-based processing, where data is processed one frame at a time. Each frame of data contains sequential samples from an independent channel. Frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing

multisample frames of data, you can improve the computational time of your signal processing algorithms. Set the number of samples per frame to 4000.

```
Sine1.SamplesPerFrame = 4000;
Sine2.SamplesPerFrame = 4000;
```

Filter the Noisy Sine Wave Signal

Add zero-mean white Gaussian noise with a standard deviation of 0.1 to the sum of sine waves. Filter the result using the FIR filter. While running the simulation, the spectrum analyzer shows that frequencies above 8000 Hz in the source signal are attenuated. The resulting signal maintains the peak at 1 kHz because it falls in the passband of the lowpass filter.

```
for i = 1 : 1000
    x = step(Sine1)+step(Sine2)+0.1.*randn(Sine1.SamplesPerFrame,1);
    y = step(FIRLowPass,x);
    step(SpecAna,x,y);
end
release(SpecAna)
```
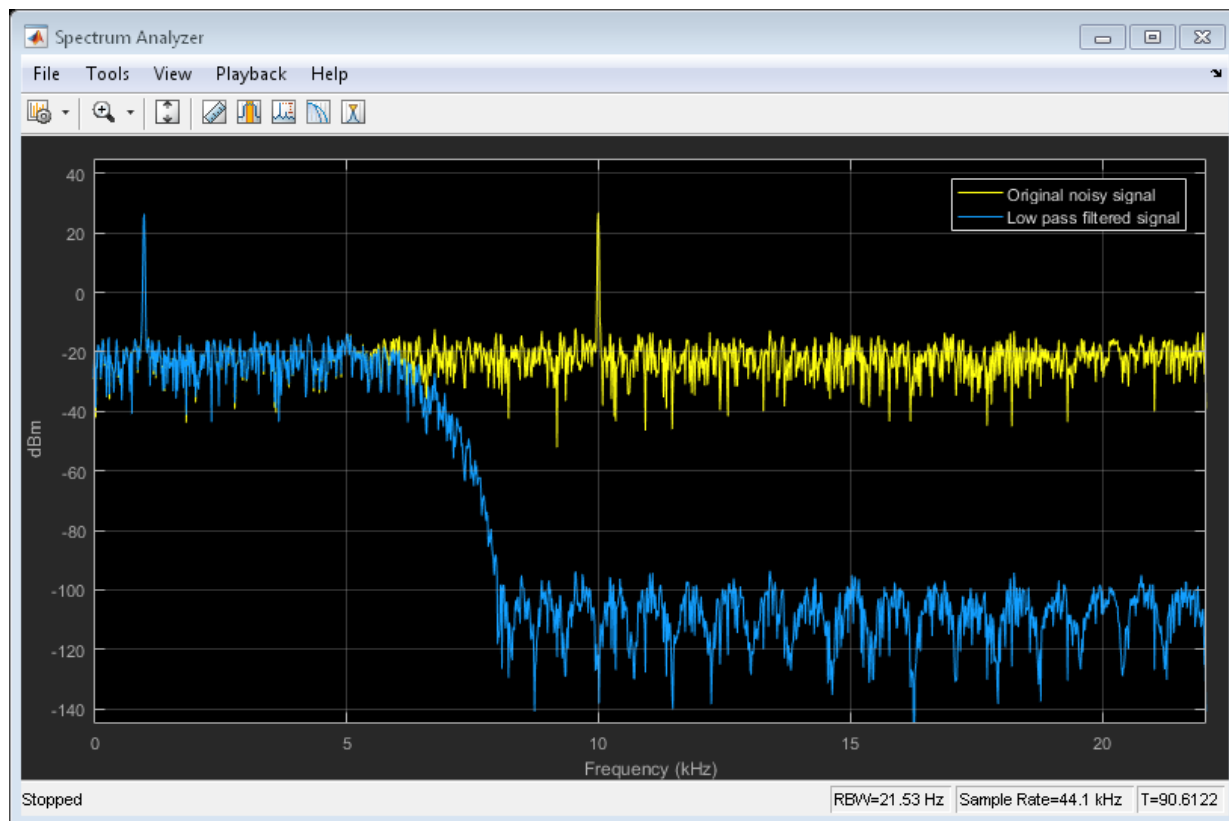
See Also

"Lowpass Filter Design in MATLAB" on page 1-17 | "Filter Frames of a Noisy Sine Wave Signal in Simulink" on page 1-10 | "Introduction to Streaming Signal Processing in MATLAB" on page 1-2 | "Design Multirate Filters" on page 1-46

Filter Frames of a Noisy Sine Wave Signal in Simulink

This example shows how to lowpass filter a noisy signal in Simulink® and visualize the original and filtered signals with a spectrum analyzer. For a MATLAB® version of this example, see "Filter Frames of a Noisy Sine Wave Signal in MATLAB" on page 1-7.

In this section...
"Open Model" on page 1-10
"Inspect Model" on page 1-11
"Compare Original and Filtered Signal" on page 1-14

Open Model

To create a new blank model and open the library browser:

1 On the MATLAB **Home** tab, click **Simulink**, and choose the **Basic Filter** model template.

2 Click **Create Model** to create a basic filter model opens with settings suitable for use with DSP System Toolbox™. To access the library browser, click the **Library Browser** button on the model toolbar.
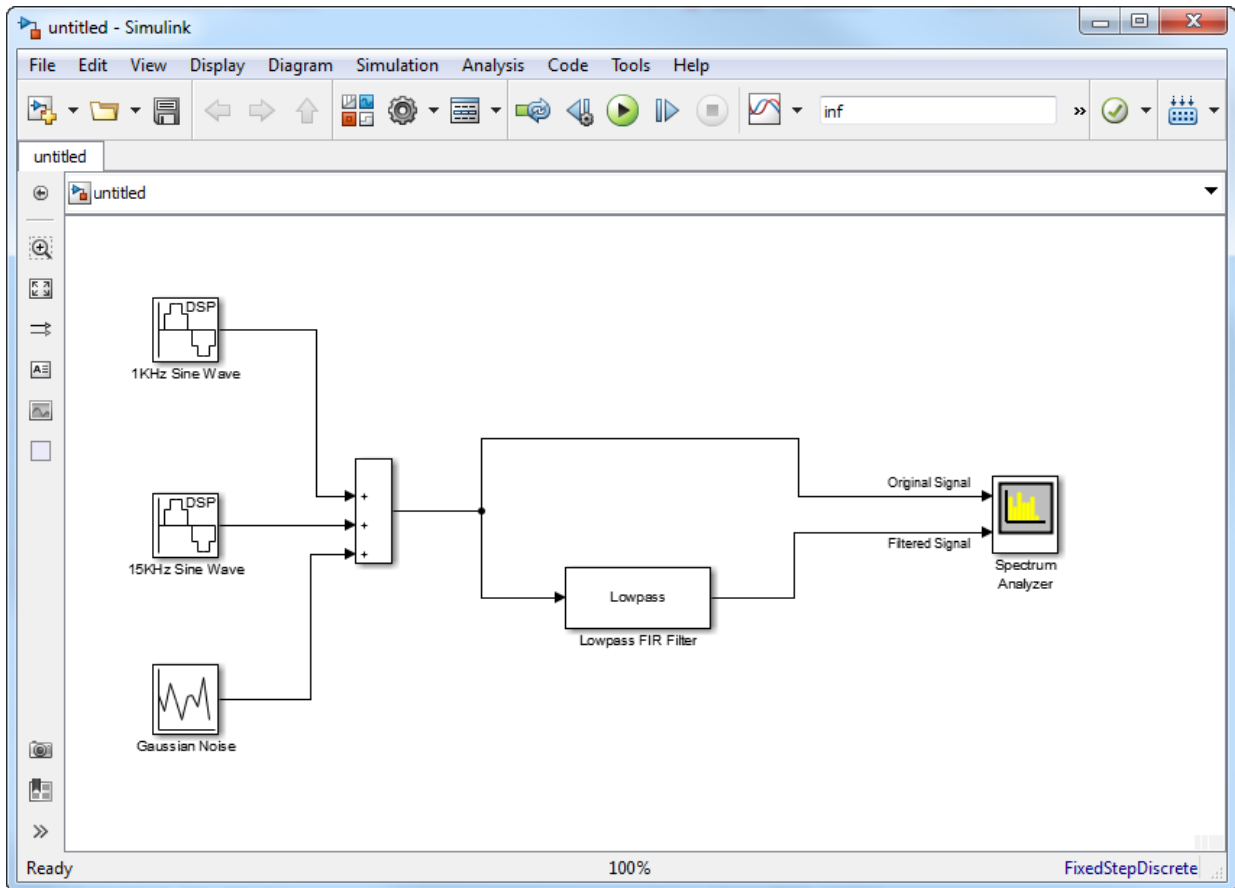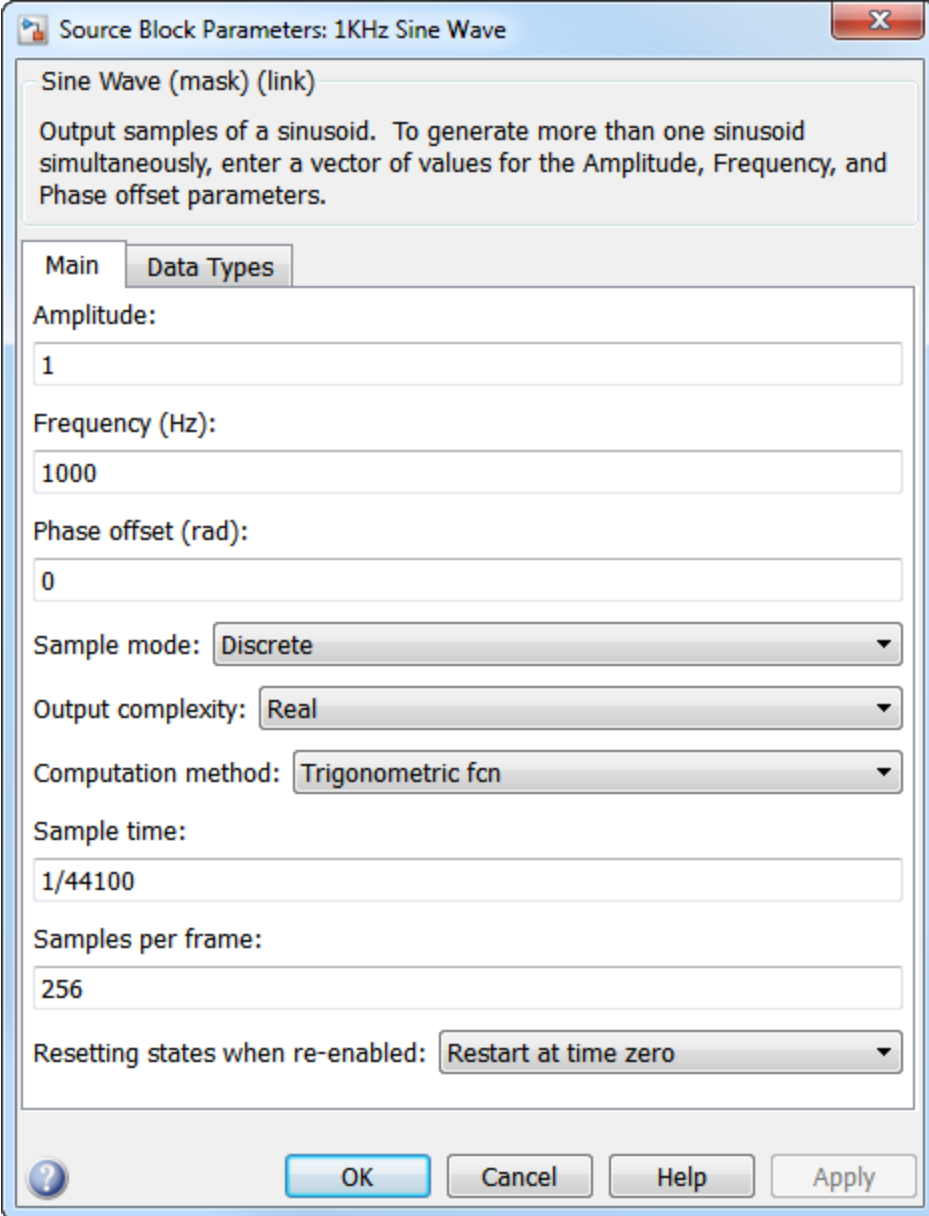
The new model using the template settings and contents appears in the Simulink Editor. The model is only in memory until you save it.

Inspect Model

Input Signal

Three source blocks comprise the input signal. The input signal consists of the sum of two sine waves and white Gaussian noise with mean 0 and variance 0.05. The frequencies of the sine waves are 1 kHz and 15 kHz. The sampling frequency is 44.1 kHz. The dialog box shows the block parameters for the 1 kHz sine wave.

Source Block Parameters: 1KHz Sine Wave

Sine Wave (mask) (link)

Output samples of a sinusoid. To generate more than one sinusoid simultaneously, enter a vector of values for the Amplitude, Frequency, and Phase offset parameters.

Main | Data Types

Amplitude:

1

Frequency (Hz):

1000

Phase offset (rad):

0

Sample mode: Discrete

Output complexity: Real

Computation method: Trigonometric fcn

Sample time:

1/44100

Samples per frame:

256

Resetting states when re-enabled: Restart at time zero

OK | Cancel | Help | Apply

Lowpass Filter

The lowpass filter is modeled using a `Lowpass Filter` block. The example uses a generalized Remez FIR filter design algorithm. The filter has a passband frequency of 8000 Hz, a stopband frequency of 10,000 Hz, a passband ripple of 0.1 dB, and a stopband attenuation of 80 dB.

Block Parameters: Lowpass FIR Filter

Lowpass Filter

Design a FIR or IIR lowpass filter

Source code

Main   Data Types

Parameters

Filter type: FIR

☑ Design minimum order filter

Passband edge frequency (Hz):   8e3

Stopband edge frequency (Hz):   1e4

Maximum passband ripple (dB):   0.1

Minimum stopband attenuation (dB):   80

☐ Inherit sample rate from input

Input sample rate (Hz):   44100

View Filter Response

Simulate using: Interpreted execution

OK   Cancel   Help   Apply

The Lowpass Filter block uses frame-based processing to process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can improve the computational time of your signal processing algorithms.

Compare Original and Filtered Signal

Use a `Spectrum Analyzer` to compare the power spectra of the original and filtered signals. The spectrum units are in dBm.

To run the simulation, in the model, click **Run**. To stop the simulation, in the `Spectrum Analyzer` block, click **Stop**. Alternatively, you can execute the following code to run the simulation for 200 frames of data.

```
set_param(model,'StopTime','256/44100 * 200')
sim(model);
```
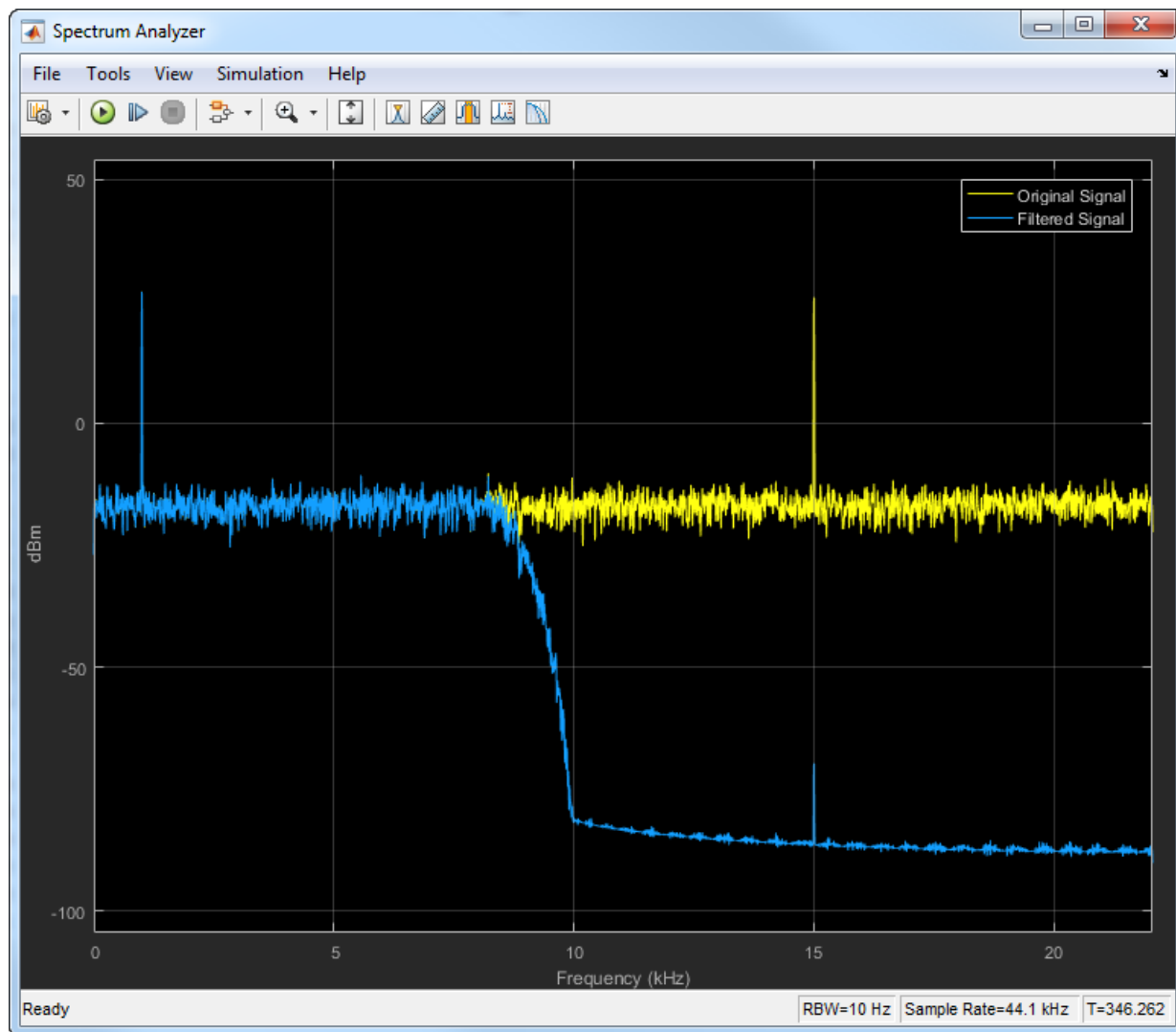
Frequencies above 10 kHz in the source signal are attenuated. The resulting signal maintains the peak at 1 kHz because it falls in the passband of the lowpass filter.

See Also

"Filter Frames of a Noisy Sine Wave Signal in MATLAB" on page 1-7 | "Lowpass IIR Filter Design in Simulink" on page 1-28 | "Tunable Lowpass Filtering of Noisy Input in Simulink" on page 1-67 | "Design Multirate Filters" on page 1-46

Lowpass Filter Design in MATLAB

This example shows how to design lowpass filters. The example highlights some of the most commonly used command-line tools in the DSP System Toolbox. Alternatively, you can use the Filter Builder app to implement all the designs presented here. For more design options, see Designing Low Pass FIR Filters.

Introduction

When designing a lowpass filter, the first choice you make is whether to design an FIR or IIR filter. You generally choose FIR filters when a linear phase response is important. FIR filters also tend to be preferred for fixed-point implementations because they are typically more robust to quantization effects. FIR filters are also used in many high-speed implementations such as FPGAs or ASICs because they are suitable for pipelining. IIR filters (in particular biquad filters) are used in applications (such as audio signal processing) where phase linearity is not a concern. IIR filters are generally computationally more efficient in the sense that they can meet the design specifications with fewer coefficients than FIR filters. IIR filters also tend to have a shorter transient response and a smaller group delay. However, the use of minimum-phase and multirate designs can result in FIR filters comparable to IIR filters in terms of group delay and computational efficiency.

FIR Lowpass Designs - Specifying the Filter Order

There are many practical situations in which you must specify the filter order. One such case is if you are targeting hardware which has constrained the filter order to a specific number. Another common scenario is when you have computed the available computational budget (MIPS) for your implementation and this affords you a limited filter order. FIR design functions in the Signal Processing Toolbox (including `fir1`, `firpm`, and `firls`) are all capable of designing lowpass filters with a specified order. In the DSP System Toolbox, the preferred function for lowpass FIR filter design with a specified order is firceqrip. This function designs optimal equiripple lowpass/highpass FIR filters with specified passband/stopband ripple values and with a specified passband-edge frequency. The stopband-edge frequency is determined as a result of the design.

Design a lowpass FIR filter for data sampled at 48 kHz. The passband-edge frequency is 8 kHz. The passband ripple is 0.01 dB and the stopband attenuation is 80 dB. Constrain the filter order to 120.

```
N  = 120;
Fs = 48e3;
```

```
Fp  = 8e3;
Ap  = 0.01;
Ast = 80;
```

Obtain the maximum deviation for the passband and stopband ripples in linear units.

```
Rp  = (10^(Ap/20) - 1)/(10^(Ap/20) + 1);
Rst = 10^(-Ast/20);
```

Design the filter using `firceqrip` and view the magnitude frequency response.

```
NUM = firceqrip(N,Fp/(Fs/2),[Rp Rst],'passedge');
fvtool(NUM,'Fs',Fs)
```
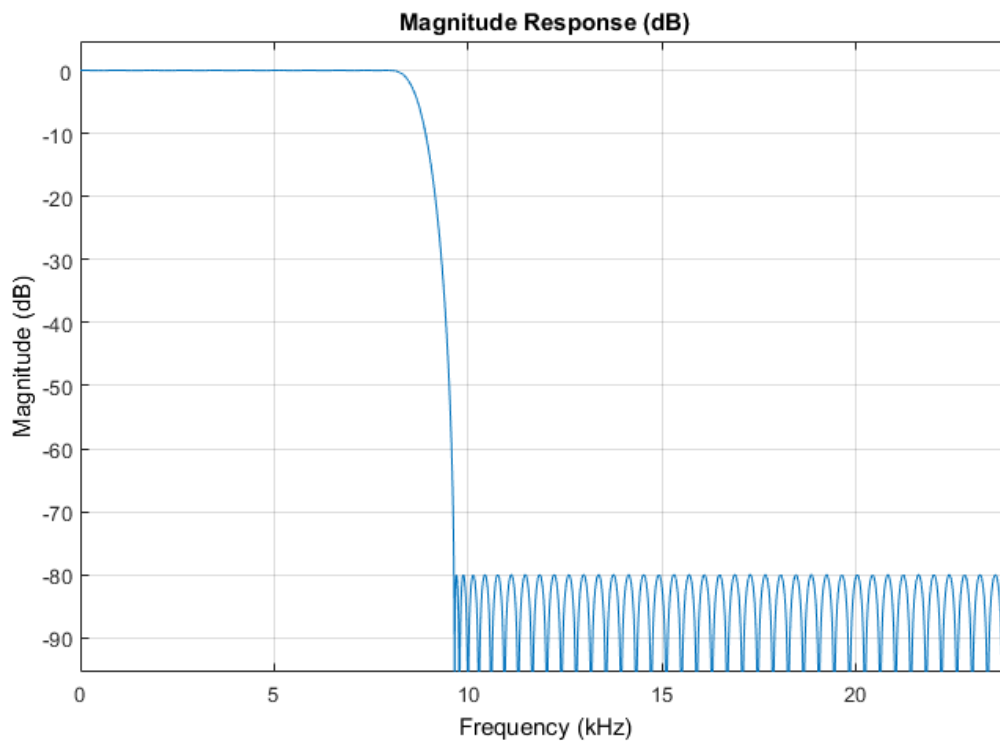

The resulting stopband-edge frequency is about 9.64 kHz.

Minimum-Order Designs

Another design function for optimal equiripple filters is `firgr`. `firgr` can design a filter that meets passband/stopband ripple constraints as well as a specified transition width with the smallest possible filter order. For example, if the stopband-edge frequency is specified as 10 kHz, the resulting filter has an order of 100 rather than the 120th-order filter designed with `firceqrip`. The smaller filter order results from the larger transition band.

Specify the stopband-edge frequency of 10 kHz. Obtain a minimum-order FIR filter with a passband ripple of 0.01 dB and 80 dB of stopband attenuation.

```
Fst    = 10e3;
NumMin = firgr('minorder',[0 Fp/(Fs/2) Fst/(Fs/2) 1], [1 1 0 0],[Rp,Rst]);
```

Plot the magnitude frequency responses for the minimum-order FIR filter obtained with `firgr` and the 120th-order filter designed with `firceqrip`. The minimum-order design results in a filter with order 100. The transition region of the 120th-order filter is, as expected, narrower than that of the filter with order 100.

```
hvft = fvtool(NUM,1,NumMin,1,'Fs',Fs);
legend(hvft,'N = 120','N = 100')
```
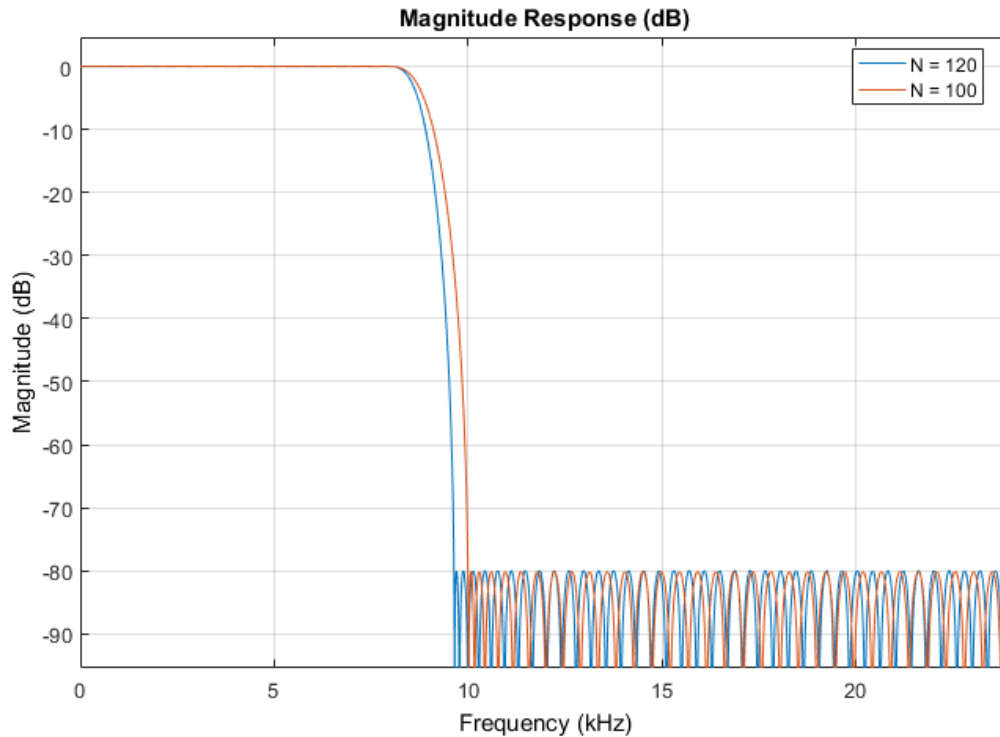
Filtering Data

To apply the filter to data, you can use the `filter` command or you can use `dsp.FIRFilter`. `dsp.FIRFilter` has the advantage of managing state when executed in a loop. `dsp.FIRFilter` also has fixed-point capabilities and supports C code generation, HDL code generation, and optimized code generation for ARM® Cortex® M and ARM Cortex A.

Filter 10 seconds of white noise with zero mean and unit standard deviation in frames of 256 samples with the 120th-order FIR lowpass filter. View the result on a spectrum analyzer.

```
LP_FIR = dsp.FIRFilter('Numerator',NUM);
SA     = dsp.SpectrumAnalyzer('SampleRate',Fs,'SpectralAverages',5);
tic
```

```
while toc < 10
    x = randn(256,1);
    y = step(LP_FIR,x);
    step(SA,y);
end
```
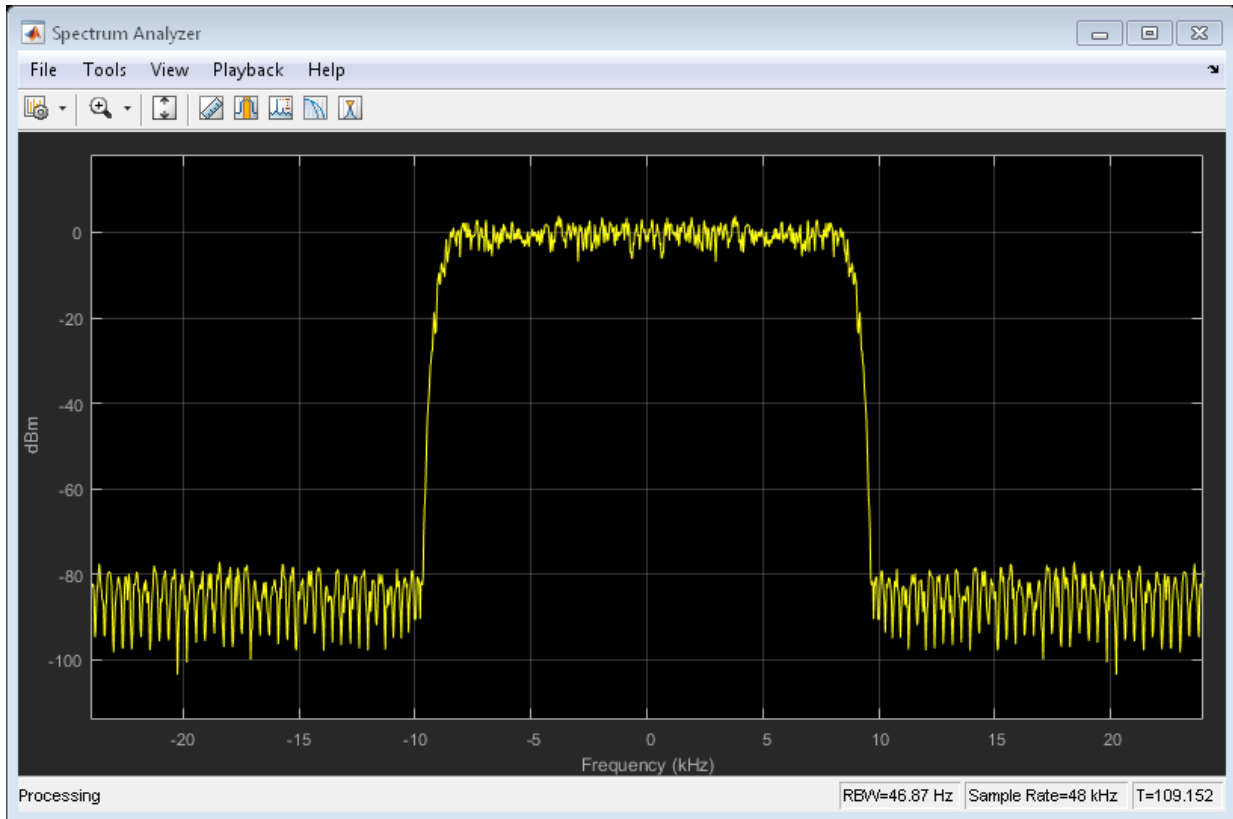

Using `dsp.LowpassFilter`

`dsp.LowpassFilter` is an alternative to using `firceqrip` and `firgr` in conjuction with `dsp.FIRFilter`. Basically, `dsp.LowpassFilter` condenses the two step process into one. `dsp.LowpassFilter` provides the same advantages that `dsp.FIRFilter` provides in terms of fixed-point support, C code generation support, HDL code generation support, and ARM Cortex code generation support.

1-21

Design a lowpass FIR filter for data sampled at 48 kHz. The passband-edge frequency is 8 kHz. The passband ripple is 0.01 dB and the stopband attenuation is 80 dB. Constrain the filter order to 120. Create a `dsp.FIRFilter` based on your specifications.

```
LP_FIR = dsp.LowpassFilter('SampleRate',Fs,...
    'DesignForMinimumOrder',false,'FilterOrder',N,...
    'PassbandFrequency',Fp,'PassbandRipple',Ap,'StopbandAttenuation',Ast);
```

The coefficients in `LP_FIR` are identical to the coefficients in `NUM`.

```
NUM_LP = tf(LP_FIR);
```

You can use `LP_FIR` to filter data directly, as shown in the preceding example. You can also analyze the filter using FVTool or measure the response using `measure`.

```
fvtool(LP_FIR,'Fs',Fs);
measure(LP_FIR)


ans =

Sample Rate      : 48 kHz
Passband Edge    : 8 kHz
3-dB Point       : 8.5843 kHz
6-dB Point       : 8.7553 kHz
Stopband Edge    : 9.64 kHz
Passband Ripple  : 0.01 dB
Stopband Atten.  : 79.9981 dB
Transition Width : 1.64 kHz
```
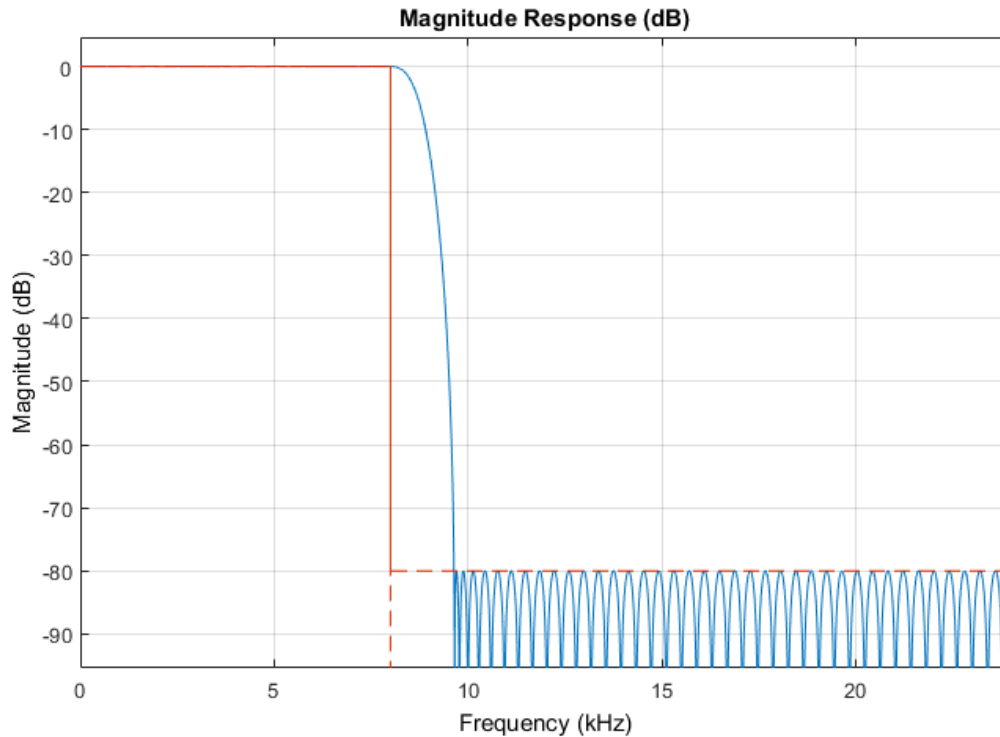
Minimum-Order Designs with dsp.LowpassFilter

You can use `dsp.LowpassFilter` to design minimum-order filters and use `measure` to verify that the design meets the prescribed specifications. The order of the filter is again 100.

```
LP_FIR_minOrd = dsp.LowpassFilter('SampleRate',Fs,...
    'DesignForMinimumOrder',true,'PassbandFrequency',Fp,...
    'StopbandFrequency',Fst,'PassbandRipple',Ap,'StopbandAttenuation',Ast);
measure(LP_FIR_minOrd)
Nlp = order(LP_FIR_minOrd)


ans =

Sample Rate       : 48 kHz
```

```
Passband Edge     : 8 kHz
3-dB Point        : 8.7136 kHz
6-dB Point        : 8.922 kHz
Stopband Edge     : 10 kHz
Passband Ripple   : 0.0098641 dB
Stopband Atten.   : 80.122 dB
Transition Width  : 2 kHz


Nlp =

   100
```

Designing IIR Filters

Elliptic filters are the IIR counterpart to optimal equiripple FIR filters. Accordingly, you can use the same specifications to design elliptic filters. The filter order you obtain for an IIR filter is much smaller than the order of the corresponding FIR filter.

Design an elliptic filter with the same sampling frequency, cutoff frequency, passband-ripple constraint, and stopband attenuation as the 120th-order FIR filter. Reduce the filter order for the elliptic filter to 10.

```
N = 10;
LP_IIR = dsp.LowpassFilter('SampleRate',Fs,'FilterType','IIR',...
    'DesignForMinimumOrder',false,'FilterOrder',N,...
    'PassbandFrequency',Fp,'PassbandRipple',Ap,'StopbandAttenuation',Ast);
```

Compare the FIR and IIR designs. Compute the cost of the two implementations.

```
hfvt = fvtool(LP_FIR,LP_IIR,'Fs',Fs);
legend(hfvt,'FIR Equiripple, N = 120', 'IIR Elliptic, N = 10');
cost_FIR = cost(LP_FIR)
cost_IIR = cost(LP_IIR)


cost_FIR =

                  NumCoefficients: 121
                        NumStates: 120
    MultiplicationsPerInputSample: 121
          AdditionsPerInputSample: 120
```

```
cost_IIR =

              NumCoefficients: 25
                    NumStates: 20
    MultiplicationsPerInputSample: 25
         AdditionsPerInputSample: 20
```

Magnitude Response (dB)
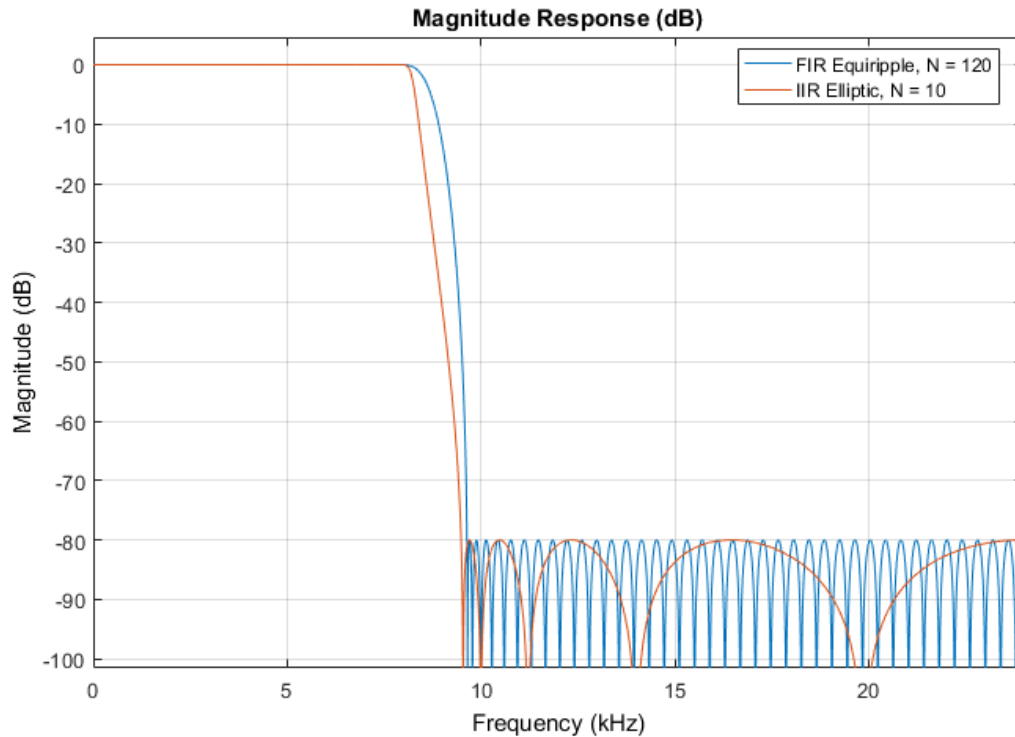
The FIR and IIR filters have similar magnitude responses. The cost of the IIR filter is about 1/6 the cost of the FIR filter.

Running the IIR Filters

The IIR filter is designed as a biquad filter. To apply the filter to data, use the same commands as in the FIR case.

Filter 10 seconds of white Gaussian noise with zero mean and unit standard deviation in frames of 256 samples with the 10th-order IIR lowpass filter. View the result on a spectrum analyzer.

```
SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'SpectralAverages',5);
tic
while toc < 10
    x = randn(256,1);
    y = step(LP_IIR,x);
    step(SA,y);
end
```
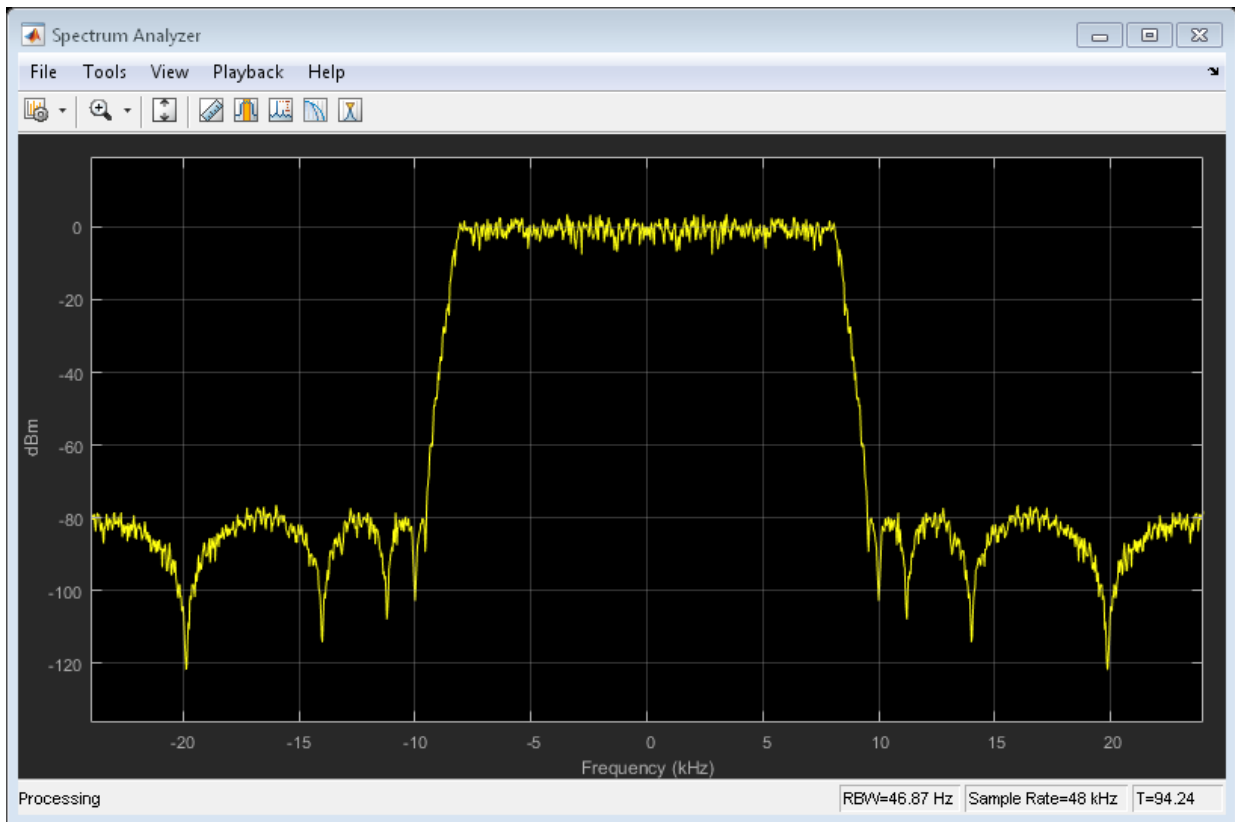
Variable Bandwidth FIR and IIR Filters

You can also design filters that allow you to change the cutoff frequency at run-time. `dsp.VariableBandwidthFIRFilter` and `dsp.VariableBandwidthIIRFilter` can be used for such cases.

See Also

"Tunable Lowpass Filtering of Noisy Input in Simulink" on page 1-67 | "Lowpass IIR Filter Design in Simulink" on page 1-28 | "Design Multirate Filters" on page 1-46 | "Filter Frames of a Noisy Sine Wave Signal in MATLAB" on page 1-7

Lowpass IIR Filter Design in Simulink

This example shows how to design classic lowpass IIR filters in Simulink.

The example first presents filter design using `filterbuilder`. The critical parameter in this design is the cutoff frequency, the frequency at which filter power decays to half (-3 dB) the nominal passband value. The example shows how to replace a Butterworth design with either a Chebyshev or elliptic filter of the same order and obtain a steeper roll-off at the expense of some ripple in the passband and/or stopband of the filter. The example also explores minimum-order designs.

The example then shows how to design and use lowpass filters in Simulink using the interface available from the Lowpass Filter block..

Finally, the example showcases the `Variable Bandwidth IIR Filter`, which enables you to change the filter cutoff frequency at run time.

In this section...
"Lowpass Filter Block" on page 1-44
"Variable Bandwidth IIR Filter Block" on page 1-45

`filterbuilder`

`filterbuilder` starts user interface for building filters. `filterbuilder` uses a specification-centered approach to find the best algorithm for the desired response. It also enables you to create a Simulink block from the specified design.

To start designing IIR lowpass filter blocks using `filterbuilder`, execute the command `filterbuilder('lp')`. A Lowpass Design dialog box opens.

Butterworth Filter

Design an eighth order Butterworth lowpass filter with a cutoff frequency of 5 kHz, assuming a sample rate of 44.1 KHz.

Set the **Impulse response** to IIR, the **Order mode** to Specify, and the **Order** to 8. To specify the cutoff frequency, set **Frequency constraints** to Half power (3 dB) frequency. To specify the frequencies in Hz, set **Frequency units** to Hz, **Input sample rate** to 44100, and **Half power (3 dB) frequency** to 5000. Set the **Design method** to Butterworth.

Lowpass Design

Lowpass Design

Design a lowpass filter.

Filter output variable name: Hlp

View Filter Response

| Main | Data Types | Code Generation |

Filter specifications

Impulse response: IIR

Order mode: Specify

Order: 8   ☐ Denominator order: 20

Filter type: Single-rate

Frequency specifications

Frequency constraints: Half power (3dB) frequency

Frequency units: Hz   Input sample rate: 44100

Half power (3dB) frequency: 5000

Magnitude specifications

Magnitude constraints: Unconstrained

Algorithm

Design method: Butterworth

☑ Scale SOS filter coefficients to reduce chance of overflow

Filter implementation

Structure: Direct-form II SOS

☐ Use a System object to implement filter

Click **Apply**. To visualize the filter's frequency response, click **View Filter Response**. The filter is maximally flat. There is no ripple in the passband or in the stopband. The filter response is within the specification mask (the red dotted line).
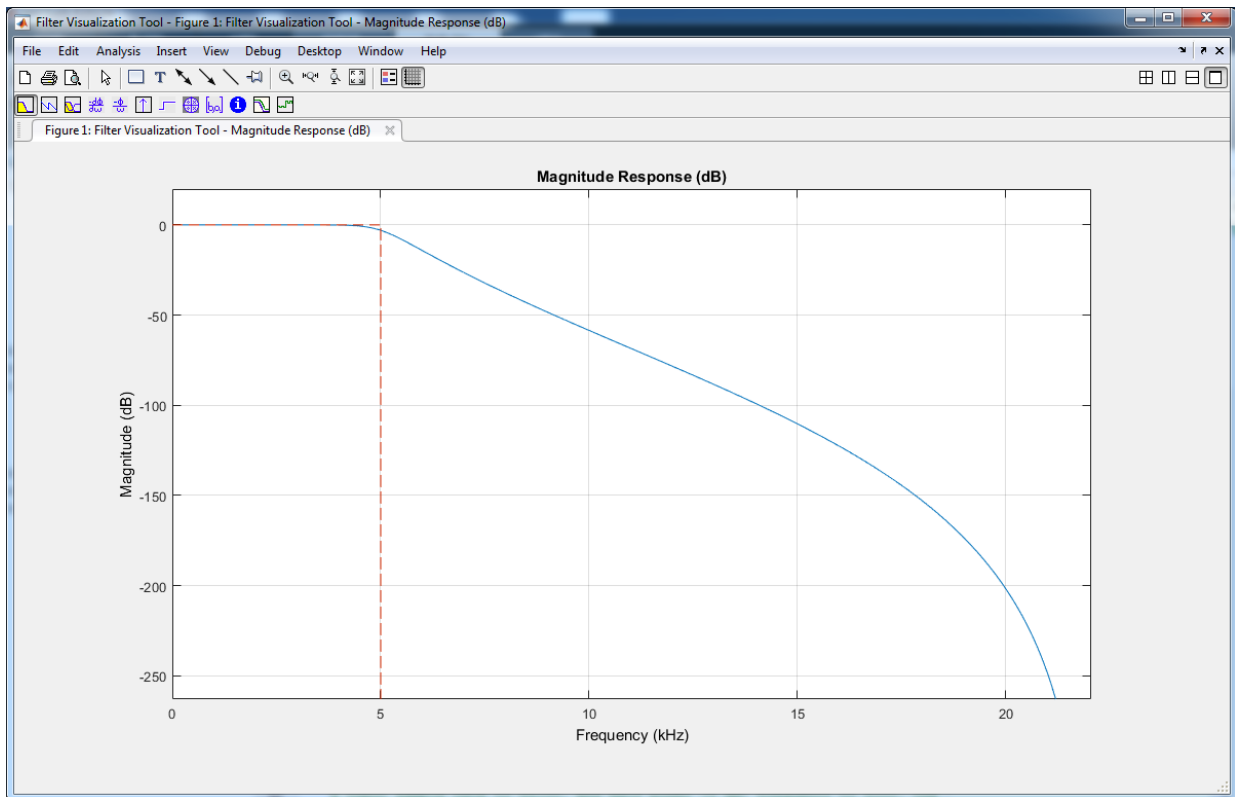
Generate a block from this design and use it in a model. Open the model `ex_iir_design`. In `filterbuilder`, on the **Code Generation** tab, click **Generate Model**. In the Export to Simulink window, specify the **Block name** as `Butter` and **Destination** as `Current`. You can also choose to build the block using basic elements such as delays and gains, or use one of the DSP System Toolbox filter blocks. This example uses the filter block.

Export to Simulink (Direct-Form II, Second-Order Sections, order = 8)

Model

Block name: Butter

Destination: Current

User Defined: Untitled

☐ Overwrite generated 'Butter' block

☐ Build model using basic elements

Optimization

☐ Optimize for zero gains

☐ Optimize for unity gains

☐ Optimize for negative gains

☐ Optimize delay chains

☑ Optimize for unity scale values

Input processing: Columns as channels (frame based)

Realize Model

Click **Realize model** to generate the Simulink block. You can now connect the block's input and output ports to the source and sink blocks in the `ex_iir_design` model.
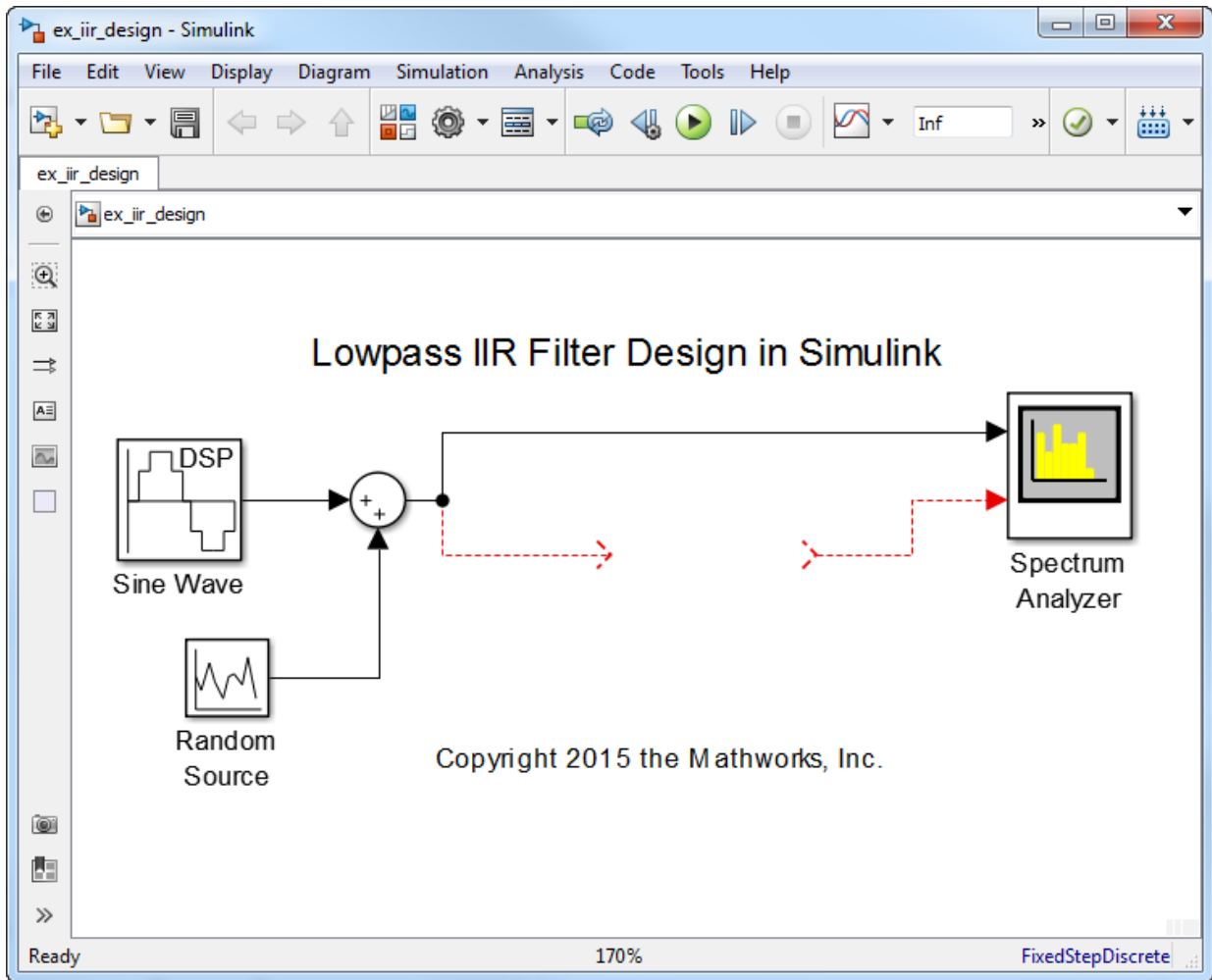
In the model, a noisy sine wave sampled at 44.1 kHz passes through the filter. The sine wave is corrupted by Gaussian noise with zero mean and a variance of 10^{-5}. Run the model. The view in the Spectrum Analyzer shows the original and filtered signals.
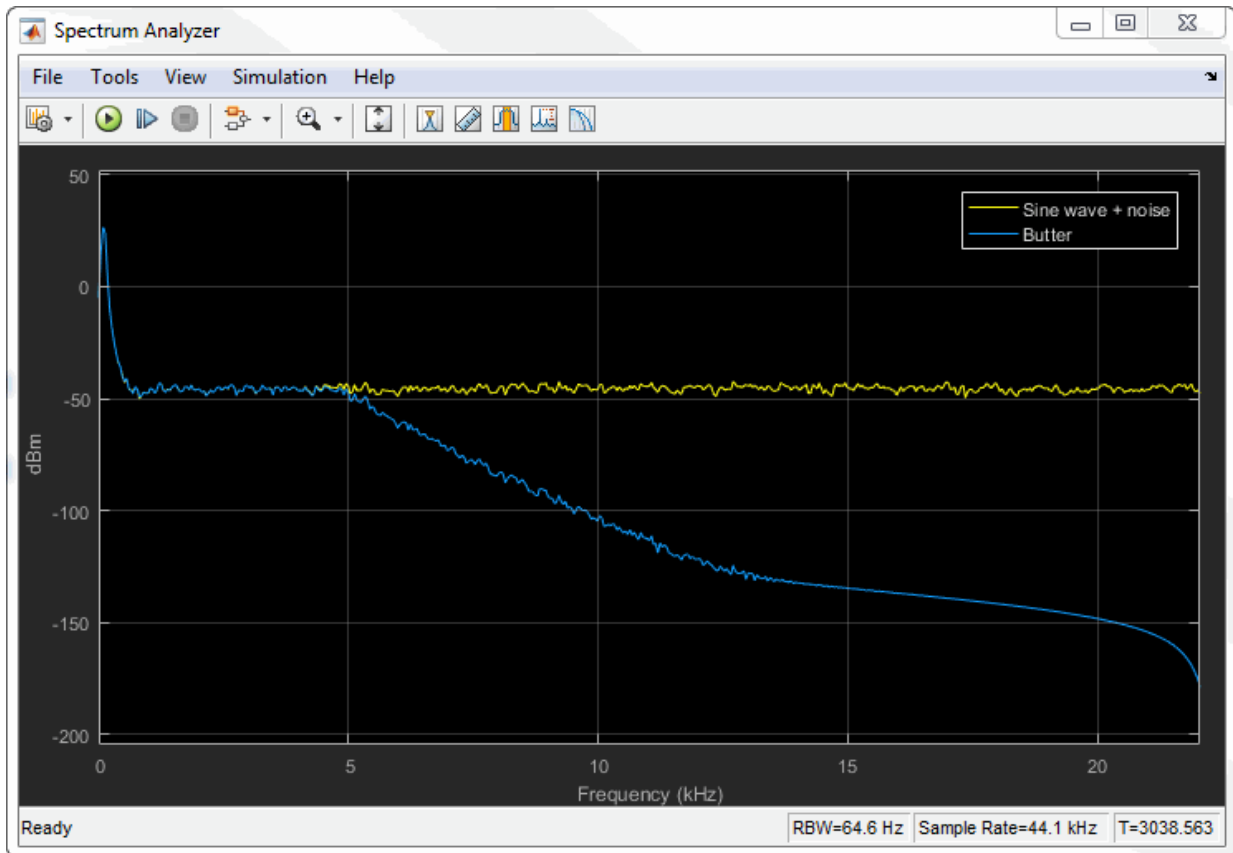
Chebyshev Type I Filter

Now design a Chebyshev Type I filter. A Chebyshev type I design allows you to control the passband. There are still no ripples in the stopband. Larger ripples enable a steeper roll-off. In this model, the peak-to-peak ripple is specified as 0.5 dB.

In the **Main** tab of `filterbuilder`, set the

1 **Magnitude Constraints** to `Passband ripple`.
2 **Passband ripple** to `0.5`.
3 **Design method** to `Chebyshev type I`.

Click **Apply** and then click **View Filter Response**.

Zooming in on the passband, you can see that the ripples are contained in the range [-0.5, 0] dB.
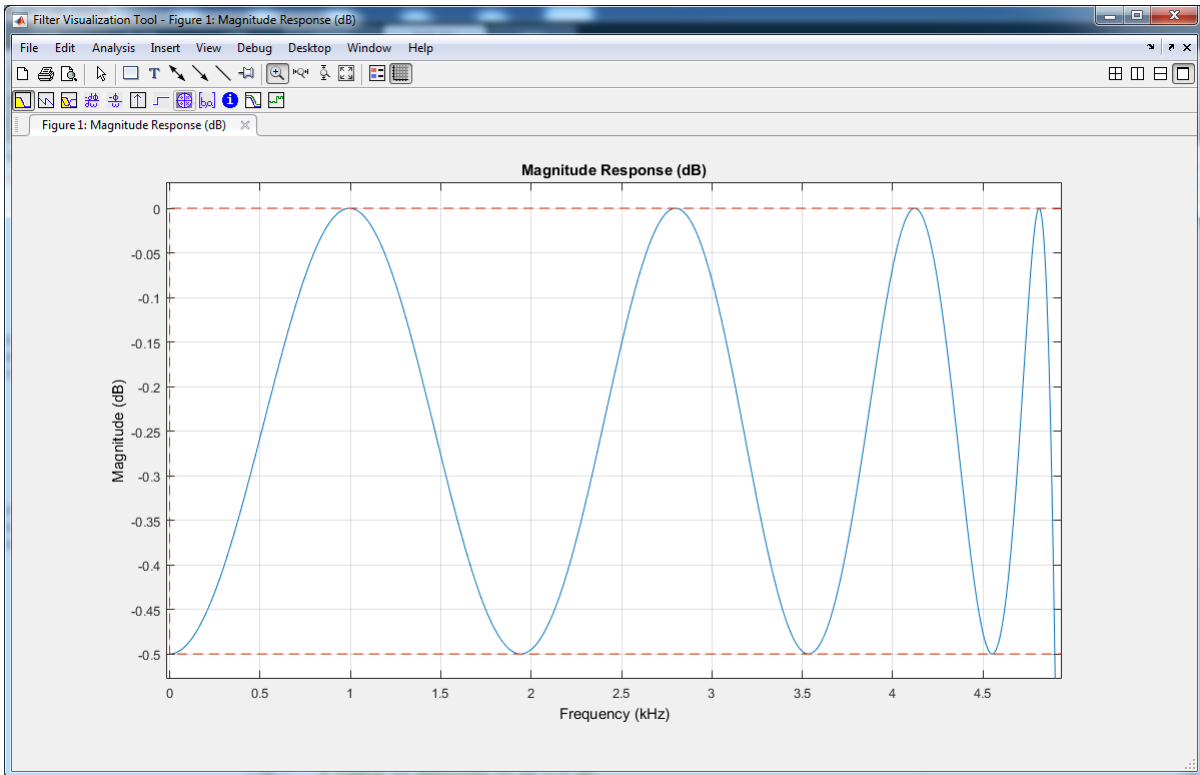
Similar to the Butterworth filter, you can generate a block from this design by clicking **Generate Model** on the **Code Generation** tab, and then clicking **Realize model**.

Chebyshev Type II Filter

A Chebyshev type II design allows you to control the stopband attenuation. There are no ripples in the passband. A smaller stopband attenuation enables a steeper roll-off. In this example, the stopband attenuation is 80 dB. Set the `filterbuilder` **Main** tab as shown, and click **Apply**.

Click **View Filter Response**.

To generate a block from this design, on the **Code Generation** tab, click **Generate Model**, and then click **Realize model**.

Elliptic Filter

An elliptic filter can provide steeper roll-off compared to previous designs by allowing ripples in both the stopband and passband. To illustrate this behavior, use the same passband and stopband characteristics specified in the Chebyshev designs. Set the `filterbuilder` **Main** tab as shown, and click **Apply**.

Lowpass Design ✕

Lowpass Design

Design a lowpass filter.

Filter output variable name: Hlp   View Filter Response

| Main | Data Types | Code Generation |

Filter specifications

Impulse response: IIR ▼

Order mode: Specify ▼

Order: 8   ☐ Denominator order: 20

Filter type: Single-rate ▼

Frequency specifications

Frequency constraints: Half power (3dB) frequency ▼

Frequency units: Hz ▼   Input sample rate: 44100

Half power (3dB) frequency: 5000

Magnitude specifications

Magnitude constraints: Passband ripple and stopband attenuation ▼

Magnitude units: dB ▼

Passband ripple: .5   Stopband attenuation: 80

Algorithm

Design method: Elliptic ▼

☑ Scale SOS filter coefficients to reduce chance of overflow

Filter implementation

Structure: Direct-form II SOS ▼

☐ Use a System object to implement filter

To generate a block from this design, on the **Code Generation** tab, click **Generate Model**, and then click **Realize model**.

Minimum-Order Designs

To specify the passband and stopband in terms of frequencies and the amount of tolerable ripple, use a minimum-order design. As an example, verify that the **Order mode** of the Butterworth filter is set to `Minimum`, and set **Design method** to `Butterworth`. Set the passband and stopband frequencies to `0.1*22050` Hz and `0.3*22050` Hz, and the passband ripple and the stopband attenuation to 1 dB and 60 dB, respectively. A seventh-order filter is necessary to meet the specifications with a Butterworth design. By following the same approach for other design methods, you can verify that a fifth-order filter is required for Chebyshev type I and type II designs. A fourth-order filter is sufficient for the elliptic design.

Lowpass Design

Lowpass Design

Design a lowpass filter.

Filter output variable name: Hlp2

[View Filter Response]

| Main | Data Types | Code Generation |

Filter specifications

Impulse response: IIR

Order mode: Minimum

Filter type: Single-rate

Frequency specifications

Frequency units: Hz   Input sample rate: 44100

Passband frequency: 0.1*22050   Stopband frequency: 0.3*22050

Magnitude specifications

Magnitude units: dB

Passband ripple: 1   Stopband attenuation: 60

Algorithm

Design method: Butterworth

☑ Scale SOS filter coefficients to reduce chance of overflow

▶ Design options

Filter implementation

Structure: Direct-form II SOS

☐ Use a System object to implement filter

This figure shows the magnitude response for the seventh-order Butterworth design.
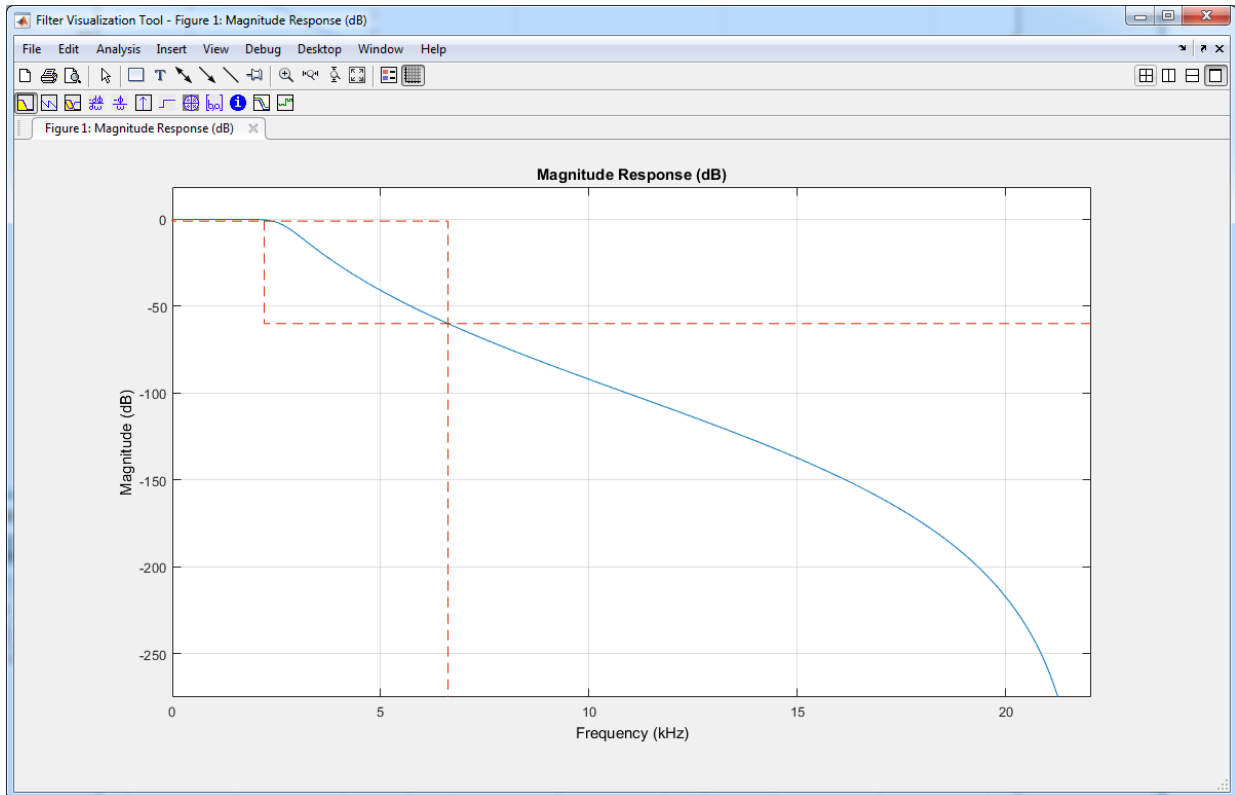
The pole-zero plot for the seventh-order Butterworth design shows the expected clustering of 7 poles around an angle of zero radians on the unit circle and the corresponding 7 zeros at an angle of π radians.
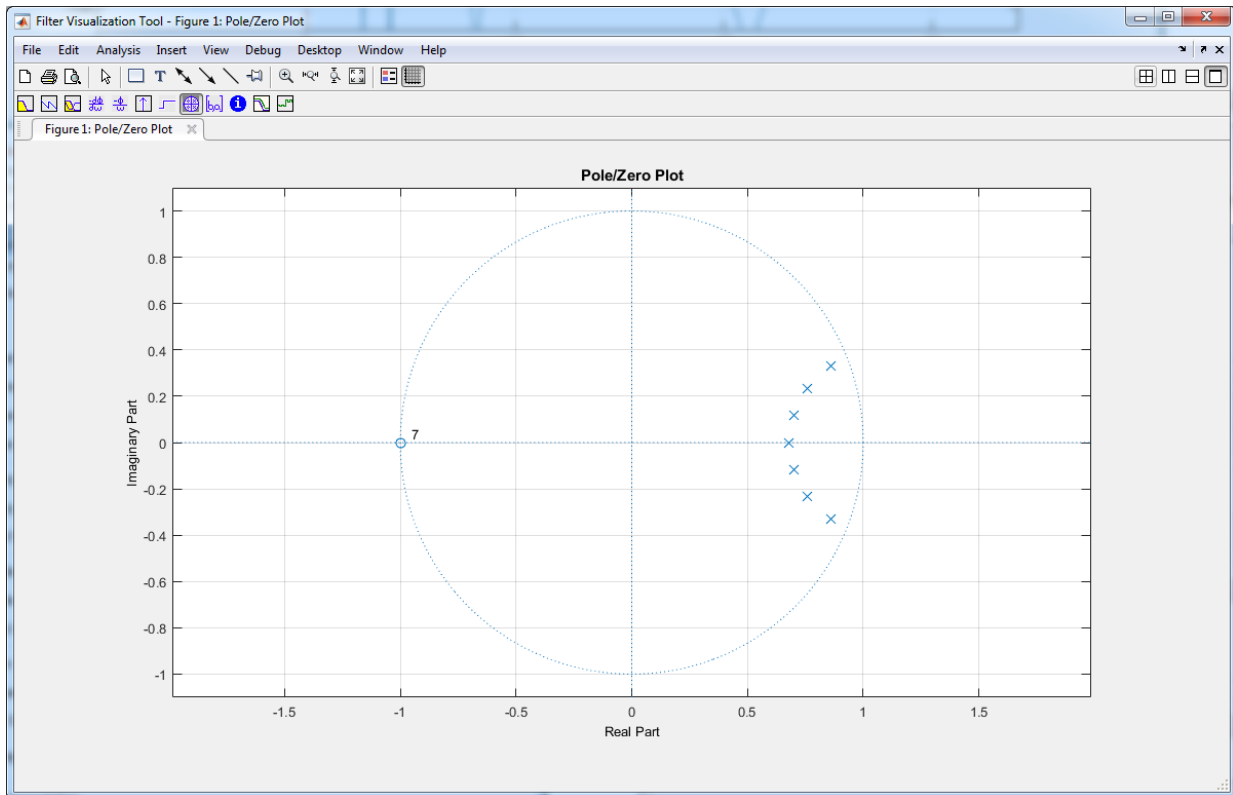
Lowpass Filter Block

As an alternative to `filterbuilder`, you can use the `Lowpass Filter` block in your Simulink model. The `Lowpass Filter` block combines the design and implementation stages into one step. The filter designs its coefficients using the elliptical method, and allows minimum order and custom order designs.

The `Lowpass Filter` block is used in the model `ex_lowpass` to filter a noisy sine wave signal sampled at `44.1` kHz. The original and filtered signals are displayed in a spectrum analyzer.

```
model = 'ex_lowpass';
open_system(model);
set_param(model,'StopTime','1024/44100 * 1000')
```

```
sim(model);
```
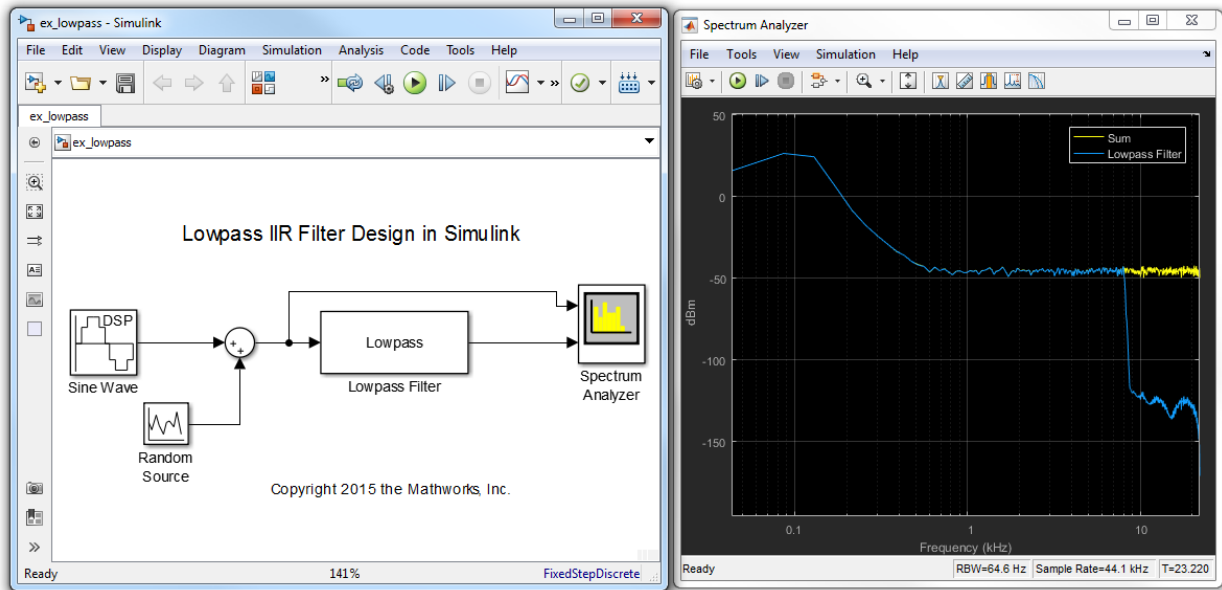

The `Lowpass Filter` block allows you to design filters that approximate arbitrarily close to Butterworth and Chebyshev filters. To approximate a Chebyshev Type I filter, make the stopband attenuation arbitrarily large, for example, `180` dB. To approximate a Chebyshev Type II filter, make the passband ripple arbitrarily small, for example, `1e-4`. To approximate a Butterworth filter, make the stopband attenuation arbitrarily large and the passband ripple arbitrarily small.

Variable Bandwidth IIR Filter Block

You can also design filters that allow you to change the cutoff frequency at run time. The `Variable Bandwidth IIR Filter` block can be used for such cases. Refer to the "Tunable Lowpass Filtering of Noisy Input in Simulink" on page 1-67 example for a model that uses this block.

See Also

"Tunable Lowpass Filtering of Noisy Input in Simulink" on page 1-67 | "Lowpass Filter Design in MATLAB" on page 1-17 | "Design Multirate Filters" on page 1-46

Design Multirate Filters

Multirate filters are filters in which different parts of the filter operate at different rates. Such filters are commonly used when the input and output sample rates differ, such as during decimation, interpolation, or a combination of both. However, multirate filters are often used in designs where the input sample rate and output sample rate are the same. In such filters, there is an internal decimation and interpolation occurring in a series of filters. Such filters can achieve both greatly reduced filter lengths and computational rates as compared to standard single-rate filter designs.

The most basic multirate filters are interpolators, decimators, and rate converters. These filters are building components of more advanced filter technologies such as filter banks and Quadrature Mirror Filter (QMF). You can design these filters in MATLAB and Simulink using the `designMultirateFIR` function.

The function uses the FIR Nyquist filter design algorithm to compute the filter coefficients. To implement these filters in MATLAB, use these coefficients as inputs to the dsp.FIRDecimator, dsp.FIRInterpolator, and dsp.FIRRateConverter System objects. In Simulink, compute these coefficients using `designMultirateFIR` in the default **Auto** mode of the `FIR Decimation`, `FIR Interpolation`, and `FIR Rate Conversion` blocks.

The inputs to the `designMultirateFIR` function are the interpolation factor and the decimation factor. Optionally, you can provide the half-polyphase length and stopband attenuation. The interpolation factor of the decimator is set to 1. Similarly, the decimation factor of the interpolator is set to 1.

These examples show how to implement an FIR decimator in MATLAB and Simulink. The same workflow can apply to an FIR interpolator and FIR rate converter as well.

In this section...
"Implement an FIR Decimator in MATLAB" on page 1-46
"Implement an FIR Decimator in Simulink" on page 1-51
"Sample Rate Conversion" on page 1-54

Implement an FIR Decimator in MATLAB

To implement an FIR Decimator, you must first design it by using the `designMultirateFIR` function. Specify the decimation factor of interest (usually greater than 1) and an interpolation factor equal to 1. You can use the default half-

polyphase length of 12 and the default stopband attenuation of 80 dB. Alternatively, you can also specify the half-polyphase length and stopband attenuation values.

Design an FIR decimator with the decimation factor set to 3 and the half-polyphase length set to 14. Use the default stopband attenuation of 80 dB.

```
b = designMultirateFIR(1,3,14);
```

Provide the coefficients vector, `b`, as an input to the `dsp.FIRDecimator` System object™.

```
FIRDecim = dsp.FIRDecimator(3,b);
fvtool(FIRDecim);
```
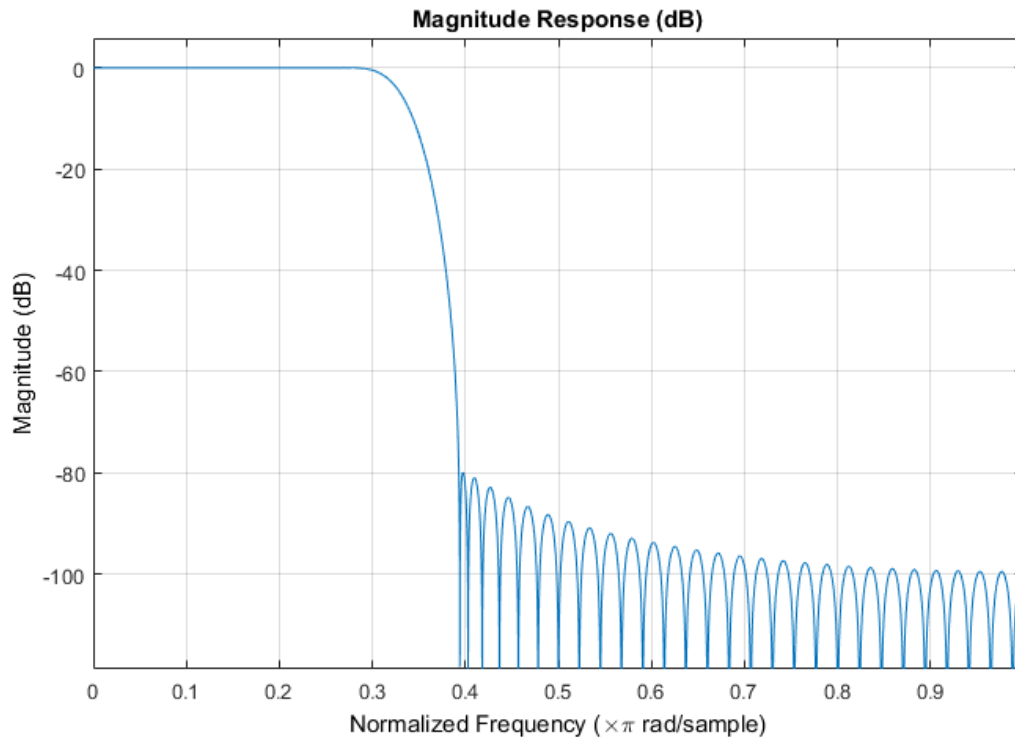

By default, the `fvtool` shows the magnitude response. Navigate through the `fvtool` toolbar to see the phase response, impulse response, group delay, and other filter analysis information.

Filter a noisy sine wave input using the `FIRDecim` object. The sine wave has frequencies at 1000 Hz and 3000 Hz 3000 Hz. The noise is a white Gaussian noise with mean zero and standard deviation 1e-5. The decimated output will have one-third the sample rate as input. Initialize two `dsp.SpectrumAnalyzer` System objects, one for the input and the other for the output.

```
f1 = 1000;
f2 = 3000;
Fs = 8000;
source = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs,...
    'SamplesPerFrame',1026);

specanainput = dsp.SpectrumAnalyzer('SampleRate',Fs,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Noisy Input signal',...
    'ChannelNames', {'Noisy Input'});
specanaoutput = dsp.SpectrumAnalyzer('SampleRate',Fs/3,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Filtered output',...
    'ChannelNames', {'Filtered output'});
```

Stream in the input and filter the signal in a processing loop.

```
for Iter = 1:100
    input = sum(step(source),2);
    noisyInput = input + (10^-5)*randn(1026,1);
    output = step(FIRDecim,noisyInput);
    step(specanainput,noisyInput)
    step(specanaoutput,output)
end
```
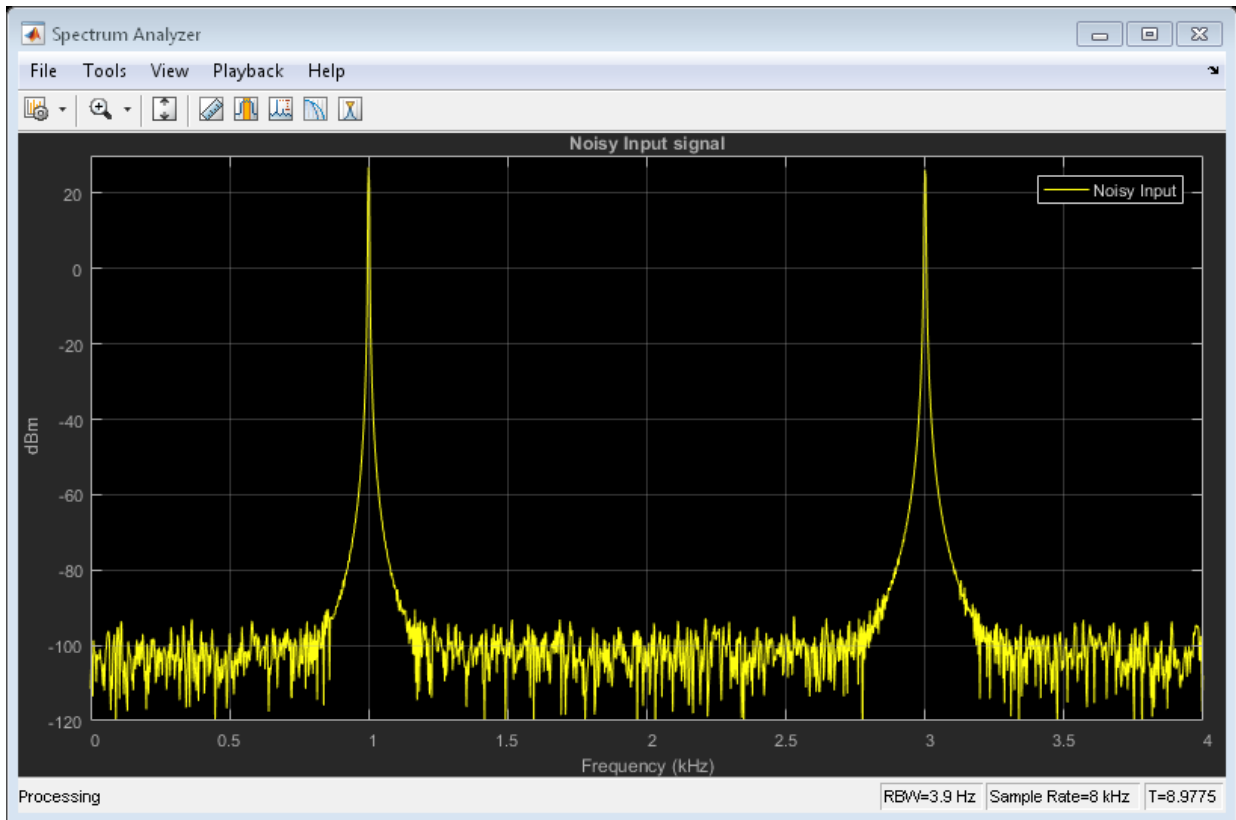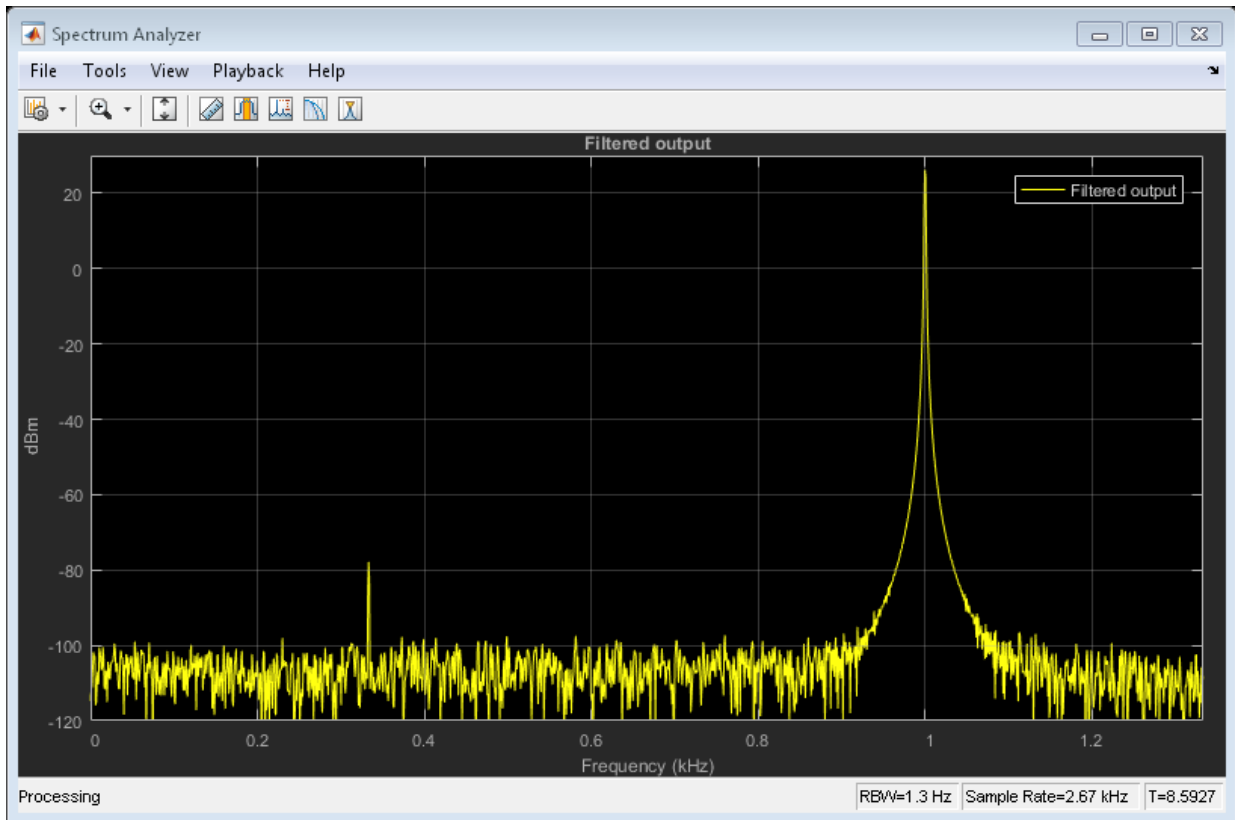
The input has two peaks: one at 1000 Hz and the other at 3000 Hz. The filter has a lowpass response with a passband frequency of 0.3*π rad/sample. With a sampling frequency of 8000 Hz, that is a passband frequency of 1200 Hz. The tone at 1000 Hz is unattenuated, because it falls in the passband of the filter. The tone at 3000 Hz is filtered out.

Similarly, you can design an FIR interpolator and FIR rate Converter by providing appropriate inputs to the designMultirateFIR function. To implement the filters, pass

the designed coefficients to the `dsp.FIRInterpolator` and `dsp.FIRRateConverter` objects.

Implement an FIR Decimator in Simulink

You can design and implement the FIR multirate filters in Simulink using the FIR Decimation, FIR Interpolation, and FIR Rate Conversion blocks. When you set **Coefficient source** to **Dialog parameters**, you can provide `designMultirateFIR(1,2)` as a parameter to specify the filter coefficients. To design the Decimator using the `designMultirateFIR` function, you must specify the decimation factor of interest (usually greater than 1) and an interpolation factor equal to 1. You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternatively, you can also specify the half-polyphase length and stopband attenuation values.

Block Parameters: FIR Decimation

FIR Decimation (mask) (link)

Apply an FIR filter to the input signal, then downsample the signal by an integer-valued factor. The block implements the FIR filter using a polyphase filter structure.

Coefficient source

- ◉ Dialog parameters
- ○ Input port
- ○ Filter object
- ○ Auto

Main | Data Types

Parameters

FIR filter coefficients: `designMultirateFIR(1,2)`

Decimation factor: `2`

Filter structure: `Direct form`

Input processing: `Columns as channels (frame based)`

Rate options: `Enforce single-rate processing`

View Filter Response

OK | Cancel | Help | Apply
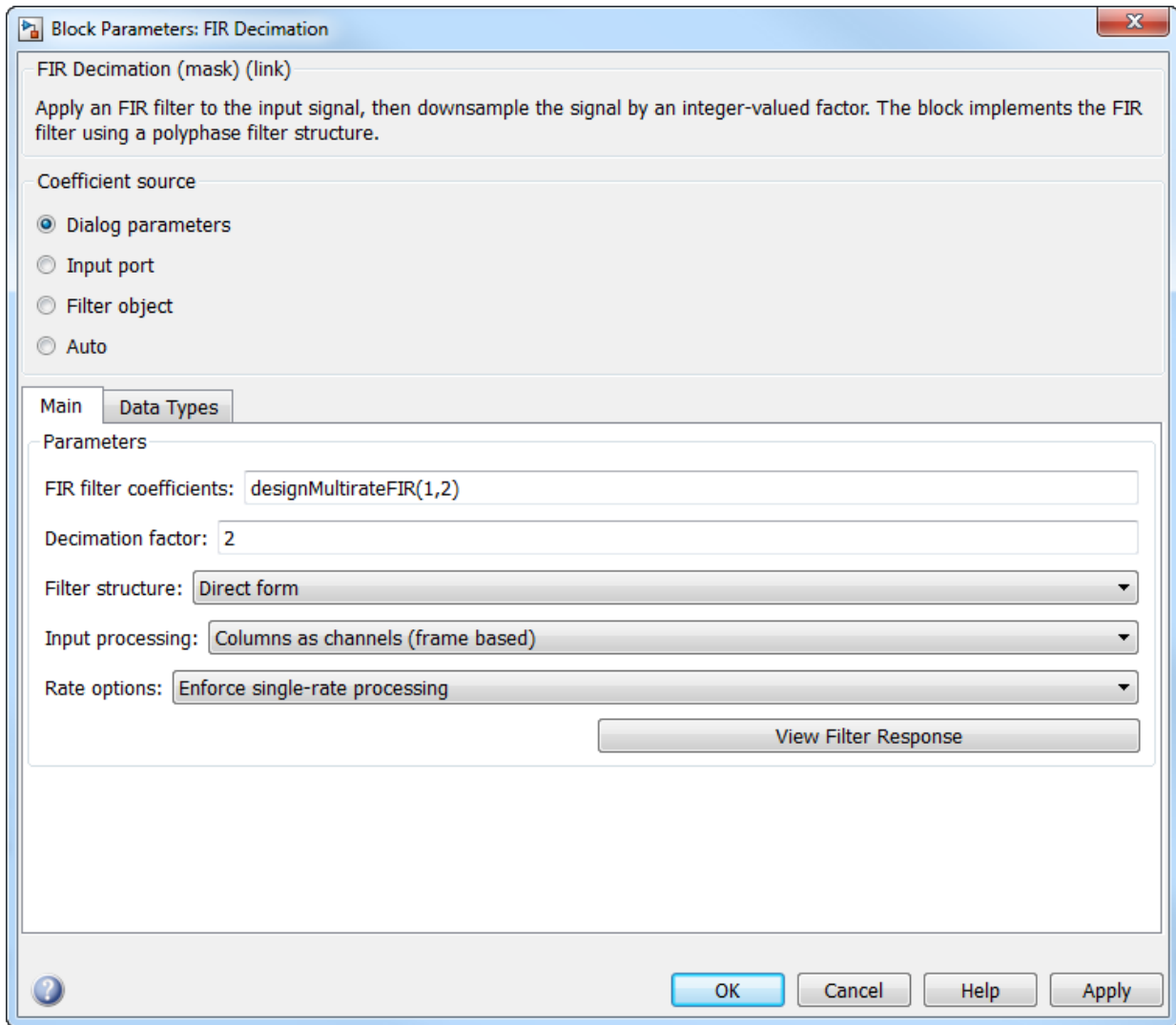
The block chooses the coefficients computed using the `designMultirateFIR` function.

Similarly, you can design an FIR interpolator and FIR rate converter by providing appropriate inputs to the `designMultirateFIR` function.

When you set **Coefficient source** to **Auto**, the block computes the coefficients using the `designMultirateFIR` function. The function uses the decimation factor specified in the block dialog box.

Function Block Parameters: FIR Decimation

FIR Decimation (mask) (link)

Apply an FIR filter to the input signal, then downsample the signal by an integer-valued factor. The block implements the FIR filter using a polyphase filter structure.

Coefficient source

○ Dialog parameters

○ Input port

○ Filter object

◉ Auto

| Main | Data Types |

Parameters

Decimation factor: 2

Filter structure: Direct form ▼

Input processing: Columns as channels (frame based) ▼

Rate options: Enforce single-rate processing ▼

View Filter Response

OK   Cancel   Help   Apply

You can design an FIR Interpolator and an FIR Rate Converter using a similar approach in the corresponding blocks.

Sample Rate Conversion

Sample rate conversion is a process of converting the sample rate of a signal from one sampling rate to another sampling rate. Multistage filters minimize the amount of computation involved in a sample rate conversion. To perform an efficient multistage rate conversion, the `dsp.SampleRateConverter` object:

1 accepts input sample rate and output sample rate as inputs.

2 partitions the design problem into optimal stages.

3 designs all the filters required by the various stages.

4 implements the design.

The design makes sure that aliasing does not occur in the intermediate steps.

In this example, change the sample rate of a noisy sine wave signal from an input rate of 192 kHz to an output rate of 44.1 kHz. Initialize a sample rate converter object.

```
SRC = dsp.SampleRateConverter;
```

Display the filter information.

```
info(SRC)
```

```
ans =

Overall Interpolation Factor   : 147
Overall Decimation Factor      : 640
Number of Filters              : 3
Multiplications per Input Sample: 27.667188
Number of Coefficients         : 8631
Filters:
   Filter 1:
   dsp.FIRDecimator     - Decimation Factor   : 2
   Filter 2:
   dsp.FIRDecimator     - Decimation Factor   : 2
   Filter 3:
   dsp.FIRRateConverter - Interpolation Factor: 147
```

```
                             - Decimation Factor   : 160
```

SRC is a three-stage filter: two FIR decimators followed by an FIR rate converter.

Initialize the sine wave source. The sine wave has two tones: one at 2000 Hz and the other at 5000 Hz.

```
source = dsp.SineWave ('Frequency',[2000 5000],'SampleRate',192000,...
    'SamplesPerFrame',1280);
```

Initialize the spectrum analyzer to see the input and output signals.

```
Fsin = SRC.InputSampleRate;
Fsout = SRC.OutputSampleRate;
specanainput = dsp.SpectrumAnalyzer('SampleRate',Fsin,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Input signal',...
    'ChannelNames', {'Input'});
specanaoutput = dsp.SpectrumAnalyzer('SampleRate',Fsout,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Rate Converted output',...
    'ChannelNames', {'Rate Converted output'});
```

Stream in the input signal and convert the signal's sample rate.

```
for Iter = 1 : 5000
    input = sum(step(source),2);
    noisyinput = input + (10^-5)*randn(1280,1);
    output = step(SRC,noisyinput);
    step(specanainput,noisyinput);
    step(specanaoutput,output);
end
```
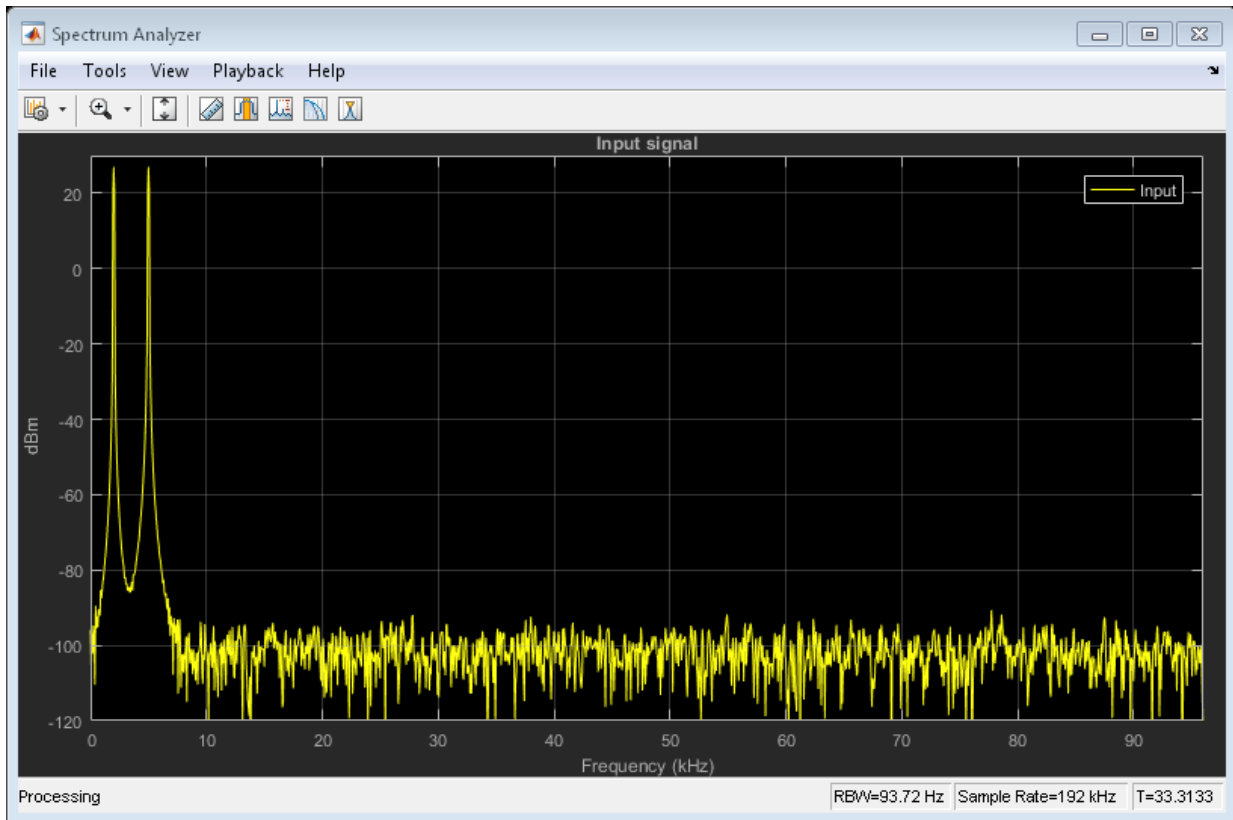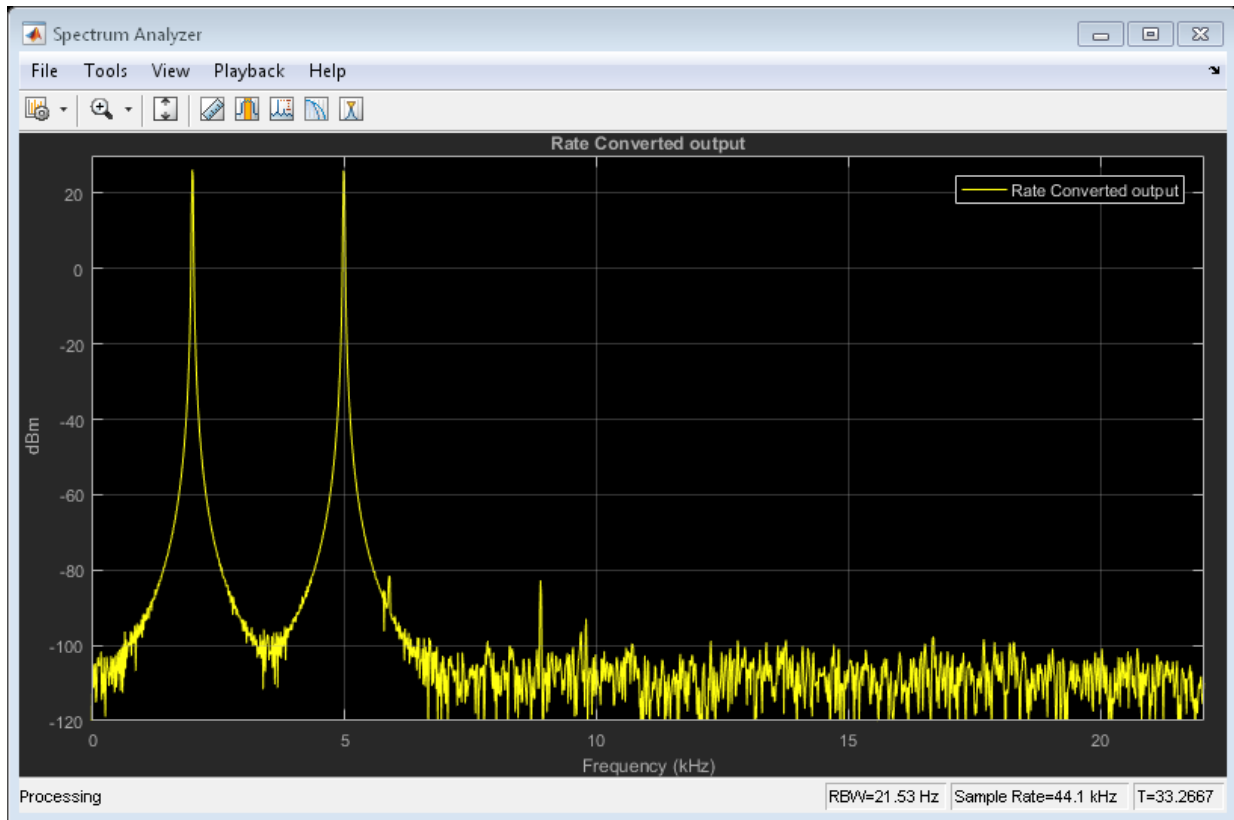
The spectrum shown is one-sided in the range [0 Fs/2]. For the spectrum analyzer showing the input, Fs/2 is 192000/2. For the spectrum analyzer showing the output, Fs/2 is 44100/2. Hence, the sample rate of the signal changed from 192 kHz to 44.1 kHz.

References

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

More About

- "Multirate Filters" on page 6-2
- "Multistage Filters" on page 6-6

Create Moving Average System object

Introduction

This example shows how to create a System object™ that implements a moving average filter. The example shows how to use the System object in MATLAB and Simulink through the `MATLAB System` block. `MovingAverageFilter` is a simple moving average System object filter, which computes the unweighted mean of the previous `WindowLength` input samples, where `WindowLength` is the length of the moving average window.

The System object accepts single-precision and double-precision 2-D input matrices. Each column of the input matrix is treated as an independent (1-D) channel. The first dimension of the input defines the length of the channel (or the input frame size). `MovingAverageFilter` independently computes the moving average of each input channel over time.

"System object Usage in MATLAB" on page 1-64 and "System object Usage in Simulink" on page 1-65 show how to use your System object with data.

Create the Class Definition

In the MATLAB **Home** tab select **New -> System Object -> Simulink Extension** to open a System object template. This template includes customizations of the System object for use in the `MATLAB System` block. You can edit the template file, using it as guideline, to create your own System object.

Replace all occurrences of `Untitled` in the file with `MovingAverageFilter` and save the file as `MovingAverageFilter.m` in a folder where you have write permission. You need to add this folder to the MATLAB path to use the System object. For convenience, the entire System object is provided in the file `dspdemo.MovingAverageFilter.m`. To view this file enter

```
edit dspdemo.MovingAverageFilter
```
at the MATLAB command prompt. The prefix `dspdemo` on `dspdemo.MovingAverageFilter` is a package name. Packages are special folders that can contain class folders, function and class definition files, and other packages. Package folders always begin with the + character such as `+dspdemo`. Packages define the scope of the contents of the package folder (that is, a namespace in which names must be unique). This means function and class names need to be unique only within the package. Using a package provides a means to organize classes and functions and to select names for these components that other packages can reuse. You do not have to use packages when creating your System object. For more information on packages in MATLAB, see "Packages Create Namespaces". The remainder of this example shows you how to create the `MovingAverageFilter` object from the System object template without using a package. However, you can also review and use the completed version, `dspdemo.MovingAverageFilter`.

Moving Average Filter Properties

The `MovingAverageFilter` object has one public property that controls the length of the moving average. Because the algorithm depends on this value being constant once data processing begins, the property is defined as nontunable. Additionally, the property only accepts real, positive integers. To ensure correct input, add the `PositiveInteger` attribute to the property. The default value of this property is 5.

```
properties (PositiveInteger, Nontunable)
        % WindowLength Moving average filter length
        %    Specify the length of the moving average filter as a
        %    scalar positive integer value. The default value of this
        %    property is 5.
```

```
        WindowLength = 5
end
```

The state of the moving average filter is defined with the `DiscreteState` attribute. Get the value of the state by calling the `getDiscreteState` method.

```
properties (DiscreteState)
        State;
end
```

A moving average filter is an FIR Filter with numerator coefficients equal to `ones(WindowLength,1)/WindowLength`. Because the coefficients do not change during the streaming operation, the coefficients are defined in a property for optimization purposes. Additionally, to ensure the coefficients are not accessible to users of the System object, use the `private` attribute.

```
properties (Access = private, Nontunable)
        pCoefficients;
end
```

Finally, the System object operates on a possibly multichannel input and therefore requires a property for the number of channels. This property is not accessible to users and therefore you use the `private` attribute. The value of this property is determined from the number of columns in the input.

```
properties (Access = private)
        % pNumChannels Property used to cache the number of input channels
        % (columns). Varying the number of channels during the streaming
        % operation is not allowed (since it modifes the number of required
        % states). The default of -1 means that the streaming operation has
        % not started yet (i.e. the number ofo channels is still unknown).
        pNumChannels = -1;
    end
```

Moving Average Filter Constructor

The System object constructor is a method that has the same name as the class (`MovingAverageFilter` in this example). Within that method, use the `setProperties` method to allow standard name-value pair handling at construction, `filt = MovingAverageFilter('WindowLength',10)`.

```
methods
        % Constructor
        function obj = MovingAverageFilter(varargin)
```

```
                % Support name-value pair arguments when constructing the
                % object.
                setProperties(obj,nargin,varargin{:});
            end
    end
```

Moving Average Filter Setup

The `setupImpl` method sets up the object and implements one-time initialization tasks. The filter coefficients are computed based on the specified window length. The filter's states are initialized to zero. Note that there are `WindowLength-1` states per input channel. If you would like to initialize the states to a custom value, you can create a public `InitialConditions` property and use the property value to set the object state (`obj.State`) in `setupImpl`. Finally, the number of channels is determined from the number of columns in the input.

```
function setupImpl(obj,x)
            numChannels = size(x,2);
            obj.State = zeros(obj.WindowLength-1,numChannels,'like',x);
            % Cache the number of channels
            obj.pNumChannels = numChannels;
            obj.pCoefficients = ones(1,obj.WindowLength)/obj.WindowLength;
end
```

Note: You must set `Access = protected` for this method.

Moving Average Filter Step

The object's algorithm is defined in the `stepImpl` method. The algorithm in `stepImpl` is executed when the user of the System object calls `step` at the command line. In this example, `stepImpl` calculates the output and updates the object's state values using the `filter` function.

```
function Y = stepImpl(obj,X)
    % Compute output and update state
    [Y,obj.State] = filter(obj.pCoefficients,1,X,obj.State);
end
```

Note: You must set `Access = protected` for this method.

Moving Average Filter Reset

The state reset equations are defined in the `resetImpl` method. In this example, the states are reset to zero. If you want to reset the states to a custom value, you can create a public `InitialConditions` property and use the property value to reset state in `resetImpl`.

```
function resetImpl(obj)
    obj.State(:) = 0;
end
```

Note: You must set `methods(Access = protected)` for this method.

Input Validation

`validateInputsImpl` validates inputs to the `step` method at initialization and at each subsequent call to `step` where the input attributes (such as dimensions, data type or complexity) change. In this example, `validattributes` ensures that the input is a 2-D matrix with floating-point data.

```
function validateInputsImpl(obj, u)
            validateattributes(u,{'double','single'}, {'2d',...
                'nonsparse'},'','input');
            % The number of input channels is not allowed to change. If
            % pNumChannels = -1. This means that the streaming operation
            % has not started yet (i.e. setupImpl has not been invoked
            % yet). Do not perform the check in that case.
            coder.internal.errorIf(obj.pNumChannels~=-1 && obj.pNumChannels ~= size(u,2
end
```

Note: You must set `methods(Access = protected)` for this method.

Object Saving and Loading

`saveObjectImpl` defines what property and state values are saved in a MAT-file when you call save on that object. If you do not define a `saveObjectImpl` method for your System object class, only public properties and properties with the `DiscreteState` attribute are saved. Save the state of an object only if the object is locked. When you load

the saved object, the object loads in that locked state. In this System object, the filter coefficients are saved if the object is locked.

```
function s = saveObjectImpl(obj)
            s = saveObjectImpl@matlab.System(obj);
            if isLocked(obj)
                s.pCoefficients  =  obj.pCoefficients;
                s.pNumChannels  =  obj.pNumChannels;
            end
end
```

`loadObjectImpl` defines what System object property and state values are loaded when you load a MAT-file. `loadObjectImpl` should correspond to your `saveObjectImpl` to ensure that all saved properties and data are loaded.

```
function loadObjectImpl(obj,s,wasLocked)
            if wasLocked
                obj.pCoefficients = s.pCoefficients;
                obj.pNumChannels = s.pNumChannels;
            end
            loadObjectImpl@matlab.System(obj,s,wasLocked);
end
```

Note: You must set `Access = protected` for this method.

System object Usage in MATLAB

This example uses the System object to remove noise from a noisy pulse sequence. The length of the moving average filter is 30 samples. If you are using the predefined `dspdemo.MovingAverageFilter`, substitute that name for `MovingAverageFilter` in the class constructor, for example `movingAverageFilter = dspdemo.MovingAverageFilter('WindowLength',30);`.

```
movingAverageFilter = MovingAverageFilter('WindowLength',30);
scope = dsp.TimeScope('SampleRate',1e3,...
    'TimeSpan',256 * .01,...
    'ShowGrid',true,...
    'NumInputPorts',2,...
    'LayoutDimensions',[2 1]);
for i=1:100
    input =  (1-2*randi([0 1],1)) * ones(256,1) + 0.5 * randn(256,1);
```

```
    output = step(movingAverageFilter,input);
    step(scope,input,output)
end
```

Simulink Customization Methods

You need to define a few more methods to be able to use the System object in a Simulink MATLAB System block. These methods are not required if you use the System object only in MATLAB. getOutputSizeImpl returns the sizes of each output port. For System objects with one input and one output and where you want the input and output sizes to be the same, you do not need to implement this method. In the case of MovingAverageFilter, there is one input and output and the size of each is the same. Therefore, remove this method from the class definition of MovingAverageFilter.

getDiscreteStateSpecificationImpl returns the size, data type, and complexity of a property. This property must be a discrete-state property. You must define this method if your System object has discrete-state properties and is used in the MATLAB System block. In this example, the method is used to define the State property.

```
function [sz,dt,cp] = getDiscreteStateSpecificationImpl(obj,~)
    inputSize = propagatedInputSize(obj,1);
    sz = [obj.WindowLength-1 inputSize(2)];
    dt =  propagatedInputDataType(obj,1);
    cp =  propagatedInputComplexity(obj,1);
end
```

Note: You must set Access = protected for this method.

System object Usage in Simulink

To use your System object in a Simulink model, drag a MATLAB System block from the User-Defined Functions library in Simulink to your model.

Open the block dialog box and set the System object name to MovingAverageFilter. The model ex_movingaverage_filter illustrates the use of the System object in Simulink to filter a noisy pulse sequence.

```
model = 'ex_movingaverage_filter';
open_system(model);
```
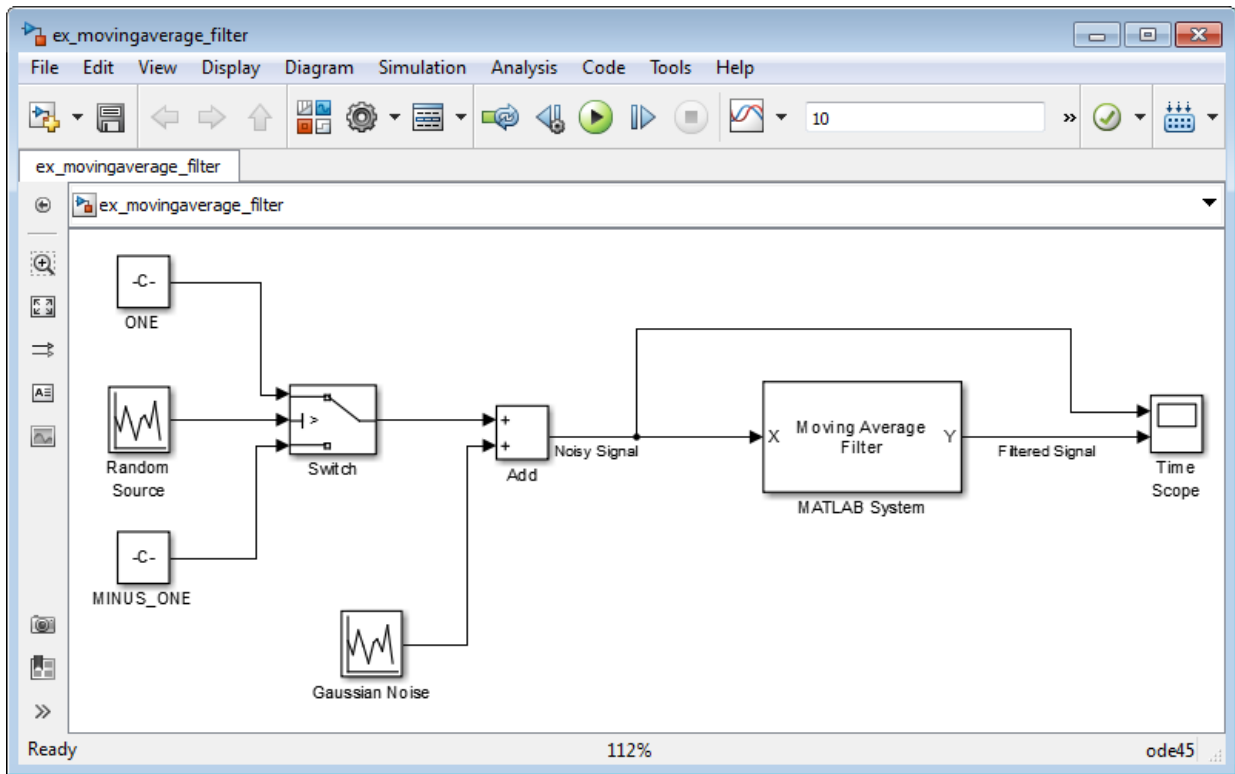
Run the model by clicking the **Run** button in the model or entering:

```
sim(model)
```

Tunable Lowpass Filtering of Noisy Input in Simulink

In this section...

This example shows how to filter a noisy chirp signal with a lowpass filter that has a tunable passband frequency. The filter is a `Variable Bandwidth IIR Filter` block with **Filter type** set to `Lowpass`. This type of filter enables you to change the passband frequency during simulation without having to redesign the whole filter. The filter algorithm recomputes the filter coefficients whenever the passband frequency changes.

Open Lowpass Filter Model

```
model = 'ex_tunable_chirp_lowpass';
open_system(model);
```
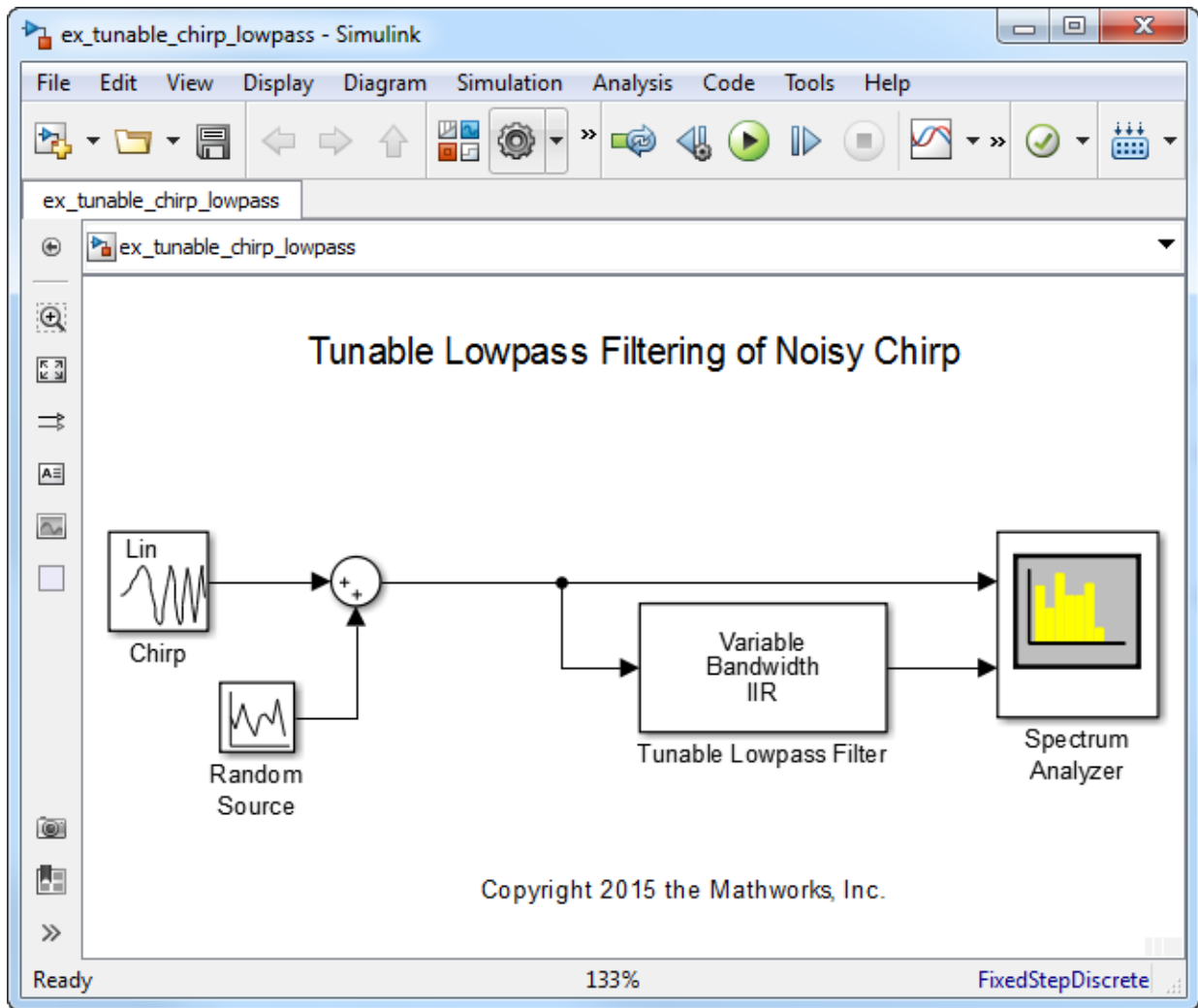
The input signal is a noisy chirp sampled at 44.1 kHz. The chirp has an initial frequency of 5000 Hz and a target frequency of 8000 Hz.

Block Parameters: Chirp

Chirp (mask) (link)

Linear, Logarithmic, and Quadratic modes generate a swept-frequency cosine with instantaneous frequency values specified by the frequency and time parameters. The Swept cosine mode generates a swept-frequency cosine with a linear instantaneous output frequency that may differ from the one specified by the frequency and time parameters.

Parameters

Frequency sweep: Linear

Sweep mode: Bidirectional

Initial frequency (Hz):

5000

Target frequency (Hz):

8000

Target time (s):

1

Sweep time (s):

1

Initial phase (rad):

0

Sample time:

1/44100

Samples per frame:

1024

Output data type: Double

The `Variable Bandwidth IIR Filter` block has a lowpass frequency response, with the passband frequency set to 2000 Hz.
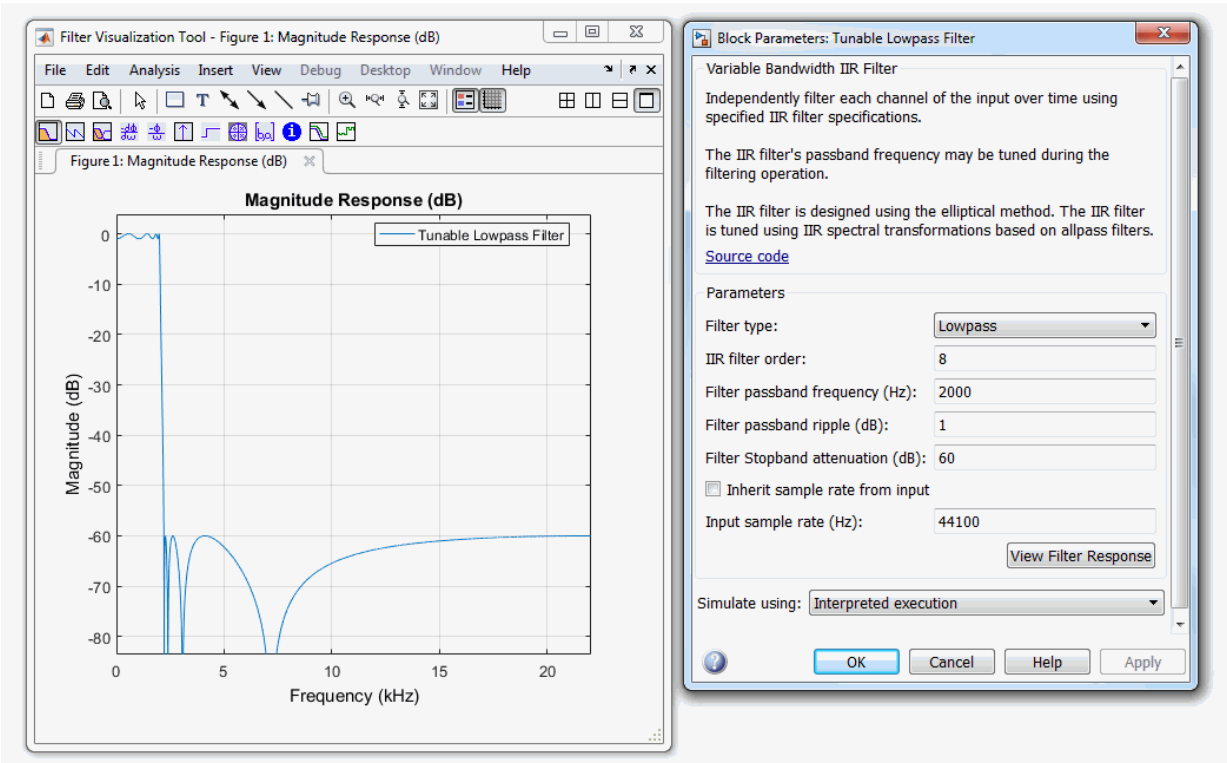
Simulate the Model

After you configure the block parameters, simulate the model. In the initial configuration, the chirp sweeps from 5000 Hz to 8000 Hz which falls in the stopband of the filter. When the chirp input passes through this filter, the filter attenuates the chirp.
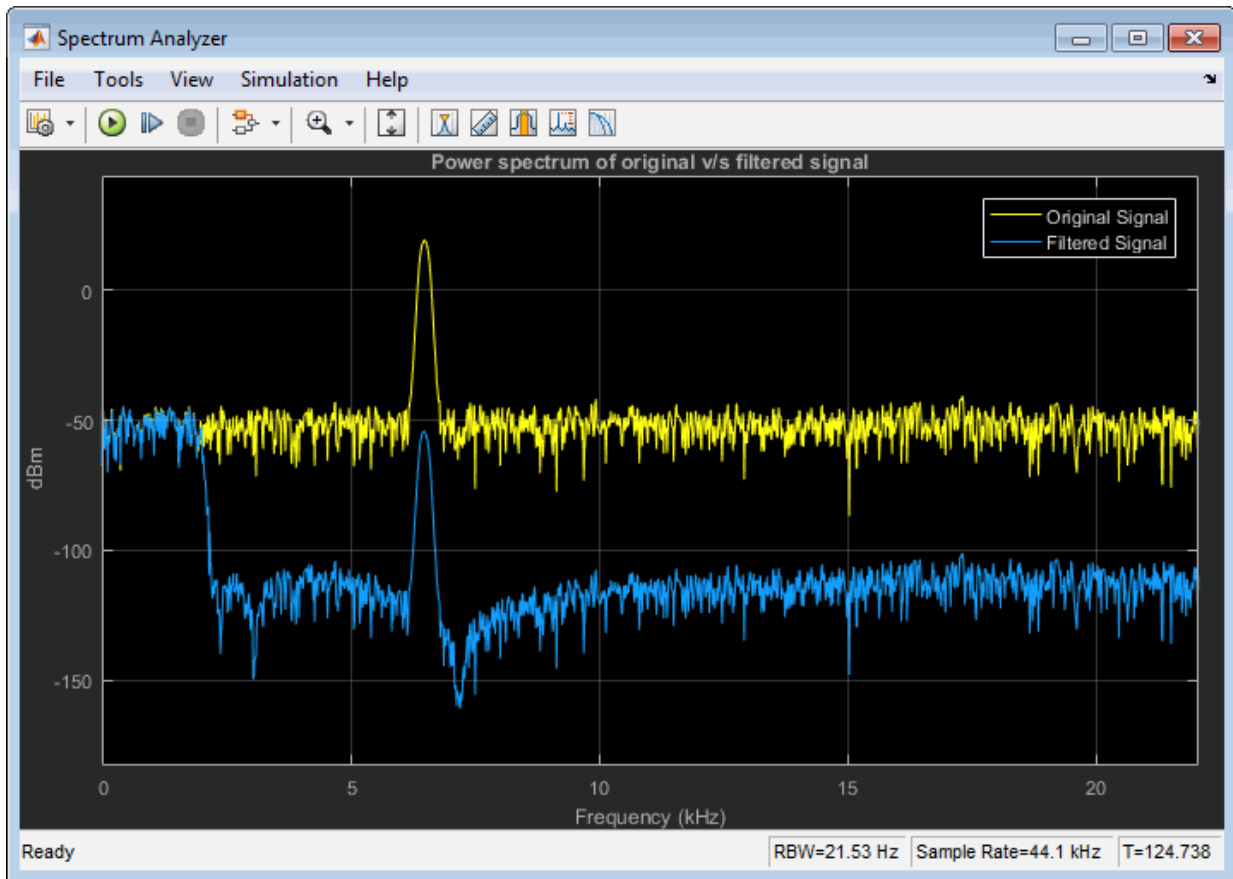
To tune the Passband frequency of the filter, in the `Variable Bandwidth IIR Filter` block dialog box, change **Filter passband frequency (Hz)** to 6000 Hz. Click **Apply** and the output of the Spectrum Analyzer changes immediately.

The chirp's sweep frequency ranges from 5000 to 8000 Hz. Part of this frequency range is in the passband and the remaining part is in the stopband. While in the filter's passband frequency, the chirp is unaffected.
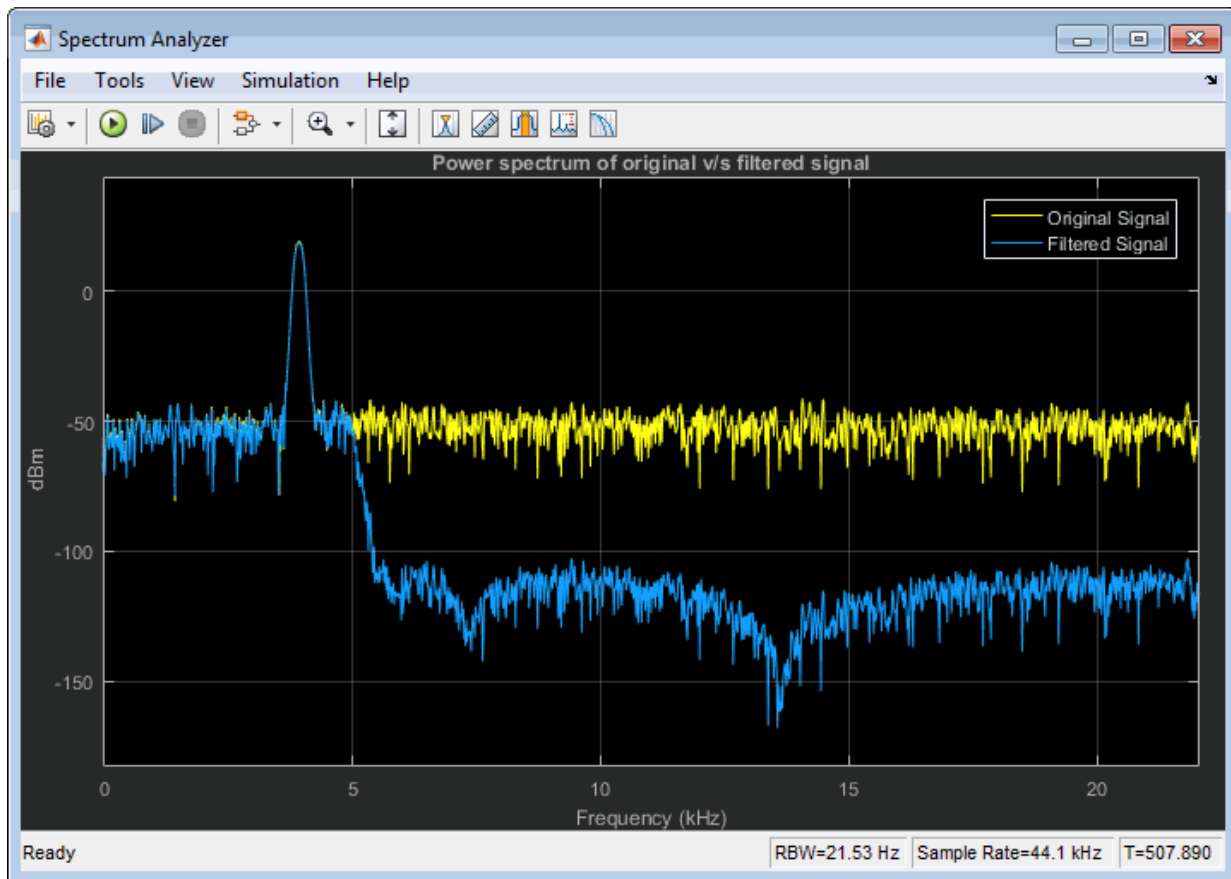
While in the filter's stopband frequency, the chirp is attenuated.
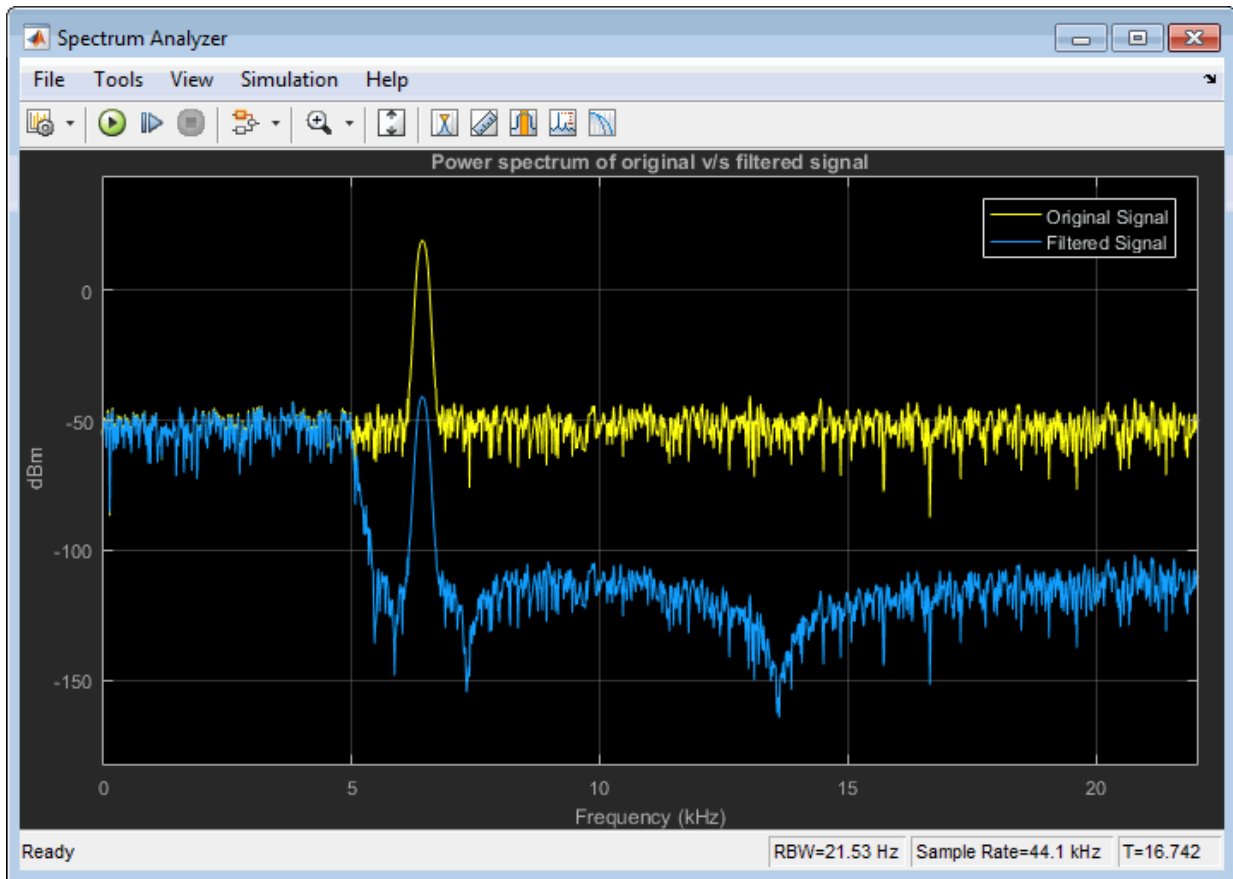
During simulation, you can tune any of the tunable parameters in the model and see the effect on the filtered output real time.

See Also

"Lowpass IIR Filter Design in Simulink" on page 1-28 | "Design Multirate Filters" on page 1-46 | "Filter Frames of a Noisy Sine Wave Signal in MATLAB" on page 1-7 | "Filter Frames of a Noisy Sine Wave Signal in Simulink" on page 1-10 | "Introduction to Streaming Signal Processing in MATLAB" on page 1-2

Signal Processing Algorithm Acceleration in MATLAB

In this section...
"Algorithm" on page 1-74
"Generate the MEX file using codegen" on page 1-76
"Generate the MEX file using dspunfold" on page 1-77

This example shows how to accelerate MATLAB code on a desktop. You can generate a MATLAB executable (MEX-function) from an entire MATLAB function or specific parts of the MATLAB function. When you run the MEX file instead of the original MATLAB code, simulation speed increases significantly. The speedup factor depends on the MATLAB constructs. For some operations such as matrix multiplication, you gain no speedup through MEX file. The original code is faster than the equivalent MEX file.

This page describes the workflow to generate a MEX file using the functions: `codegen` and `dspunfold`. To use these functions, you must have the MATLAB Coder™ software installed.

Algorithm

This example uses the MATLAB function `kalmanfilter`. This function models a kalman filter that tracks a noisy sinusoid signal. The input is a sine wave sampled at 192 kHz. Each frame of the signal contains 64000 samples. The input signal is noisy due to the addition of normally distributed random noise with zero mean and 0.02 standard deviation. The kalman filter uses the noisy signal and estimates the original sinusoid signal. The part of the algorithm which contains the kalman filter can be quite time consuming, especially if the frame size of the input signal is large. You can speed this part of the algorithm by replacing it with its equivalent MEX version.

```
type kalmanfilter
```

```
function kalmanfilter
% This function tracks a noisy sinusoid signal using Kalman filter
%
% Copyright 2015 The MathWorks, Inc.
%
% Create the Signal Source
Sig = dsp.SineWave('SamplesPerFrame',64000,'SampleRate',192000);
R = 0.02;          % Measurement noise covariance
```

```
% Setup the Kalman filter
Kalman = dsp.KalmanFilter('ProcessNoiseCovariance', 0.0001,...
            'MeasurementNoiseCovariance', R,...
            'InitialStateEstimate', 5,...
            'InitialErrorCovarianceEstimate', 1,...
            'ControlInputPort',false);

for i = 1 : 5
    trueVal = step(Sig);                  % Actual values
    noisyVal = trueVal + sqrt(R)*randn;   % Noisy measurements
    estVal = step(Kalman,noisyVal);       % Estimated by Kalman filter
end
end
```

Create a temporary directory. Run the function in the temporary directory and measure the time it takes to complete the simulation.

```
mydir = pwd; addpath(mydir); cd(tempdir);
tic;
kalmanfilter;
t1 = toc;
fprintf('MATLAB Simulation Time: %d\n',t1);

MATLAB Simulation Time: 1.039289e+01
```

Turn on the profiler to see the time it takes to execute each line in the kalmanfilter function.

```
profile on
kalmanfilter
profile viewer
```

In the profiler that opens up, click the kalmanfilter function. You can see that most of time is spent on the line: estVal = step(Kalman,noisyVal). If we can accelerate this line, the simulation time of the entire algorithm reduces significantly. In order to accelerate this line, replace this line with its MEX equivalent.

First step is to create a function foo from this line.

```
type foo


function estVal = foo(noisyVal,R)
% This is the entry point function that contains kalman filtering.
persistent Kalman
```

```
if (isempty(Kalman))
    Kalman = dsp.KalmanFilter('ProcessNoiseCovariance', 0.0001,...
        'MeasurementNoiseCovariance', R,...
        'InitialStateEstimate', 5,...
        'InitialErrorCovarianceEstimate', 1,...
        'ControlInputPort',false); %Create Kalman filter
end

estVal = step(Kalman,noisyVal);    % Estimated by Kalman filter
```

Then, generate an equivalent MEX version.

Generate the MEX file using `codegen`

Before calling codegen, make sure the inputs required by `foo` are defined in the workspace.

```
Sig = dsp.SineWave('SamplesPerFrame',64000,'SampleRate',192000); % Create the Signal So
R = 0.02;              % Measurement noise covariance
trueVal = step(Sig);                  % Actual values
 noisyVal = trueVal + sqrt(R)*randn; % Noisy measurements
```

Call `codegen` on `foo.m`. This generates a single-threaded mex file named `foo_mex`.

```
codegen foo -args {noisyVal,R}
```

Call `foo_mex` from the original function `kalmanfilter`. Rename this function as `kalmanfilter_codegen`.

```
type kalmanfilter_codegen
```

```
function kalmanfilter_codegen
% This function replaced the kalman fitlering with its
% MEX equivalent. The MEX file is generated using codegen.
%
% Copyright 2015 The MathWorks, Inc.
%
% Create Signal Source
Sig = dsp.SineWave('SamplesPerFrame',64000,'SampleRate',192000); % Create the Signal So
R = 0.02;          % Measurement noise covariance

for i = 1 : 5
```

```
    trueVal = step(Sig);              % Actual values
    noisyVal = trueVal + sqrt(R)*randn; % Noisy measurements
    estVal = foo_mex(noisyVal,R);
end
end
```

Run `kalmanfilter_codegen`. Measure the simulation time.

```
tic;
kalmanfilter_codegen
t2 = toc;
fprintf('Code Generation Simulation Time: %d\n',t2);
fprintf('Speedup Factor with codegen: %6.2f\n',t1/t2);

Code Generation Simulation Time: 5.503122e-02
Speedup Factor with codegen: 188.85
```

See the profiler output.

```
profile on
kalmanfilter_codegen
profile viewer
```

You can see that the simulation time for `kalmanfilter_codegen` is much lower than the simulation time for `kalmanfilter`.

On the profiler window, click `kalmanfilter_codegen`. Time to run `estVal = foo_mex(noisyVal,R)` has significantly reduced.

Generate the MEX file using `dspunfold`

Next, call `dspunfold` on `foo`. This generates a multithreaded mex file `foo_mt` using the concept of DSP unfolding.

```
dspunfold foo -args {noisyVal,coder.Constant(R)} -s auto


State length: [autodetect] frames, Repetition: 1, Output latency: 16 frames, Threads: 8
Analyzing: foo.m
Creating single-threaded MEX file: foo_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 1 frames ... Sufficient
Minimal state length is 1 frames
```

```
Creating multi-threaded MEX file: foo_mt.mexw64
Creating analyzer file: foo_analyzer.p
```

Call this mex file from the original function `kalmanfilter`. Rename this function as `kalmanfilter_dspunfold`.

type `kalmanfilter_dspunfold`

```
function kalmanfilter_dspunfold
% This function replaced the kalman fitlering with its
% MEX equivalent. The MEX file is generated using dspunfold.
%
% Copyright 2015 The MathWorks, Inc.
%
% Create Signal Source
Sig = dsp.SineWave('SamplesPerFrame',64000,'SampleRate',192000);
R = 0.02;          % Measurement noise covariance
for i = 1 : 5
    trueVal = step(Sig);               % Actual values
    noisyVal = trueVal + sqrt(R)*randn; % Noisy measurements
    estVal = foo_mt(noisyVal,R);
end
end
```

Run `kalmanfilter_dspunfold`. Measure the simulation time.

```
tic;
kalmanfilter_dspunfold
t3 = toc;
fprintf('dspunfold Simulation Time: %d\n',t3);
fprintf('Speedup Factor with dspunfold: %6.2f\n',t1/t3);
```

```
dspunfold Simulation Time: 1.165916e-01
Speedup Factor with dspunfold:  89.14
```

See the profiler output profile on `kalmanfilter_dspunfold` profile viewer. You can see that time to run `estVal = foo_mt(noisyVal,R)` is even lesser. To increase the speed up further, increase the repetition factor. The tradeoff is that the output latency also increases.

```
dspunfold foo -args {noisyVal,coder.Constant(R)} -s auto -r 3
tic;
kalmanfilter_dspunfold
```

```
t4 = toc;
fprintf('dspunfold Simulation Time with repetition factor: %d\n',t4);
fprintf('Speedup factor with dspunfold with repetition factor: %6.2f\n',t1/t4);
```

```
State length: [autodetect] frames, Repetition: 3, Output latency: 48 frames, Threads: 8
Analyzing: foo.m
Creating single-threaded MEX file: foo_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 1 frames ... Sufficient
Minimal state length is 1 frames
Creating multi-threaded MEX file: foo_mt.mexw64
Creating analyzer file: foo_analyzer.p
dspunfold Simulation Time with repetition factor: 5.082688e-02
Speedup factor with dspunfold with repetition factor: 204.48
```

See the profiler output profile on `kalmanfilter_dspunfold` profile viewer. For this particular algorithm, the speed up gain is the largest with dspunfold since the MEX file generated uses the multicore CPU architecture of the host computer. codegen comes next, still performing better than the original simulation.

After running the above example, you can move the control to your original directory using:

```
cd(mydir)
```

Multi-Threaded MEX File Generation Using DSP Unfolding

This example shows how to use the `dspunfold` function to generate a multi-threaded MEX file from a MATLAB function using DSP unfolding technology. The MATLAB function can contain an algorithm which is stateless (has no states) or stateful (has states).

NOTE: The following example assumes that the current host computer has at least two physical CPU cores. The presented screenshots, speedup, and latency values were collected using a host computer with eight physical CPU cores.

Required MathWorks® products:

- DSP System Toolbox
- MATLAB Coder

Use `dspunfold` with a MATLAB Function Containing a Stateless Algorithm

Consider the MATLAB function `dspunfoldDCTExample`. This function computes the DCT of an input signal and returns the value and index of the maximum energy point.

```
function [peakValue,peakIndex] = dspunfoldDCTExample(x)
% Stateless MATLAB function computing the dct of a signal (e.g. audio), and
% returns the value and index of the highest energy point

% Copyright 2015 The MathWorks, Inc.

X = dct(x);
[peakValue,peakIndex] = max(abs(X));
```

To accelerate the algorithm, a common approach is to generate a MEX file using the `codegen` function. This example shows how to do so when using an input of 4096 doubles. The generated MEX file, `dspunfoldDCTExample_mex`, is singlethreaded.

```
codegen dspunfoldDCTExample -args {(1:4096)'}
```

To generate a multi-threaded MEX file, use the `dspunfold` function. The argument `-s 0` indicates that the algorithm in `dspunfoldDCTExample` is stateless.

```
dspunfold dspunfoldDCTExample -args {(1:4096)'} -s O
```

This command generates these files:

- Multi-threaded MEX file `dspunfoldDCTExample_mt`
- Single-threaded MEX file `dspunfoldDCTExample_st`, which is identical to the MEX file obtained using the `codegen` function
- Self-diagnostic analyzer function `dspunfoldDCTExample_analyzer`

Additional three MATLAB files are also generated, containing the help for each of the above files.

To measure the speedup of the multi-threaded MEX file relative to the single-threaded MEX file, see the example function `dspunfoldBenchmarkDCTExample`.

```
function dspunfoldBenchmarkDCTExample
% Function used to measure the speedup of the multi-threaded MEX file
% dspunfoldDCTExample_mt obtained using dspunfold vs the single-threaded MEX
% file dspunfoldDCTExample_st.

% Copyright 2015 The MathWorks, Inc.

clear dspunfoldDCTExample_mt;  % for benchmark precision purpose
numFrames = 1e5;
inputFrame = (1:4096)';

% exclude first run from timing measurements
dspunfoldDCTExample_st(inputFrame);
tic;  % measure execution time for the single-threaded MEX
for frame = 1:numFrames
    dspunfoldDCTExample_st(inputFrame);
end
timeSingleThreaded = toc;

% exclude first run from timing measurements
dspunfoldDCTExample_mt(inputFrame);
tic;  % measure execution time for the multi-threaded MEX
for frame = 1:numFrames
    dspunfoldDCTExample_mt(inputFrame);
end
timeMultiThreaded = toc;
fprintf('Speedup = %.1fx\n',timeSingleThreaded/timeMultiThreaded);
```

dspunfoldBenchmarkDCTExample measures the execution time taken by dspunfoldDCTExample_st and dspunfoldDCTExample_mt to process numFrames frames. Finally, it prints the speedup, which is the ratio between the multi-threaded MEX file execution time and single-threaded MEX file execution time. Run the example.

```
dspunfoldBenchmarkDCTExample;
```

```
Speedup = 4.7x
```

To improve the speedup even more, increase the repetition value. To modify the repetition value, use the -r flag. For more information on the repetition value, see the dspunfold function reference page. For an example on how to specify the repetition value, see the section 'Using dspunfold with a MATLAB Function Containing a Stateful Algorithm'.

DSP unfolding generates a multi-threaded MEX file, which buffers multiple signal frames and then processes these frames simultaneously, using multiple cores. This process introduces some deterministic output latency. Executing help dspunfoldDCTExample_mt displays more information about the multi-threaded MEX file, including the value of the output latency. For this example, the output of the multi-threaded MEX file has a latency of 16 frames relative to its input, which is not the case for the single-threaded MEX file.

Run dspunfoldShowLatencyDCTExample example. The generated plot displays the outputs of the single-threaded and multi-threaded MEX files. Notice that the output of the multi-threaded MEX is delayed by 16 frames, relative to that of the single-threaded MEX.

```
dspunfoldShowLatencyDCTExample;
```
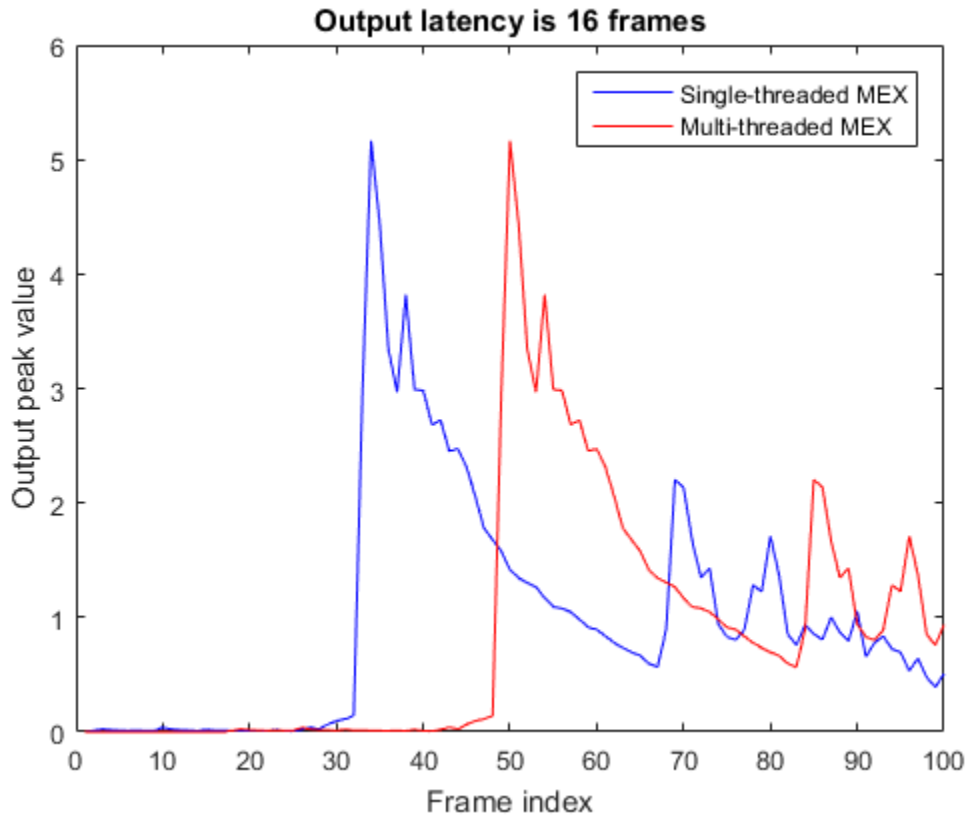
Using `dspunfold` with a MATLAB Function Containing a Stateful Algorithm

The MATLAB function `dspunfoldFIRExample` executes two FIR filters.

```
type dspunfoldFIRExample
```

```
function y = dspunfoldFIRExample(u,c1,c2)
% Stateful MATLAB function executing two FIR filters

% Copyright 2015 The MathWorks, Inc.
```

```
persistent FIRSTFIR SECONDFIR
if isempty(FIRSTFIR)
    FIRSTFIR = dsp.FIRFilter('NumeratorSource','Input port');
    SECONDFIR = dsp.FIRFilter('NumeratorSource','Input port');
end
t = step(FIRSTFIR,u,c1);
y = step(SECONDFIR,t,c2);
```

To build the multi-threaded MEX file, you must provide the state length corresponding to the two FIR filters. Specify -s 1 to indicate that the state length does not exceed 1 frame.

```
firCoeffs1 = fir1(127,0.8);
firCoeffs2 = fir1(256,0.2,'High');
dspunfold dspunfoldFIRExample -args {(1:2048)',firCoeffs1,firCoeffs2} -s 1
```

Executing this code generates:

- Multi-threaded MEX file dspunfoldFIRExample_mt
- Single-threaded MEX file dspunfoldFIRExample_st
- Self-diagnostic analyzer function dspunfoldFIRExample_analyzer
- The corresponding MATLAB help files for these three files

The output latency of the multi-threaded MEX file is 16 frames. To measure the speedup, execute dspunfoldBenchmarkFIRExample.

```
dspunfoldBenchmarkFIRExample;
```

```
Speedup = 3.9x
```

To improve the speedup of the multi-threaded MEX file even more, specify the exact state length in samples. To do so, you must specify which input arguments to dspunfoldFIRExample are frames. In this example, the first input is a frame because the elements of this input are sequenced in time. Therefore it can be further divided into subframes. The last two inputs are not frames because the FIR filters coefficients cannot be subdivided without changing the nature of the algorithm. The value of the dspunfoldFIRExample MATLAB function state length is the sum of the state length of the two FIR filters (127 + 256 = 383). Using the -f argument, mark the first input argument as true (frame), and the last two input arguments as false (nonframes)

```
dspunfold dspunfoldFIRExample -args {(1:2048)',firCoeffs1,firCoeffs2} -s 383 -f [true,t
```

Again, measure the speedup for the resulting multi-threaded MEX using the dspunfoldBenchmarkFIRExample function. Notice that the speedup increased because

the exact state length was specified in samples, and `dspunfold` was able to subdivide
the frame inputs.

```
dspunfoldBenchmarkFIRExample;
```

```
Speedup = 6.3x
```

Oftentimes, the speedup can be increased even more by increasing the repetition (-
r) provided when invoking `dspunfold`. The default repetition value is 1. When you
increase this value, the multi-threaded MEX buffers more frames internally before the
processing starts. Increasing the repetition factor increases the efficiency of the multi-
threading, but at the cost of a higher output latency.

```
dspunfold dspunfoldFIRExample -args {(1:2048)',firCoeffs1,firCoeffs2} -s 383 -f [true,t
```

Again, measure the speedup for the resulting multi-threaded MEX, using the
`dspunfoldBenchmarkFIRExample` function. Speedup increases again, but
the output latency is now 80 frames. The general output latency formula is
`2*Threads*Repetition` frames. In these examples, the number of `Threads` is equal to
the number of physical CPU cores.

```
dspunfoldBenchmarkFIRExample;
```

```
Speedup = 7.7x
```

Detecting State Length Automatically

To request that `dspunfold` autodetect the state length, specify `-s auto`. This option
generates an efficient multi-threaded MEX file, but with a significant increase in the
generation time, due to the extra analysis that it requires.

```
dspunfold dspunfoldFIRExample -args {(1:2048)',firCoeffs1,firCoeffs2} -s auto -f [true,
```

```
State length: [autodetect] samples, Repetition: 5, Output latency: 40 frames, Threads:
Analyzing: dspunfoldFIRExample.m
Creating single-threaded MEX file: dspunfoldFIRExample_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 2048 samples ... Sufficient
Checking 1024 samples ... Sufficient
Checking 512 samples ... Sufficient
Checking 256 samples ... Insufficient
Checking 384 samples ... Sufficient
```

```
Checking 320 samples ... Insufficient
Checking 352 samples ... Insufficient
Checking 368 samples ... Insufficient
Checking 376 samples ... Insufficient
Checking 380 samples ... Insufficient
Checking 382 samples ... Insufficient
Checking 383 samples ... Sufficient
Minimal state length is 383 samples
Creating multi-threaded MEX file: dspunfoldFIRExample_mt.mexw64
Creating analyzer file: dspunfoldFIRExample_analyzer.p
```

dspunfold checks different state lengths, using as inputs the values provided with the -args option. The function aims to find the minimum state length for which the outputs of the multi-threaded MEX and single-threaded MEX are the same. Notice that it found 383, as the minimal state length value, which matches the expected value, manually computed before.

Verify Generated Multi-Threaded MEX Using the Generated Analyzer

When creating a multi-threaded MEX file using dspunfold, the single-threaded MEX file is also created along with an analyzer function. For the stateful example in the previous section, the name of the analyzer is dspunfoldFIRExample_analyzer.

The goal of the analyzer is to provide a quick way to measure the speedup of the multi-threaded MEX relative to the single-threaded MEX, and also to check if the outputs of the multi-threaded MEX and single-threaded MEX match. Outputs usually do not match when an incorrect state length value is specified.

Execute the analyzer for the multi-threaded MEX file, dspunfoldFIRExample_mt, generated previously using the -s auto option.

```
firCoeffs1_1 = fir1(127,0.8);
firCoeffs1_2 = fir1(127,0.7);
firCoeffs1_3 = fir1(127,0.6);
firCoeffs2_1 = fir1(256,0.2,'High');
firCoeffs2_2 = fir1(256,0.1,'High');
firCoeffs2_3 = fir1(256,0.3,'High');
dspunfoldFIRExample_analyzer((1:2048*3)',[firCoeffs1_1;firCoeffs1_2;firCoeffs1_3],[fir

Analyzing multi-threaded MEX file dspunfoldFIRExample_mt.mexw64  ...
Latency = 80 frames
Speedup = 7.8x
```

Each input to the analyzer corresponds to the inputs of the `dspunfoldFIRExample_mt` MEX file. Notice that the length (first dimension) of each input is greater than the expected length. For example, `dspunfoldFIRExample_mt` expects a frame of 2048 doubles for its first input, while 2048*3 samples were provided to `dspunfoldFIRExample_analyzer`. The analyzer interprets this input as 3 frames of 2048 samples. The analyzer alternates between these 3 input frames circularly while checking if the outputs of the multi-threaded and single-threaded MEX files match.

The table shows the inputs used by the analyzer at each step of the numerical check. The total number of steps invoked by the analyzer is 240 or `3*latency`, where `latency` is 80 in this case.

	Input 1	Input 2	Input 3
Step 1	`(1:2048)'`	`firCoeffs1_1`	`firCoeffs2_1`
Step 2	`(2049:4096)'`	`firCoeffs1_2`	`firCoeffs2_2`
Step 3	`(4097:6144)'`	`firCoeffs1_3`	`firCoeffs2_3`
Step 4	`(1:2048)'`	`firCoeffs1_1`	`firCoeffs2_1`
...	...	...	...

NOTE: For the analyzer to correctly check for the numerical match between the multi-threaded MEX and single-threaded MEX, provide at least two frames with different values for each input. For inputs that represent parameters, such as filter coefficients, the frames can have the same values for each input. In this example, you could have specified a single set of coefficients for the second and third inputs.

References

[1] Unfolding (DSP implementation)

See Also
"Generate a Multi-Threaded MEX File from a MATLAB Function using DSP Unfolding" | "Workflow for Generating a Multi-Threaded MEX File using dspunfold" on page 10-22 | "Why Does the Analyzer Choose the Wrong State Length?" on page 10-31 | "How Is dspunfold Different from parfor?" on page 10-20 | `dspunfold`

Fixed-Point Filter Design in MATLAB

This example shows how to design filters for use with fixed-point input. The example analyzes the effect of coefficient quantization on filter design. You must have the Fixed-Point Designer software™ to run this example.

Introduction

Fixed-point filters are commonly used in digital signal processors where data storage and power consumption are key limiting factors. With the constraints you specify, DSP System Toolbox software allows you to design efficient fixed-point filters. The filter for this example is a lowpass equiripple FIR filter. Design the filter first for floating-point input to obtain a baseline. You can use this baseline for comparison with the fixed-point filter.

FIR Filter Design

The lowpass FIR filter has the following specifications:

- Sample rate: 2000 Hz

- Center frequency: 450 Hz

- Transition width: 100 Hz

- Equiripple design

- Maximum 1 dB of ripple in the passband

- Minimum 80 dB of attenuation in the stopband

```
samplingFrequency = 2000;
centerFrequency = 450;
transitionWidth = 100;
passbandRipple = 1;
stopbandAttenuation = 80;

designSpec = fdesign.lowpass('Fp,Fst,Ap,Ast',...
    centerFrequency-transitionWidth/2, ...
    centerFrequency+transitionWidth/2, ...
    passbandRipple,stopbandAttenuation, ...
    samplingFrequency);
LPF = design(designSpec,'equiripple','SystemObject',true)
```

```
LPF =

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1x52 double]
    InitialConditions: 0

  Use get to show all properties
```

View the baseline frequency response. The dotted red lines show the design specifications used to create the filter.

```
fvtool(LPF)
```
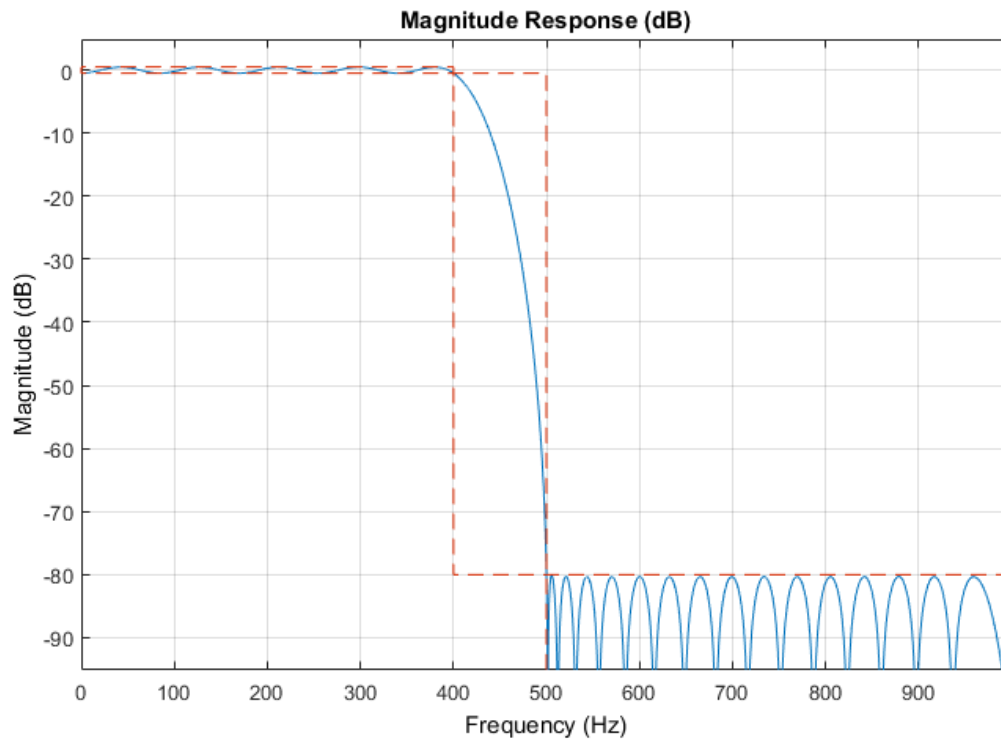
Full-Precision Fixed-Point Operation

The fixed-point properties of the filter are contained in the `Fixed-point properties` section in the display of the object. By default, the filter uses full-precision arithmetic to deal with fixed-point inputs. With full-precision arithmetic, the filter uses as many bits for the product, accumulator, and output as needed to prevent any overflow or rounding. If you do not want to use full-precision arithmetic, you can set the `FullPrecisionOverride` property to `false` and then set the product, accumulator, and output data types independently.

```
rng default
inputWordLength = 16;
fixedPointInput = fi(randn(100,1),true,inputWordLength);
floatingPointInput = double(fixedPointInput);
floatingPointOutput = step(LPF,floatingPointInput);

release(LPF)
fullPrecisionOutput = step(LPF,fixedPointInput);
norm(floatingPointOutput-double(fullPrecisionOutput),'inf')


ans =

   6.8994e-05
```

The result of full-precision fixed-point filtering comes very close to floating point, but the results are not exact. The reason for this is coefficient quantization. In the fixed-point filter, the `CoefficientsDataType` property has the same word length (16) for the coefficients and the input. The frequency response of the filter in full-precision mode shows this more clearly. The `measure` function shows that the minimum stopband attenuation of this filter with quantized coefficients is 76.6913 dB, less than the 80 dB specified for the floating-point filter.

```
LPF.CoefficientsDataType
fvtool(LPF)
measure(LPF)


ans =

Same word length as input
```

```
ans =

Sample Rate       : 2 kHz
Passband Edge     : 400 Hz
3-dB Point        : 416.2891 Hz
6-dB Point        : 428.1081 Hz
Stopband Edge     : 500 Hz
Passband Ripple   : 0.96325 dB
Stopband Atten.   : 76.6913 dB
Transition Width  : 100 Hz
```

Magnitude Response (dB)
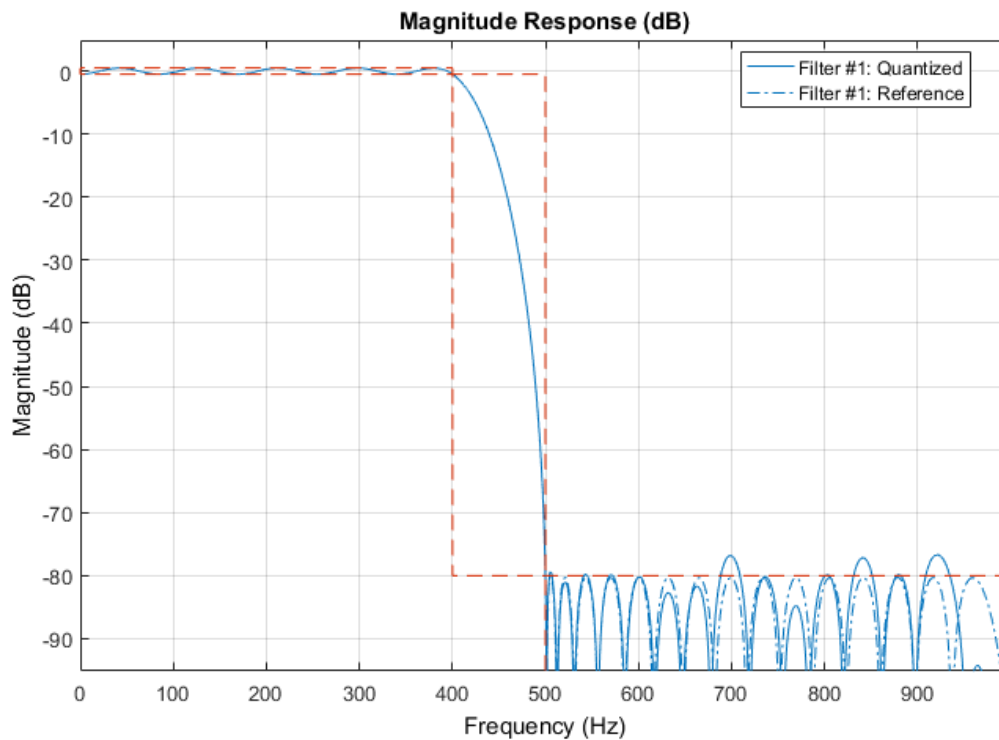
The filter was last used with fixed-point input and is still in a locked state. For that reason, `fvtool` displays the fixed-point frequency response. The dash-dot response is that of the reference floating-point filter, and the solid plot is the response of the filter that was used with fixed-point input. The desired frequency response cannot be matched

because the coefficient word length has been restricted to 16 bits. This accounts for the difference between the floating-point and fixed-point designs. Increasing the number of bits allowed for the coefficient word length makes the quantization error smaller and enables you to match the design requirement for 80 dB of stopband attenuation. Use a coefficient word length of 24 bits to achieve an attenuation of 80.1275 dB.

```
LPF24bitCoeff = design(designSpec,'equiripple','SystemObject',true);
LPF24bitCoeff.CoefficientsDataType = 'Custom';
coeffNumerictype = numerictype(fi(LPF24bitCoeff.Numerator,true,24));
LPF24bitCoeff.CustomCoefficientsDataType = numerictype(true, ...
          coeffNumerictype.WordLength,coeffNumerictype.FractionLength);
fullPrecisionOutput32bitCoeff = step(LPF24bitCoeff,fixedPointInput);
norm(floatingPointOutput-double(fullPrecisionOutput32bitCoeff),'inf')

fvtool(LPF24bitCoeff)
measure(LPF24bitCoeff)


ans =

   4.1077e-07


ans =

Sample Rate      : 2 kHz
Passband Edge    : 400 Hz
3-dB Point       : 416.2901 Hz
6-dB Point       : 428.1091 Hz
Stopband Edge    : 500 Hz
Passband Ripple  : 0.96329 dB
Stopband Atten.  : 80.1275 dB
Transition Width : 100 Hz
```
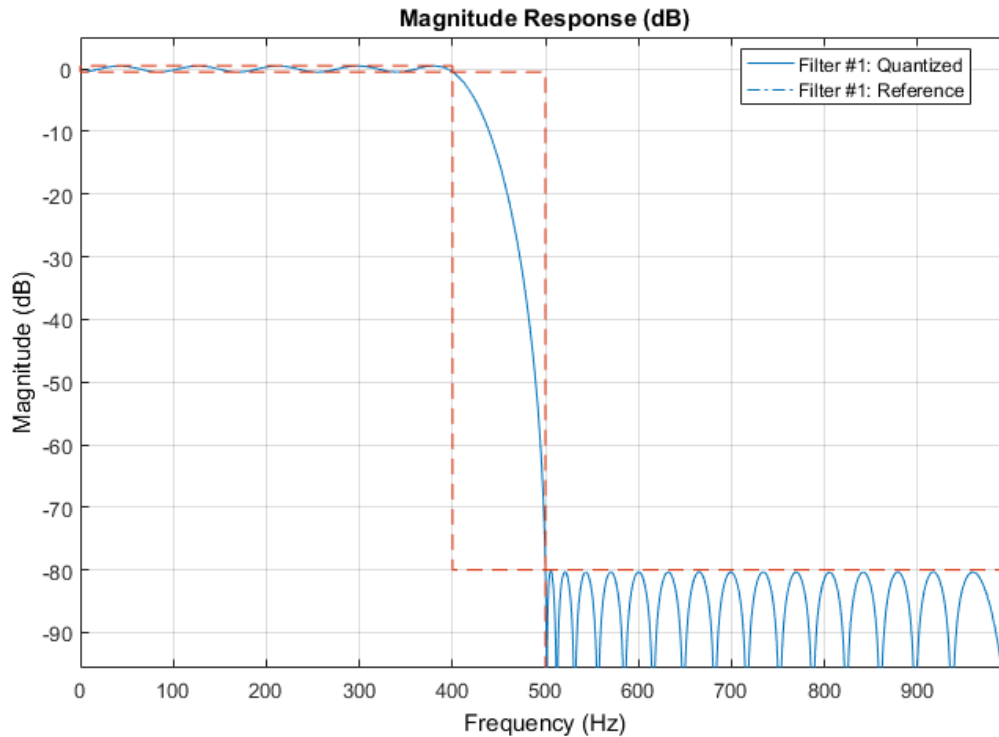
Design Parameters and Coefficient Quantization

In many fixed-point design applications, the coefficient word length is not flexible. For example, supposed you are restricted to work with 14 bits. In such cases, the requested minimum stopband attenuation of 80 dB cannot be reached. A filter with 14-bit coefficient quantization can achieve a minimum attenuation of only 67.2987 dB.

```
LPF14bitCoeff = design(designSpec,'equiripple','SystemObject',true);
coeffNumerictype = numerictype(fi(LPF14bitCoeff.Numerator,true,14));
LPF14bitCoeff.CoefficientsDataType = 'Custom';
LPF14bitCoeff.CustomCoefficientsDataType = numerictype(true, ...
          coeffNumerictype.WordLength,coeffNumerictype.FractionLength);
measure(LPF14bitCoeff,'Arithmetic','fixed')
```

```
ans =
```

```
Sample Rate      : 2 kHz
Passband Edge    : 400 Hz
3-dB Point       : 416.2939 Hz
6-dB Point       : 428.1081 Hz
Stopband Edge    : 500 Hz
Passband Ripple  : 0.96405 dB
Stopband Atten.  : 67.2987 dB
Transition Width : 100 Hz
```

For FIR filters in general, each bit of coefficient word length provides approximately 5 dB of stopband attenuation. Accordingly, if your filter's coefficients are always quantized to 14 bits, you can expect the minimum stopband attenuation to be only around 70 dB. In such cases, it is more practical to design the filter with stopband attenuation less than 70 dB. Relaxing this requirement results in a design of lower order.

```
designSpec.Astop = 60;
LPF60dBStopband = design(designSpec,'equiripple','SystemObject',true);
LPF60dBStopband.CoefficientsDataType = 'Custom';
coeffNumerictype = numerictype(fi(LPF60dBStopband.Numerator,true,14));
LPF60dBStopband.CustomCoefficientsDataType = numerictype(true, ...
          coeffNumerictype.WordLength,coeffNumerictype.FractionLength);
measure(LPF60dBStopband,'Arithmetic','fixed')

order(LPF14bitCoeff)
order(LPF60dBStopband)


ans =

Sample Rate      : 2 kHz
Passband Edge    : 400 Hz
3-dB Point       : 419.3391 Hz
6-dB Point       : 432.9718 Hz
Stopband Edge    : 500 Hz
Passband Ripple  : 0.92801 dB
Stopband Atten.  : 59.1829 dB
Transition Width : 100 Hz


ans =

    51
```

```
ans =

    42
```

The filter order decreases from 51 to 42, implying that fewer taps are required to implement the new FIR filter. If you still want a high minimum stopband attenuation without compromising on the number of bits for coefficients, you must relax the other filter design constraint: the transition width. Increasing the transition width might enable you to get higher attenuation with the same coefficient word length. However, it is almost impossible to achieve more than 5 dB per bit of coefficient word length, even after relaxing the transition width.

```
designSpec.Astop = 80;
transitionWidth = 200;
designSpec.Fpass = centerFrequency-transitionWidth/2;
designSpec.Fstop = centerFrequency+transitionWidth/2;
LPF300TransitionWidth = design(designSpec,'equiripple', ...
                                'SystemObject',true);
LPF300TransitionWidth.CoefficientsDataType = 'Custom';
coeffNumerictype = numerictype(fi(LPF300TransitionWidth.Numerator, ...
                            true, 14));
LPF300TransitionWidth.CustomCoefficientsDataType = numerictype(true, ...
        coeffNumerictype.WordLength,coeffNumerictype.FractionLength);
measure(LPF300TransitionWidth,'Arithmetic','fixed')
```

```
ans =

Sample Rate        : 2 kHz
Passband Edge      : 350 Hz
3-dB Point         : 385.4095 Hz
6-dB Point         : 408.6465 Hz
Stopband Edge      : 550 Hz
Passband Ripple    : 0.74045 dB
Stopband Atten.    : 74.439 dB
Transition Width   : 200 Hz
```

As you can see, increasing the transition width to 200 Hz allows 74.439 dB of stopband attenuation with 14-bit coefficients, compared to the 67.2987 dB attained when the transition width was set to 100 Hz. An added benefit of increasing the transition width is that the filter order also decreases, in this case from 51 to 27.

```
order(LPF300TransitionWidth)

ans =

    27
```

Visualizing Multiple Signals Using Logic Analyzer

This example shows how to visualize multiple signals of a programmable FIR filter using a logic analyzer. For more information on the model used in this example and how to configure the model to generate HDL code, see "Generate HDL Code for Programmable FIR Filter".

Model Programmable FIR Filter

Open the example model.

```
modelname = 'dspprogfirhdl';
open_system(modelname);
```
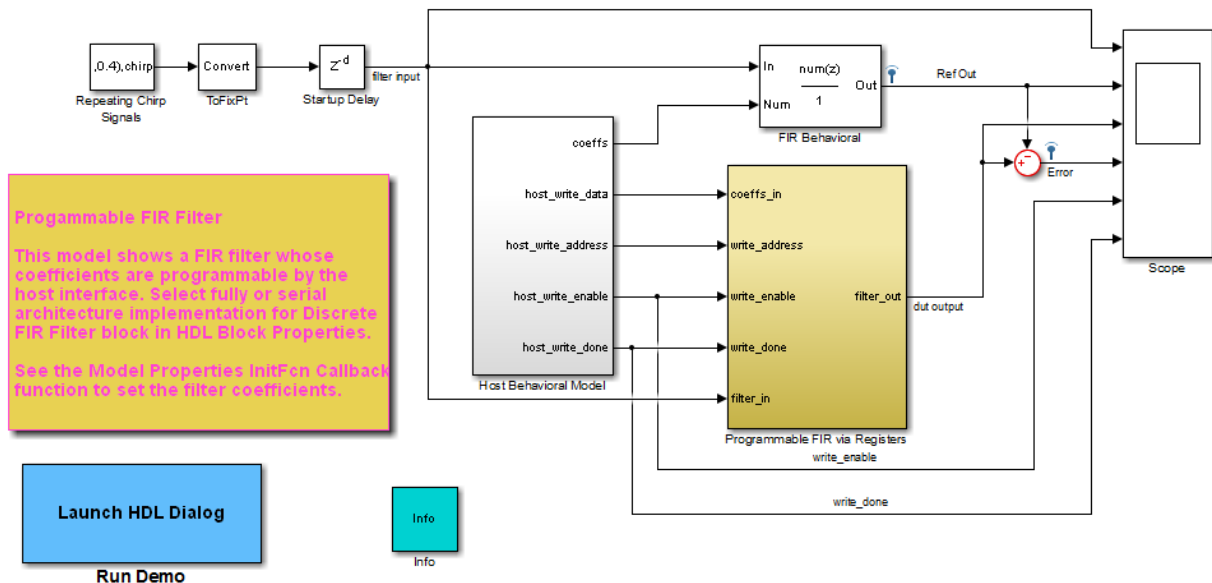

Copyright 2011-2012 The MathWorks, Inc.

Consider two FIR filters, one with a lowpass response and the other with a highpass response. The coefficients can be specified using the **InitFcn*** callback function. To specify the callback, select **File > Model Properties > Model Properties**. In the dialog box, in the **Callbacks** tab, select **InitFcn***.

The `Programmable FIR via Registers` block loads the lowpass coefficients from the `Host Behavioral Model` block and processes the input chirp samples first. The block then loads the highpass coefficients and processes the same chirp samples again.

Open the `Programmable FIR via Registers` block.

```
systemname = [modelname '/Programmable FIR via Registers'];
open_system(systemname);
```
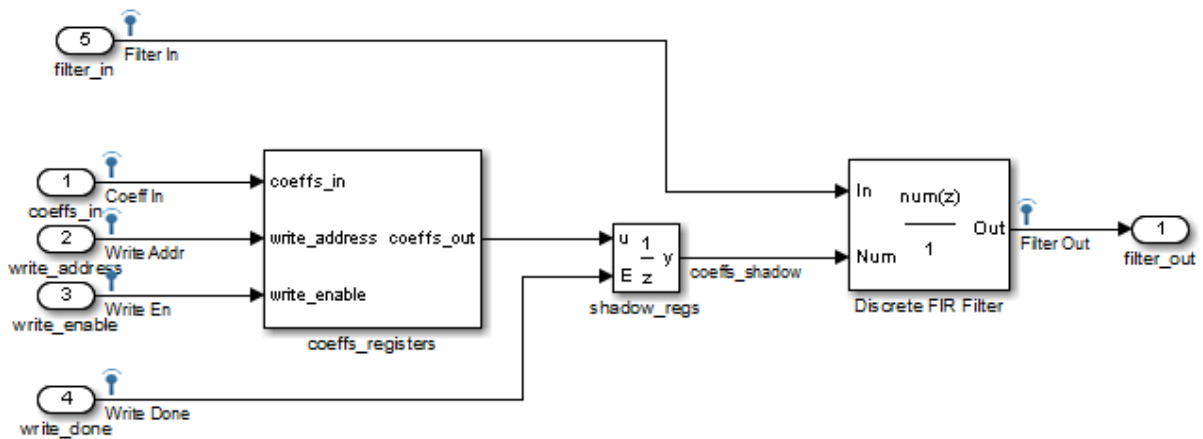

Simulation

Run the example model.

```
sim(modelname)
```

Open the scope.

```
open_system([modelname '/Scope']);
```
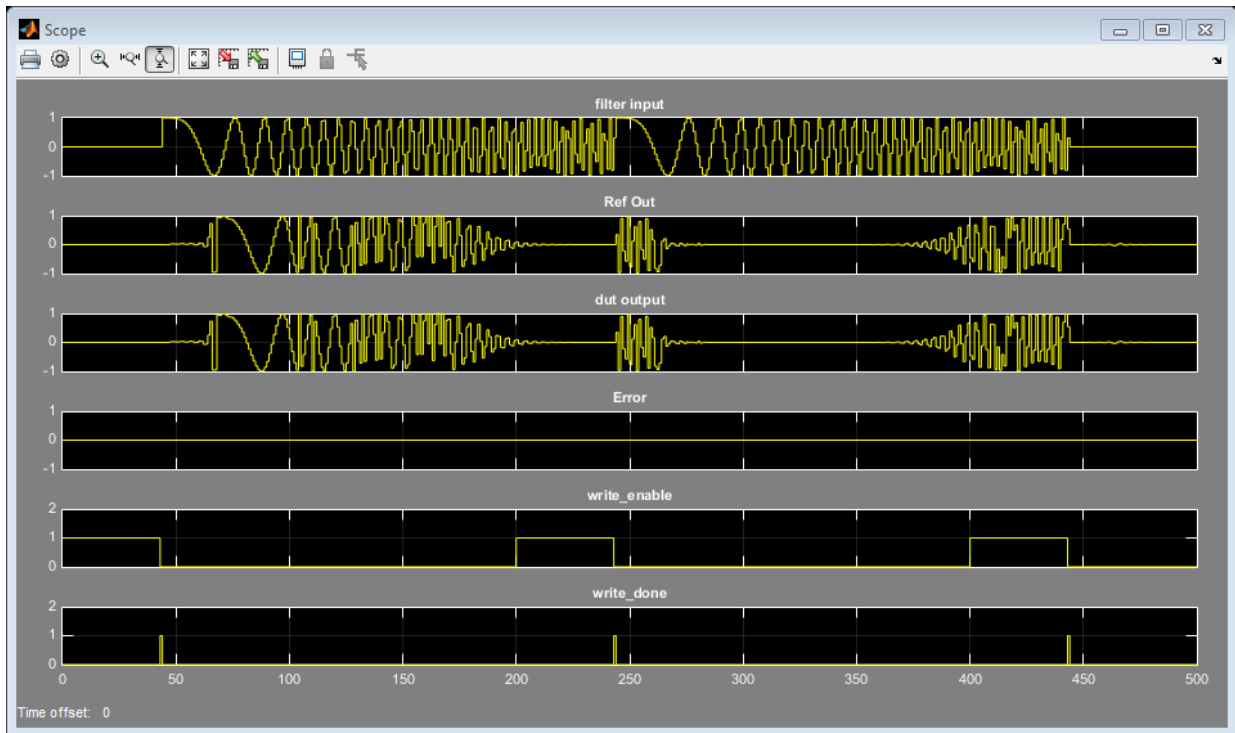
Compare the DUT (Design under Test) output with the reference output.

Use the Logic Analyzer

The logic analyzer enables you to view multiple signals in one window. It also makes it easy to detect signal transitions.

The signals of interest (input coefficient, write address, write enable, write done, filter in, filter out, reference out, and error) have been logged to a `Simulink.SimulationData.Dataset` named `dspprogfirhdl_logsout` in the MATLAB workspace.

The helper function `dspprogfirhdlReorderDataset` creates a waveform for every input provided in the `dspprogfirhdl_logsout` data set. Tto make the default order of display more meaningful, the data set is reordered to make the coefficient signals appear first and the data signals next. The error values are converted to logical values.

```
logsout_r = dspprogfirhdlReorderDataset(dspprogfirhdl_logsout);
```

Display the data in the logic analyzer. Use the helper function
`analyzeLogicFromSimulink`, which launches a logic analyzer, adds in a waveform for
every input, labels the waveform, and displays the data. `analyzeLogicFromSimulink`
returns a handle to the logic analyzer that you can use to modify the display.

```
analyzer = analyzeLogicFromSimulink(logsout_r);
```
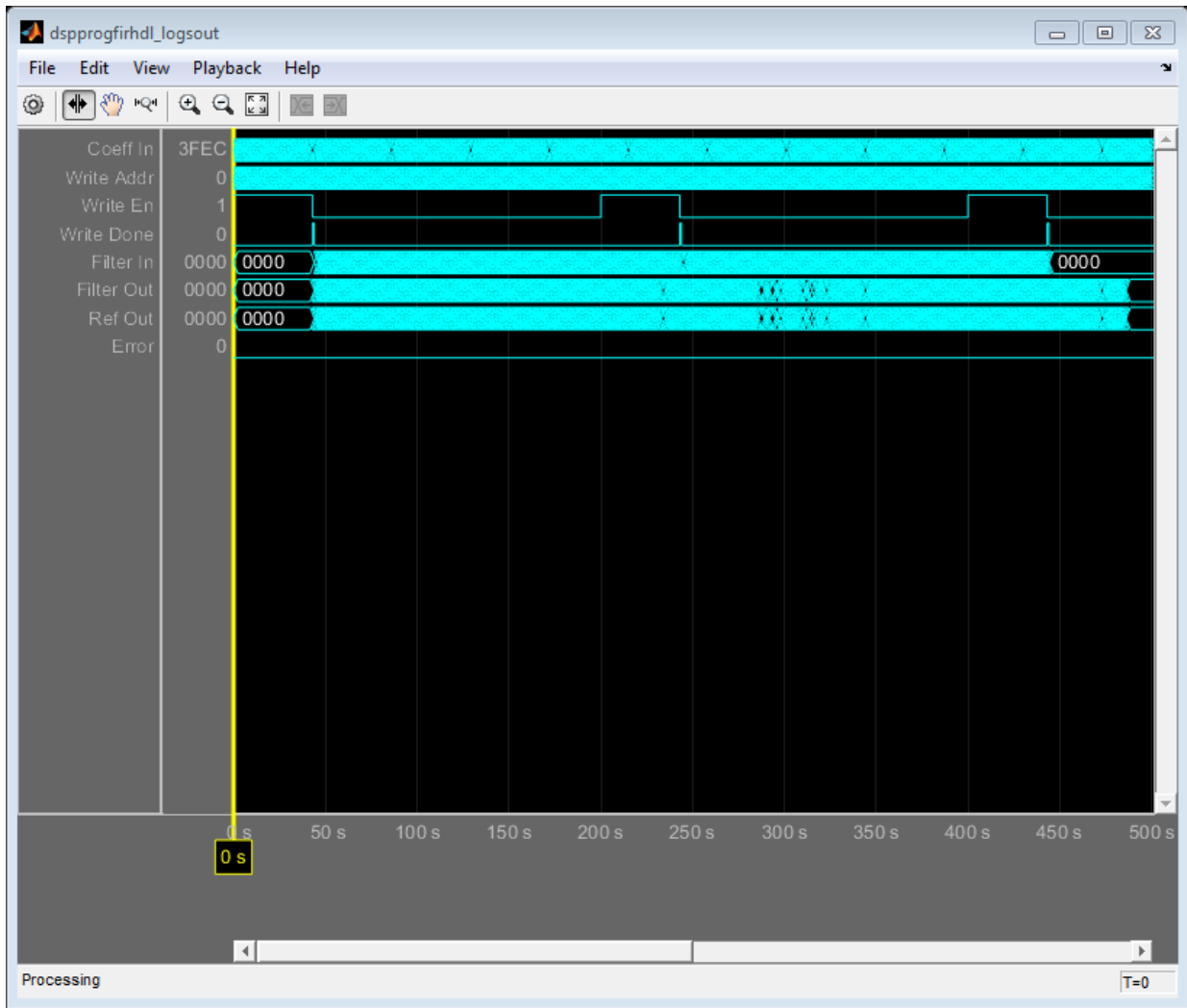
Modify the Display

In the logic analyzer, you can modify the height of all the displayed channels, the spacing between the channels, and the time span of the display.

```
analyzer.DisplayChannelSpacing = 2;
```

```
analyzer.DisplayChannelHeight  = 2;
analyzer.TimeSpan = 12;
```
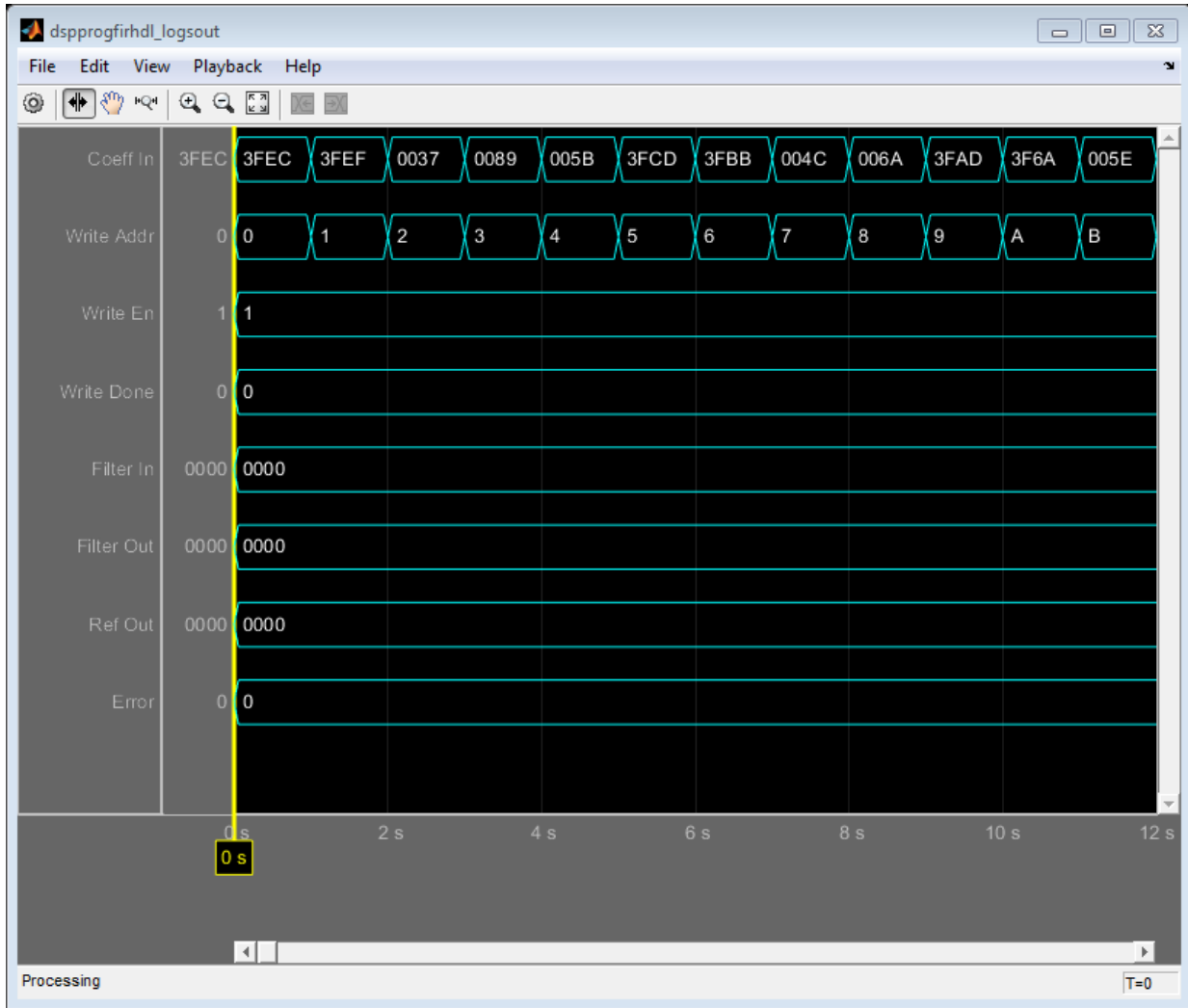

You can also control the display on a per-waveform or per-divider basis. To modify an individual waveform or divider, use the tag associated with it. You can obtain the tag associated with a waveform or divider as the return value

of `addWave` and `addDivider`, or you can get all the tags for the waveforms and dividers using the `getDisplayChannelTags` method.

Get the tags for all the displayed waveforms.

```
displayTags = analyzer.getDisplayChannelTags;
```
View the write address (second waveform in the display) in decimal mode.

```
modifyDisplayChannel(analyzer, 'DisplayChannelTag', displayTags{2}, ...
    'Radix', 'Signed decimal');
```
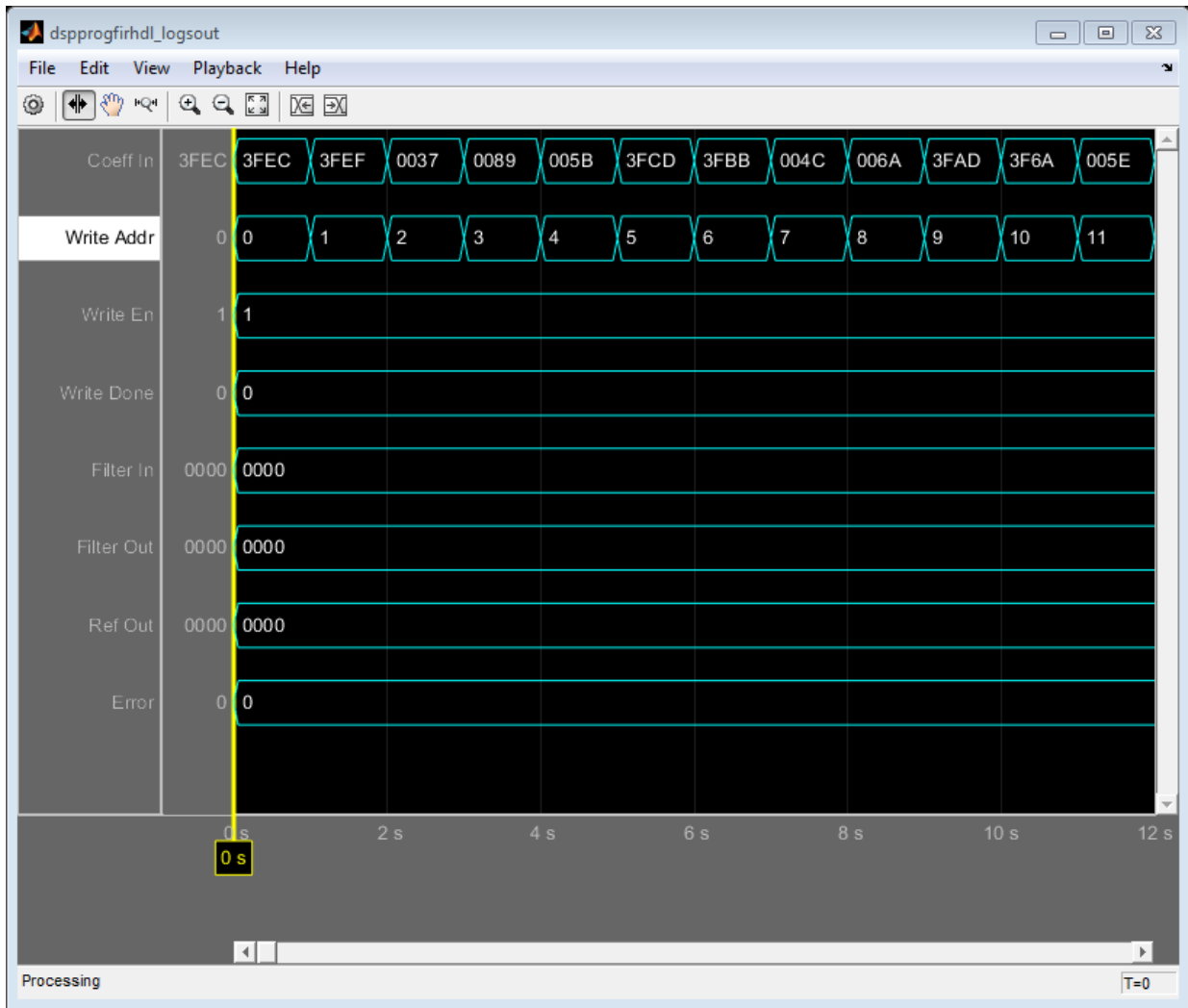
Another useful mode of visualization in the logic analyzer is the analog format. View the
`Filter In`, `Filter Out`, and `Ref Out` signals in analog format.

```
modifyDisplayChannel(analyzer, 'DisplayChannelTag', displayTags{5}, ...
    'Format', 'Analog');
modifyDisplayChannel(analyzer, 'DisplayChannelTag', displayTags{6}, ...
    'Format', 'Analog');
```

```
modifyDisplayChannel(analyzer, 'DisplayChannelTag', displayTags{7}, ...
    'Format', 'Analog');
analyzer.TimeSpan = 500;
```
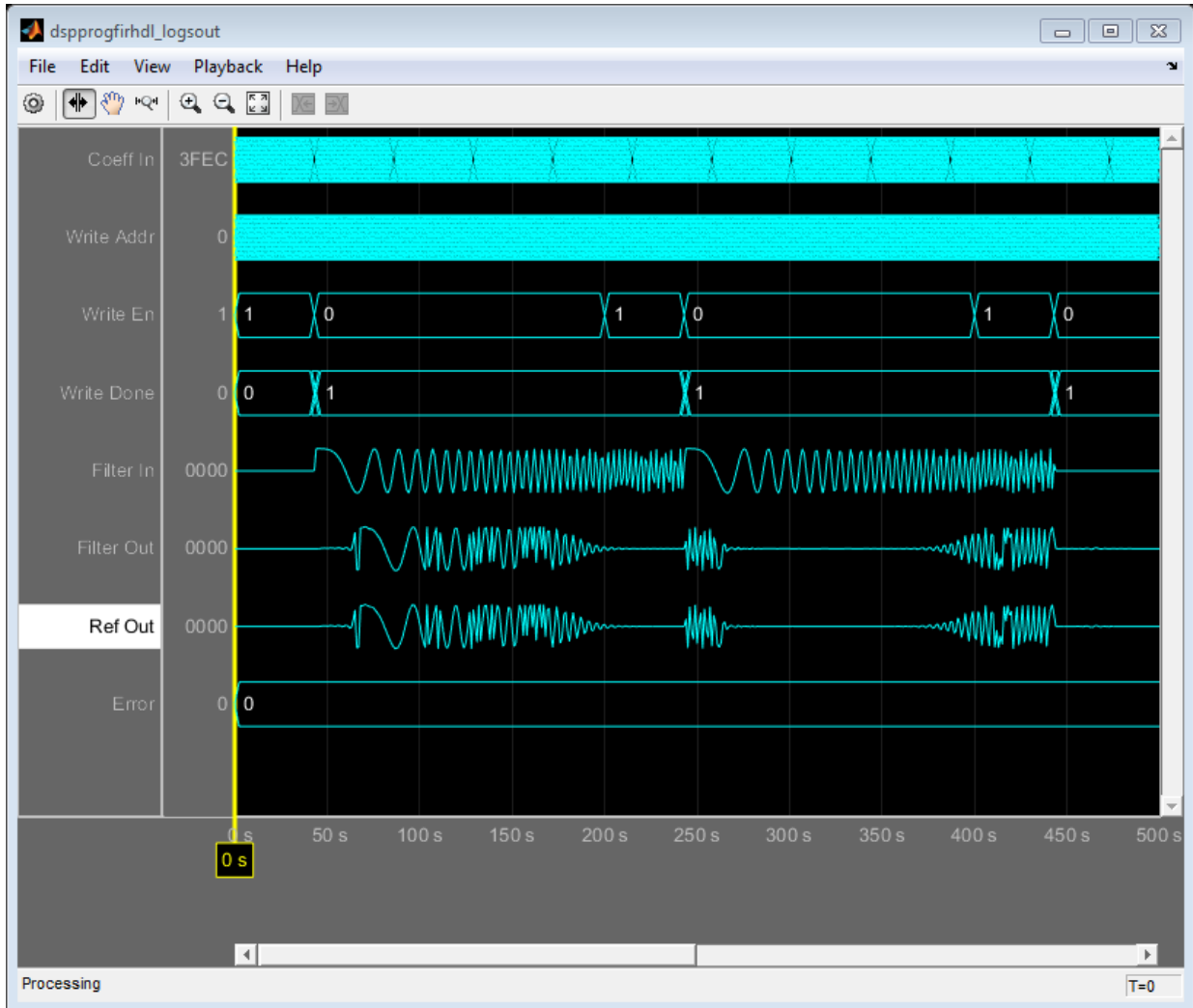

You can also add dividers and waveforms. By default, a new waveform or divider is added to the bottom of the display. If you want, you can pass in the display channel while

adding in the waveform or divider. You can also move the waveform or divider after it has been added to the display.

Add dividers for the coefficient and data signals. The divider for the coefficient signals is set to be shown in Display Channel 1. The divider for the data signals is added to the bottom of the display.

```
addDivider(analyzer, 'DisplayChannel', 1, 'Name', 'Coeff');
divTagData = addDivider(analyzer, 'Name', 'Data');
```
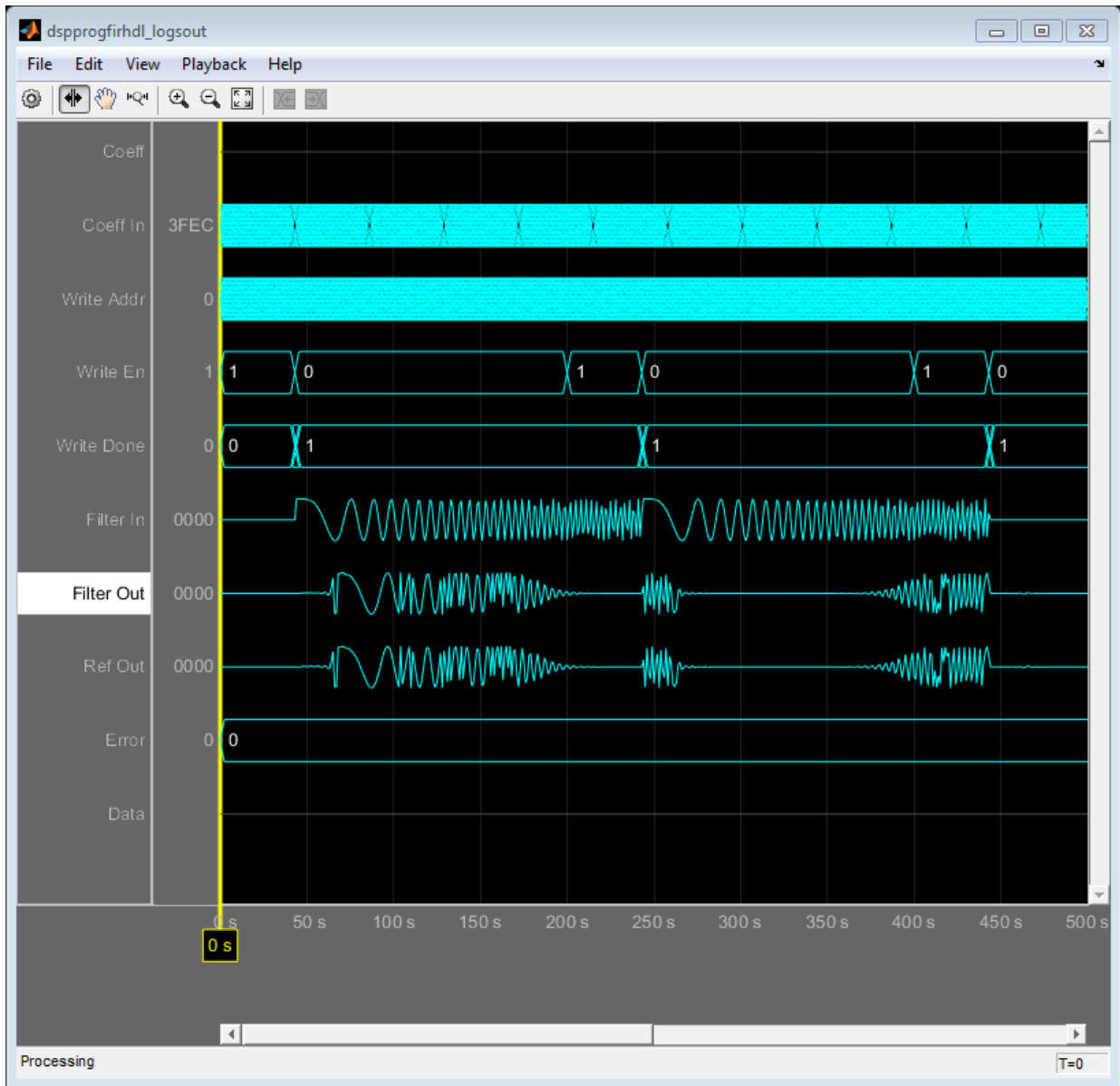
Increase the size of the window so that all the waveforms and dividers are visible.

```
pos = analyzer.Position;
analyzer.Position = [pos(1) pos(2) pos(3) pos(4)+100];
```

Using the tag for a given waveform or divider, you can use `moveDisplayChannel` to move it to the display channel of your choice.

Move the data signal divider to display channel 6.

```
moveDisplayChannel(analyzer, 'DisplayChannelTag', divTagData, 'DisplayChannel', 6);
```
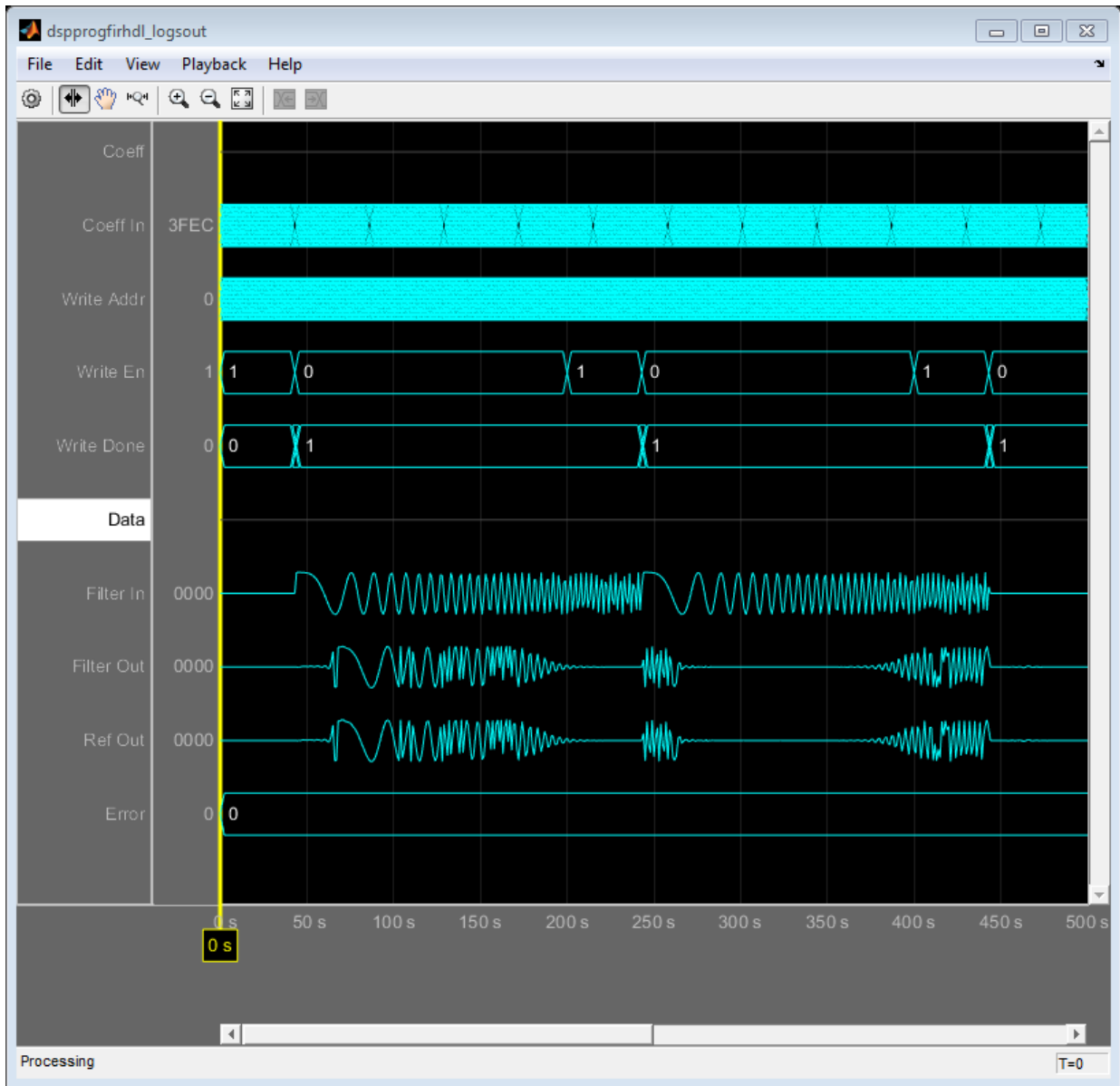
You can also operate on individual channels by right-clicking the channel name.
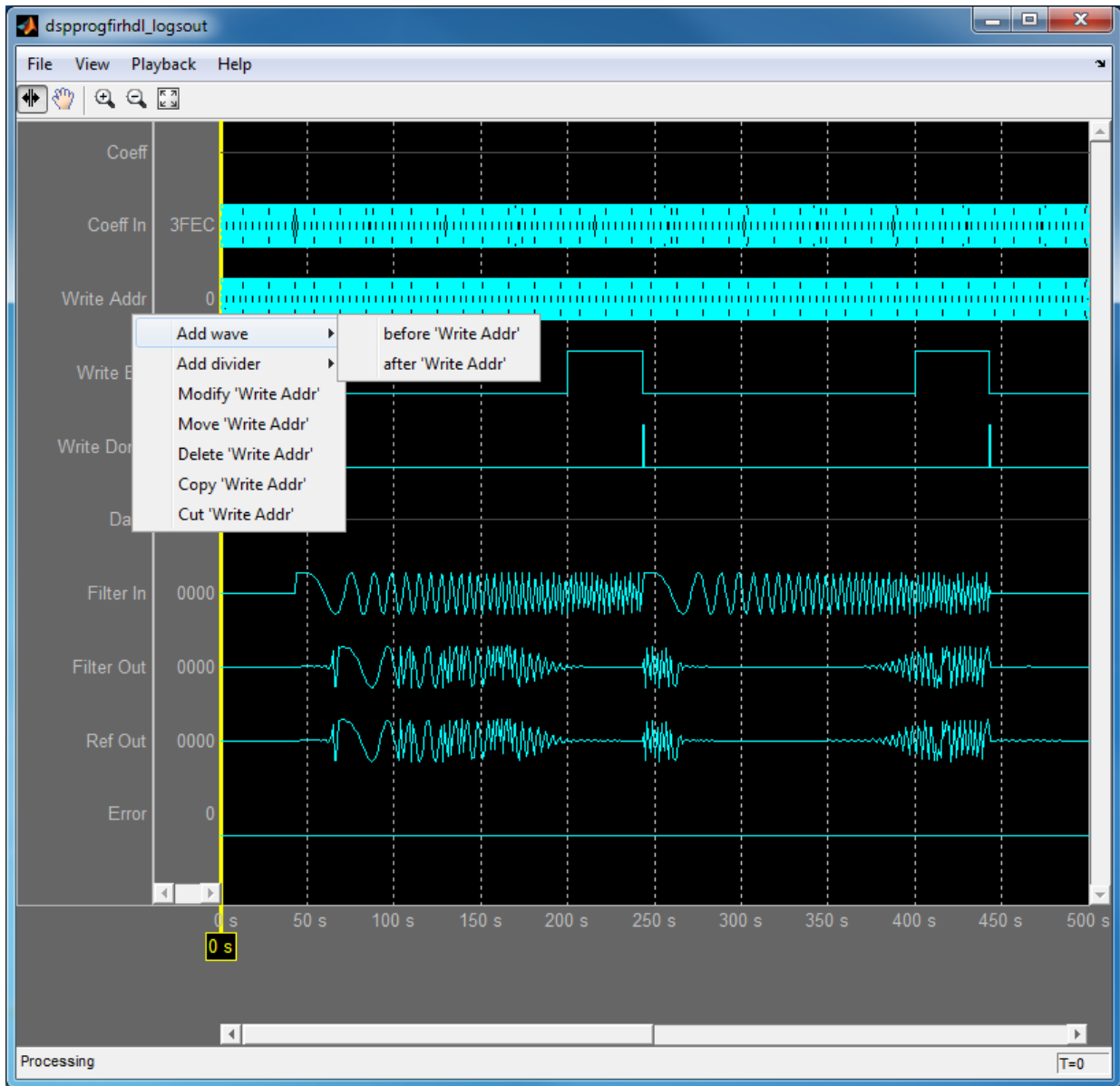
See dsp.LogicAnalyzer for more details.

Signal Visualization and Measurements in MATLAB

This example shows how to visualize and measure signals in the time and frequency domain in MATLAB using a time scope and spectrum analyzer. This example does not demonstrate the full capabilities of the time scope or spectrum analyzer. To learn more about the measurement capabilites of the time scope, see the Simulink example Time Scope Measurements. To learn more about the use of measurements with the spectrum analyzer, see the Simulink example Spectrum Analyzer Measurements.

Signal Visualization in Time and Frequency Domains

Create a sine wave with a frequency of 100 Hz sampled at 1000 Hz. Generate five seconds of the 100-Hz sine wave with additive white Gaussian noise with mean 0 and standard deviation 0.05. The signal is processed in one-second frames. Send the signal to a time scope and spectrum analyzer for display and measurement.

```
SampPerFrame = 1000;
Fs = 1000;
SW = dsp.SineWave('Frequency', 100, ...
  'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
TS = dsp.TimeScope('SampleRate', Fs, 'TimeSpan', 0.1, ...
    'YLimits', [-2, 2], 'ShowGrid', true);
SA = dsp.SpectrumAnalyzer('SampleRate', Fs);
tic;
while toc < 5
  sigData = step(SW) + 0.05*randn(SampPerFrame,1);
  step(TS, sigData);
  step(SA, sigData);
end
release(TS);
release(SA);
```
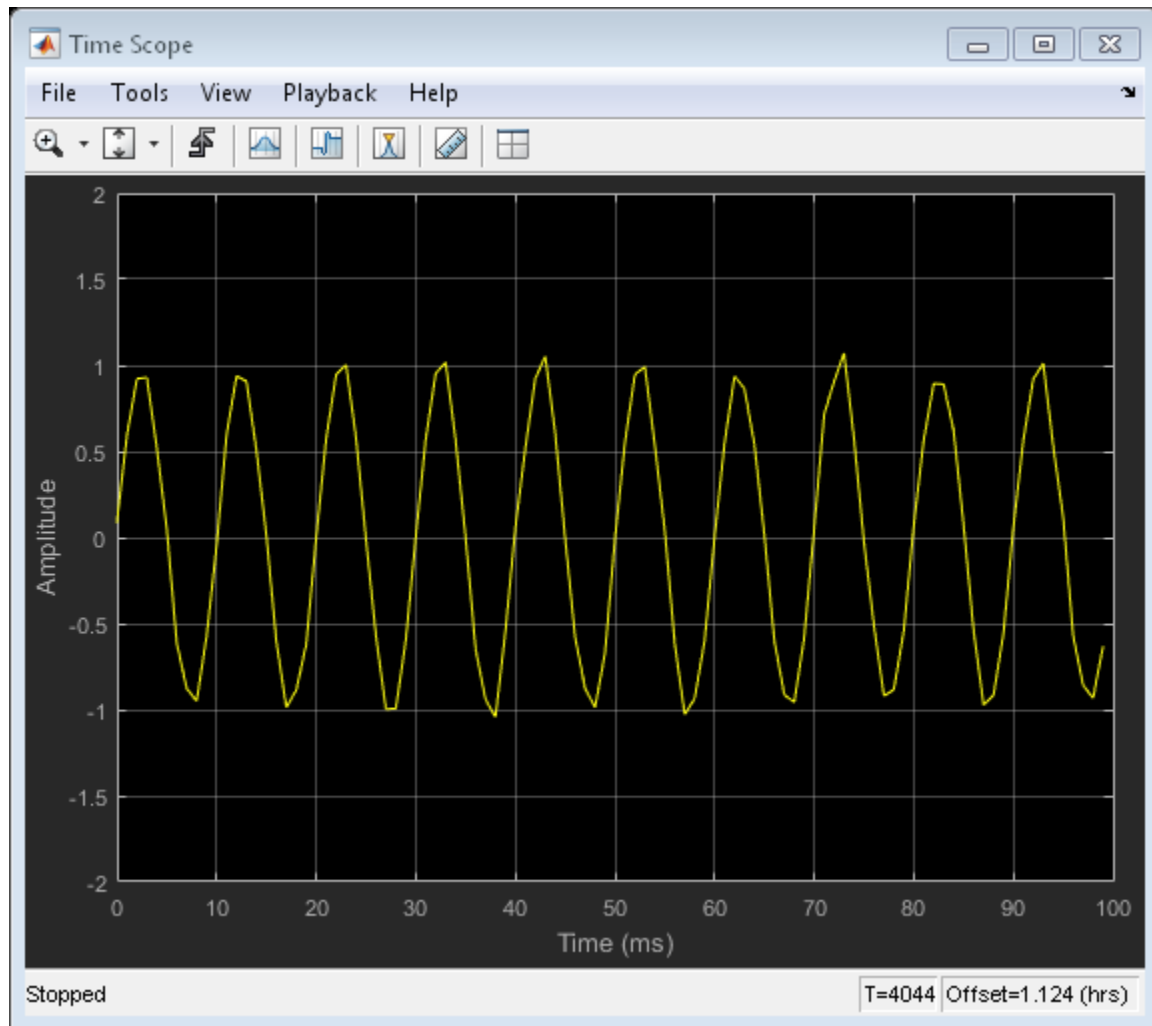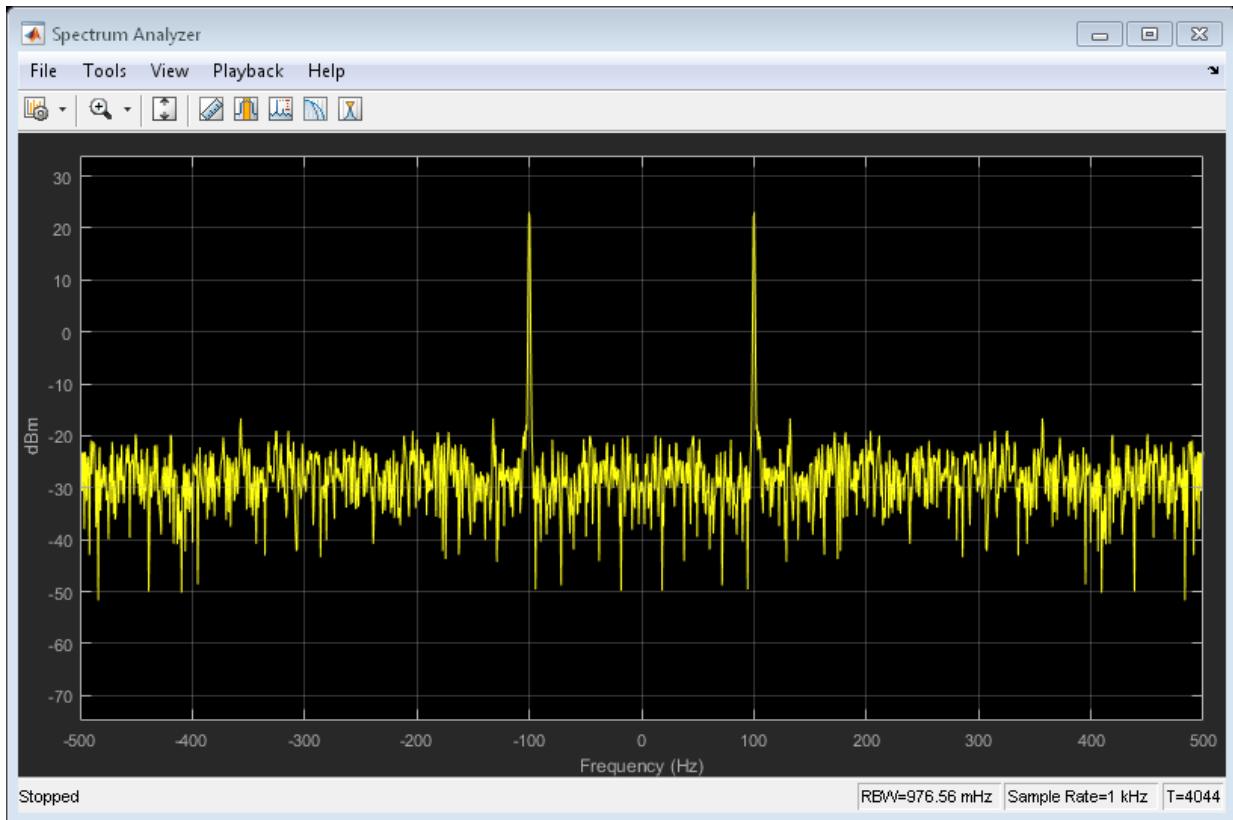
Time-Domain Measurements

Using the time scope, you can make a number of signal measurements.

The following measurements are available:

- **Cursor Measurements** - puts screen cursors on all scope displays.
- **Signal Statistics** - displays maximum, minimum, peak-to-peak difference, mean, median, RMS values of a selected signal, and the times at which the maximum and minimum occur.
- **Bilevel Measurements** - displays information about a selected signal's transitions, overshoots or undershoots, and cycles.

- **Peak Finder** - displays maxima and the times at which they occur.

You can enable and disable these measurements from the time scope toolbar or from the **Tools > Measurements** menu.
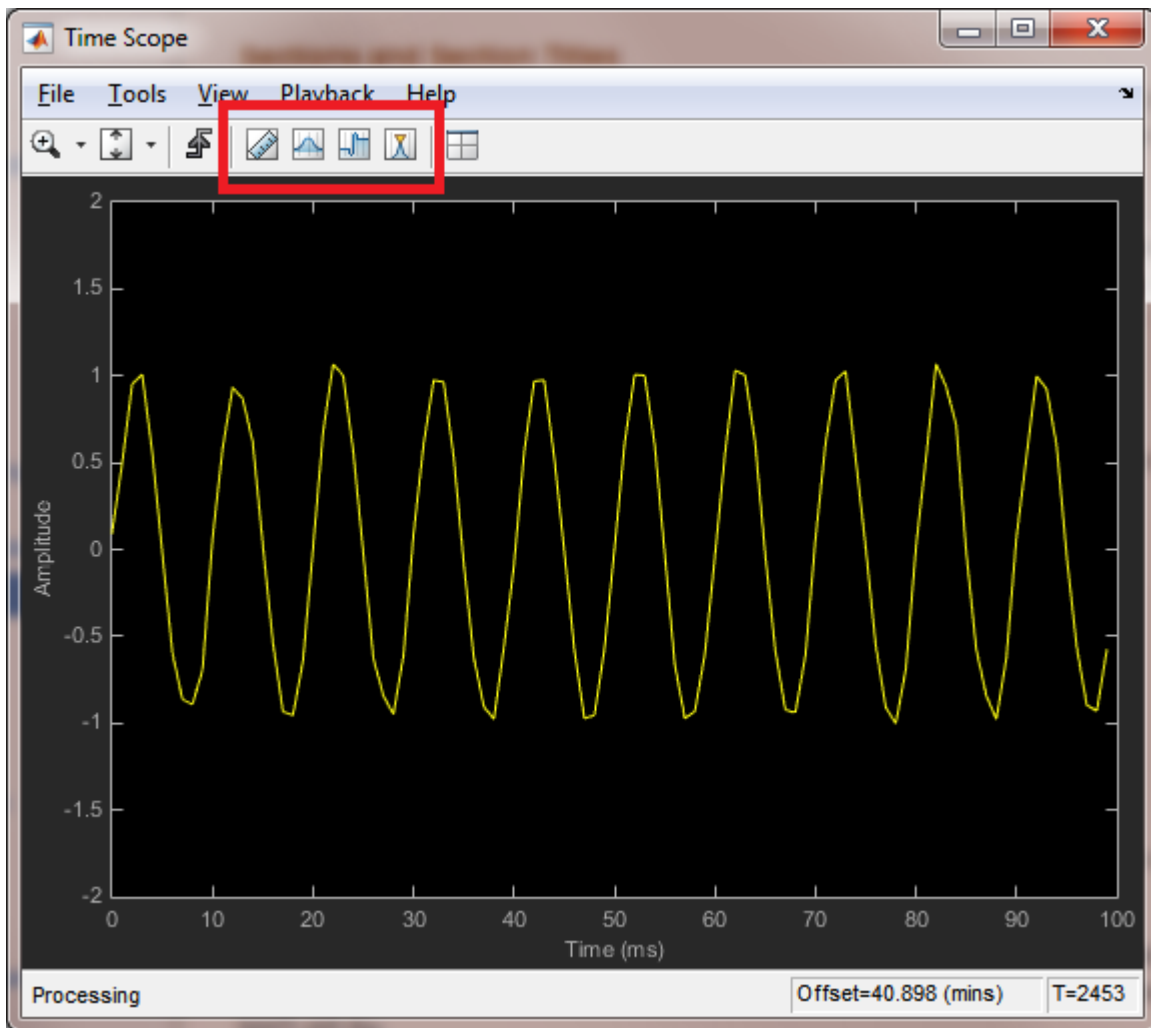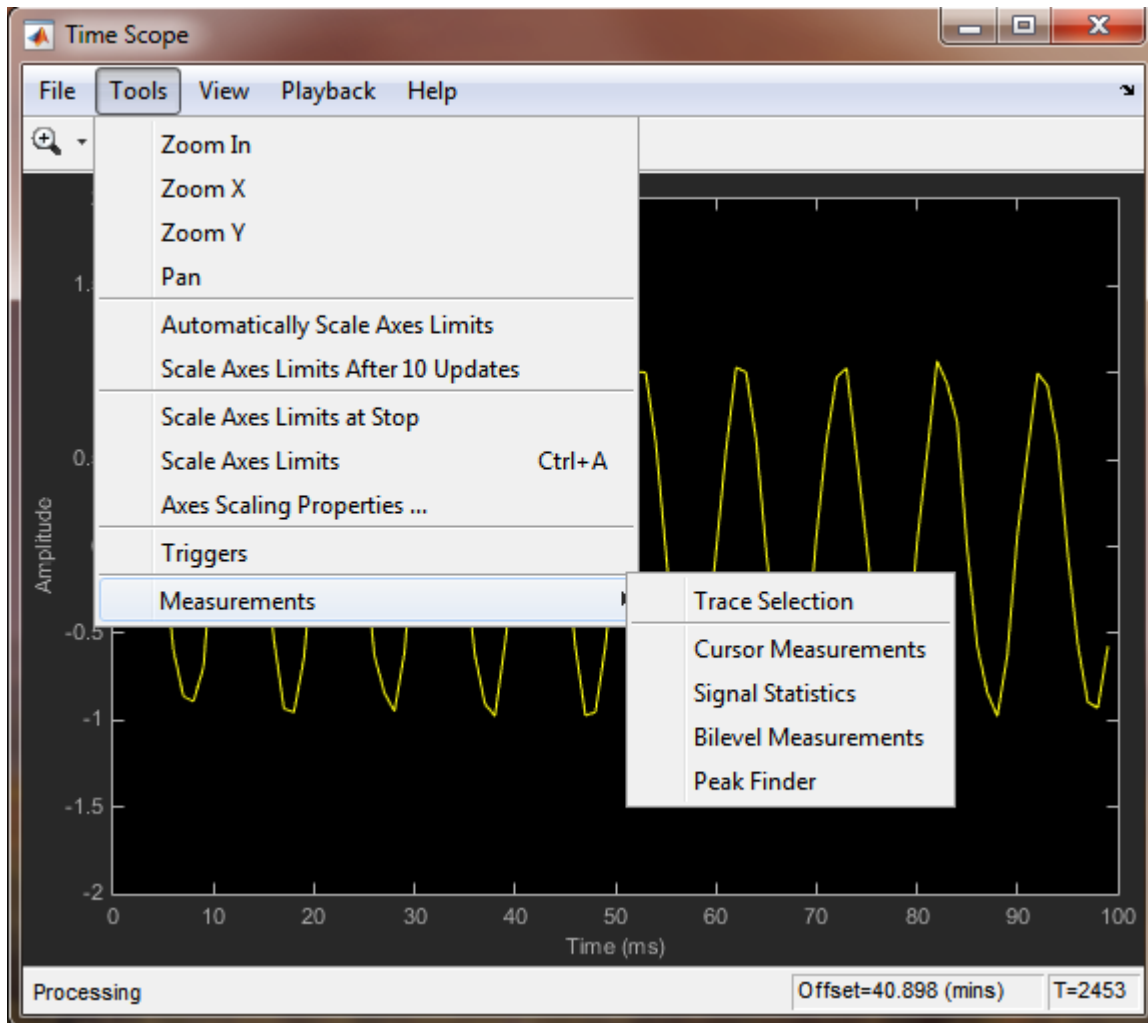
To illustrate the use of measurements in the time scope, simulate an ECG signal. Use the `ecg` function to generate 2700 samples of the signal. Use a Savitzky-Golay filter to smooth the signal and periodically extend the data to obtain approximately 11 periods.

```
x = 3.5*ecg(2700).';
y = repmat(sgolayfilt(x,0,21),[1 13]);
sigData = y(1:30000)';
```

Display the signal in the time scope and use the Peak Finder, Cursor and Signal Statistics measurements. Assume a sample rate of 4 kHz.

```
TS_ECG = dsp.TimeScope('SampleRate', 4000, ...
    'TimeSpanSource', 'Auto', 'ShowGrid', true);
step(TS_ECG, sigData);
TS_ECG.YLimits = [-4, 4];
release(TS_ECG);
```
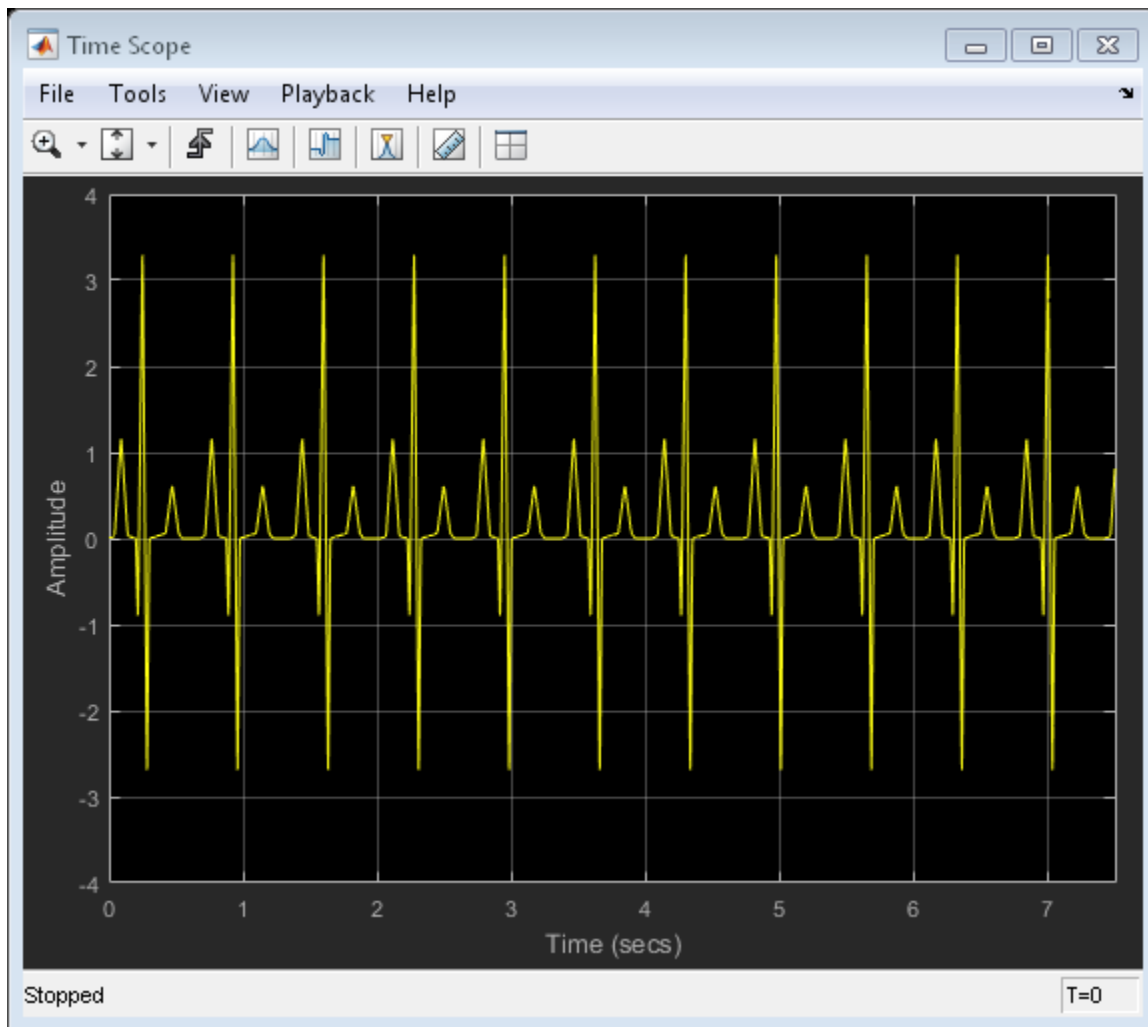
Peak Measurements

Enable **Peak Measurements** by clicking the corresponding toolbar icon or by clicking the **Tools > Measurements > Peak Finder** menu item. Click **Settings** in the Peak Finder panel to expand the Settings pane. Enter 10 for **Max Num of Peaks** and press Enter. The time scope displays in the Peaks pane a list of 10 peak amplitude values and the times at which they occur.
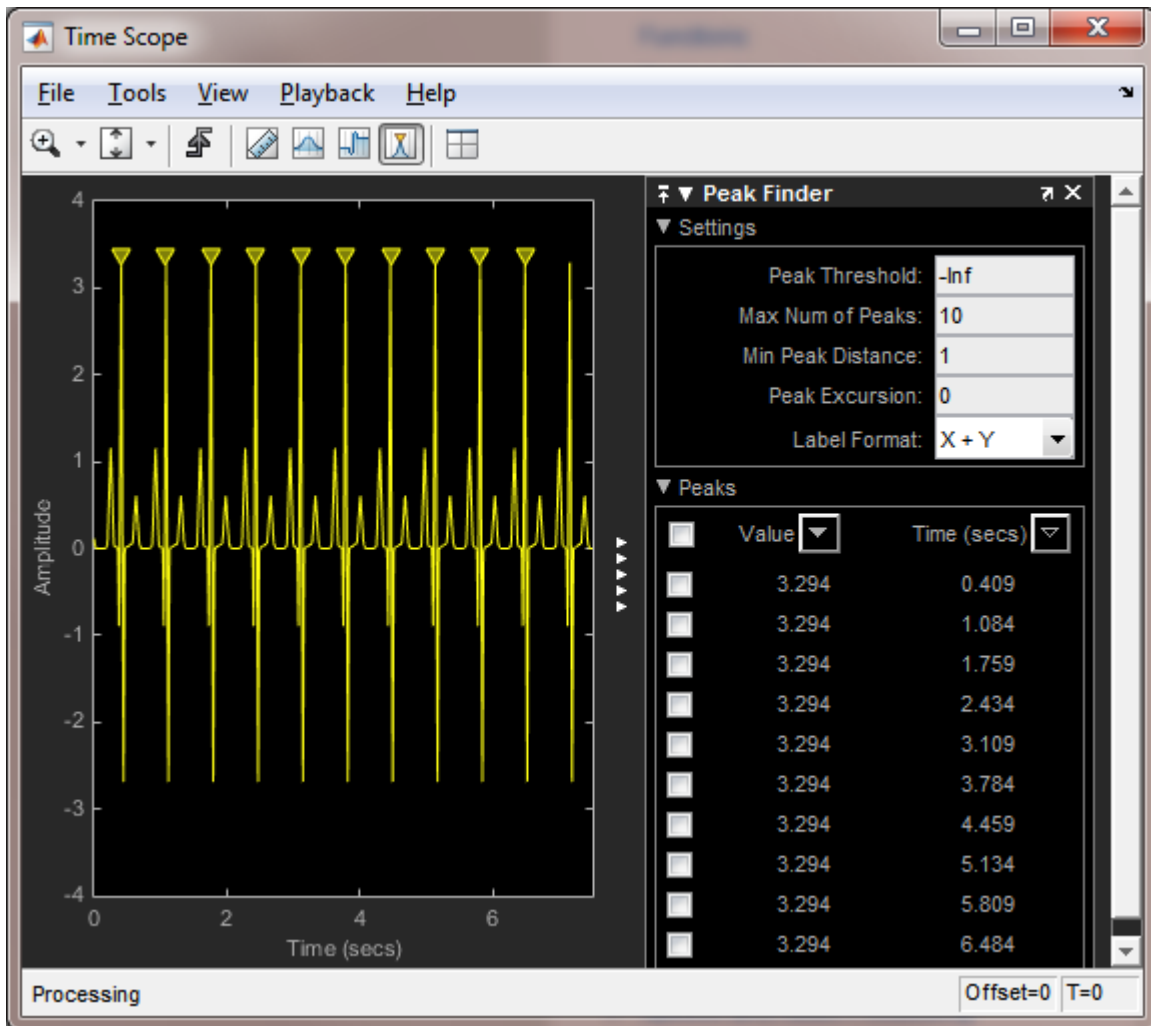
There is a constant time difference of 0.675 seconds between each heartbeat. Therefore, the heart rate of the ECG signal is given by the following equation:

$$\frac{60\,\text{sec/min}}{0.675\,\text{sec/beat}} = 88.89\,\text{beats/min (bpm)}$$

Cursor Measurements

Enable **Cursor Measurements** by clicking the corresponding toolbar icon or by clicking the **Tools > Measurements > Cursor Measurements** menu item. The Cursor Measurements panel opens and displays two cursors in the time scope. You can drag the cursors and use them to measure the time between events in the waveform. In the following figure, cursors are used to measure the time interval between peaks in the ECG waveform. The ΔT measurement in the Cursor Measurements panel demonstrates that the time interval between the two peaks is 0.675 seconds corresponding to a heart rate of 1.482 Hz or 88.9 beats/min.
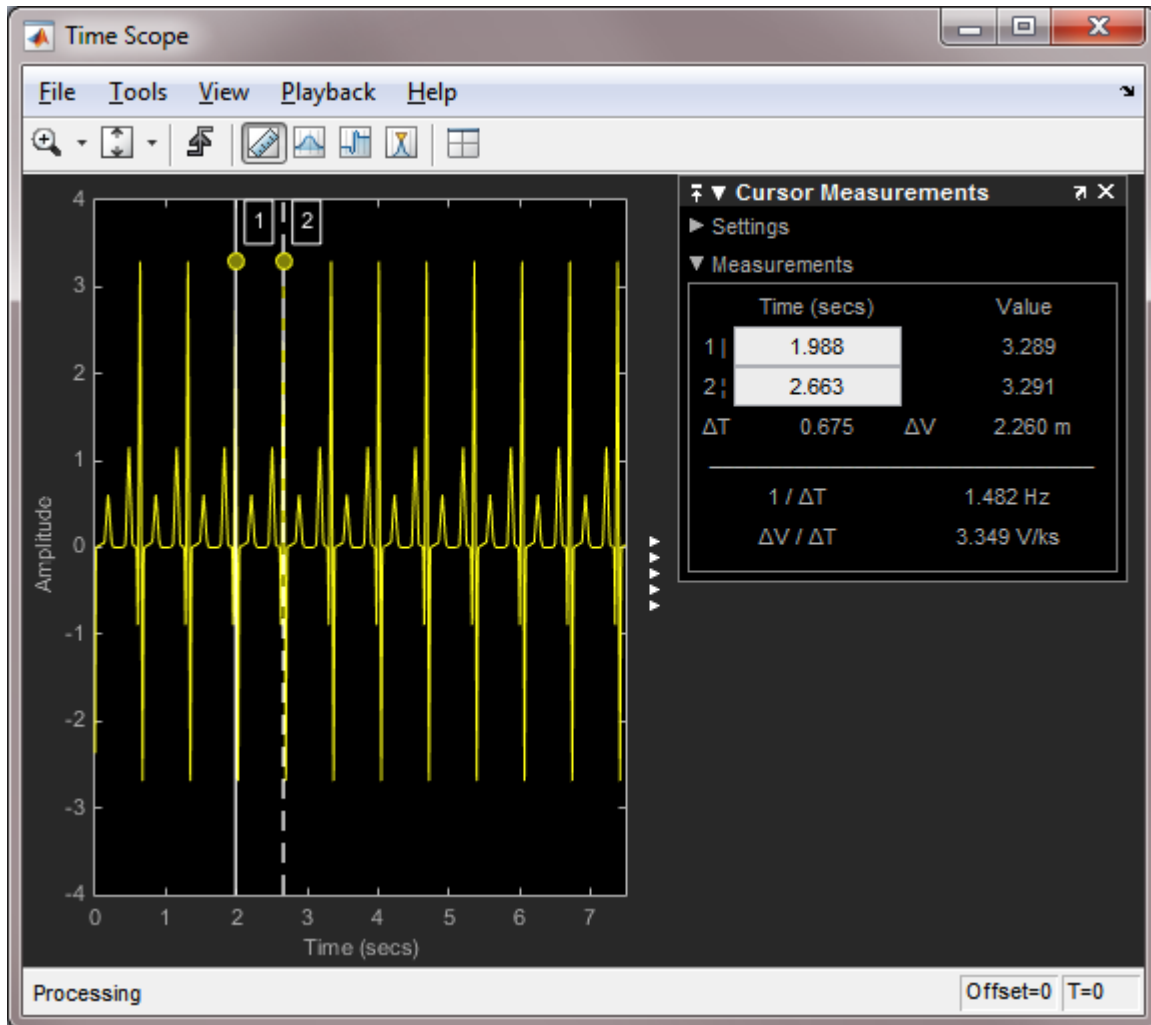
Signal Statistics and **Bilevel Measurements**

You can also select **Signal Statistics** and **Bilevel Measurements** from the **Tools > Measurements** menu. Signal Statistics can be used to determine the signal's minimum and maximum values as well as other metrics like the peak-to-peak, mean, median, and RMS values. Bilevel Measurements can be used to determine information about rising

and falling transitions, transition aberrations, overshoot and undershoot information, pulse width, and duty cycle.

Frequency-Domain Measurements

This section explains how to make frequency domain measurements with the spectrum analyzer.

The spectrum analyzer provides the following measurements:

- **Cursor Measurements** - places cursors on the spectrum display.
- **Peak Finder** - displays maxima and the frequencies at which they occur.
- **Channel Measurements** - displays occupied bandwidth and ACPR channel measurements.
- **Distortion Measurements** - displays harmonic and intermodulation distortion measurements.
- **CCDF Measurements** - displays complimentary cumulative distribution function measurements.

You can enable and disable these measurements from the spectrum analyzer toolbar or from the **Tools > Measurements** menu.
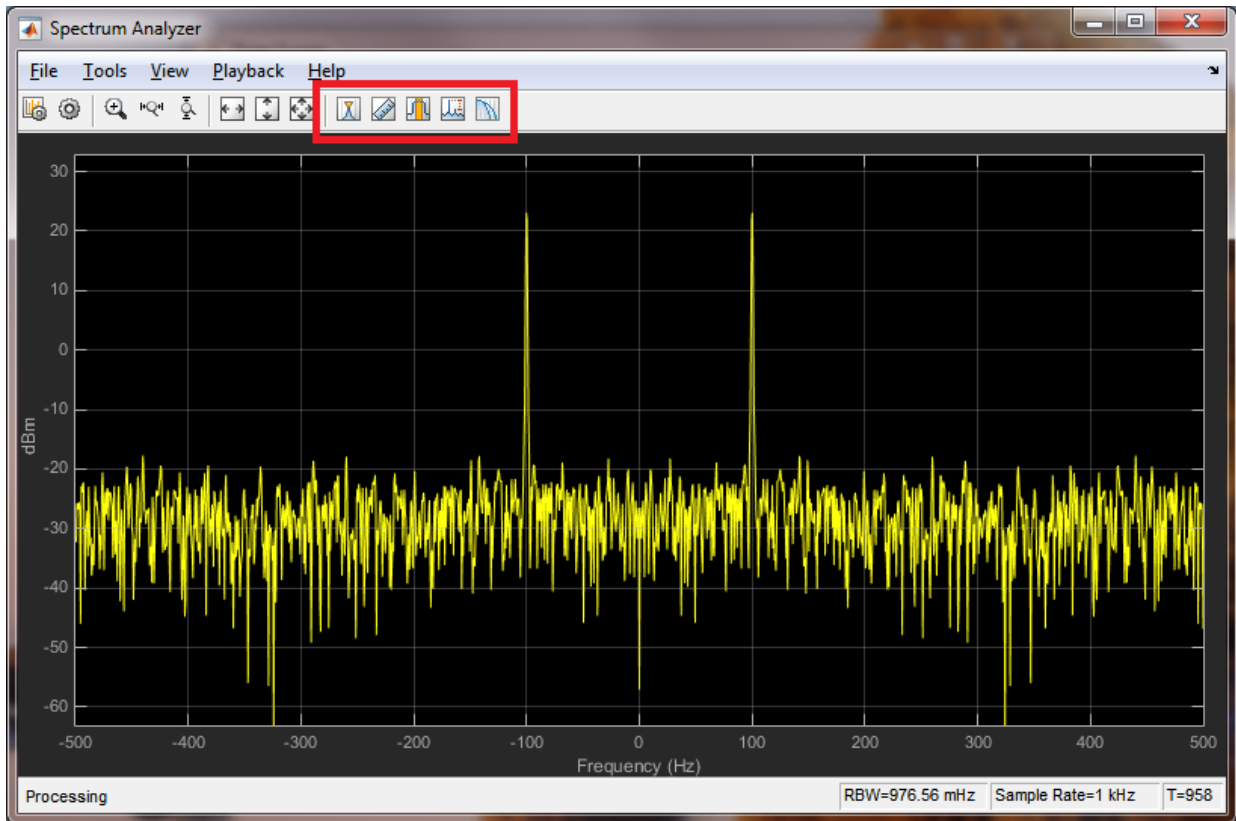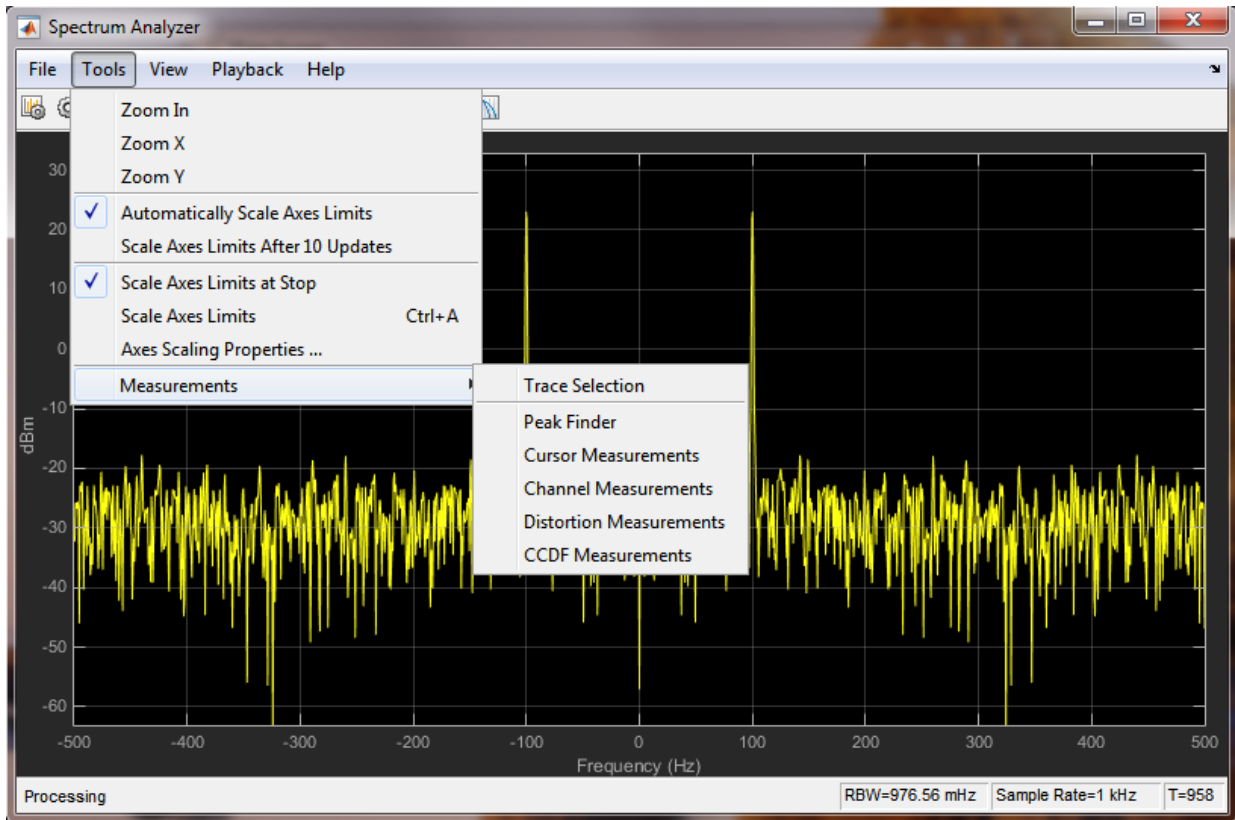
Distortion Measurements

To illustrate the use of measurements with the spectrum analyzer, create a 2.5 kHz sine wave sampled at 48 kHz with additive white Gaussian noise. Evaluate a high-order polynomial (9th degree) at each signal value to model non-linear distortion. Display the signal in a spectrum analyzer.

```
Fs = 48e3;
SW = dsp.SineWave('Frequency', 2500, ...
  'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
SA_Distortion = dsp.SpectrumAnalyzer('SampleRate', Fs, ...
    'PlotAsTwoSidedSpectrum', false);
y = [1e-6 1e-9 1e-5 1e-9 1e-6 5e-8 0.5e-3 1e-6 1 3e-3];
tic;
while toc < 5
```

```
  x = step(SW) + 1e-8*randn(SampPerFrame,1);
  sigData = polyval(y, x);
  step(SA_Distortion, sigData);
end
clear SA_Distortion;
```

Enable the harmonic distortion measurements by clicking the corresponding icon in the toolbar or by clicking the **Tools > Measurements > Distortion Measurements** menu item. In the Distortion Measurements, change the value for **Num. Harmonics** to 9 and check the **Label Harmonics** checkbox. In the panel, you see the value of the fundamental close to 2500 Hz and 8 harmonics as well as their SNR, SINAD, THD and SFDR values, which are referenced with respect to the fundamental output power.
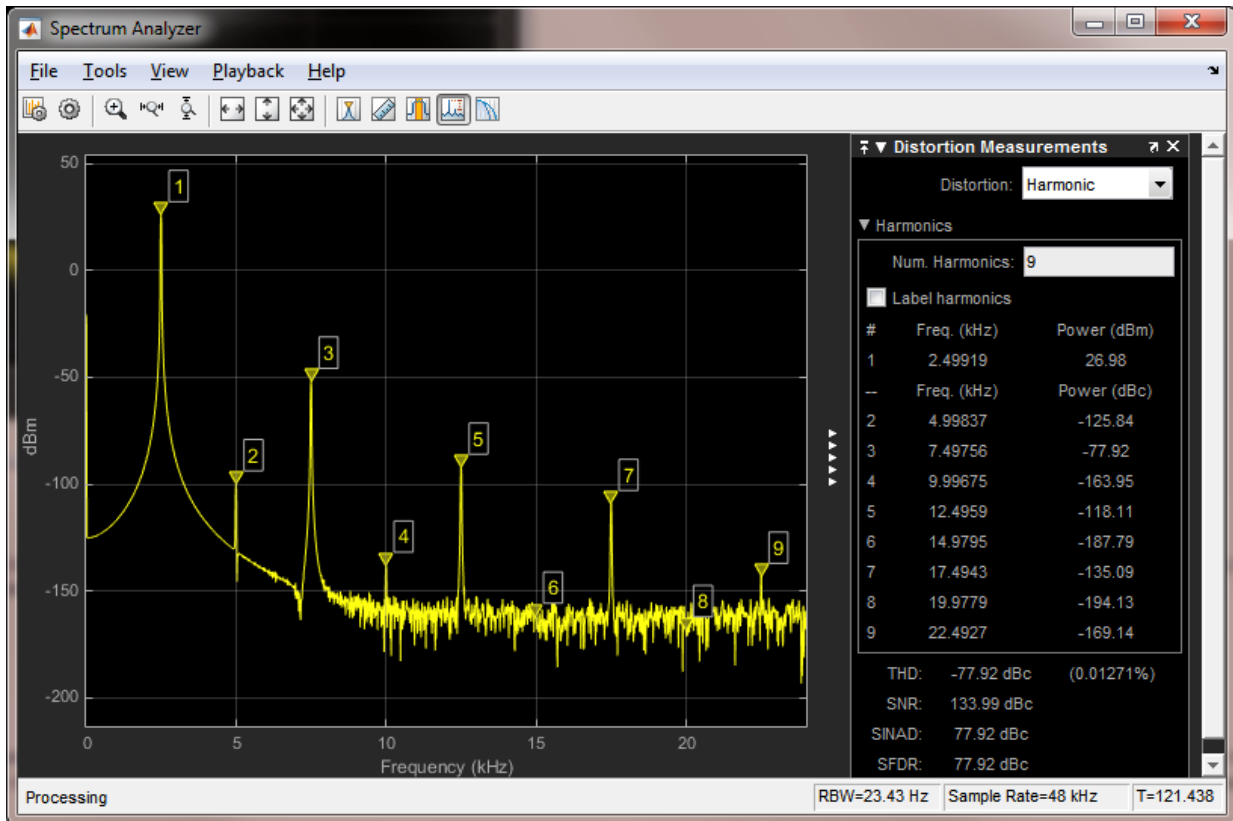
Peak Finder

You can track time-varying spectral components by using the Peak Finder measurement dialog. You can show and optionally label up to 100 peaks. You can invoke the Peak Finder dialog from the **Tools > Measurements > Peak Finder** menu item, or by clicking the corresponding icon in the toolbar.

To illustrate the use of **Peak Finder**, create a signal consisting of the sum of three sine waves with frequencies of 5, 15, and 25 kHz and amplitudes of 1, 0.1, and 0.01 respectively. The data is sampled at 100 kHz. Add $N(0, 10^{-8})$ white Gaussian noise to the sum of sine waves and display the one-sided power spectrum in the spectrum analyzer.

```
Fs = 100e3;
SW1 = dsp.SineWave(1e0,   5e3, 0, 'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
SW2 = dsp.SineWave(1e-1, 15e3, 0, 'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
SW3 = dsp.SineWave(1e-2, 25e3, 0, 'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
SA_Peak = dsp.SpectrumAnalyzer('SampleRate', Fs, 'PlotAsTwoSidedSpectrum', false);
tic;
while toc < 5
    sigData = step(SW1) + step(SW2) + step(SW3) + 1e-4*randn(SampPerFrame,1);
    step(SA_Peak, sigData);
end
clear SA_Peak;
```

Enable the **Peak Finder** to label the three sine wave frequencies. The frequency values and powers in dBm are displayed in the **Peak Finder** panel. You can increase or decrease the maximum number of peaks, specify a minimum peak distance, and change other settings from the **Settings** pane in the Peak Finder Measurement panel.
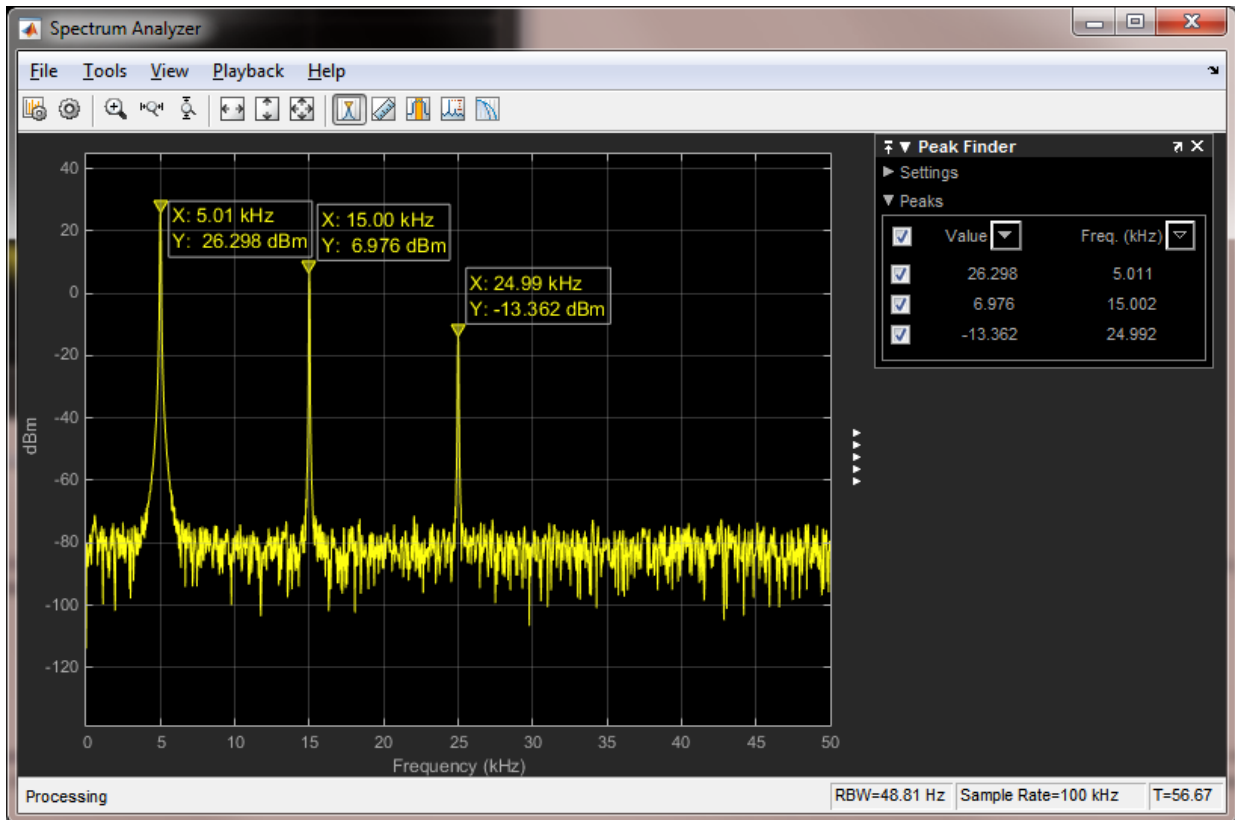
Filter Frames of a Noisy Sine Wave Signal using Testbench Generator

This example shows how to use the Streaming Testbench Generator app to generate DSP algorithm testbenches. The DSP algorithm generated in this example is similar to the algorithm in the Filter Frames of a Noisy Sine Wave Signal in MATLAB example. That example filters a noisy sine wave signal using a FIR lowpass filter and displays the power spectrum using a spectrum analyzer.

Streaming Testbench Generator Example App

The Streaming Testbench Generator app helps you develop and test streaming signal processing algorithms by enabling you to quickly generate testbenches. To launch the Testbench Generator, enter `testbenchGeneratorExampleApp` at the MATLAB command prompt. The command launches an interface through which you can:

1 Select a set of **sources** and **sinks**.

2 Enter the function name of your custom **User Algorithm**.

3 Customize the properties of each of the added sources and sinks.

Each source is treated as a separate input to your algorithm, but you can associate more than one sink with the same output from your algorithm.

`testbenchGeneratorExampleApp`
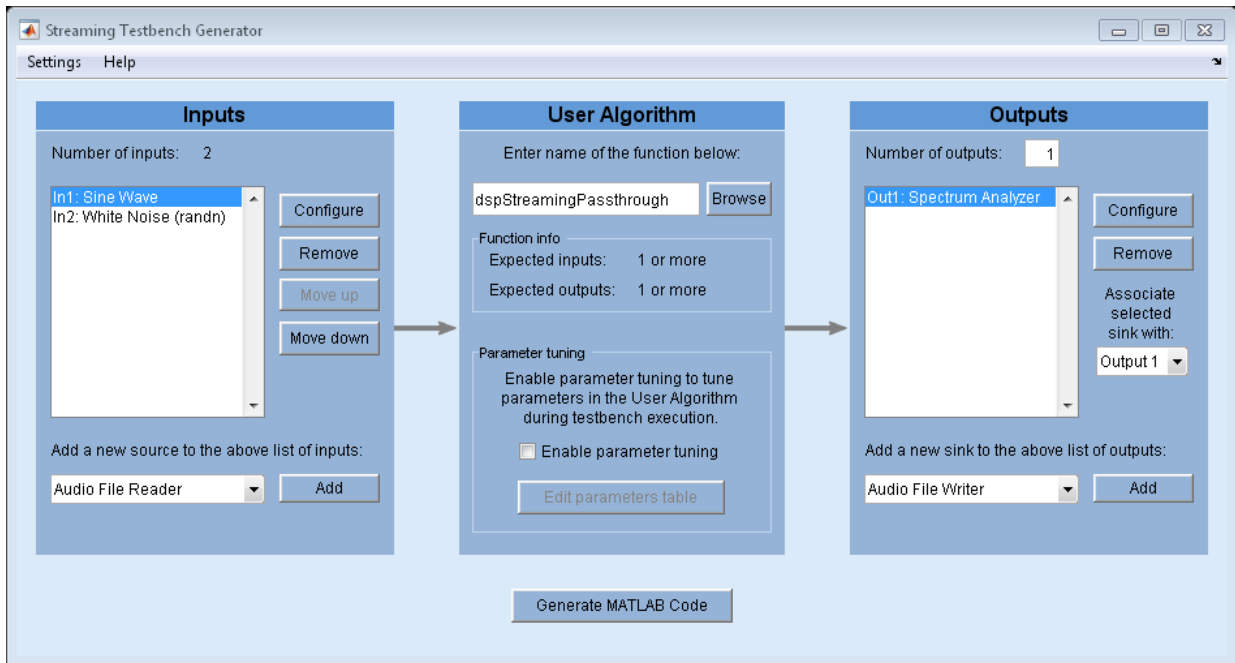
Inputs - Sine Waves and White Noise

By default, the testbench generator selects a two-channel sine wave source and a white Gaussian noise source. The two channels of the sine wave source have frequencies of 1 kHz and 10 kHz. The sampling frequency is 44.1 kHz. The white Gaussian noise input has mean 0 and standard deviation 0.1. The data is processed in frames of 1024 samples. To add more sources, use the list under **Add a new source to the above list of inputs** to select one of the supported sources. Alternatively, you can add your custom System object source by selecting **Custom System object** from the list and clicking **Add**. The added source appears in the list of inputs.
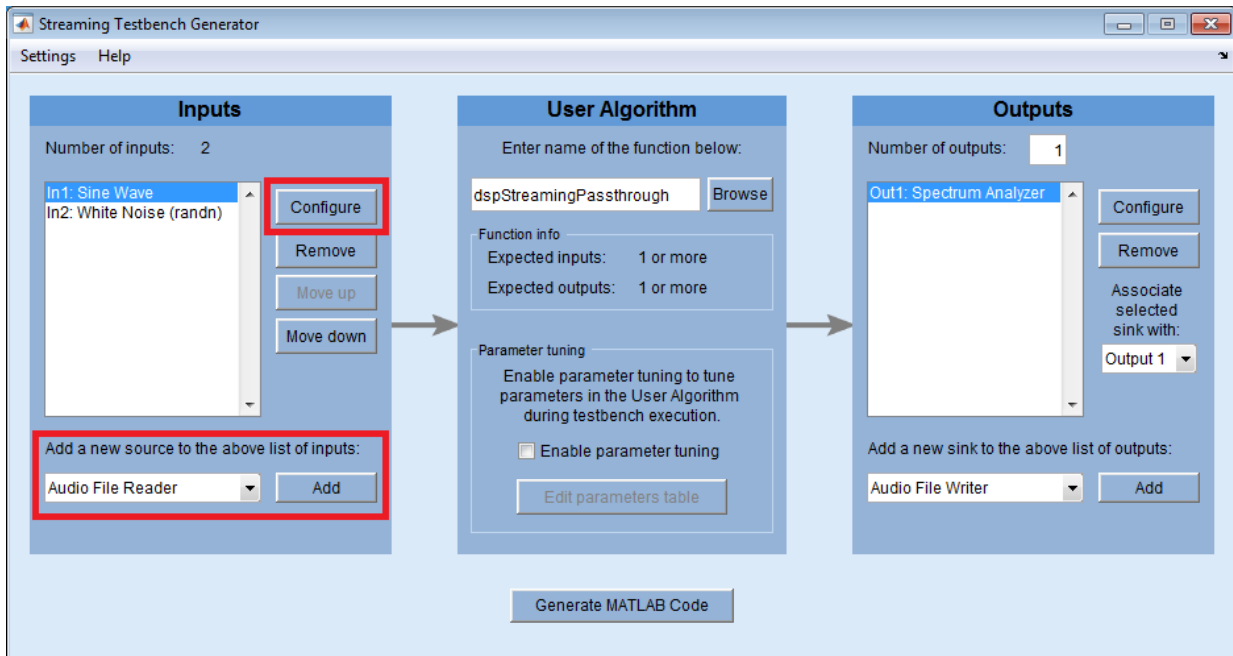
After adding a source, you can select it and click **Configure** to change the selected source's properties.
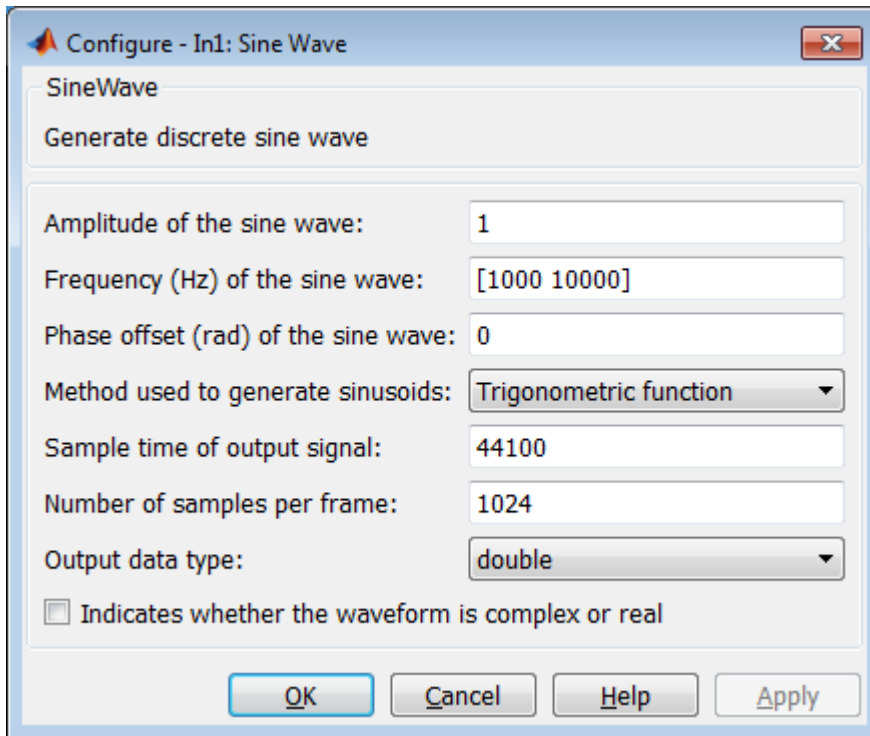
User Algorithm - Lowpass Filter

The default user algorithm `dspStreamingPassthrough` is a generic function that just passes through the inputs to the outputs. The user algorithm used in this example is a more meaningful function `hTestbenchLowpass`. You can view the code for this function by entering

```
edit hTestbenchLowpass
```

at the MATLAB command prompt. `hTestbenchLowpass` accepts two inputs, lowpass filters the sum of those two inputs, and returns the filtered signal. It uses a constrained equiripple FIR filter design with a cutoff frequency of 5 kHz. The ripples in the passband and stopband are equal to 0.05 and 0.001. Filtering is performed using `dsp.FIRFilter`, which is optimized for streaming.

Type `hTestbenchLowpass` in the User Algorithm text box replacing the default `dspStreamingPassthrough`. Alternatively, you can bring up a new testbench generator

session by entering `testbenchGeneratorExampleApp('hTestbenchLowpass')` at the MATLAB command prompt.
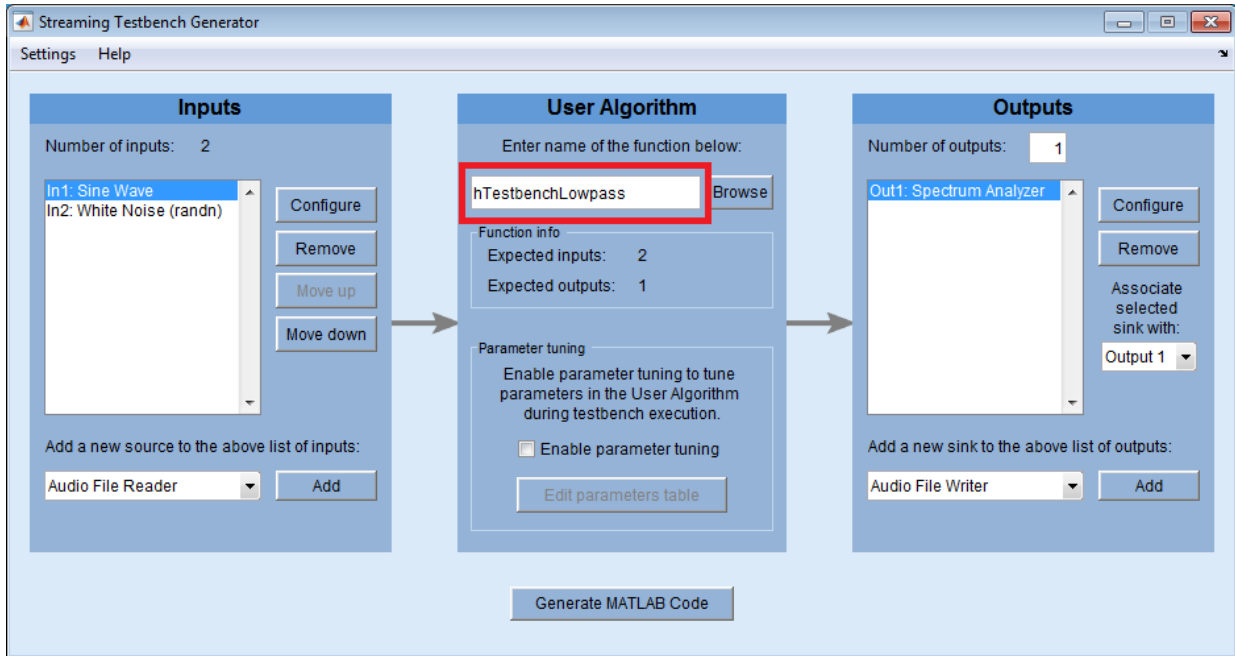
Output

The power spectrum of the output is displayed on a spectrum analyzer in dBm. You can add more sinks to visualize or post-process the outputs. Similar to inputs, you can use the list under **Add a new sink to the above list of outputs** to add a new sink, and click **Configure** to modify the properties of the selected sink.

You can associate a single output from the user algorithm with one or more sinks. For example, you can visualize the same output signal with both a time scope and spectrum analyzer. To do this, add the required sinks and make sure you associate all of the sinks to desired output from the user algorithm by changing the value under the **Associate selected sink with** list.
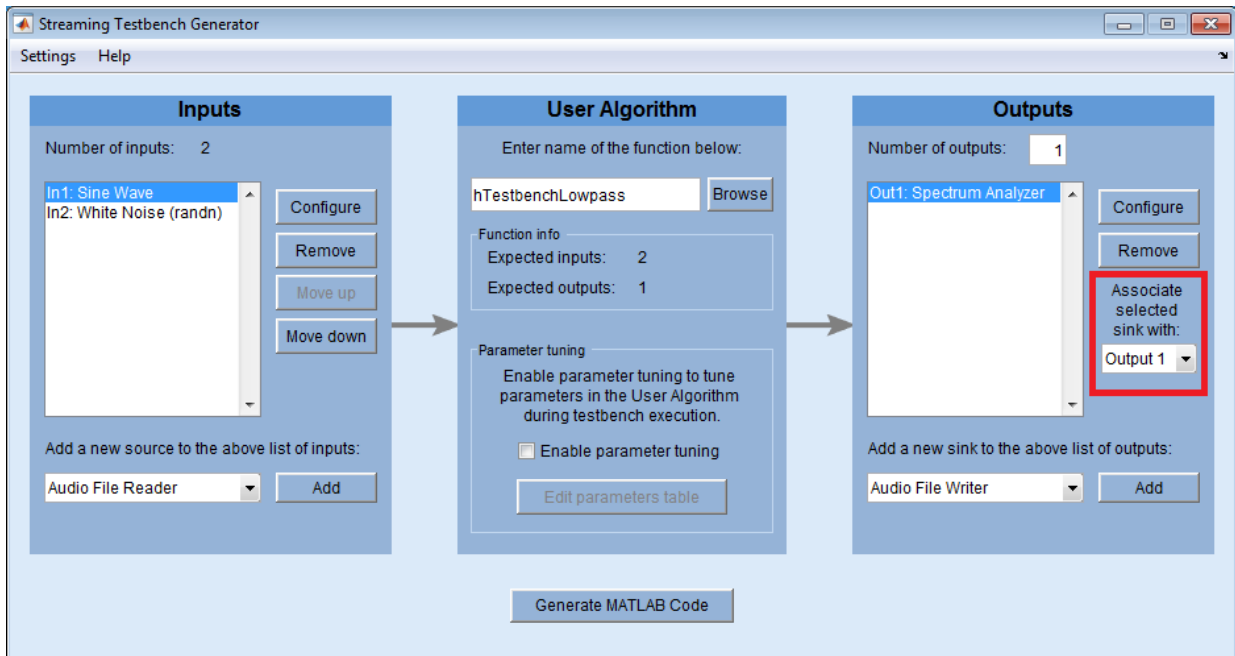
Generate Code and Simulate

After you add and configure the sources and sinks and enter a function name in the **User Algorithm** text box, the testbench generator is ready to generate testbench MATLAB code. To generate code, click on the **Generate MATLAB Code** button. A new untitled document opens in the MATLAB editor containing the generated testbench code.

You can edit the generated code to customize it before executing it. For the default example, the generated code is included below. Executing this testbench code, you see in the spectrum analyzer that the frequencies above 4 kHz in the source signal are attenuated. The resulting signal maintains the peak at 1 kHz because 1 kHz falls in the passband of the lowpass filter.

```
% Streaming testbench script
% Generated by Streaming Testbench Generator

% Initialization
numIterations = 10000;

% Construct sources (for all inputs)
```

```matlab
src1 = dsp.SineWave('Frequency',[1000 10000], ...
    'SampleRate',44100, ...
    'SamplesPerFrame',1024);

% Construct sinks (for all outputs)
sink1 = dsp.SpectrumAnalyzer('SampleRate',44100, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'ShowLegend',true);

% Stream processing loop
clear hTestbenchLowpass;
for i = 1:numIterations
    % Sources
    in1 = step(src1);
    in2 = 0.1*randn(1024,2);

    % User Algorithm
    out1 = hTestbenchLowpass(in1,in2);

    % Sinks
    step(sink1,out1);
end

% Clean up
release(src1);
release(sink1);
```
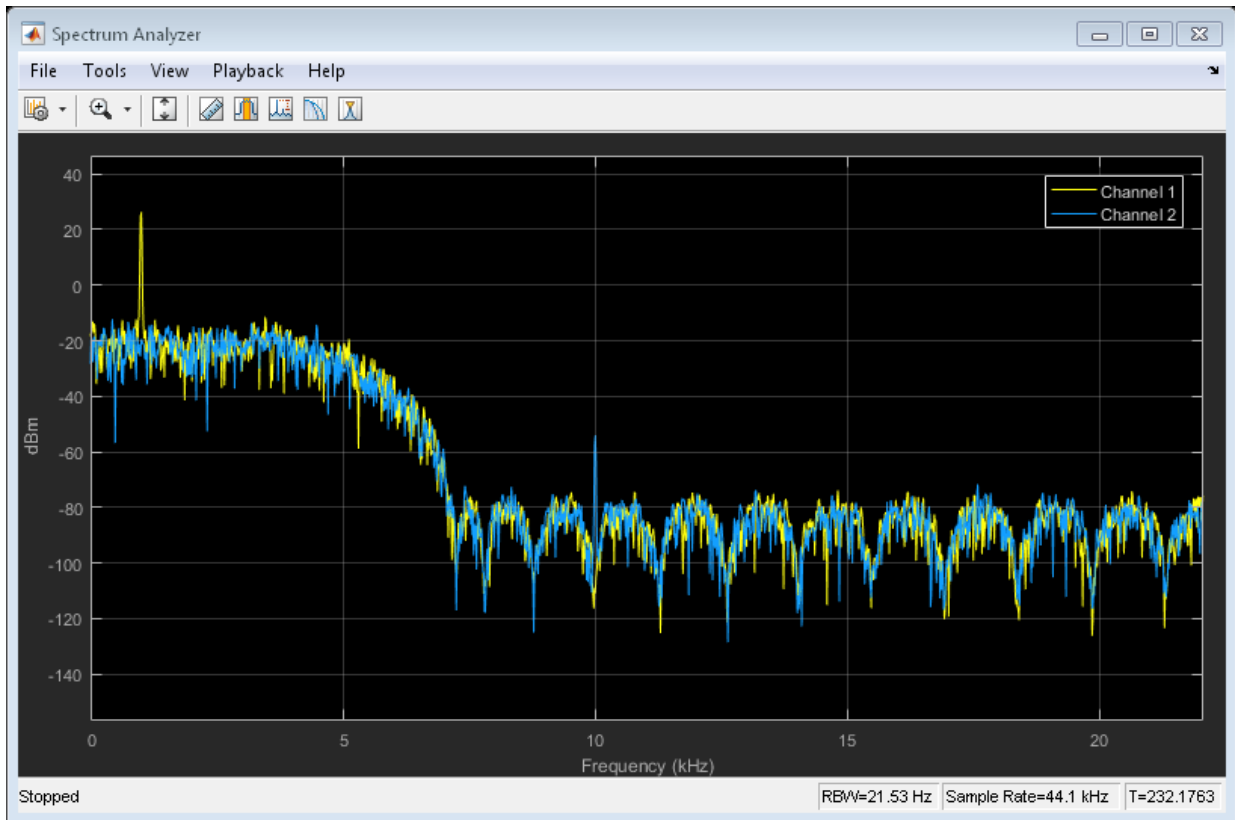
More Customizations in Testbench Generator

The testbench generator offers additional top-level customizations, which you can configure using the **Testbench Generator Settings** dialog box. To open this dialog box, select **Settings > Testbench Generator Settings ...**.
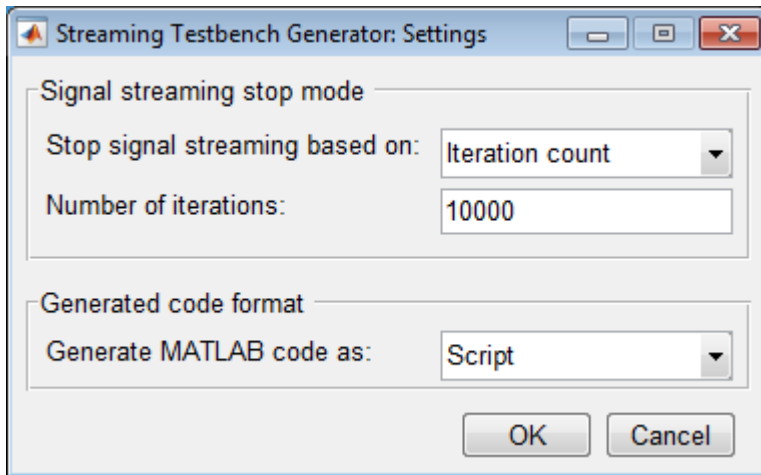
You can also tune some of the parameters used in your algorithm during testbench execution. To use the **Parameter Tuning UI**, check the **Enable parameter tuning** check box under the **User Algorithm** and click **Edit parameters table** to add the details of your tunable parameters before you generate testbench code. Also, make sure that your user algorithm handles parameter tuning during execution. See the MATLAB code for `hTestbenchVariableBandwithFIR` for an example of how to make your user algorithm work with parameter tuning.

Create Composite System object

This example shows how to create a System object composed of other System objects. In this example, build a multi-notch filter using two dsp.NotchPeakFilter System objects. Multi-notch filters are used in many applications. Examples include audio phasers and applications that require the removal of multiple interfering tones.

Create Multi-Notch Filter

"Create Moving Average System object" on page 1-59 explains in detail how to write a System object using a template file. In this example, the entire System object is provided for convenience in dspdemo.CompositeObj_MultiNotch. To view the MATLAB code, at the command prompt enter:

edit dspdemo.CompositeObj_MultiNotch

dspdemo.CompositeObj_MultiNotch has five public properties. Public properties are accessible to every user. The first property, SampleRate, is a nontunable property. Nontunable properties cannot change when the filter is locked (after you use step on the filter). The remaining properties, CenterFrequency1, CenterFrequency2, QualityFactor1, and QualityFactor2 control settings in the two notch filters contained in dspdemo.CompositeObj_MultiNotch. These four properties are tunable. You can change the values of these properties while streaming data.

Set Up the Multi-Notch Filters

dspdemo.CompositeObj_MultiNotch uses dsp.NotchPeakFilter to design the two notch filters. The notch filters are set up in the setupImpl method.

```
methods (Access=protected)
        function setupImpl(obj,~)
            % Construct two notch filters with default values
            obj.NotchFilter1 = dsp.NotchPeakFilter(...
                'Specification', 'Quality factor and center frequency',...
                'CenterFrequency',400);
            obj.NotchFilter2 = dsp.NotchPeakFilter(...
                'Specification', 'Quality factor and center frequency',...
                'CenterFrequency',800);
        end
end
```

Contain System Objects as Private Properties

The ability to create more than one instance of a System object and having each instance manage its own state is one of the biggest advantages of using System objects over functions. The private properties `NotchFilter1` and `NotchFilter2` are used to store the two notch filters.

```
properties (Access=private)
        % This example class contains two notch filters (more can be added
        % in the same way)
        NotchFilter1
        NotchFilter2
end
```

Work with Dependent Properties

The `SampleRate` property as well as the remaining four public properties are implemented as *dependent* properties in this example. Whenever you assign a value to one of the dependent properties, the value is set in the corresponding single-notch filter. When you read one of the dependent properties, the value is read from the corresponding single-notch filter. Find the following code block in `dspdemo.CompositeObj_MultiNotch`.

```
properties (Dependent)
        %CenterFrequency1 Center frequency of first notch
        %   Specify the first notch center frequency as a finite positive
        %   numeric scalar in Hertz. The default is 400 Hz. This property
        %   is tunable.
        CenterFrequency1;
        %QualityFactor1 Quality factor of first notch
        %   Specify the quality factor (Q factor) for the first notch
        %   filter. The default value is 5. This property is tunable.
        QualityFactor1;
        %CenterFrequency2 Center frequency of second notch
        %   Specify the second notch center frequency as a finite positive
        %   numeric scalar in Hertz. The default is 800 Hz. This property
        %   is tunable.
        CenterFrequency2;
        %QualityFactor2 Quality factor of second notch
        %   Specify the quality factor (Q factor) for the first notch
        %   filter. The default value is 5. This property is tunable.
        QualityFactor2;
end
```

Use the Multi-Notch Filter - Initialization

To use `dspdemo.CompositeObj_MultiNotch`, initialize the filter and any other required components. In this example, initialize an audio file reader, a multi-notch filter, a transfer function estimator, an array plotter, and an audio player.

```
FrameSize = 1024;
AFR = dsp.AudioFileReader('guitar10min.ogg','SamplesPerFrame',FrameSize);
Fs  = AFR.SampleRate;

MNF = dspdemo.CompositeObj_MultiNotch('SampleRate',Fs);

TFE = dsp.TransferFunctionEstimator(...
    'FrequencyRange','onesided','SpectralAverages',5);
AP  = dsp.ArrayPlot('PlotType','Line','YLimits',[-85 15],...
    'SampleIncrement',Fs/FrameSize);
P   = audioDeviceWriter;
```

Use the Multi-Notch Filter - Streaming

To illustrate the tunability of the two notch filter center frequencies, vary the center frequencies sinusoidally in a loop. The starting center frequencies are 500 and 2000 Hz. The first center frequency oscillates over the range [100, 900] Hz with a frequency of 0.2 Hz. The second center frequency oscillates over the range [1200, 2800] Hz with a frequency of 0.5 Hz. Because `CenterFrequency1` and `CenterFrequency2` are dependent properties, modifying their values in the loop changes the center frequencies in the two notch filters contained in `dspdemo.CompositeObj_MultiNotch`. To visualize the multi-notch filter, estimate and plot the transfer function continuously. The quality factors remain constant. The simulation runs for 20 seconds.

```
MNF.QualityFactor1 = .5;
MNF.QualityFactor2 = 1;

f0 = 0.2;
f1 = 0.5;

k = 0;
tic,
while toc < 20
    x = step(AFR);
    t = k*FrameSize/Fs;
    k = k+1;
```

```
MNF.CenterFrequency1 =  500 + 400*sin(2*pi*f0*t);
MNF.CenterFrequency2 = 2000 + 800*sin(2*pi*f1*t);
CF1(k) = MNF.CenterFrequency1;
CF2(k) = MNF.CenterFrequency2;

y = step(MNF,x);

H = step(TFE,x(:,1),y(:,1));
magdB = 20*log10(abs(H));
step(AP,magdB);
step(P,y);
end
```
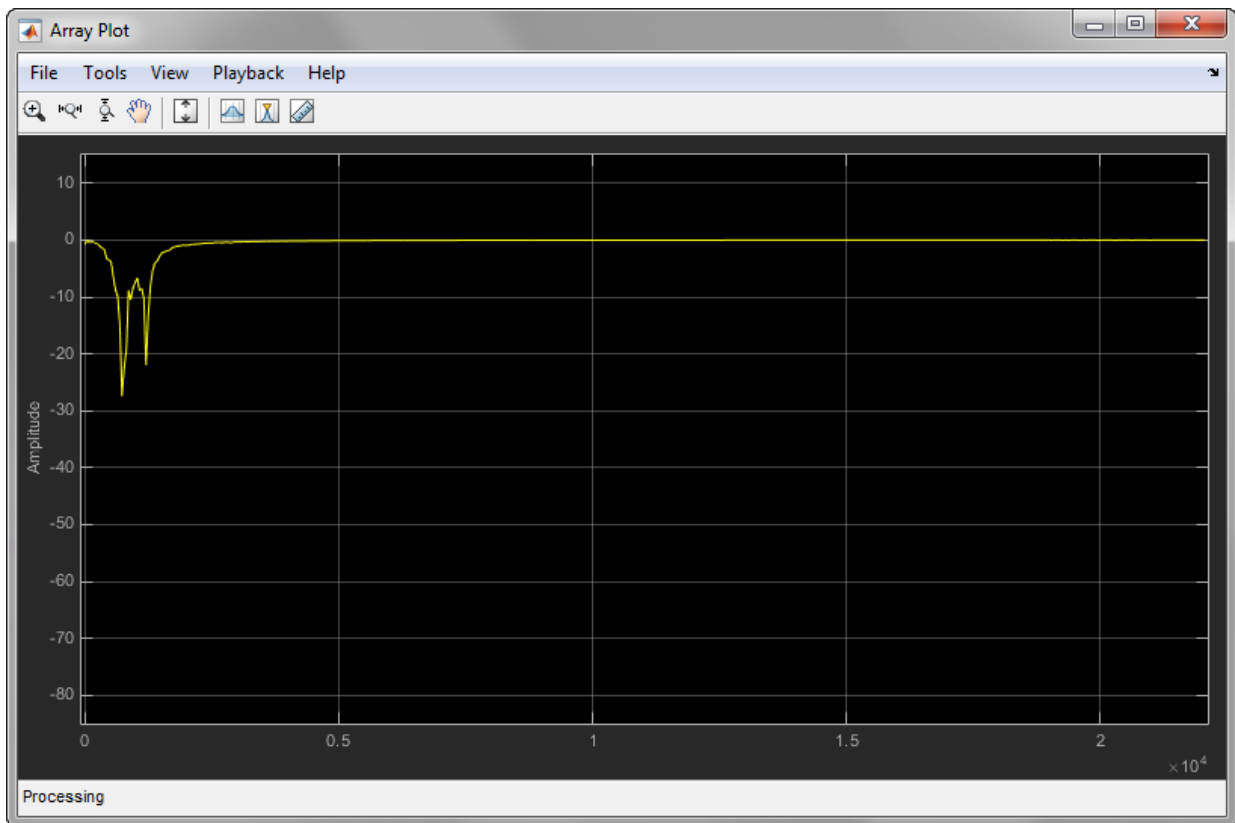

Execute this code to show how the two notch center frequencies varied over the
simulation.

```
subplot(2,1,1)
plot(CF1);
title('Center Frequency 1');
ylabel('Notch CF (Hz)');
subplot(2,1,2)
plot(CF2);
title('Center Frequency 2');
ylabel('Notch CF (Hz)');
xlabel('Iteration')
```

2
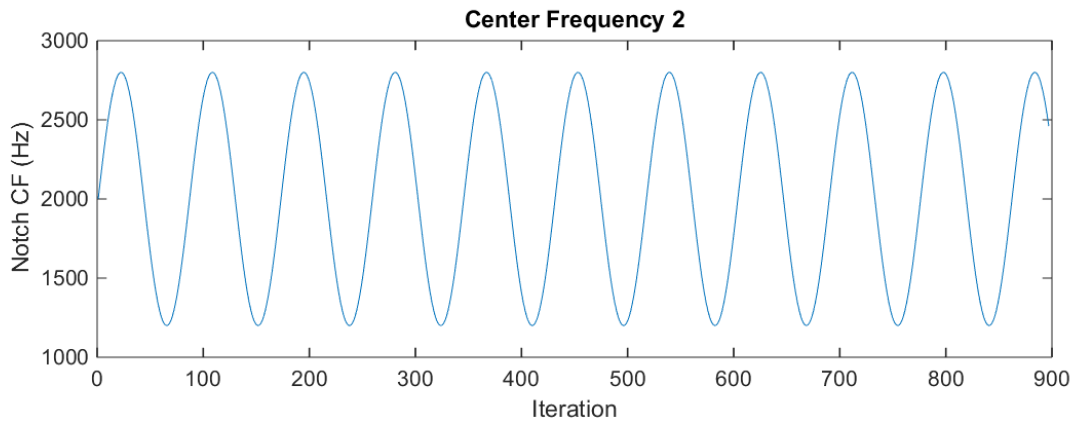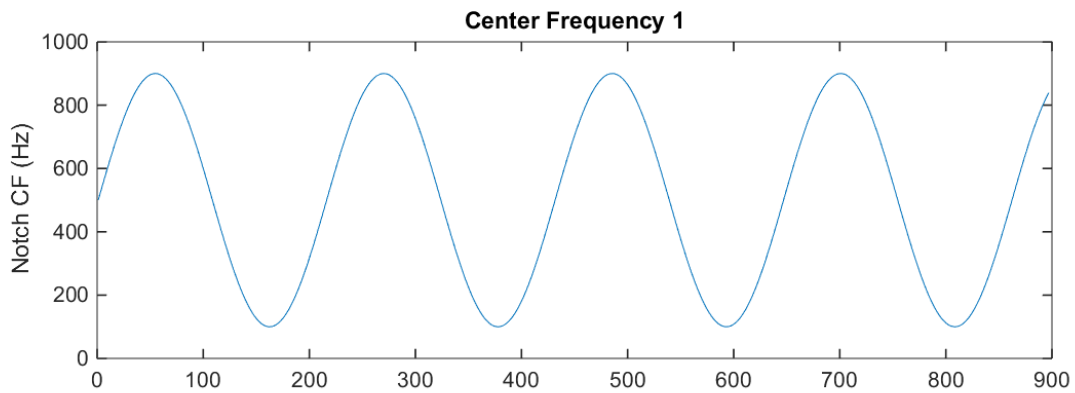
Input, Output, and Display

Learn how to input, output and display data and signals with DSP System Toolbox.

Discrete-Time Signals

In this section...
"Time and Frequency Terminology" on page 2-2
"Recommended Settings for Discrete-Time Simulations" on page 2-3
"Other Settings for Discrete-Time Simulations" on page 2-6

Time and Frequency Terminology

Simulink models can process both discrete-time and continuous-time signals. Models built with DSP System Toolbox software are often intended to process discrete-time signals only. A discrete-time signal is a sequence of values that correspond to particular instants in time. The time instants at which the signal is defined are the signal's *sample times*, and the associated signal values are the signal's *samples*. Traditionally, a discrete-time signal is considered to be undefined at points in time between the sample times. For a periodically sampled signal, the equal interval between any pair of consecutive sample times is the signal's *sample period*, T_s. The *sample rate*, F_s, is the reciprocal of the sample period, or $1/T_s$. The sample rate is the number of samples in the signal per second.

The 7.5-second triangle wave segment below has a sample period of 0.5 second, and sample times of 0.0, 0.5, 1.0, 1.5, ...,7.5. The sample rate of the sequence is therefore 1/0.5, or 2 Hz.
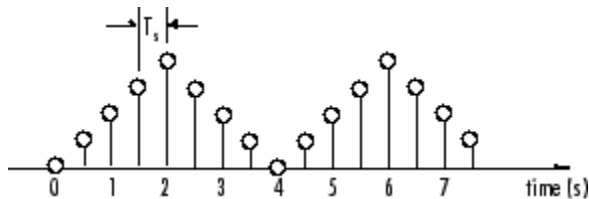
A number of different terms are used to describe the characteristics of discrete-time signals found in Simulink models. These terms, which are listed in the following table, are frequently used to describe the way that various blocks operate on sample-based and frame-based signals.

Term	Symbol	Units	Notes
Sample period	T_s T_{si}	Seconds	The time interval between consecutive samples in a sequence, as the input to a block (T_{si}) or the output from a block (T_{so}).

Term	Symbol	Units	Notes
	T_{so}		
Frame period	T_f T_{fi} T_{fo}	Seconds	The time interval between consecutive frames in a sequence, as the input to a block (T_{fi}) or the output from a block (T_{fo}).
Signal period	T	Seconds	The time elapsed during a single repetition of a periodic signal.
Sample frequency	F_s	Hz (samples per second)	The number of samples per unit time, $F_s = 1/T_s$.
Frequency	f	Hz (cycles per second)	The number of repetitions per unit time of a periodic signal or signal component, $f = 1/T$.
Nyquist rate		Hz (cycles per second)	The minimum sample rate that avoids aliasing, usually twice the highest frequency in the signal being sampled.
Nyquist frequency	f_{nyq}	Hz (cycles per second)	Half the Nyquist rate.
Normalized frequency	f_n	Two cycles per sample	Frequency (linear) of a periodic signal normalized to half the sample rate, $f_n = \omega/\pi = 2f/F_s$.
Angular frequency	Ω	Radians per second	Frequency of a periodic signal in angular units, $\Omega = 2\pi f$.
Digital (normalized angular) frequency	ω	Radians per sample	Frequency (angular) of a periodic signal normalized to the sample rate, $\omega = \Omega/F_s = \pi f_n$.

Note In the Block Parameters dialog boxes, the term *sample time* is used to refer to the *sample period*, T_s. For example, the **Sample time** parameter in the Signal From Workspace block specifies the imported signal's sample period.

Recommended Settings for Discrete-Time Simulations

Simulink allows you to select from several different simulation solver algorithms. You can access these solver algorithms from a Simulink model:

1 In the Simulink model window, from the **Simulation** menu, select **Model Configuration Parameters**. The **Configuration Parameters** dialog box opens.

2 In the **Select** pane, click **Solver**.

The selections that you make here determine how discrete-time signals are processed in Simulink. The recommended **Solver options** settings for signal processing simulations are

- **Type**: `Fixed-step`
- **Solver**: `Discrete (no continuous states)`
- **Fixed step size (fundamental sample time)**: `auto`
- **Tasking mode for periodic sample times**: `SingleTasking`
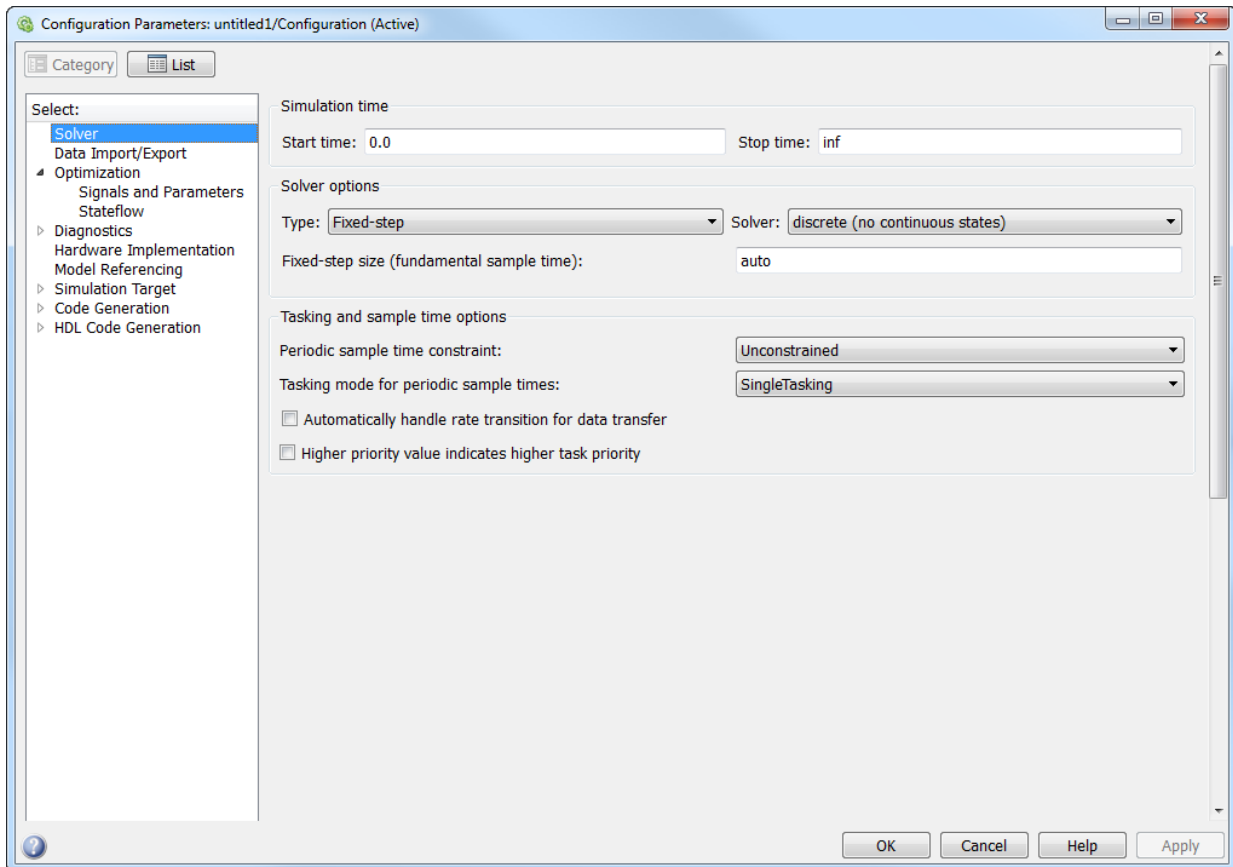
You can automatically set the above solver options for all new models by using DSP Simulink model templates. For more information, see "Configure the Simulink Environment for Signal Processing Models" in the DSP System Toolbox documentation.

In `Fixed-step SingleTasking` mode, discrete-time signals differ from the prototype described in "Time and Frequency Terminology" on page 2-2 by remaining defined between sample times. For example, the representation of the discrete-time triangle wave looks like this.
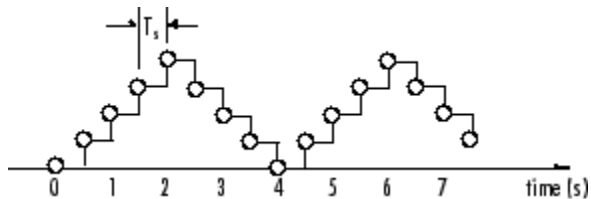
The above signal's value at t=3.112 seconds is the same as the signal's value at t=3 seconds. In `Fixed-step SingleTasking` mode, a signal's sample times are the instants where the signal is allowed to change values, rather than where the signal is defined. Between the sample times, the signal takes on the value at the previous sample time.

As a result, in `Fixed-step SingleTasking` mode, Simulink permits cross-rate operations such as the addition of two signals of different rates. This is explained further in "Cross-Rate Operations" on page 2-7.

Other Settings for Discrete-Time Simulations

It is useful to know how the other solver options available in Simulink affect discrete-time signals. In particular, you should be aware of the properties of discrete-time signals under the following settings:

- **Type:** `Fixed-step`, **Mode:** `MultiTasking`
- **Type:** `Variable-step` (the Simulink default solver)
- **Type:** `Fixed-step`, **Mode:** `Auto`

When the `Fixed-step MultiTasking` solver is selected, discrete signals in Simulink are undefined between sample times. Simulink generates an error when operations attempt to reference the undefined region of a signal, as, for example, when signals with different sample rates are added.

When the `Variable-step` solver is selected, discrete time signals remain defined between sample times, just as in the `Fixed-step SingleTasking` case described in "Recommended Settings for Discrete-Time Simulations" on page 2-3. When the `Variable-step` solver is selected, cross-rate operations are allowed by Simulink.

In the `Fixed-step Auto` setting, Simulink automatically selects a tasking mode, single-tasking or multitasking, that is best suited to the model. See "Simulink Tasking Mode" on

page 3-68 for a description of the criteria that Simulink uses to make this decision. For the typical model containing multiple rates, Simulink selects the multitasking mode.

Cross-Rate Operations

When the `Fixed-step MultiTasking` solver is selected, discrete signals in Simulink are undefined between sample times. Therefore, to perform cross-rate operations like the addition of two signals with different sample rates, you must convert the two signals to a common sample rate. Several blocks in the Signal Operations and Multirate Filters libraries can accomplish this task. See "Convert Sample and Frame Rates in Simulink" on page 3-19 for more information. Rate change can happen implicitly, depending on diagnostic settings. See "Multitask rate transition", "Single task rate transition". However, this is not recommended. By requiring explicit rate conversions for cross-rate operations in discrete mode, Simulink helps you to identify sample rate conversion issues early in the design process.

When the `Variable-step` solver or `Fixed-step SingleTasking` solver is selected, discrete time signals remain defined between sample times. Therefore, if you sample the signal with a rate or phase that is different from the signal's own rate and phase, you will still measure meaningful values:

1   At the MATLAB command line, type `ex_sum_tut1`.

     The Cross-Rate Sum Example model opens. This model sums two signals with different sample periods.
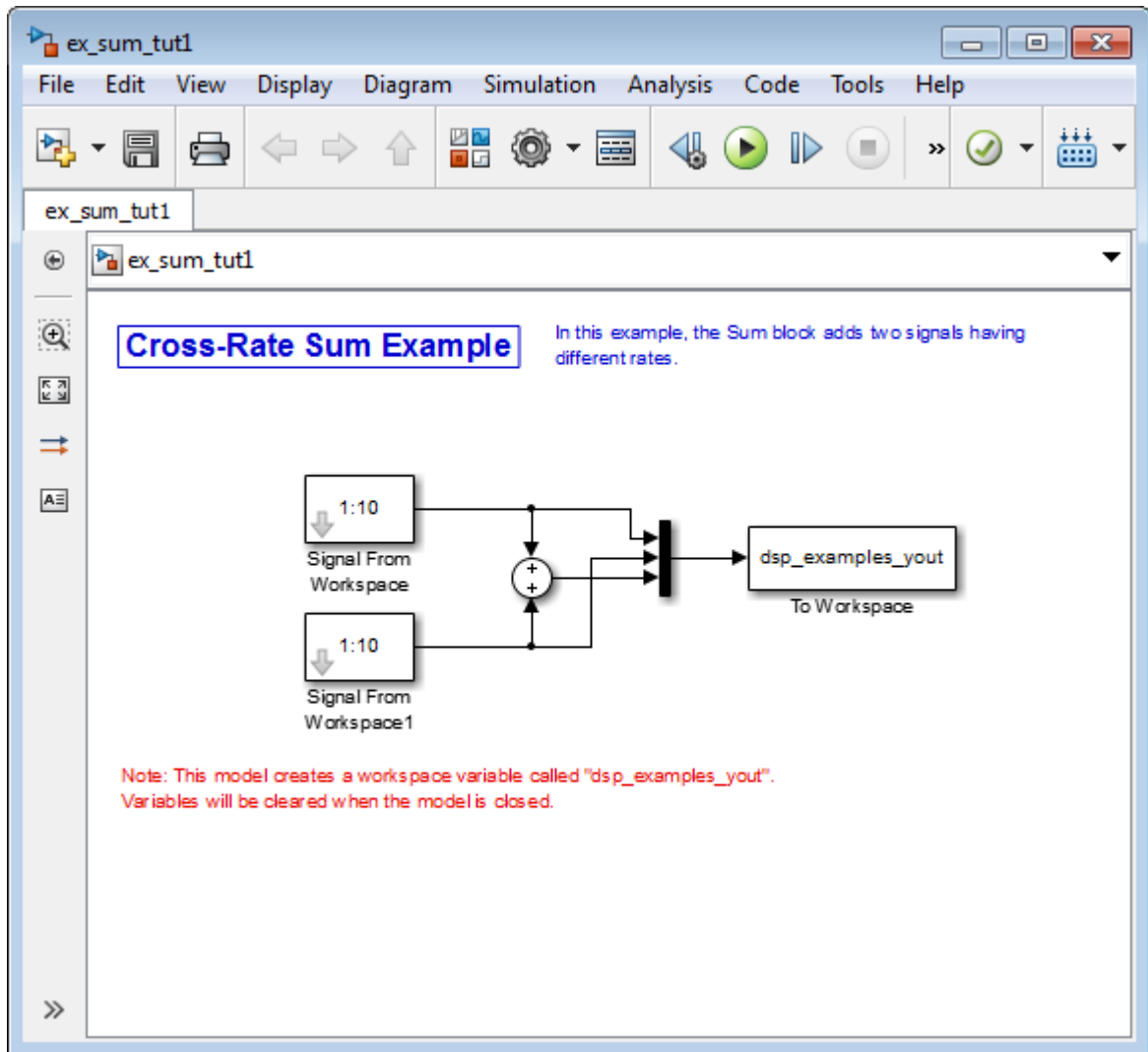
2. Double-click the upper Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

3. Set the **Sample time** parameter to 1.

   This creates a fast signal, $(T_s=1)$, with sample times 1, 2, 3, ...

4 Double-click the lower Signal From Workspace block

5 Set the **Sample time** parameter to 2.

This creates a slow signal, (T_s=2), with sample times 1, 3, 5, ...

6 From the **Display** menu choose **Sample Time** > **Colors**.

Checking the **Colors** option allows you to see the different sampling rates in action. For more information about the color coding of the sample times see "View Sample Time Information" in the Simulink documentation.

7 Run the model.

Note Using the DSP Simulink model templates with cross-rate operations generates errors even though the `Fixed-step SingleTasking` solver is selected. This is due to the fact that **Single task rate transition** is set to `error` in the **Sample Time** pane of the **Diagnostics** section of the **Configuration Parameters** dialog box.

---

8 At the MATLAB command line, type `dsp_examples_yout`.

The following output is displayed:

```
dsp_examples_yout =
      1      1      2
      2      1      3
      3      2      5
      4      2      6
      5      3      8
      6      3      9
      7      4     11
      8      4     12
      9      5     14
     10      5     15
      0      6      6
```

The first column of the matrix is the fast signal, (T_s=1). The second column of the matrix is the slow signal (T_s=2). The third column is the sum of the two signals. As expected, the slow signal changes once every 2 seconds, half as often as the fast signal. Nevertheless, the slow signal is defined at every moment because Simulink holds the previous value of the slower signal during time instances that the block doesn't run.

In general, for `Variable-step` and `Fixed-step SingleTasking` modes, when you measure the value of a discrete signal between sample times, you are observing the value of the signal at the previous sample time.

Continuous-Time Signals

In this section...
"Continuous-Time Source Blocks" on page 2-11
"Continuous-Time Nonsource Blocks" on page 2-11

Continuous-Time Source Blocks

Most signals in a signal processing model are discrete-time signals. However, many blocks can also operate on and generate continuous-time signals, whose values vary continuously with time. Source blocks are those blocks that generate or import signals in a model. Most source blocks appear in the Sources library. The sample period for continuous-time source blocks is set internally to zero. This indicates a continuous-time signal. The Simulink `Signal Generator` and `Constant` blocks are examples of continuous-time source blocks. Continuous-time signals are rendered in black when, from the **Display** menu, you point to **Sample Time** and select **Colors**.

When connecting continuous-time source blocks to discrete-time blocks, you might need to interpose a Zero-Order Hold block to discretize the signal. Specify the desired sample period for the discrete-time signal in the **Sample time** parameter of the Zero-Order Hold block.
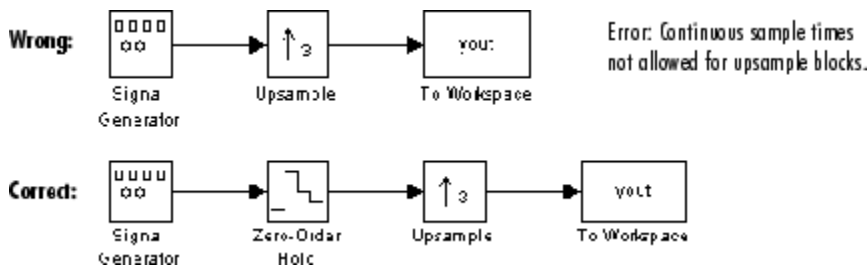
Continuous-Time Nonsource Blocks

Most nonsource blocks in DSP System Toolbox software accept continuous-time signals, and all nonsource blocks inherit the sample period of the input. Therefore, continuous-time inputs generate continuous-time outputs. Blocks that are not capable of accepting continuous-time signals include the Biquad Filter, Discrete FIR Filter, FIR Decimation, and FIR Interpolation blocks.

Create Signals for Sample-Based Processing

In this section...
"Create Signals Using Constant Block" on page 2-13
"Create Signals Using Signal From Workspace Block" on page 2-14

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.

When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an M-by-N matrix as multichannel signal with $M*N$ independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

This page discusses creating signals for sample-based processing using the `Constant` block and the `Signal From Workspace` block. Note that the block receiving this signal implements sample-based processing or frame-based processing on the signal based on the parameters set in the block dialog box.

Create Signals Using Constant Block

1   Create a new Simulink model.

2   From the Sources library, click-and-drag a Constant block into the model.

3   From the Sinks library, click-and-drag a Display block into the model.

4   Connect the two blocks.

5   Double-click the Constant block, and set the block parameters as follows:

- **Constant value** = [1 2 3; 4 5 6]
- **Interpret vector parameters as 1–D** = Clear this check box
- **Sample time** = 1

Based on these parameters, the Constant block outputs a constant, discrete-valued, 2-by-3 matrix signal with a sample period of 1 second.

The Constant block's **Constant value** parameter can be any valid MATLAB variable or expression that evaluates to a matrix.

6   Save these parameters and close the dialog box by clicking **OK**.

7   From the **Display** menu, point to **Signals & Ports** and select **Signal Dimensions**.

8   Run the model and expand the Display block so you can view the entire signal.

You have now successfully created a six-channel signal with a sample period of 1 second.

To view the model you just created, and to learn how to create a 1–D vector signal from the block diagram you just constructed, continue to the next section.

Create an Unoriented Vector Signal

You can create an unoriented vector by modifying the block diagram you constructed in the previous section:

1   To add another signal to your model, copy the block diagram you created in the previous section and paste it below the existing signal in your model.

2   Double-click the Constant1 block, and set the block parameters as follows:

- **Constant value** = [1 2 3 4 5 6]
- **Interpret vector parameters as 1–D** = Check this box
- **Sample time** = 1

> **3** Save these parameters and close the dialog box by clicking **OK**.
>
> **4** Run the model and expand the Display1 block so you can view the entire signal.
>
> Your model should now look similar to the following figure. You can also open this model by typing `ex_usingcnstblksb` at the MATLAB command line.
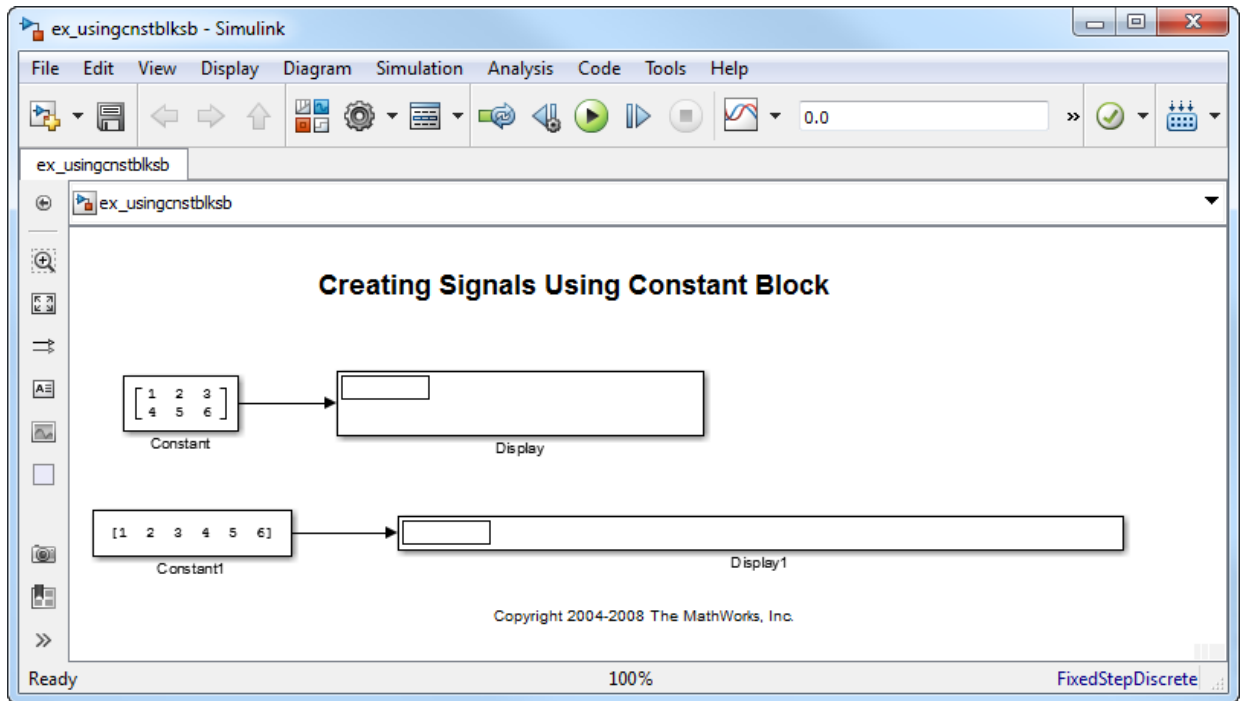
The Constant1 block generates a length-6 unoriented vector signal. This means that the output is not a matrix. However, most nonsource signal processing blocks interpret a length-M unoriented vector as an M-by-1 matrix (column vector).

Create Signals Using `Signal From Workspace` Block

This topic discusses how to create a four-channel signal for sample-based processing with a sample period of 1 second using the `Signal From Workspace` block:

1 Create a new Simulink model.

2 From the Sources library, click-and-drag a Signal From Workspace block into the model.

3 From the Simulink Sinks library, click-and-drag a To Workspace block into the model.

4 Connect the two blocks.

5 Double-click the Signal From Workspace block, and set the block parameters as follows:

- **Signal** = `cat(3,[1 -1;0 5],[2 -2;0 5],[3 -3;0 5])`
- **Sample time** = `1`
- **Samples per frame** = `1`
- **Form output after final data value by** = `Setting to zero`

Based on these parameters, the Signal From Workspace block outputs a four-channel signal with a sample period of 1 second. After the block has output the signal, all subsequent outputs have a value of zero. The four channels contain the following values:

- Channel 1: 1, 2, 3, 0, 0,...
- Channel 2: -1, -2, -3, 0, 0,...
- Channel 3: 0, 0, 0, 0, 0,...
- Channel 4: 5, 5, 5, 0, 0,...

6 Save these parameters and close the dialog box by clicking **OK**.

7 From the **Display** menu, point to **Signals & Ports**, and select **Signal Dimensions**.

8 Run the model.

The following figure is a graphical representation of the model's behavior during simulation. You can also open the model by typing `ex_usingsfwblksb` at the MATLAB command line.

Creating Sample-Based Signals Using the Signal From Workspace Block

cat(3,[1 -1;0 5],[2 -2;0 5],[3 -3;0 5]) $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} 3 & -3 \\ 0 & 5 \end{bmatrix}\begin{bmatrix} 2 & -2 \\ 0 & 5 \end{bmatrix}\begin{bmatrix} 1 & -1 \\ 0 & 5 \end{bmatrix}$ yout   Info

Signal From Workspace   t=3   t=2   t=1   t=0   To Workspace   Doc Example Info

First matrix output
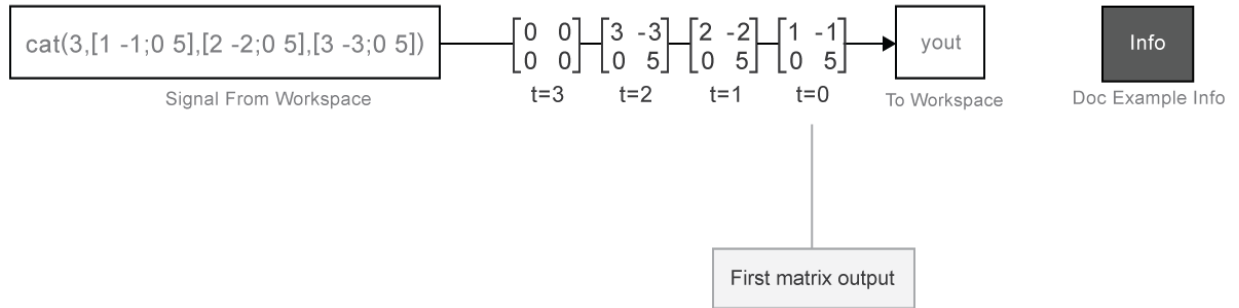
9 At the MATLAB command line, type yout.

The following is a portion of the output:

```
yout(:,:,1) =

1     -1
0      5

yout(:,:,2) =

2     -2
0      5

yout(:,:,3) =

3     -3
0      5

yout(:,:,4) =

0      0
0      0
```

You have now successfully created a four-channel signal with sample period of 1 second using the Signal From Workspace block. This signal is used for sample-based processing.

More About

Sample-Based Row Vector Processing Changes

There are DSP System Toolbox blocks that handle sample-based row vector inputs in a special way.

One category of these blocks have a **Treat sample-based row input as a column** check box which allows you to explicitly specify how the block should treat sample-based row vector inputs. Expand the following section for a full list of these blocks.

Blocks with a Check Box

- `Maximum`
- `Mean`
- `Median`
- `Minimum`
- `Normalization`
- `RMS`
- `Standard Deviation`
- `Variance`

In a future release these blocks will produce a warning when you provide them with a sample-based row vector input, and eventually, their behavior will change.

You can prepare your models for the upcoming change by using `slupdate`. If the function detects any blocks that have a **Treat sample-based row input as a column** check box, it performs the following actions:

- If the input to the block is a sample-based row vector, and the **Treat sample-based row input as a column** check box is selected, the `slupdate` function places a `Transpose` block in front of the affected block. The Transpose block transposes the sample-based row vector into a column vector, which is then input into the affected block. Transposing the input signal ensures that your model will produce the same results in future releases.
- If the **Treat sample-based row input as a column** check box is not selected, or if the input to the block is not a sample-based row vector, function takes no action. Your model will continue to work as expected in future releases.

If the `slupdate` function detects any blocks that automatically treat sample-based row vectors as a column, it performs the following actions:

- If the input to the block is a sample-based row vector, the `slupdate` function places a `Transpose` block in front of the affected block. The Transpose block transposes the sample-based row vector into a column vector, which is then input into the affected block. Transposing the input signal ensures that your model will produce the same results in future releases.

- If the input to the block is not a sample-based row vector, the `slupdate` function takes no action. Your model will continue to work as expected in future releases.

See Also

"Frame-based processing changes"

Create Signals for Frame-Based Processing

In this section...
"Create Signals Using Sine Wave Block" on page 2-21
"Create Signals Using Signal From Workspace Block" on page 2-24

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.

When you configure a block to perform frame-based processing, the block interprets an M-by-1 vector as a single-channel signal containing M samples per frame. Similarly, the block interprets an M-by-N matrix as a multichannel signal with N independent channels and M samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.

Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

This page discusses creating signals for frame-based processing using the `Sine Wave` block and the `Signal From Workspace` block. Note that the block receiving this signal implements sample-based processing or frame-based processing on the signal based on the parameters set in the block dialog box.

Create Signals Using Sine Wave Block

1 Create a new Simulink model.

2 From the Sources library, click-and-drag a Sine Wave block into the model.

3 From the Matrix Operations library, click-and-drag a Matrix Sum block into the model.

4 From the Simulink Sinks library, click-and-drag a To Workspace block into the model.

5 Connect the blocks in the order in which you added them to your model.

6 Double-click the Sine Wave block, and set the block parameters as follows:

- **Amplitude** = [1 3 2]
- **Frequency** = [100 250 500]
- **Sample time** = 1/5000
- **Samples per frame** = 64

Based on these parameters, the Sine Wave block outputs three sinusoids with amplitudes 1, 3, and 2 and frequencies 100, 250, and 500 Hz, respectively. The sample period, 1/5000, is 10 times the highest sinusoid frequency, which satisfies the Nyquist criterion. The frame size is 64 for all sinusoids, and, therefore, the output has 64 rows.

7 Save these parameters and close the dialog box by clicking **OK**.

You have now successfully created a three-channel signal, with 64 samples per each frame, using the Sine Wave block. The rest of this procedure describes how to add these three sinusoids together.

8 Double-click the Matrix Sum block. Set the **Sum over** parameter to Specified dimension, and set the **Dimension** parameter to 2. Click **OK**.

9 From the **Display** menu, point to **Signals & Ports**, and select **Signal Dimensions**.

10 Run the model.

Your model should now look similar to the following figure. You can also open the model by typing ex_usingsinwaveblkfb at the MATLAB command line.
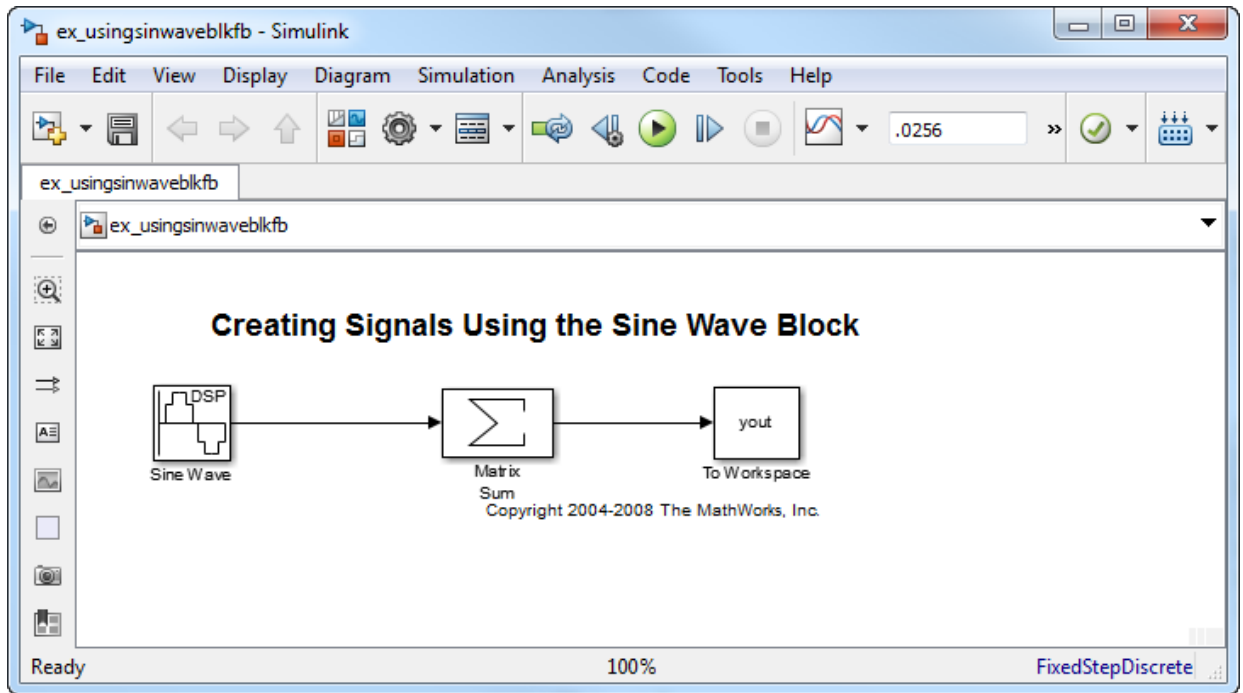
The three signals are summed point-by-point by a Matrix Sum block. Then, they are exported to the MATLAB workspace.

11 At the MATLAB command line, type `plot(yout(1:100))`.

Your plot should look similar to the following figure.
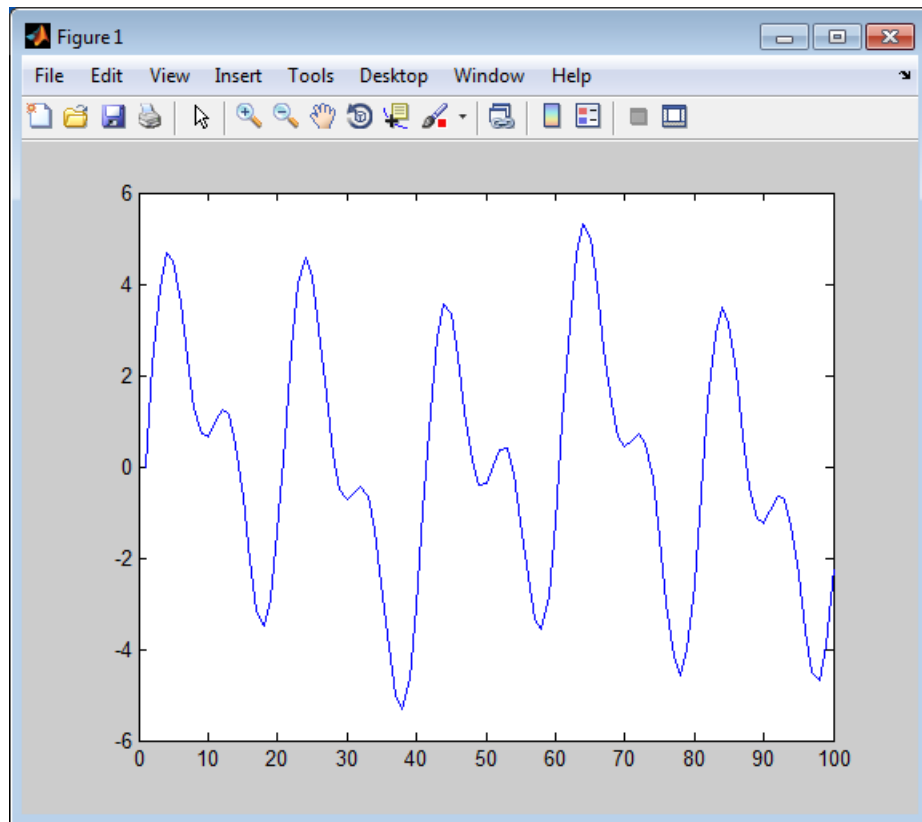
This figure represents a portion of the sum of the three sinusoids. You have now added the channels of a three-channel signal together and displayed the results in a figure window.

Create Signals Using `Signal From Workspace` Block

Frame-based processing can significantly improve the performance of your model by decreasing the amount of time it takes your simulation to run. This topic describes how to create a two-channel signal with a sample period of 1 second, a frame period of 4 seconds, and a frame size of 4 samples using the `Signal From Workspace` block.

1   Create a new Simulink model.

2   From the Sources library, click-and-drag a `Signal From Workspace` block into the model.

3   From the Simulink Sinks library, click-and-drag a `To Workspace` block into the model.

4   Connect the two blocks.

5   Double-click the `Signal From Workspace` block, and set the block parameters as follows.

- **Signal** = `[1:10; 1 1 0 0 1 1 0 0 1 1]'`
- **Sample time** = `1`
- **Samples per frame** = `4`
- **Form output after final data value by** = `Setting to zero`

Based on these parameters, the `Signal From Workspace` block outputs a two-channel signal with a sample period of 1 second, a frame period of 4 seconds, and a frame size of four samples. After the block outputs the signal, all subsequent outputs have a value of zero. The two channels contain the following values:

- Channel 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0,...
- Channel 2: 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0,...

6   Save these parameters and close the dialog box by clicking **OK**.

7   From the **Display** menu, point to **Signals & Ports**, and select **Signal Dimensions**.

8   Run the model.

The following figure is a graphical representation of the model's behavior during simulation. You can also open the model by typing `ex_usingsfwblkfb` at the MATLAB command line.
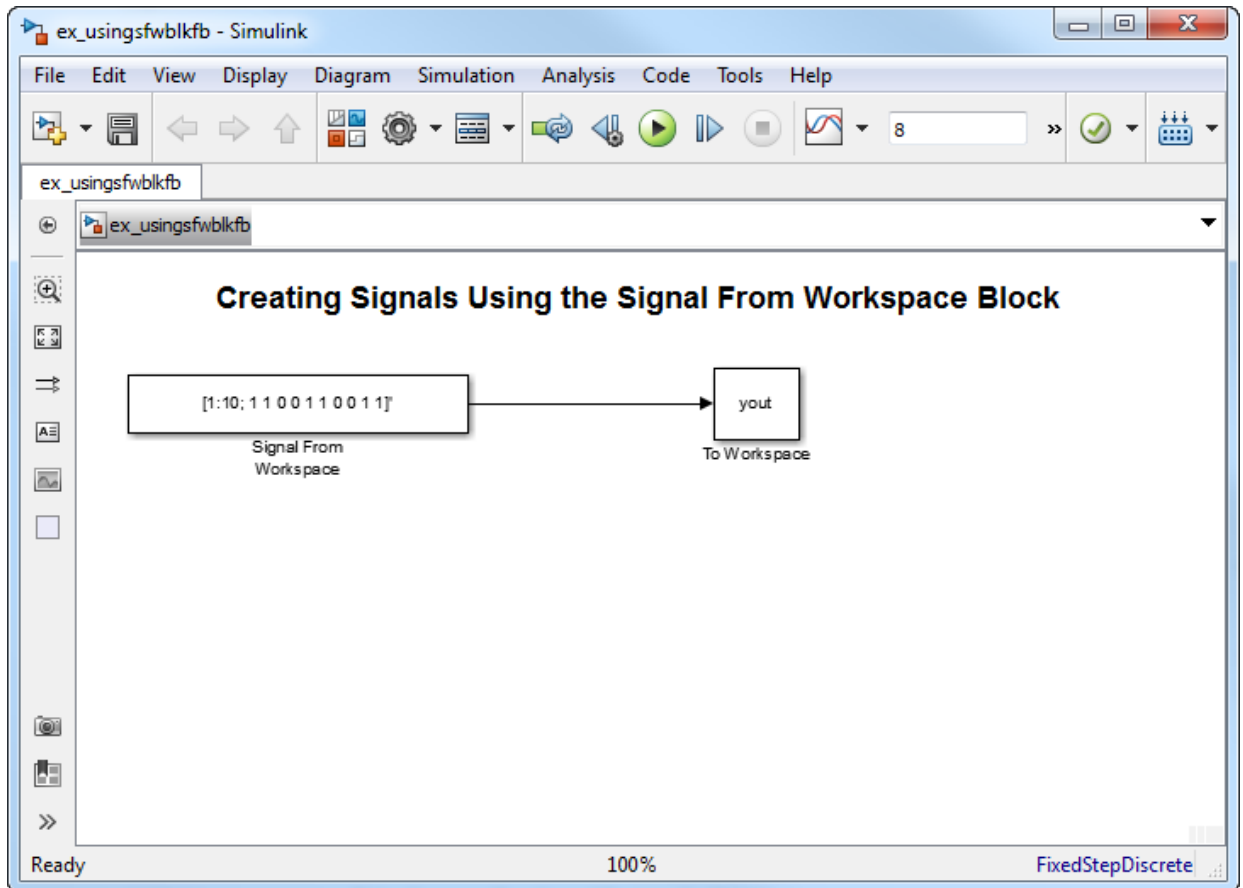
9 At the MATLAB command line, type `yout`.

The following is the output displayed at the MATLAB command line.

```
yout =

        1       1
        2       1
        3       0
        4       0
        5       1
        6       1
```

```
 7    0
 8    0
 9    1
10    1
 0    0
 0    0
```

Note that zeros were appended to the end of each channel. You have now successfully created a two-channel signal and exported it to the MATLAB workspace.

More About

- "Create Signals for Sample-Based Processing" on page 2-12
- "Create Multichannel Signals for Sample-Based Processing" on page 2-28
- "Create Multichannel Signals for Frame-Based Processing" on page 2-35
- "Deconstruct Multichannel Signals for Sample-Based Processing" on page 2-40
- "Deconstruct Multichannel Signals for Frame-Based Processing" on page 2-49

Create Multichannel Signals for Sample-Based Processing
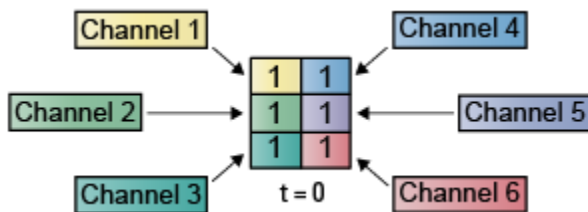
In this section...
"Multichannel Signals for Sample-Based Processing" on page 2-29
"Create Multichannel Signals by Combining Single-Channel Signals" on page 2-29
"Create Multichannel Signals by Combining Multichannel Signals" on page 2-32

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.

When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an M-by-N matrix as multichannel signal with $M*N$ independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

Multichannel Signals for Sample-Based Processing

When you want to perform the same operations on several independent signals, you can group those signals together as a multichannel signal. For example, if you need to filter each of four independent signals using the same direct-form II transpose filter, you can combine the signals into a multichannel signal, and connect the signal to a single Digital Filter Design block. The block applies the filter to each channel independently.

Multiple independent signals can be combined into a single multichannel signal using the `Concatenate` block. In addition, several multichannel signals can be combined into a single multichannel signal using the same technique.

Create Multichannel Signals by Combining Single-Channel Signals

You can combine individual signals into a multichannel signal by using the `Matrix Concatenate` block in the Simulink Math Operations library:

1 Open the Matrix Concatenate Example 1 model by typing

`ex_cmbsnglchsbsigs`

at the MATLAB command line.
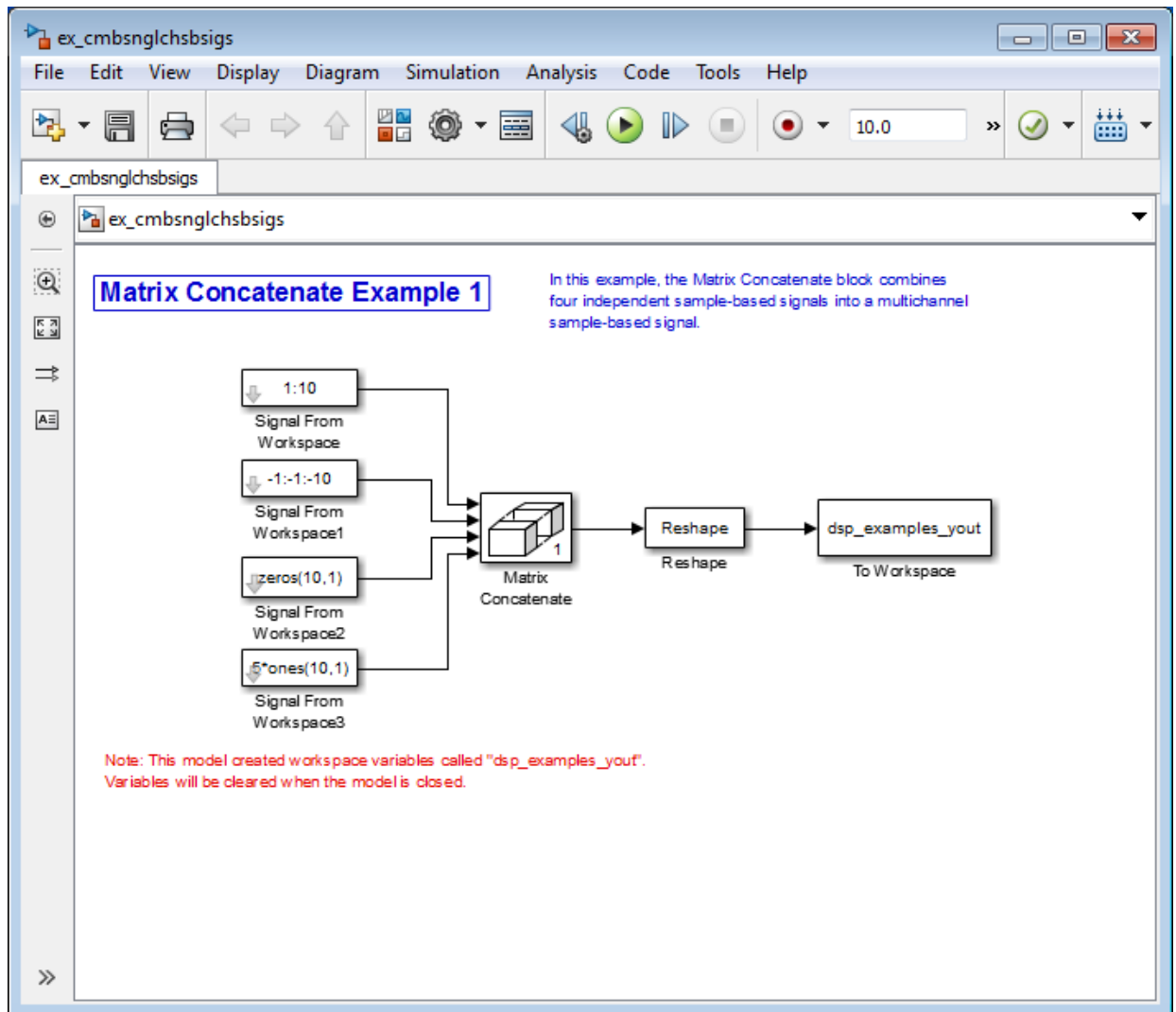
2   Double-click the Signal From Workspace block, and set the **Signal** parameter to
    1:10. Click **OK**.

3   Double-click the Signal From Workspace1 block, and set the **Signal** parameter to
    -1:-1:-10. Click **OK**.

4 Double-click the Signal From Workspace2 block, and set the **Signal** parameter to `zeros(10,1)`. Click **OK**.

5 Double-click the Signal From Workspace3 block, and set the **Signal** parameter to `5*ones(10,1)`. Click **OK**.

6 Double-click the Matrix Concatenate block. Set the block parameters as follows, and then click **OK**:

- **Number of inputs** = 4
- **Mode** = Multidimensional array
- **Concatenate dimension** = 1

7 Double-click the Reshape block. Set the block parameters as follows, and then click **OK**:

- **Output dimensionality** = Customize
- **Output dimensions** = [2,2]

8 Run the model.

Four independent signals are combined into a 2-by-2 multichannel matrix signal.

Each 4-by-1 output from the Matrix Concatenate block contains one sample from each of the four input signals at the same instant in time. The Reshape block rearranges the samples into a 2-by-2 matrix. Each element of this matrix is a separate channel.

Note that the Reshape block works column wise, so that a column vector input is reshaped as shown below.
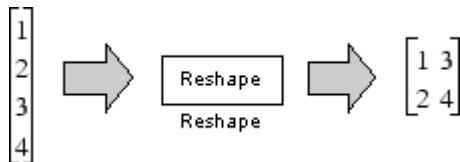
The 4-by-1 matrix output by the Matrix Concatenate block and the 2-by-2 matrix output by the Reshape block in the above model represent the same four-channel signal. In some cases, one representation of the signal may be more useful than the other.

9 At the MATLAB command line, type `dsp_examples_yout`.

The four-channel signal is displayed as a series of matrices in the MATLAB Command Window. Note that the last matrix contains only zeros. This is because every `Signal From Workspace` block in this model has its **Form output after final data value by** parameter set to `Setting to Zero`.

Create Multichannel Signals by Combining Multichannel Signals

You can combine existing multichannel signals into larger multichannel signals using the Simulink `Matrix Concatenate` block:

1 Open the Matrix Concatenate Example 2 model by typing

`ex_cmbmltichsbsigs`

at the MATLAB command line.
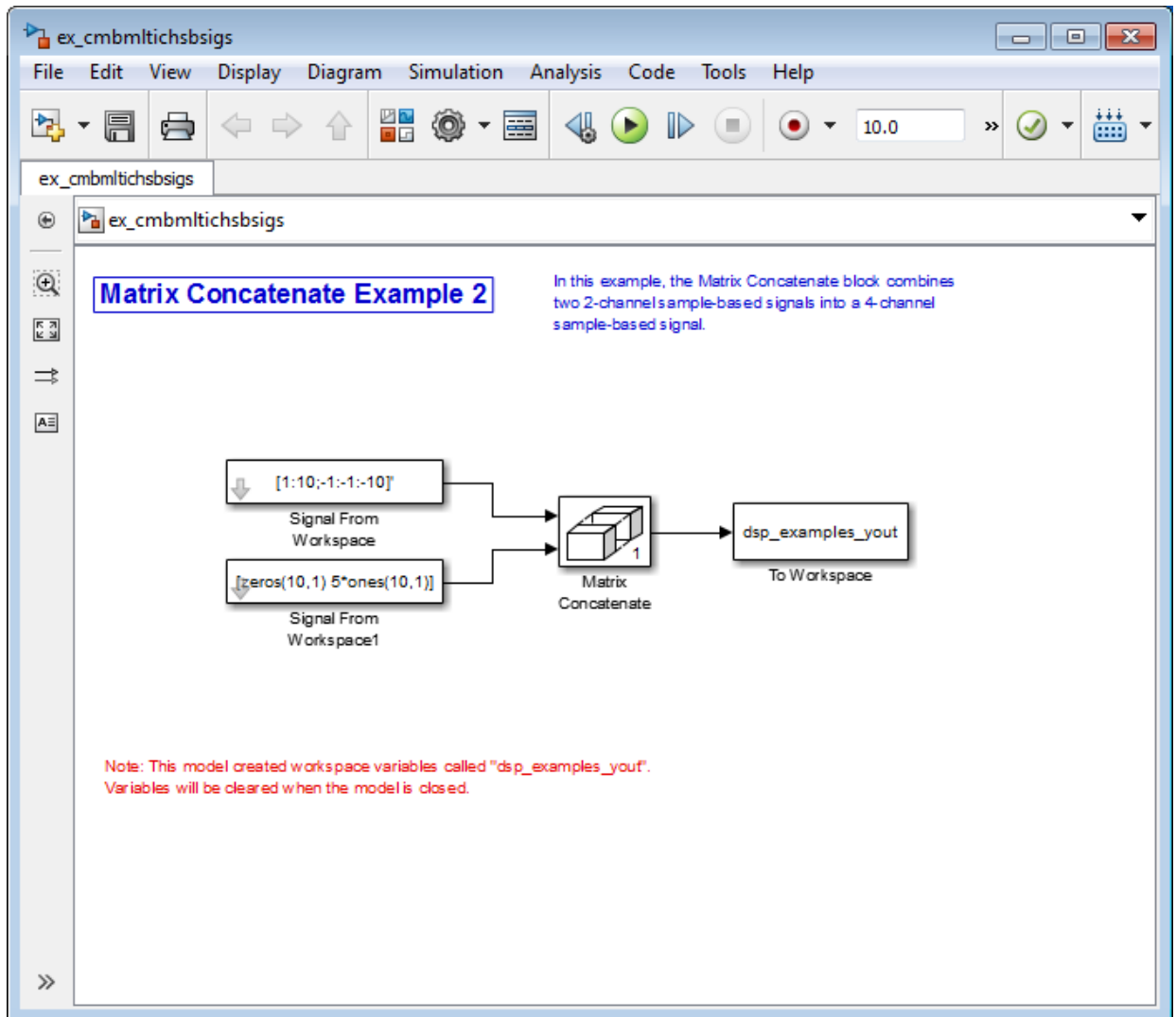
2   Double-click the Signal From Workspace block, and set the **Signal** parameter to
    [1:10;-1:-1:-10]'. Click **OK**.

3   Double-click the Signal From Workspace1 block, and set the **Signal** parameter to
    [zeros(10,1) 5*ones(10,1)]. Click **OK**.

4 Double-click the Matrix Concatenate block. Set the block parameters as follows, and then click **OK**:

- **Number of inputs** = 2
- **Mode** = Multidimensional array
- **Concatenate dimension** = 1

5 Run the model.

The model combines both two-channel signals into a four-channel signal.

Each 2-by-2 output from the Matrix Concatenate block contains both samples from each of the two input signals at the same instant in time. Each element of this matrix is a separate channel.

More About

- "Create Signals for Sample-Based Processing" on page 2-12
- "Create Signals for Frame-Based Processing" on page 2-20
- "Create Multichannel Signals for Frame-Based Processing" on page 2-35
- "Deconstruct Multichannel Signals for Sample-Based Processing" on page 2-40
- "Deconstruct Multichannel Signals for Frame-Based Processing" on page 2-49
- "Sample- and Frame-Based Concepts" on page 3-2

Create Multichannel Signals for Frame-Based Processing

In this section...
"Multichannel Signals for Frame-Based Processing" on page 2-36
"Create Multichannel Signals Using Concatenate Block" on page 2-37

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.

When you configure a block to perform frame-based processing, the block interprets an M-by-1 vector as a single-channel signal containing M samples per frame. Similarly, the block interprets an M-by-N matrix as a multichannel signal with N independent channels and M samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.

Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

Multichannel Signals for Frame-Based Processing

When you want to perform the same operations on several independent signals, you can group those signals together as a multichannel signal. For example, if you need to filter each of four independent signals using the same direct-form II transpose filter, you can combine the signals into a multichannel signal, and connect the signal to a single `Digital Filter Design` block. The block applies the filter to each channel independently.

A signal with N channels and frame size M is represented by a matrix of size *M*-by-*N*. Multiple individual signals with the same frame rate and frame size can be combined into a single multichannel signal using the Simulink `Matrix Concatenate` block. Individual signals can be added to an existing multichannel signal in the same way.
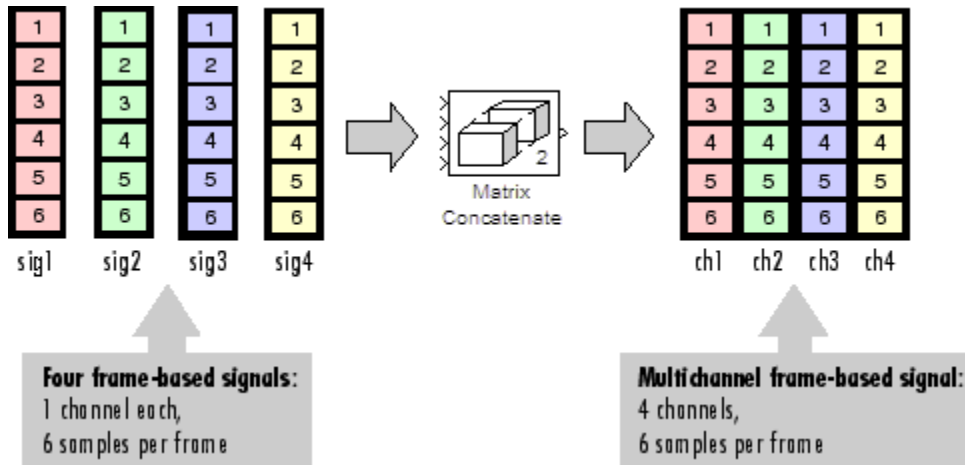
Create Multichannel Signals Using Concatenate Block

You can combine independent signals into a larger multichannel signal by using the Simulink `Concatenate` block. All signals must have the same frame rate and frame size. In this example, a single-channel signal is combined with a two-channel signal to produce a three-channel signal:

1   Open the Matrix Concatenate Example 3 model by typing

    ex_combiningfbsigs

    at the MATLAB command line.
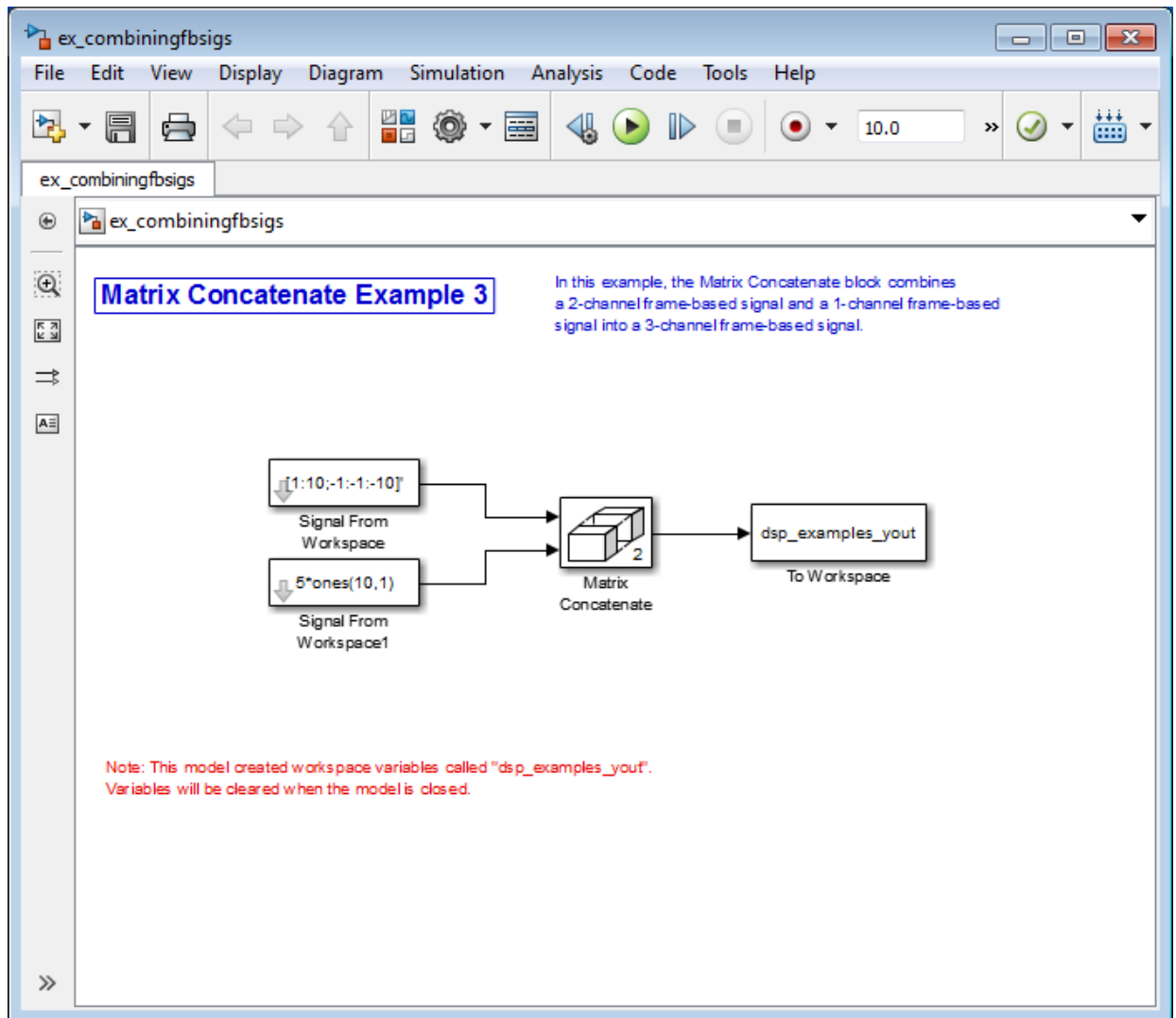
2   Double-click the Signal From Workspace block. Set the block parameters as follows:

- **Signal** = [1:10; -1: -1: -10] '
- **Sample time** = 1

- **Samples per frame** = 4

Based on these parameters, the Signal From Workspace block outputs a signal with a frame size of four.

3 Save these parameters and close the dialog box by clicking **OK**.

4 Double-click the Signal From Workspace1 block. Set the block parameters as follows, and then click **OK**:

- **Signal** = `5*ones(10,1)`
- **Sample time** = 1
- **Samples per frame** = 4

The Signal From Workspace1 block has the same sample time and frame size as the Signal From Workspace block. To combine single-channel signals into a multichannel signal, the signals must have the same frame rate and the same frame size.

5 Double-click the Matrix Concatenate block. Set the block parameters as follows, and then click **OK**:

- **Number of inputs** = 2
- **Mode** = `Multidimensional array`
- **Concatenate dimension** = 2

6 Run the model.

The 4-by-3 matrix output from the Matrix Concatenate block contains all three input channels, and preserves their common frame rate and frame size.

More About

Deconstruct Multichannel Signals for Sample-Based Processing

In this section...

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.

When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an M-by-N matrix as multichannel signal with $M*N$ independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

Split Multichannel Signals into Individual Signals

Multichannel signals, represented by matrices in the Simulink environment, are frequently used in signal processing models for efficiency and compactness. Though

most of the signal processing blocks can process multichannel signals, you may need to access just one channel or a particular range of samples in a multichannel signal. You can access individual channels of the multichannel signal by using the blocks in the Indexing library. This library includes the Selector, Submatrix, Variable Selector, Multiport Selector, and Submatrix blocks.

You can split a multichannel based signal into single-channel signals using the `Multiport Selector` block. This block allows you to select specific rows and/or columns and propagate the selection to a chosen output port. In this example, a three-channel signal of size 3-by-1 is deconstructed into three independent signals of sample period 1 second.

1 Open the Multiport Selector Example 1 model by typing `ex_splitmltichsbsigsind` at the MATLAB command line.
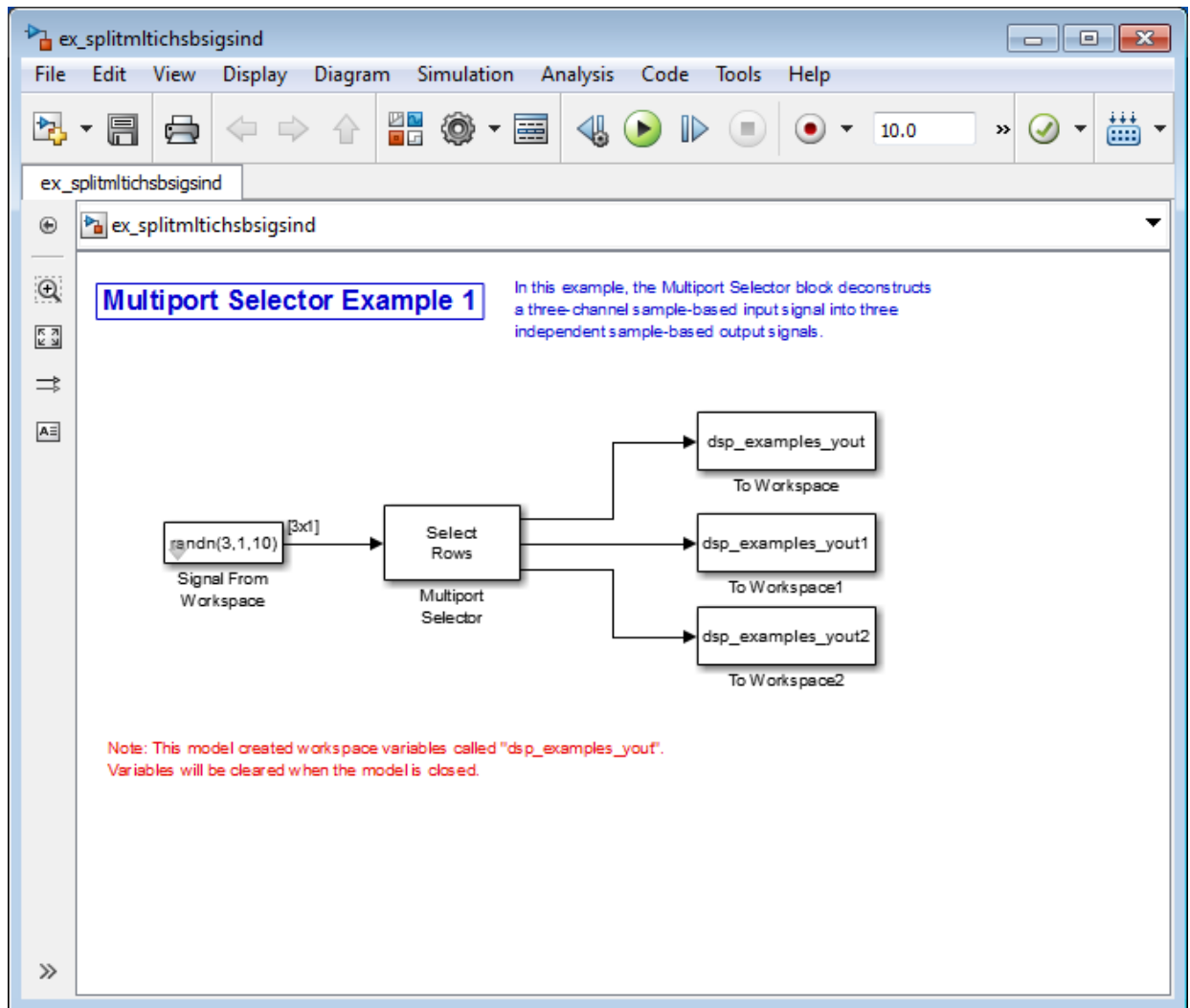
2 Double-click the Signal From Workspace block, and set the block parameters as follows:

- **Signal** = randn(3,1,10)
- **Sample time** = 1

- **Samples per frame** = 1

Based on these parameters, the Signal From Workspace block outputs a three-channel signal with a sample period of 1 second.

3 Save these parameters and close the dialog box by clicking **OK**.

4 Double-click the Multiport Selector block. Set the block parameters as follows, and then click **OK**:

- **Select** = Rows
- **Indices to output** = {1,2,3}

Based on these parameters, the Multiport Selector block extracts the rows of the input. The **Indices to output** parameter setting specifies that row 1 of the input should be reproduced at output 1, row 2 of the input should be reproduced at output 2, and row 3 of the input should be reproduced at output 3.

5 Run the model.

6 At the MATLAB command line, type `dsp_examples_yout`.

The following is a portion of what is displayed at the MATLAB command line. Because the input signal is random, your output might be different than the output show here.

```
dsp_examples_yout(:,:,1) =

    -0.1199

dsp_examples_yout(:,:,2) =

    -0.5955

dsp_examples_yout(:,:,3) =

    -0.0793
```

This signal is the first row of the input to the Multiport Selector block. You can view the other two input rows by typing `dsp_examples_yout1` and `dsp_examples_yout2`, respectively.

You have now successfully created three single-channel signals from a multichannel signal using a Multiport Selector block.

Split Multichannel Signals into Several Multichannel Signals

Multichannel signals, represented by matrices in the Simulink environment, are frequently used in signal processing models for efficiency and compactness. Though most of the signal processing blocks can process multichannel signals, you may need to access just one channel or a particular range of samples in a multichannel signal. You can access individual channels of the multichannel signal by using the blocks in the Indexing library. This library includes the Selector, Submatrix, Variable Selector, Multiport Selector, and Submatrix blocks.

You can split a multichannel signal into other multichannel signals using the Submatrix block. The Submatrix block is the most versatile of the blocks in the Indexing library because it allows arbitrary channel selections. Therefore, you can extract a portion of a multichannel signal. In this example, you extract a six-channel signal from a 35-channel signal (a matrix of size 5-by-7). Each channel contains one sample.

1   Open the Submatrix Example model by typing `ex_splitmltichsbsigsev` at the MATLAB command line.
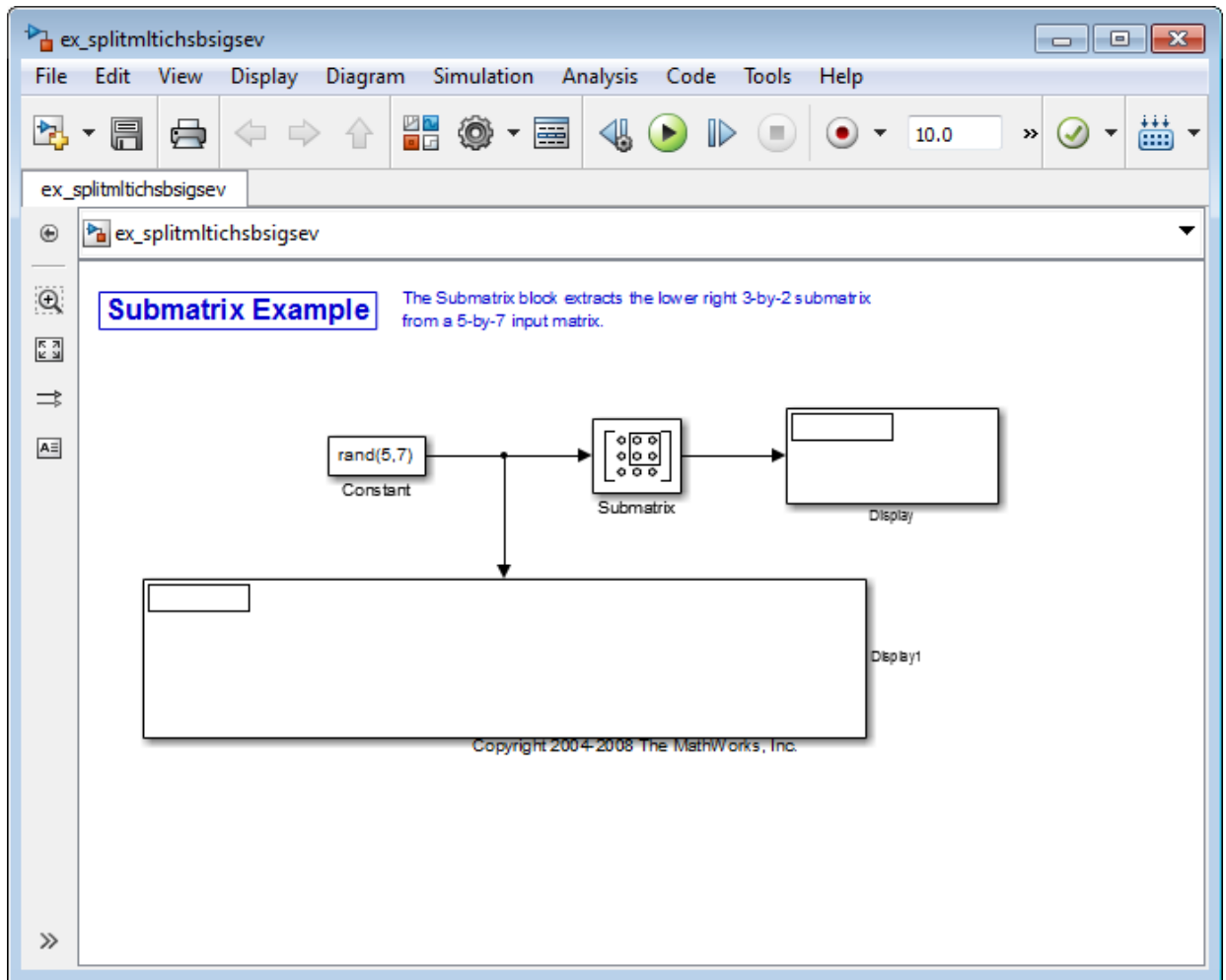
2   Double-click the Constant block, and set the block parameters as follows:

- **Constant value** = rand(5,7)
- **Interpret vector parameters as 1–D** = Clear this check box
- **Sample Time** = 1

Based on these parameters, the Constant block outputs a constant-valued signal.

3 Save these parameters and close the dialog box by clicking **OK**.

4 Double-click the Submatrix block. Set the block parameters as follows, and then click **OK**:

- **Row span** = Range of rows
- **Starting row** = Index
- **Starting row index** = 3
- **Ending row** = Last
- **Column span** = Range of columns
- **Starting column** = Offset from last
- **Starting column offset** = 1
- **Ending column** = Last

Based on these parameters, the Submatrix block outputs rows three to five, the last row of the input signal. It also outputs the second to last column and the last column of the input signal.

5 Run the model.

The model should now look similar to the following figure.
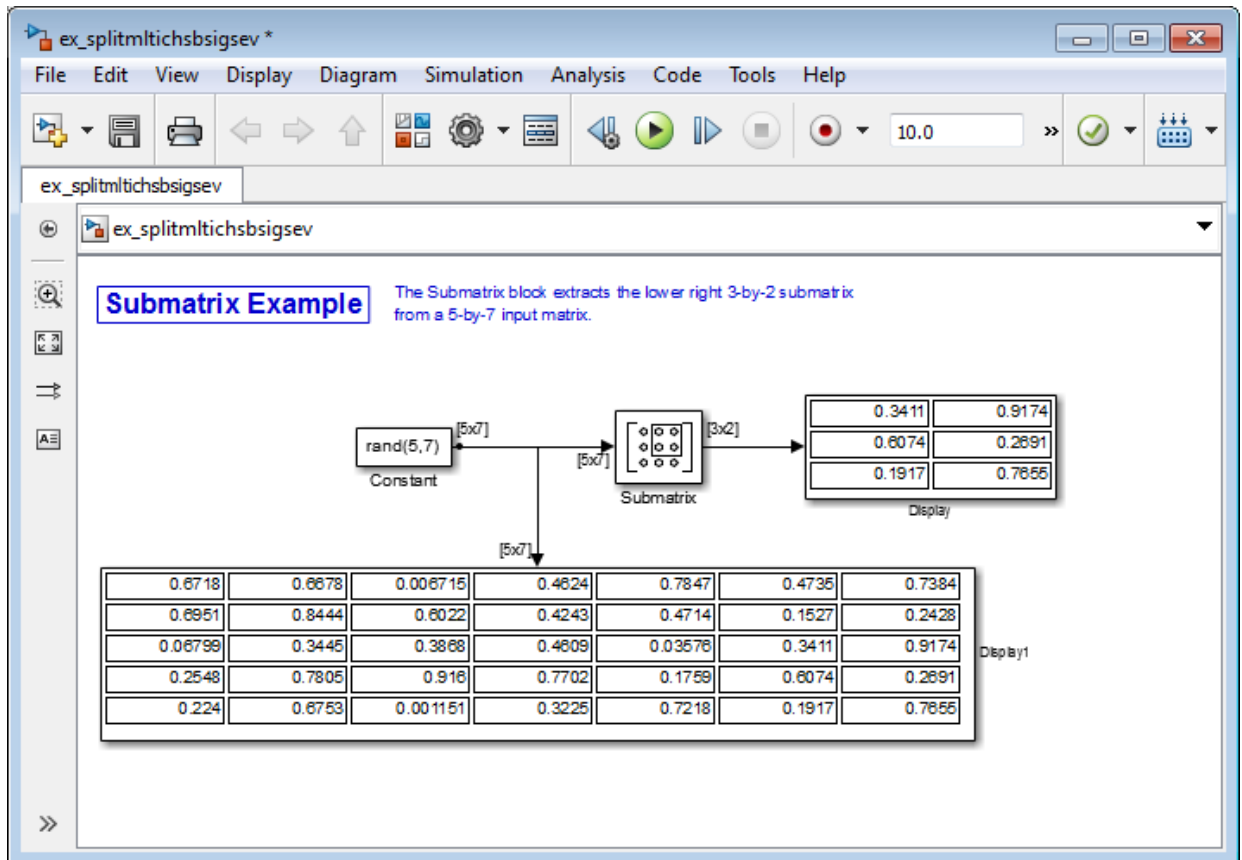
Notice that the output of the Submatrix block is equivalent to the matrix created by rows three through five and columns six through seven of the input matrix.

You have now successfully created a six-channel signal from a 35-channel signal using a Submatrix block. Each channel contains one sample.

More About

- "Create Signals for Sample-Based Processing" on page 2-12
- "Create Signals for Frame-Based Processing" on page 2-20
- "Create Multichannel Signals for Sample-Based Processing" on page 2-28

Deconstruct Multichannel Signals for Frame-Based Processing

In this section...
"Split Multichannel Signals into Individual Signals" on page 2-50
"Reorder Channels in Multichannel Signals" on page 2-54

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.

When you configure a block to perform frame-based processing, the block interprets an M-by-1 vector as a single-channel signal containing M samples per frame. Similarly, the block interprets an M-by-N matrix as a multichannel signal with N independent channels and M samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.

Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

Split Multichannel Signals into Individual Signals

Multichannel signals, represented by matrices in the Simulink environment, are frequently used in signal processing models for efficiency and compactness. Though most of the signal processing blocks can process multichannel signals, you may need to access just one channel or a particular range of samples in a multichannel signal. You can access individual channels of the multichannel signal by using the blocks in the Indexing library. This library includes the Selector, Submatrix, Variable Selector, Multiport Selector, and Submatrix blocks. It is also possible to use the Permute Matrix block, in the Matrix operations library, to reorder the channels of a frame-based signal.

You can use the `Multiport Selector` block in the Indexing library to extract the individual channels of a multichannel signal. These signals form single-channel signals that have the same frame rate and frame size of the multichannel signal.

The figure below is a graphical representation of this process.
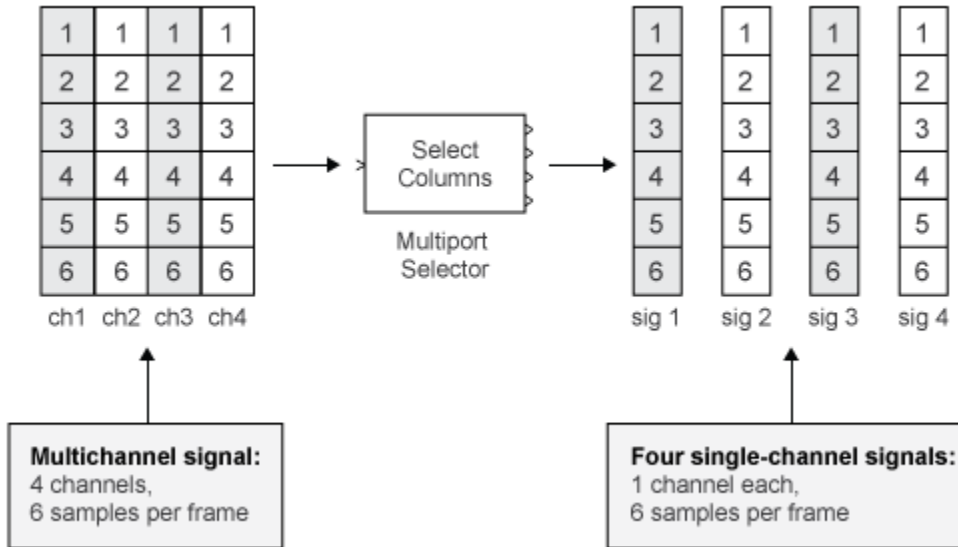
In this example, you use the Multiport Selector block to extract a single-channel signal and a two channel signal from a multichannel signal. Each channel contains four samples.

1   Open the Multiport Selector Example 2 model by typing
    ex_splitmltichfbsigsind

    at the MATLAB command line.
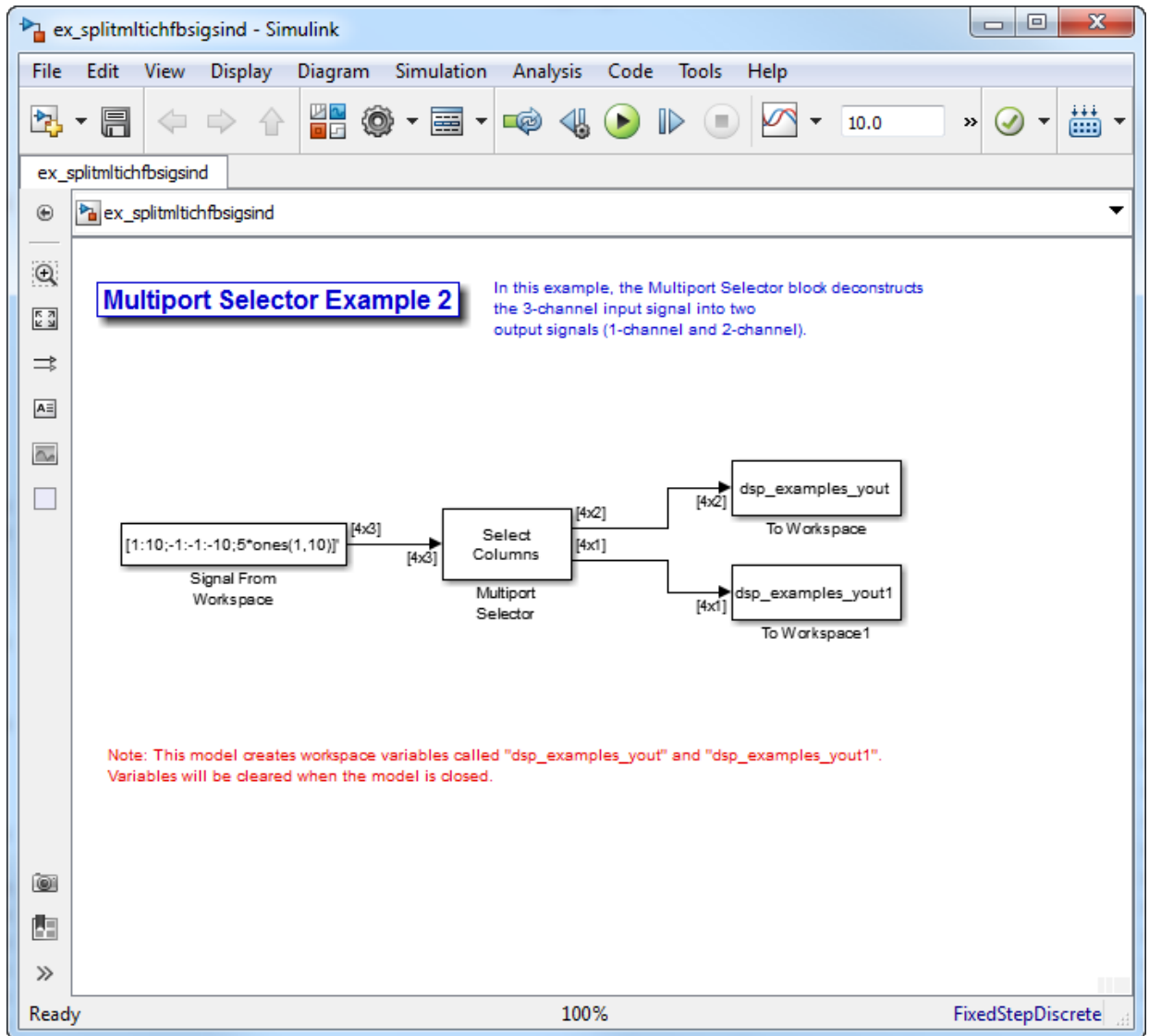
2 Double-click the Signal From Workspace block, and set the block parameters as follows:

- **Signal** = `[1:10;-1:-1:-10;5*ones(1,10)]'`
- **Samples per frame** = 4

Based on these parameters, the Signal From Workspace block outputs a three-channel signal with a frame size of four.

3 Save these parameters and close the dialog box by clicking **OK**.

4 Double-click the Multiport Selector block. Set the block parameters as follows, and then click **OK**:

- **Select** = `Columns`
- **Indices to output =** `{[1 3],2}`

Based on these parameters, the Multiport Selector block outputs the first and third columns at the first output port and the second column at the second output port of the block. Setting the **Select** parameter to `Columns` ensures that the block preserves the frame rate and frame size of the input.

5 Run the model.

The figure below is a graphical representation of how the Multiport Selector block splits one frame of the three-channel signal into a single-channel signal and a two-channel signal.

Multiport Selector Example 2

In this example, the Multiport Selector block deconstructs the 3-channel input signal into two output signals (1-channel and 2-channel)

Info

Doc Example Info

[1:10;-1:-1:-10;5^ones(1,10)]'

Signal From Workspace

[4x3]

1 -1 5
2 -2 5
3 -3 5
4 -4 5

3-channel signal

Select Columns

Multiport Selector

[4x2]

[4x1]

1 5
2 5
3 5
4 5

[4x2]

dsp_examples_yout

To Workspace

1 5
2 5
3 5
4 5

[4x1]

dsp_examples_yout1

To Workspace

2-channel signal (top) and
1-channel signal (bottom)
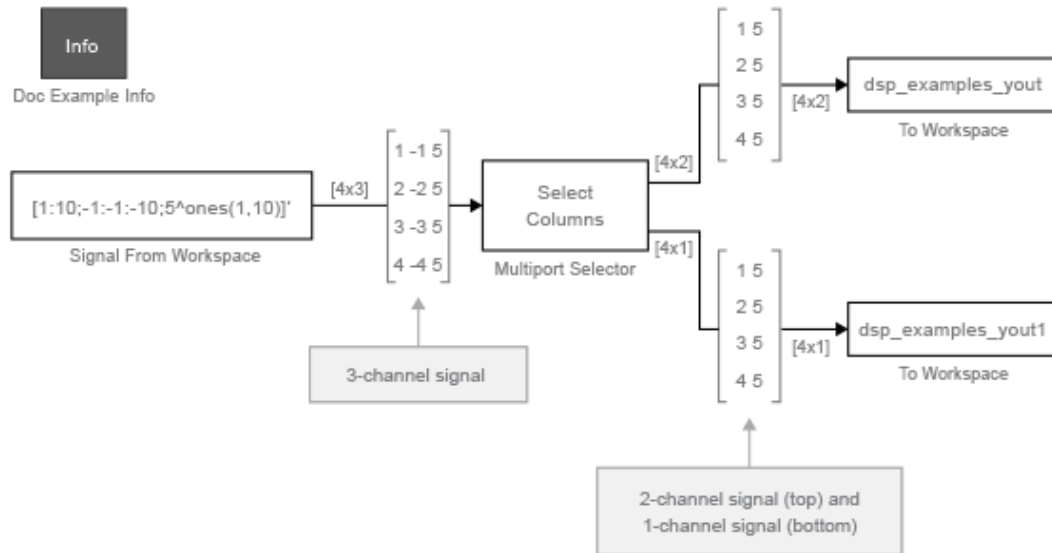
Note: This model creates workspace variables called "dsp_examples_yout" and "dsp_examples_yout1".

The Multiport Selector block outputs a two-channel signal, comprised of the first and third column of the input signal, at the first port. It outputs a single-channel comprised of the second column of the input signal, at the second port.

You have now successfully created a single-channel signal and a two-channel signal from a multichannel signal using the Multiport Selector block.

Reorder Channels in Multichannel Signals

Multichannel signals, represented by matrices in Simulink, are frequently used in signal processing models for efficiency and compactness. Though most of the signal processing blocks can process multichannel signals, you may need to access just one channel or a

particular range of samples in a multichannel signal. You can access individual channels of the multichannel signal by using the blocks in the Indexing library. This library includes the Selector, Submatrix, Variable Selector, Multiport Selector, and Submatrix blocks. It is also possible to use the Permute Matrix block, in the Matrix operations library, to reorder the channels of a frame signal.

Some DSP System Toolbox blocks have the ability to process the interaction of channels. Typically, DSP System Toolbox blocks compare channel one of signal A to channel one of signal B. However, you might want to correlate channel one of signal A with channel three of signal B. In this case, in order to compare the correct signals, you need to use the `Permute Matrix` block to rearrange the channels of your signals. This example explains how to accomplish this task.

1   Open the Permute Matrix Example model by typing `ex_reordermltichfbsigs` at the MATLAB command line.
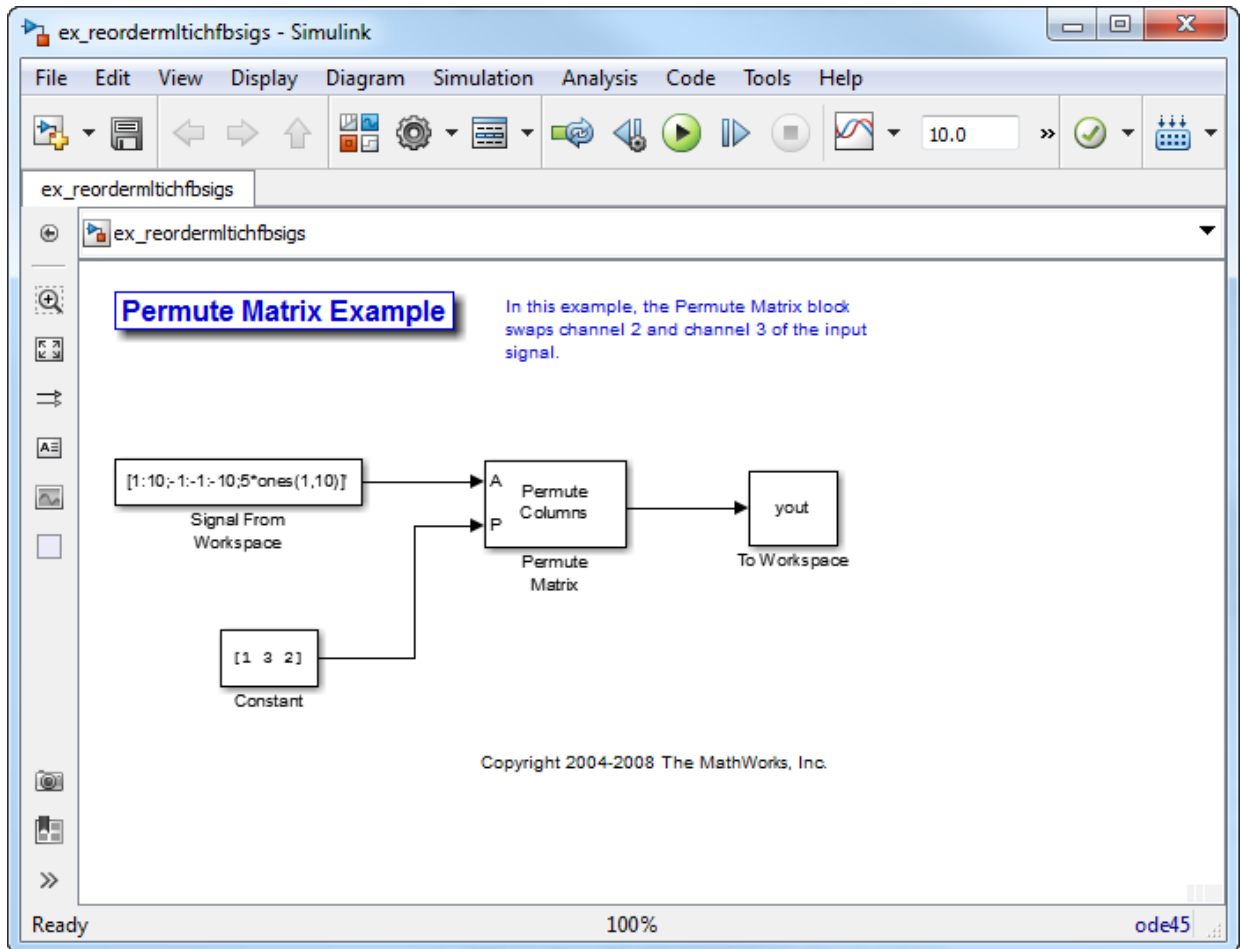
2   Double-click the Signal From Workspace block, and set the block parameters as
     follows:

   - **Signal** = [1:10;-1:-1:-10;5*ones(1,10)]'
   - **Sample time** = 1
   - **Samples per frame** = 4

Based on these parameters, the Signal From Workspace block outputs a three-channel signal with a sample period of 1 second and a frame size of 4. The frame period of this block is 4 seconds.

3 Save these parameters and close the dialog box by clicking **OK**.

4 Double-click the Constant block. Set the block parameters as follows, and then click **OK**:

  · **Constant value** = [1 3 2]

  · **Interpret vector parameters as 1–D** = Clear this check box

  · **Sample time** = 4

The discrete-time vector output by the Constant block tells the Permute Matrix block to swap the second and third columns of the input signal. Note that the frame period of the Constant block must match the frame period of the Signal From Workspace block.

5 Double-click the Permute Matrix block. Set the block parameters as follows, and then click **OK**:

  · **Permute** = Columns

  · **Index mode** = One-based

Based on these parameters, the Permute Matrix block rearranges the columns of the input signal, and the index of the first column is now one.

6 Run the model.

The figure below is a graphical representation of what happens to the first input frame during simulation.

Permute Matrix Example

In this example, the Permute Matrix block swaps channel 2 and channel 3 of the input signal.
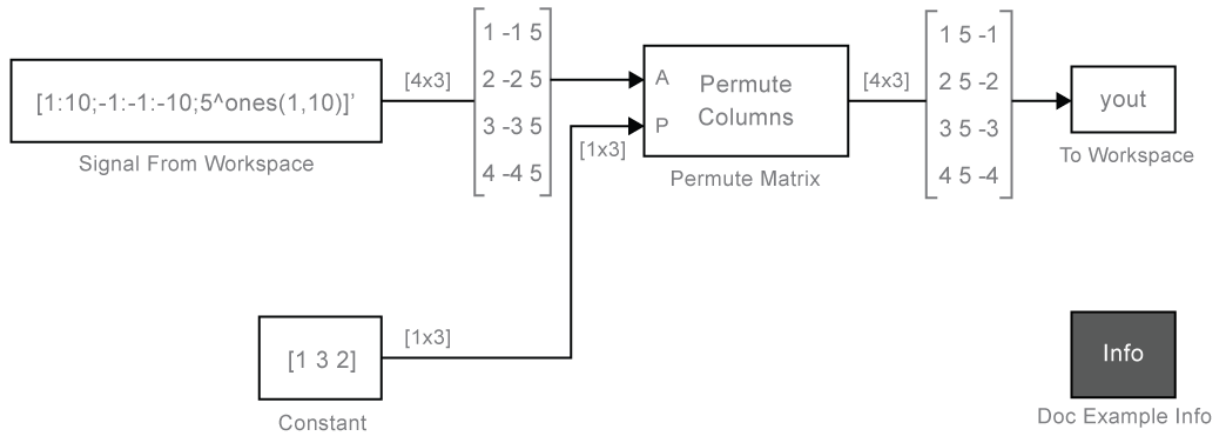
The second and third channel of the input signal are swapped.

7   At the MATLAB command line, type `yout`.

You can now verify that the second and third columns of the input signal are rearranged.

You have now successfully reordered the channels of a frame signal using the Permute Matrix block.

More About

- "Create Signals for Sample-Based Processing" on page 2-12
- "Create Signals for Frame-Based Processing" on page 2-20
- "Create Multichannel Signals for Sample-Based Processing" on page 2-28
- "Create Multichannel Signals for Frame-Based Processing" on page 2-35
- "Deconstruct Multichannel Signals for Sample-Based Processing" on page 2-40
- "Import and Export Signals for Sample-Based Processing" on page 2-60
- "Import and Export Signals for Frame-Based Processing" on page 2-72

Import and Export Signals for Sample-Based Processing

In this section...
"Import Vector Signals for Sample-Based Processing" on page 2-61
"Import Matrix Signals for Sample-Based Processing" on page 2-63
"Export Signals for Sample-Based Processing" on page 2-67

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.

When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an M-by-N matrix as multichannel signal with $M*N$ independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

Import Vector Signals for Sample-Based Processing

The Signal From Workspace block generates a vector signal for sample-based processing when the variable or expression in the **Signal** parameter is a matrix and the **Samples per frame** parameter is set to 1. Each column of the input matrix represents a different channel. Beginning with the first row of the matrix, the block outputs one row of the matrix at each sample time. Therefore, if the **Signal** parameter specifies an M-by-N matrix, the output of the Signal From Workspace block is M 1-by-N row vectors representing N channels.

The figure below is a graphical representation of this process for a 6-by-4 workspace matrix, A.
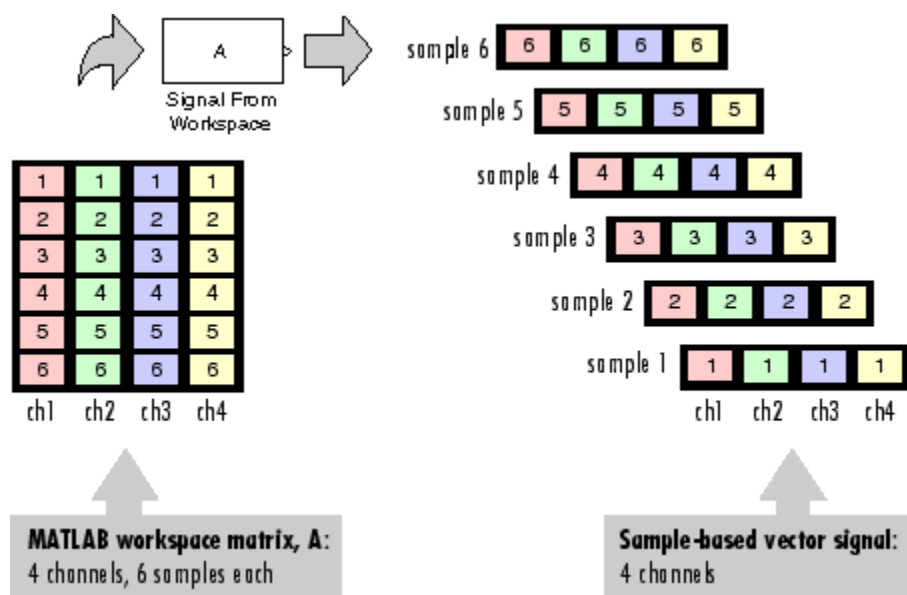
In the following example, you use the Signal From Workspace block to import the vector signal into your model.

1   Open the Signal From Workspace Example 3 model by typing
    `ex_importsbvectorsigs` at the MATLAB command line.
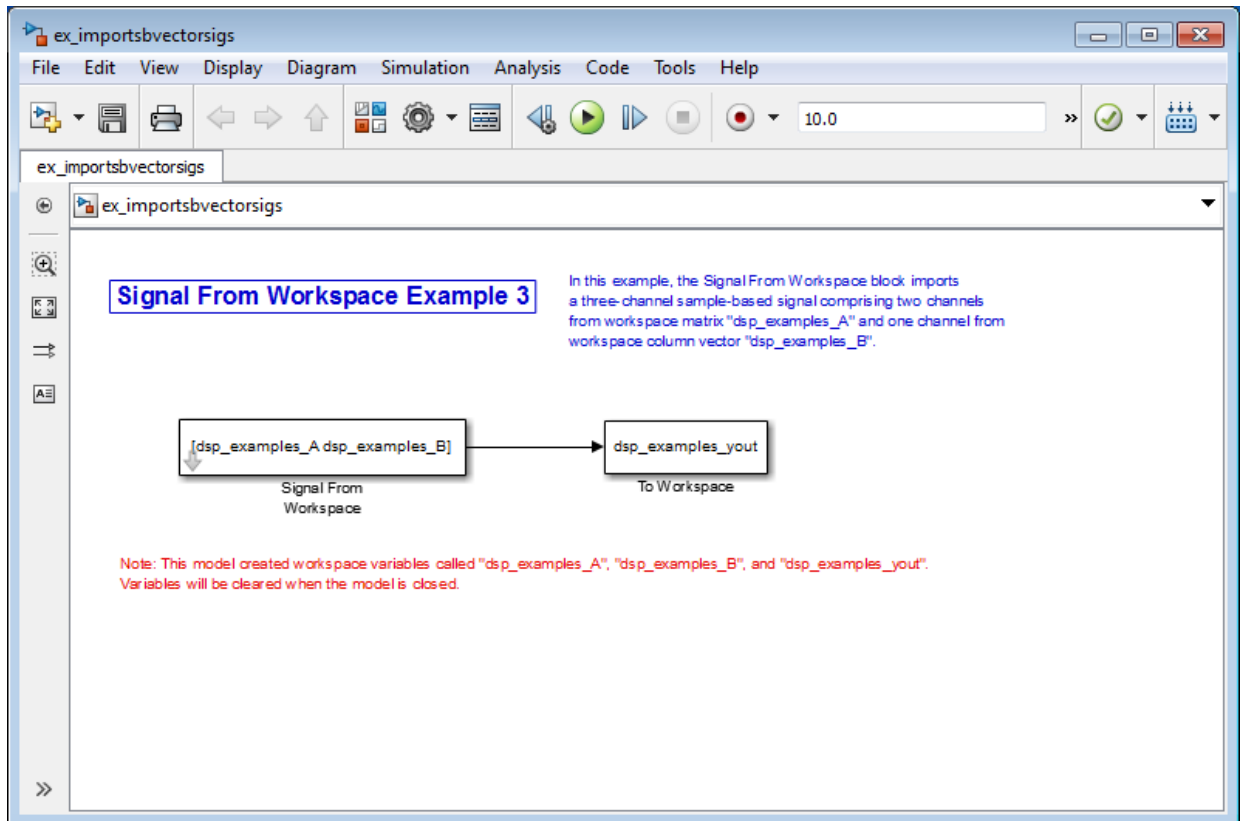
2   At the MATLAB command line, type A = [1:100;-1:-1:-100]';

The matrix A represents a two column signal, where each column is a different channel.

3   At the MATLAB command line, type B = 5*ones(100,1);

The vector B represents a single-channel signal.

4   Double-click the Signal From Workspace block, and set the block parameters as follows:

- **Signal** = [A B]
- **Sample time** = 1

- **Samples per frame** = 1
- **Form output after final data value** = `Setting to zero`

The **Signal** expression `[A B]` uses the standard MATLAB syntax for horizontally concatenating matrices and appends column vector `B` to the right of matrix `A`. The Signal From Workspace block outputs a signal with a sample period of 1 second. After the block has output the signal, all subsequent outputs have a value of zero.

5 Save these parameters and close the dialog box by clicking **OK**.

6 Run the model.

The following figure is a graphical representation of the model's behavior during simulation.
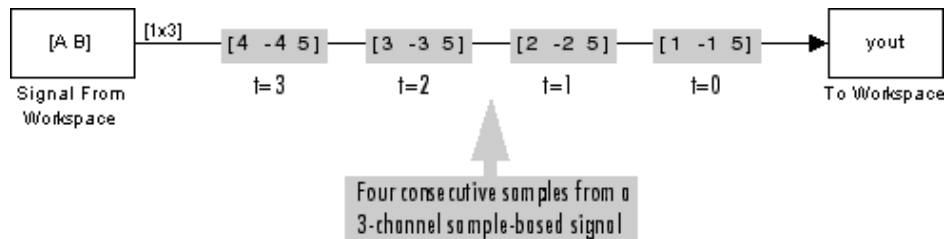
The first row of the input matrix `[A B]` is output at time `t=0`, the second row of the input matrix is output at time `t=1`, and so on.

You have now successfully imported a vector signal with three channels into your signal processing model using the Signal From Workspace block.

Import Matrix Signals for Sample-Based Processing

The Signal From Workspace block generates a matrix signal that is convenient for sample-based processing. Beginning with the first page of the array, the block outputs a single page of the array to the output at each sample time. Therefore, if the **Signal** parameter specifies an M-by-N-by-P array, the output of the Signal From Workspace block is P M-by-N matrices representing M*N channels. The block receiving this signal does sample-based processing or frame-based processing on the signal based on the parameters set in the block dialog box.

The following figure is a graphical illustration of this process for a 6-by-4-by-5 workspace array A.
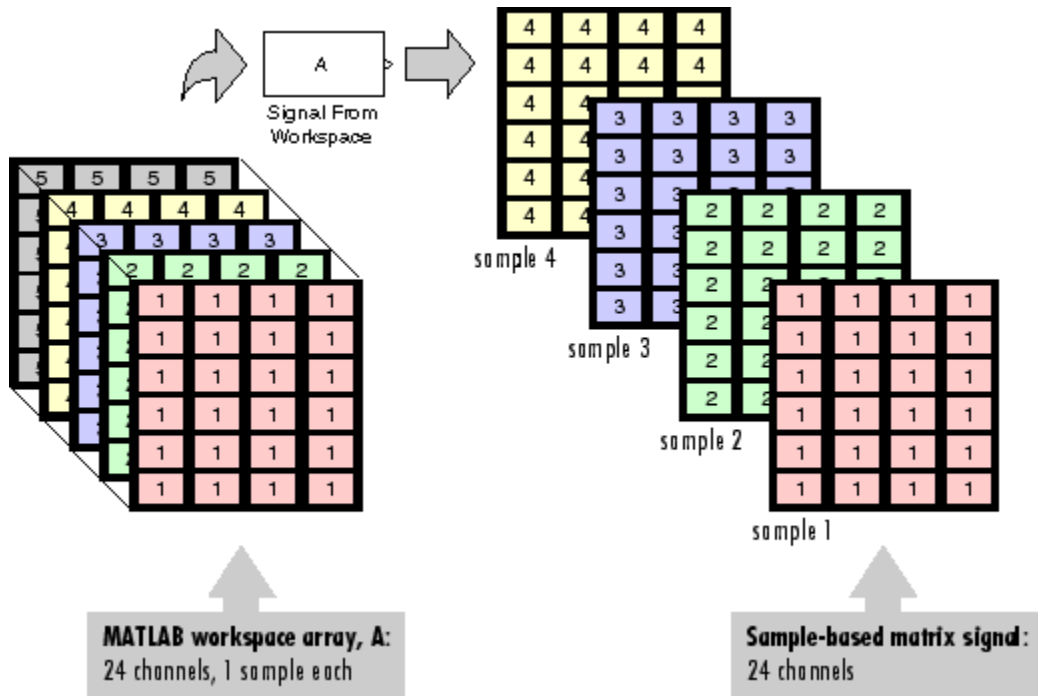
In the following example, you use the `Signal From Workspace` block to import a four-channel matrix signal into a Simulink model.

1   Open the Signal From Workspace Example 4 model by typing
    `ex_importsbmatrixsigs` at the MATLAB command line.
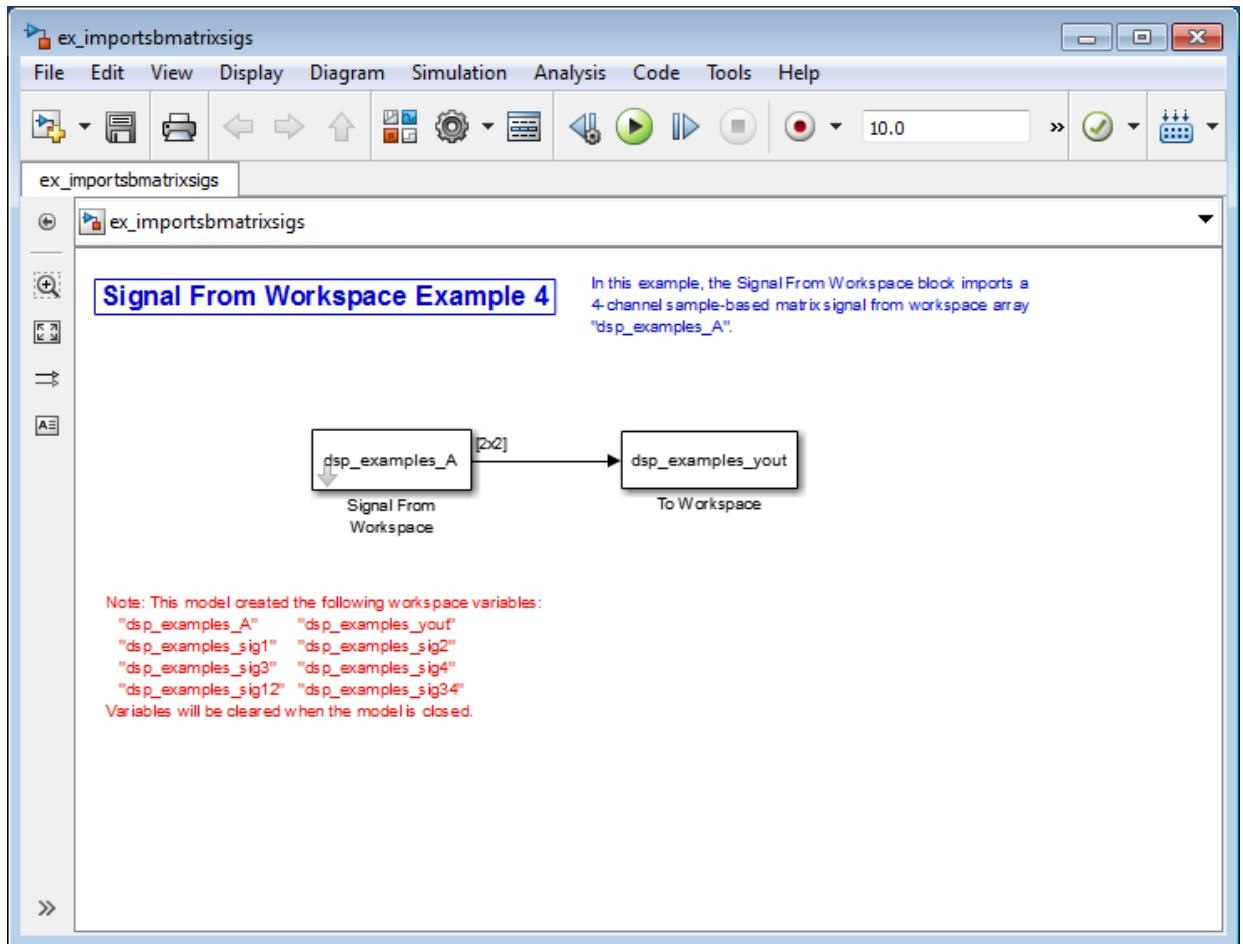
Also, the following variables are loaded into the MATLAB workspace:

```
Fs                      1x1          8           double array
dsp_examples_A          2x2x100      3200        double array
dsp_examples_sig1       1x1x100      800         double array
dsp_examples_sig12      1x2x100      1600        double array
dsp_examples_sig2       1x1x100      800         double array
```

```
dsp_examples_sig3        1x1x100        800        double array
dsp_examples_sig34       1x2x100        1600       double array
dsp_examples_sig4        1x1x100        800        double array
mtlb                     4001x1         32008      double array
```

2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **Signal** = dsp_examples_A
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = Setting to zero

The dsp_examples_A array represents a four-channel signal with 100 samples in each channel. This is the signal that you want to import, and it was created in the following way:

```
dsp_examples_sig1 = reshape(1:100,[1 1 100])
dsp_examples_sig2 = reshape(-1:-1:-100,[1 1 100])
dsp_examples_sig3 = zeros(1,1,100)
dsp_examples_sig4 = 5*ones(1,1,100)
dsp_examples_sig12 = cat(2,sig1,sig2)
dsp_examples_sig34 = cat(2,sig3,sig4)
dsp_examples_A = cat(1,sig12,sig34) % 2-by-2-by-100 array
```

3 Run the model.

The figure below is a graphical representation of the model's behavior during simulation.

Signal From Workspace Example 4

In this example, the Signal From Workspace block imports a 4-channel sample-based matrix signal from workspace array "dsp_examples_A".

Note: This model created the following workspace variables:
"dsp_examples_A"   "dsp_examples_yout"
"dsp_examples_sig1"   "dsp_examples_sig2"
"dsp_examples_sig3"   "dsp_examples_sig4"
"dsp_examples_sig12" "dsp_examples_sig34"

Four consecutive samples from a 4-channel sample-based signal

first matrix output
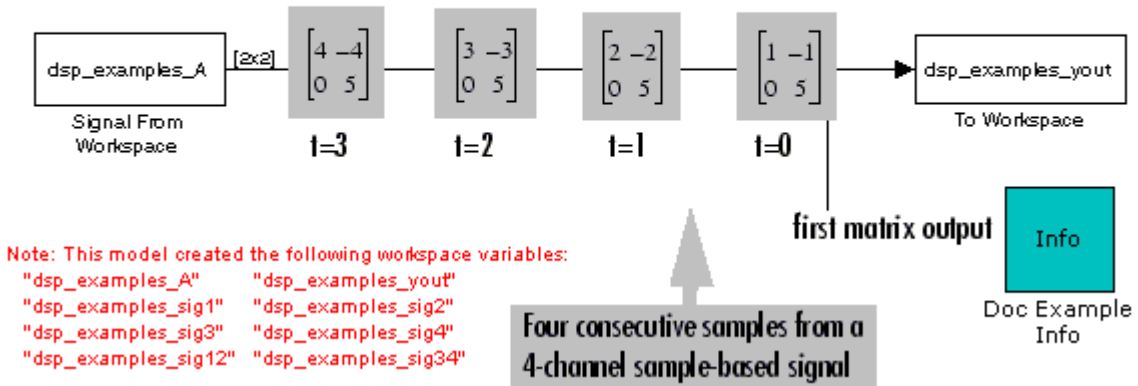
The Signal From Workspace block imports the four-channel signal from the MATLAB workspace into the Simulink model one matrix at a time.

You have now successfully imported a 4-channel matrix signal into your model using the Signal From Workspace block.

Export Signals for Sample-Based Processing

The To Workspace and Triggered To Workspace blocks are the primary blocks for exporting signals of all dimensions from a Simulink model to the MATLAB workspace.

A signal with M*N channels, is represented in Simulink as a sequence of M-by-N matrices. When the input to the To Workspace block is a signal created for sample-based processing, the block creates an M-by-N-by-P array in the MATLAB workspace containing the P most recent samples from each channel. The number of pages, P, is specified by the **Limit data points to last** parameter. The newest samples are added at the end of the array.

The following figure is the graphical illustration of this process using a 6-by-4 signal exported to workspace array A.

Sample-based matrix signal:
6-by-4 (24 channels)

MATLAB workspace array, A:
6-by-4-by-P (24 channels, P = 5)
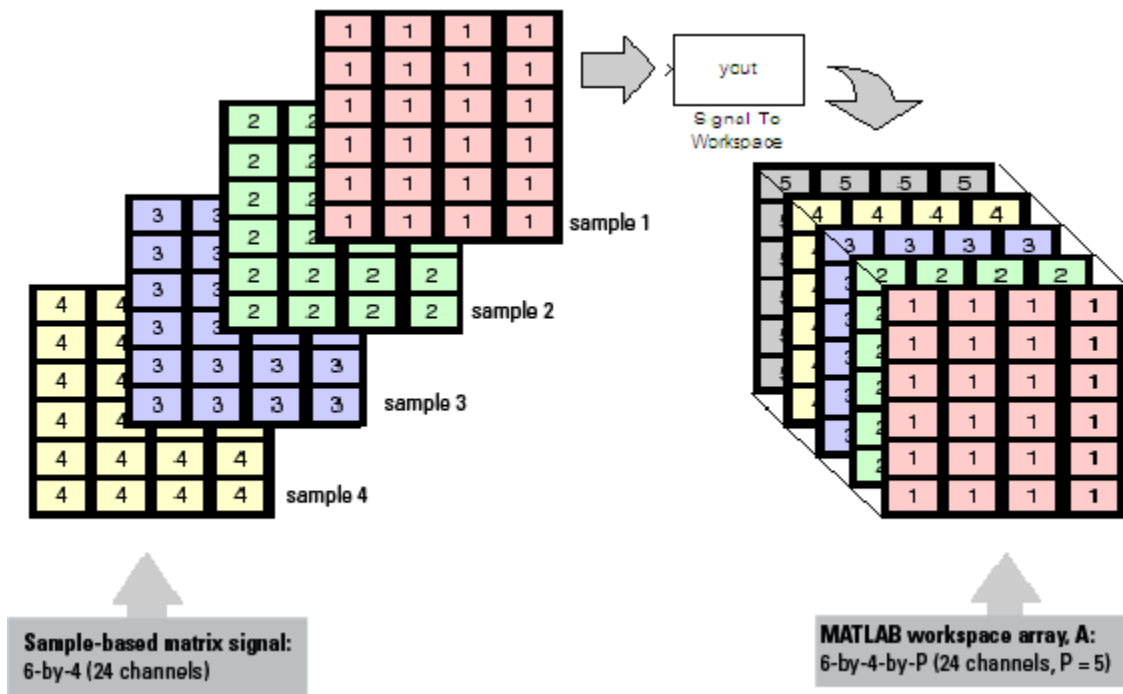
The workspace array always has time running along its third dimension, P. Samples are saved along the P dimension whether the input is a matrix, vector, or scalar (single channel case).

In the following example you use a To Workspace block to export a matrix signal to the MATLAB workspace.

1   Open the Signal From Workspace Example 6 model by typing `ex_exportsbsigs` at the MATLAB command line.
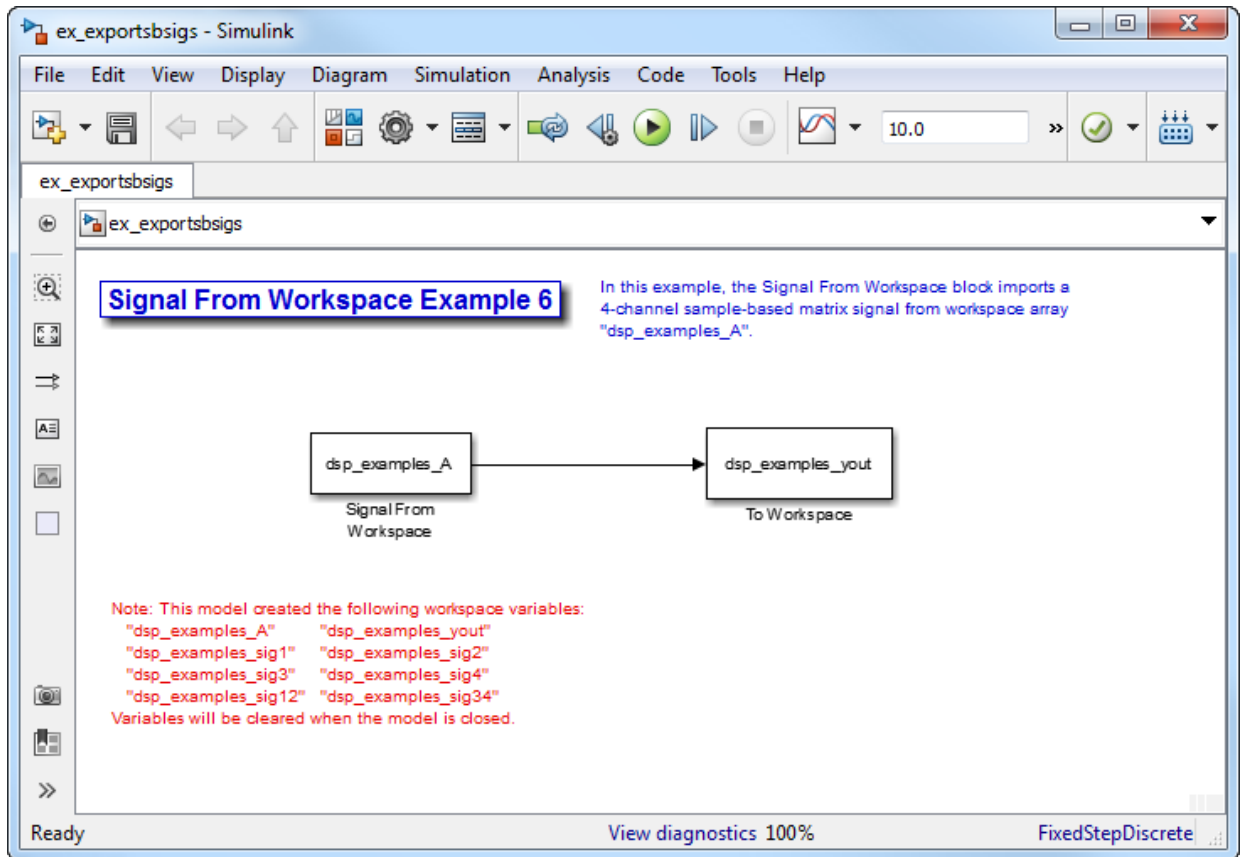
Also, the following variables are loaded into the MATLAB workspace:

```
dsp_examples_A          2x2x100       3200        double array
dsp_examples_sig1       1x1x100       800         double array
dsp_examples_sig12      1x2x100       1600        double array
dsp_examples_sig2       1x1x100       800         double array
dsp_examples_sig3       1x1x100       800         double array
dsp_examples_sig34      1x2x100       1600        double array
dsp_examples_sig4       1x1x100       800         double array
```

In this model, the Signal From Workspace block imports a four-channel matrix signal called `dsp_examples_A`. This signal is then exported to the MATLAB workspace using a To Workspace block.

2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **Signal** = `dsp_examples_A`
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = `Setting to zero`

Based on these parameters, the Signal From Workspace block outputs a signal with a sample period of 1 second. After the block has output the signal, all subsequent outputs have a value of zero.

3 Double-click the To Workspace block. Set the block parameters as follows, and then click **OK**:

- **Variable name** = `dsp_examples_yout`
- **Limit data points to last** parameter to `inf`
- **Decimation** = 1

Based on these parameters, the To Workspace block exports its input signal to a variable called `dsp_examples_yout` in the MATLAB workspace. The workspace variable can grow indefinitely large in order to capture all of the input data. The signal is not decimated before it is exported to the MATLAB workspace.

4 Run the model.

5 At the MATLAB command line, type `dsp_examples_yout`.

The four-channel matrix signal, `dsp_examples_A`, is output at the MATLAB command line. The following is a portion of the output that is displayed.

```
dsp_examples_yout(:,:,1) =

     1    -1
     0     5

dsp_examples_yout(:,:,2) =
```

```
        2    -2
        0     5

dsp_examples_yout(:,:,3) =

        3    -3
        0     5

dsp_examples_yout(:,:,4) =

        4    -4
        0     5
```

Each page of the output represents a different sample time, and each element of the matrices is in a separate channel.

You have now successfully exported a four-channel matrix signal from a Simulink model to the MATLAB workspace using the To Workspace block.

More About

- "Create Signals for Sample-Based Processing" on page 2-12
- "Create Signals for Frame-Based Processing" on page 2-20
- "Create Multichannel Signals for Sample-Based Processing" on page 2-28
- "Create Multichannel Signals for Frame-Based Processing" on page 2-35
- "Deconstruct Multichannel Signals for Sample-Based Processing" on page 2-40
- "Deconstruct Multichannel Signals for Frame-Based Processing" on page 2-49
- "Import and Export Signals for Frame-Based Processing" on page 2-72
- "Inspect Sample and Frame Rates in Simulink" on page 3-8
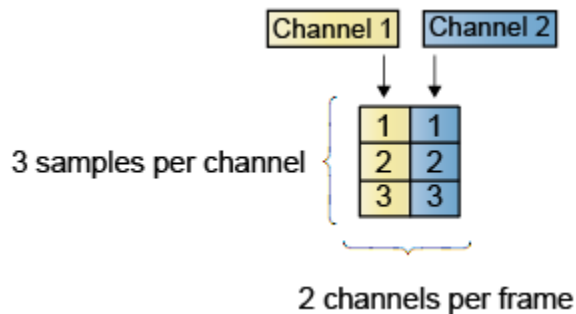- "Convert Sample and Frame Rates in Simulink" on page 3-19

Import and Export Signals for Frame-Based Processing

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.

When you configure a block to perform frame-based processing, the block interprets an M-by-1 vector as a single-channel signal containing M samples per frame. Similarly, the block interprets an M-by-N matrix as a multichannel signal with N independent channels and M samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.

Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

Import Signals for Frame-Based Processing

The Signal From Workspace block creates a multichannel signal for frame-based processing when the **Signal** parameter is a matrix, and the **Samples per frame** parameter, M, is greater than 1. Beginning with the first M rows of the matrix, the block releases M rows of the matrix (that is, one frame from each channel) to the output port every $M*T_s$ seconds. Therefore, if the **Signal** parameter specifies a W-by-N workspace matrix, the Signal From Workspace block outputs a series of M-by-N matrices representing N channels. The workspace matrix must be oriented so that its columns represent the channels of the signal.

The figure below is a graphical illustration of this process for a 6-by-4 workspace matrix, A, and a frame size of 2.

MATLAB workspace matrix, A:
4 channels, 6 samples each

Frame-based signal:
4 channels, 2 samples per frame
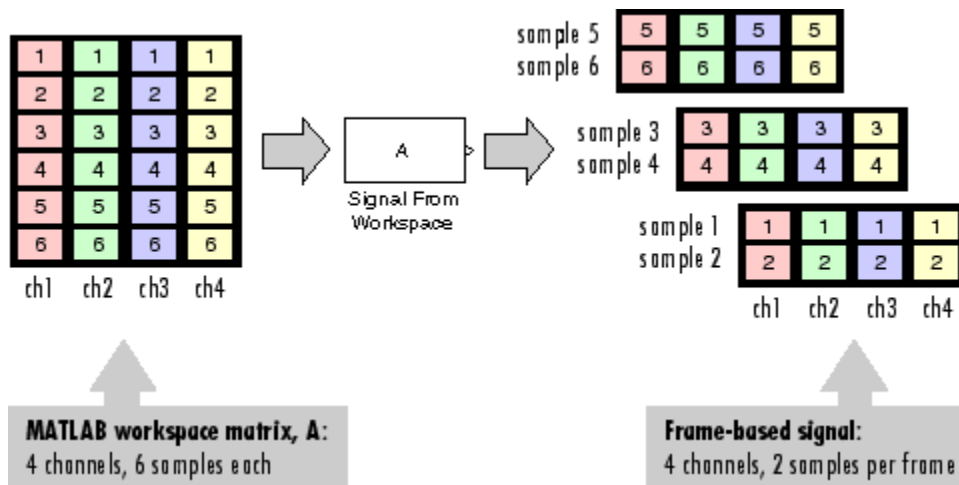
Note: Although independent channels are generally represented as columns, a single-channel signal can be represented in the workspace as either a column vector or row vector. The output from the Signal From Workspace block is a column vector in both cases.

In the following example, you use the Signal From Workspace block to create a three-channel frame signal and import it into the model:

1 Open the Signal From Workspace Example 5 model by typing

```
ex_importfbsigs
```

at the MATLAB command line.

```
dsp_examples_A = [1:100;-1:-1:-100]'; % 100-by-2 matrix
dsp_examples_B = 5*ones(100,1);    % 100-by-1 column vector
```

The variable called dsp_examples_A represents a two-channel signal with 100 samples, and the variable called dsp_examples_B represents a one-channel signal with 100 samples.

Also, the following variables are defined in the MATLAB workspace:
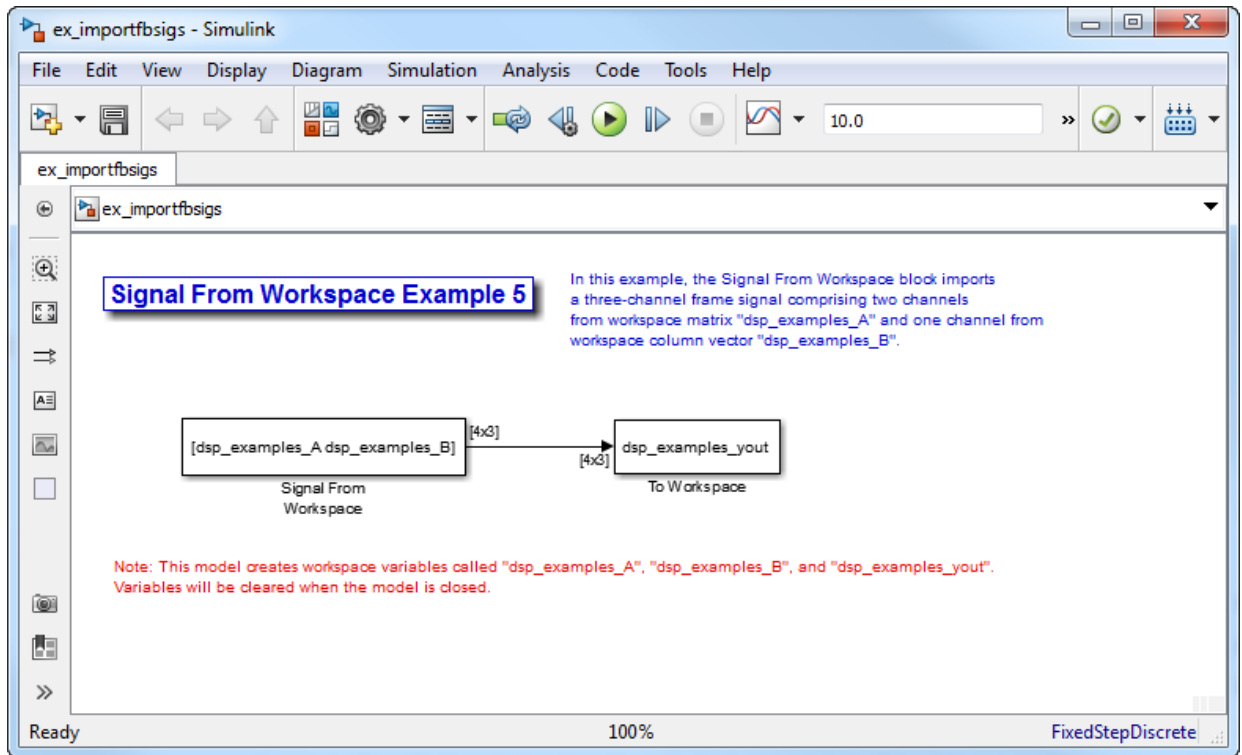
2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **Signal** parameter to [dsp_examples_A dsp_examples_B]
- **Sample time** parameter to 1
- **Samples per frame** parameter to 4
- **Form output after final data value** parameter to Setting to zero

Based on these parameters, the Signal From Workspace block outputs a signal with a frame size of 4 and a sample period of 1 second. The signal's frame period is 4 seconds. The **Signal** parameter uses the standard MATLAB syntax for horizontally concatenating matrices to append column vector dsp_examples_B to the right of matrix dsp_examples_A. After the block has output the signal, all subsequent outputs have a value of zero.

3 Run the model.

The figure below is a graphical representation of how your three-channel frame signal is imported into your model.
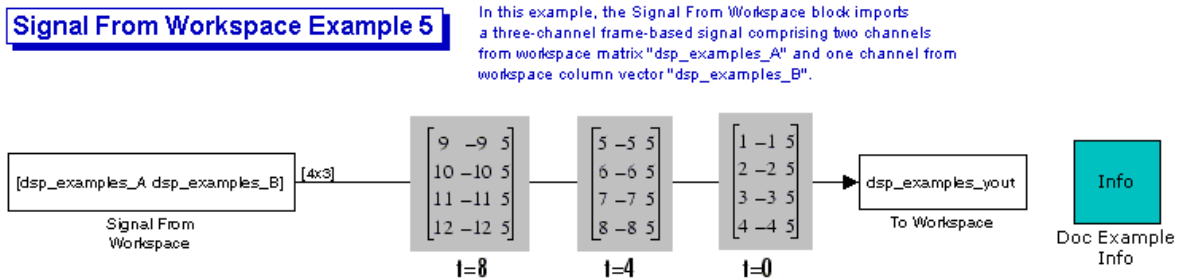
You have now successfully imported a three-channel frame signal into your model using the Signal From Workspace block.

Export Frame-Based Signals

The `To Workspace` and `Triggered To Workspace` blocks are the primary blocks for exporting signals of all dimensions from a Simulink model to the MATLAB workspace.

A signal with N channels and frame size M is represented by a sequence of M-by-N matrices. When this signal is input to the To Workspace block, the block creates a P-by-N array in the MATLAB workspace containing the P most recent samples from each channel. The number of rows, P, is specified by the **Limit data points to last** parameter. The newest samples are added at the bottom of the matrix.

The following figure is a graphical illustration of this process for three consecutive frames of a signal with a frame size of 2 that is exported to matrix A in the MATLAB workspace.
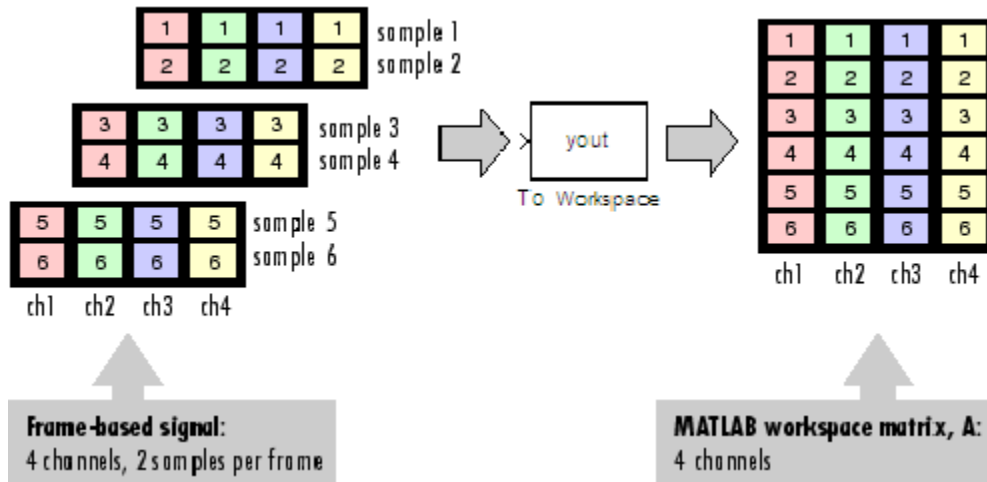
In the following example, you use a To Workspace block to export a three-channel signal with four samples per frame to the MATLAB workspace.

1 Open the Signal From Workspace Example 7 model by typing `ex_exportfbsigs` at the MATLAB command line.
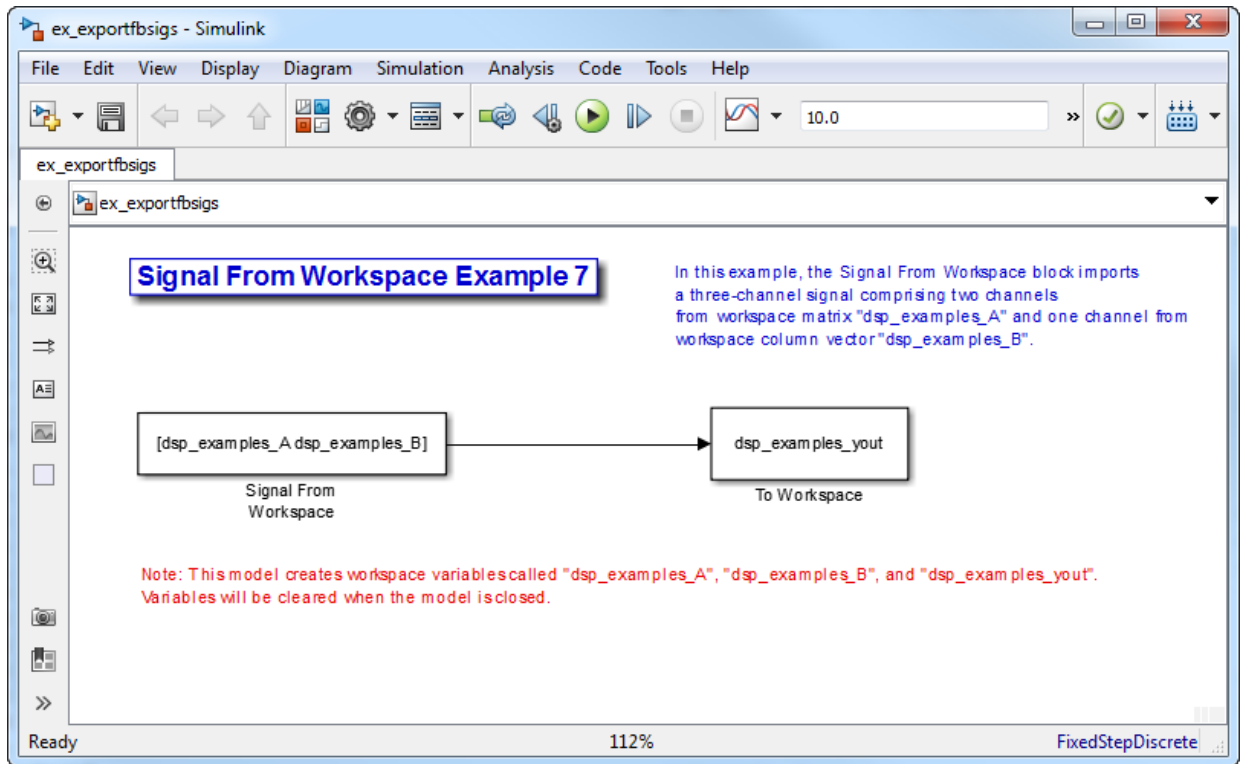
Also, the following variables are defined in the MATLAB workspace:

The variable called dsp_examples_A represents a two-channel signal with 100 samples, and the variable called dsp_examples_B represents a one-channel signal with 100 samples.

```
dsp_examples_A = [1:100;-1:-1:-100]'; % 100-by-2 matrix
dsp_examples_B = 5*ones(100,1);    % 100-by-1 column vector
```

2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **Signal** = [dsp_examples_A dsp_examples_B]
- **Sample time** = 1
- **Samples per frame** = 4

- **Form output after final data value** = `Setting to zero`

Based on these parameters, the Signal From Workspace block outputs a signal with a frame size of 4 and a sample period of 1 second. The signal's frame period is 4 seconds. The **Signal** parameter uses the standard MATLAB syntax for horizontally concatenating matrices to append column vector `dsp_examples_B` to the right of matrix `dsp_examples_A`. After the block has output the signal, all subsequent outputs have a value of zero.

3 Double-click the To Workspace block. Set the block parameters as follows, and then click **OK**:

- **Variable name** = `dsp_examples_yout`
- **Limit data points to last** = `inf`
- **Decimation** = `1`
- **Frames** = `Concatenate frames (2-D array)`

Based on these parameters, the To Workspace block exports its input signal to a variable called `dsp_examples_yout` in the MATLAB workspace. The workspace variable can grow indefinitely large in order to capture all of the input data. The signal is not decimated before it is exported to the MATLAB workspace, and each input frame is vertically concatenated to the previous frame to produce a 2-D array output.

4 Run the model.

The following figure is a graphical representation of the model's behavior during simulation.

Signal From Workspace Example 7

In this example, the Signal From Workspace block imports a three-channel frame-based signal comprising two channels from workspace matrix "dsp_examples_A" and one channel from workspace column vector "dsp_examples_B".
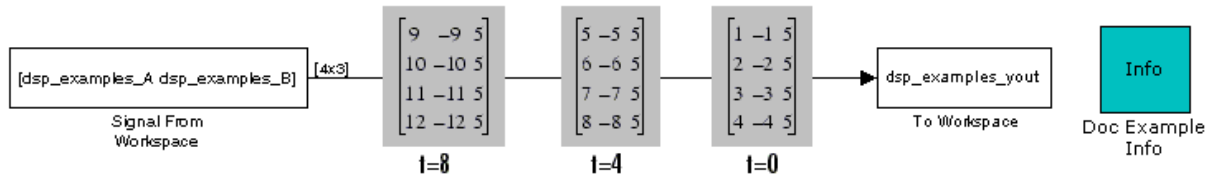
Note: This model creates workspace variables called "dsp_examples_A", "dsp_examples_B", and "dsp_examples_yout".

5   At the MATLAB command line, type dsp_examples_yout.

The output is shown below:

dsp_examples_yout =

```
     1    - 1      5
     2    - 2      5
     3    - 3      5
     4    - 4      5
     5    - 5      5
     6    - 6      5
     7    - 7      5
     8    - 8      5
     9    - 9      5
    10    - 10     5
    11    - 11     5
    12    - 12     5
```

The frames of the signal are concatenated to form a two-dimensional array.

You have now successfully output a frame signal to the MATLAB workspace using the To Workspace block.

More About

· "Create Signals for Sample-Based Processing" on page 2-12

Display Time-Domain Data

In this section...
"Configure the Time Scope Properties" on page 2-83
"Use the Simulation Controls" on page 2-88
"Modify the Time Scope Display" on page 2-89
"Inspect Your Data (Scaling the Axes and Zooming)" on page 2-91
"Manage Multiple Time Scopes" on page 2-94

The following tutorial shows you how to configure the `Time Scope` blocks in the `ex_timescope_tut` model to display time-domain signals. To get started with this tutorial, open the model by typing

`ex_timescope_tut`
at the MATLAB command line.
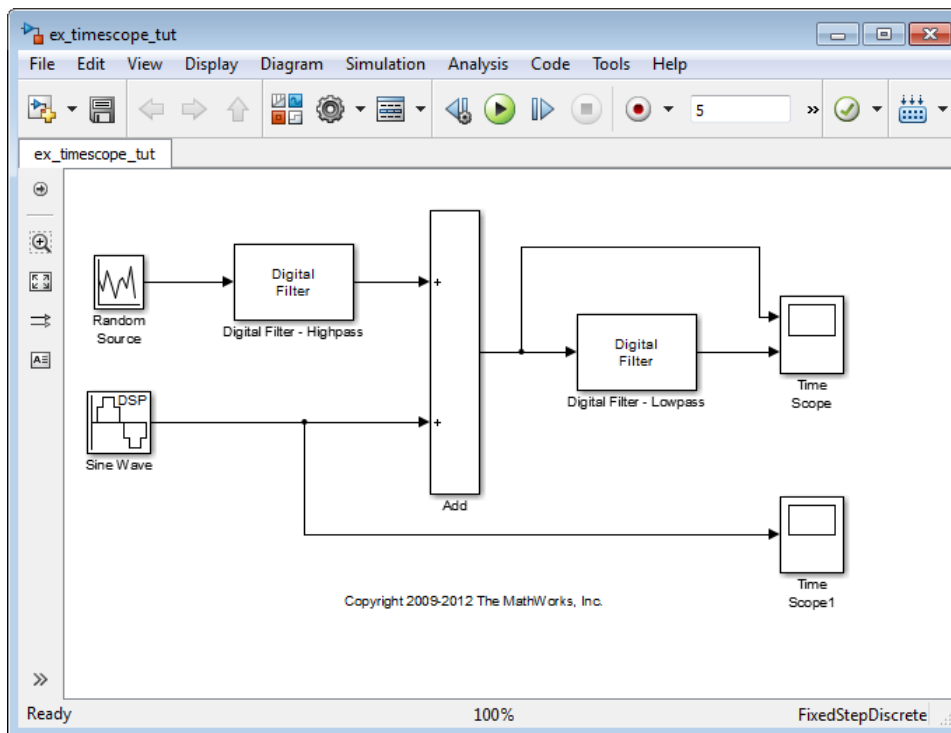
Use the following workflow to configure the Time Scope blocks in the ex_timescope_tut model:

1 "Configure the Time Scope Properties" on page 2-83
2 "Use the Simulation Controls" on page 2-88
3 "Modify the Time Scope Display" on page 2-89
4 "Inspect Your Data (Scaling the Axes and Zooming)" on page 2-91
5 "Manage Multiple Time Scopes" on page 2-94

Configure the Time Scope Properties

The Configuration Properties dialog box provides a central location from which you can change the appearance and behavior of the Time Scope block. To open the Configuration Properties dialog box, you must first open the Time Scope window by double-clicking the Time Scope block in your model. When the window opens, select **View** > **Configuration Properties**. Alternatively, in the Time Scope toolbar, click the Configuration Properties button.
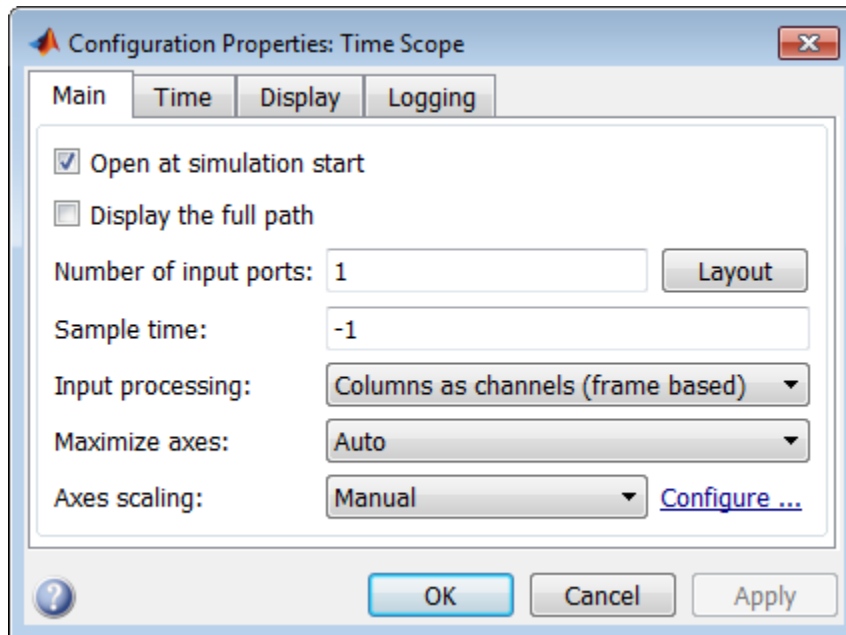
The Configuration Properties dialog box has four different tabs, **Main**, **Time**, **Display**, and **Logging**, each of which offers you a different set of options. For more information about the options available on each of the tabs, see the `Time Scope` block reference page.

Note: As you progress through this workflow, notice the blue question mark icon (🔵) in the lower-left corner of the subsequent dialog boxes. This icon indicates that context-sensitive help is available. You can get more information about any of the parameters on the dialog box by right-clicking the parameter name and selecting **What's This?**

Configure Appearance and Specify Signal Interpretation

First, you configure the appearance of the Time Scope window and specify how the Time Scope block should interpret input signals. In the Configuration Properties dialog box, click the **Main** tab. Choose the appropriate parameter settings for the **Main** tab, as shown in the following table.

Parameter	Setting
Open at simulation start	Checked
Number of input ports	2
Input processing	`Columns as channels (frame based)`
Maximize axes	`Auto`
Axes scaling	`Manual`

In this tutorial, you want the block to treat the input signal as frame-based, so you must set the **Input processing** parameter to `Columns as channels (frame based)`.

Configure Axes Scaling and Data Alignment

The **Main** tab also allows you to control when and how Time Scope scales the axes. These options also control how Time Scope aligns your data with respect to the axes. Click the link labeled **Configure...** to the right of the **Axes scaling** parameter to see additional options for axes scaling. After you click this button, the label changes to **Hide...** and new parameters appear. The following table describes these additional options.

Parameter	Description

Axes scaling	Specify when the scope automatically scales the axes. You can select one of the following options:
	
	- Manual — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
	
	  - Select **Tools > Axes Scaling Properties**.
	  - Press one of the **Scale Axis Limits** toolbar buttons.
	  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
	
	- Auto — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
	
	- After N Updates — Selecting this option causes the scope to scale the axes after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.
	
	By default, this property is set to Auto. This property is Tunable.

Scale axes limits at stop	Select this check box to scale the axes when the simulation stops. The y-axis is always scaled. The x-axis limits are only scaled if you also select the **Scale X-axis limits** check box.
Data range (%)	Allows you to specify how much white space surrounds your signal in the Time Scope window. You can specify a value for both the y- and x-axis. The higher the value you enter for the y-axis **Data range (%)**, the tighter the y-axis range is with respect to the minimum and maximum values in your signal. For example, to have your signal cover the entire y-axis range when the block scales the axes, set this value to 100.
Align	Allows you to specify where the block should align your data with respect to each axis. You can choose to have your data aligned with the top, bottom, or center of the y-axis. Additionally, if you select the **Autoscale X-axis limits** check box, you can choose to have your data aligned with the right, left, or center of the x-axis.

Set the parameters to the values shown in the following table.

Parameter	Setting
Axes scaling	Manual
Scale axes limits at stop	Checked
Data range (%)	80
Align	Center
Autoscale X-axis limits	Unchecked

Set Time Domain Properties

In the Configuration Properties dialog box, click the **Time** tab. Set the parameters to the values shown in the following table.

Parameter	Setting
Time span	One frame period
Time span overrun action	Wrap
Time units	Metric (based on Time Span)
Time display offset	0
Time-axis labels	All
Show time-axis label	Checked

The **Time span** parameter allows you to enter a numeric value, a variable that evaluates to a numeric value, or select the One frame period menu option. You can also select the Auto menu option; in this mode, Time Scope automatically calculates the appropriate value for time span from the difference between the simulation "Start time" and "Stop time" parameters. The actual range of values that the block displays on the *time*-axis depends on the value of both the **Time span** and **Time display offset** parameters. See the following figure.
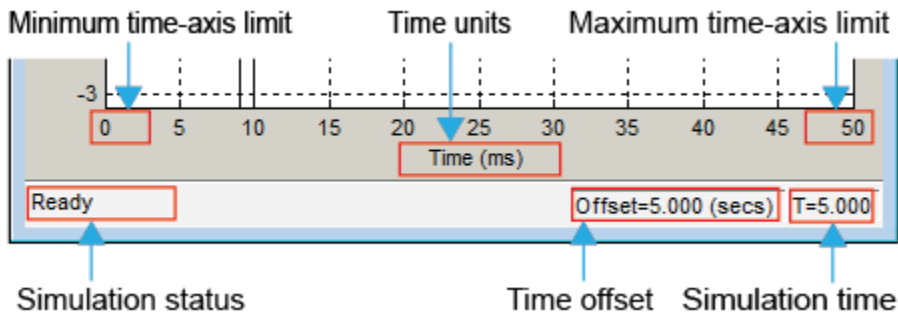
If the **Time display offset** parameter is a scalar, the value of the minimum *time*-axis limit is equal to the **Time display offset**. In addition, the value of the maximum *time*-axis limit is equal to the sum of the **Time display offset** parameter and the **Time span** parameter. For information on the other parameters in the Time Scope window, see the `Time Scope` reference page.

In this tutorial, the values on the *time*-axis range from `0` to `One frame period`, where `One frame period` is `0.05` seconds (50 ms).

Set Display Properties

In the Configuration Properties dialog box, click the **Display** tab. Set the parameters to the values shown in the following table.

Parameter	Setting
Active display	1
Title	
Show legend	Checked
Show grid	Checked
Plot signal(s) as magnitude and phase	Unchecked
Y-limits (Minimum)	`-2.5`
Y-limits (Maximum)	`2.5`
Y-label	`Amplitude`

Set Logging Properties

In the Configuration Properties dialog box, click the **Logging** tab. Set **Log data to workspace** to unchecked.

Click **OK** to save your changes and close the Configuration Properties dialog box.

Note: If you have not already done so, repeat all of these procedures for the Time Scope1 block (except leave the **Number of input ports** on the **Main** tab as 1) before continuing with the other sections of this tutorial.

Use the Simulation Controls

One advantage to using the Time Scope block in your models is that you can control model simulation directly from the Time Scope window. The buttons on the Simulation Toolbar of the Time Scope window allow you to play, pause, stop, and take steps forward or backward through model simulation. Alternatively, there are several keyboard shortcuts you can use to control model simulation when the Time Scope is your active window.

You can access a list of keyboard shortcuts for the Time Scope by selecting **Help > Keyboard Command Help**. The following procedure introduces you to these features.

1 If the Time Scope window is not open, double-click the block icon in the `ex_timescope_tut` model. Start model simulation. In the Time Scope window, on the Simulation Toolbar, click the Run button (▶) on the Simulation Toolbar. You can also use one of the following keyboard shortcuts:

  - **Ctrl+T**
  - **P**
  - **Space**

2 While the simulation is running and the Time Scope is your active window, pause the simulation. Use either of the following keyboard shortcuts:

  - **P**
  - **Space**

Alternatively, you can pause the simulation in one of two ways:

- In the Time Scope window, on the Simulation Toolbar, click the Pause button (⏸).
- From the Time Scope menu, select **Simulation** > **Pause**.

3   With the model simulation still paused, advance the simulation by a single time step. To do so, in the Time Scope window, on the Simulation Toolbar, click the Next Step button (▷).

   Next, try using keyboard shortcuts to achieve the same result. Press the **Page Down** key to advance the simulation by a single time step.

4   Resume model simulation using any of the following methods:

- From the Time Scope menu, select **Simulation** > **Continue**.
- In the Time Scope window, on the Simulation Toolbar, click the Continue button (▶).
- Use a keyboard shortcut, such as **P** or **Space**.

Modify the Time Scope Display

You can control the appearance of the Time Scope window using options from the display or from the **View** menu. Among other capabilities, these options allow you to:

- Control the display of the legend
- Edit the line properties of your signals
- Show or hide the available toolbars

Change Signal Names in the Legend

You can change the name of a signal by double-clicking the signal name in the legend. By default, the Time Scope names the signals based on the block they are coming from. For this example, set the signal names as shown in the following table.

Block Name	Original Signal Name	New Signal Name
Time Scope	Add	Noisy Sine Wave
Time Scope	Digital Filter – Lowpass	Filtered Noisy Sine Wave
Time Scope1	Sine Wave	Original Sine Wave
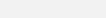
Modify Axes Colors and Line Properties

Use the Style dialog box to modify the appearance of the axes and the lines for each of the signals in your model. In the Time Scope menu, select **View** > **Style**.
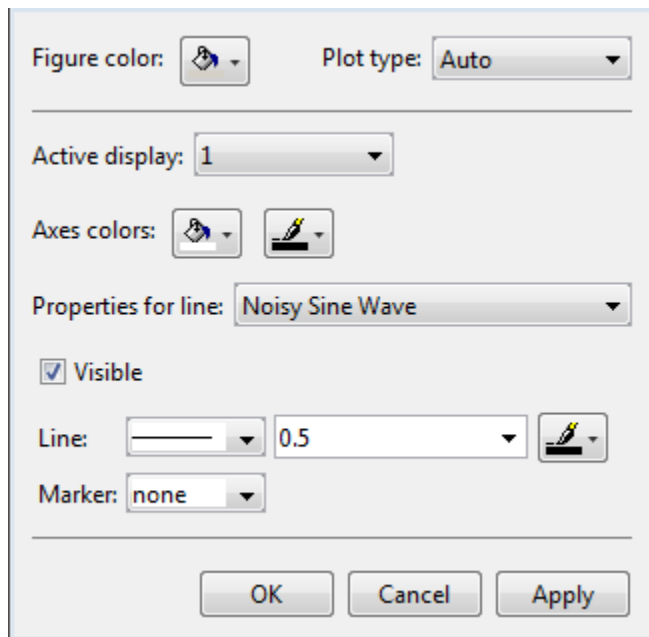
1. Change the **Plot Type** parameter to Auto for each Time Scope block. This setting ensures that Time Scope displays a line graph if the signal is continuous and a stairstep graph if the signal is discrete.

2. Change the **Axes colors** parameters for each Time Scope block. Leave the axes background color as black and set the ticks, labels, and grid colors to white.

3. Set the **Properties for line** parameter to the name of the signal for which you would like to modify the line properties. Set the line properties for each signal according to the values shown in the following table.

Block Name	Signal Name	Line	Line Width	Marker	Color
Time Scope	Noisy Sine Wave	———	0.5	none	White

Block Name	Signal Name	Line	Line Width	Marker	Color
Time Scope	Filtered Noisy Sine Wave	———————	0.5	◇	Red
Time Scope1	Original Sine Wave	———————	0.5	✳	Yellow

Show and Hide Time Scope Toolbars

You can also use the options on the **View** menu to show or hide toolbars on the Time Scope window. For example:

- To hide the simulation controls, select **View** > **Toolbar**. Doing so removes the simulation toolbar from the Time Scope window and also removes the check mark from next to the **Toolbar** option in the **View** menu.

- You can choose to show the simulation toolbar again at any time by selecting **View** > **Toolbar**.

Verify that all toolbars are visible before moving to the next section of this tutorial.

Inspect Your Data (Scaling the Axes and Zooming)

Time Scope has plot navigation tools that allow you to scale the axes and zoom in or out on the Time Scope window. The axes scaling tools allow you to specify when and how often the Time Scope scales the axes.

So far in this tutorial, you have configured the Time Scope block for manual axes scaling. Use one of the following options to manually scale the axes:

- From the Time Scope menu, select **Tools** > **Scale Axes Limits**.
- Press the Scale Axes Limits toolbar button (🔼).
- With the Time Scope as your active window, press **Ctrl + A**.

Adjust White Space Around the Signal

You can control how much space surrounds your signal and where your signal appears in relation to the axes. To adjust the amount of space surrounding your signal and realign

it with the axes, you must first open the Tools—Plot Navigation Properties dialog box. From the Time Scope menu, select **Tools** > **Axes Scaling Properties** .

In the Tools:Plot Navigation options dialog box, set the **Data range (%)** and **Align** parameters. In a previous section, you set these parameters to `80` and `Center`, respectively.

- To decrease the amount of space surrounding your signal, set the **Data range (%)** parameter on the Tools:Plot Navigation Options dialog box to `90`.
- To align your signal with the bottom of the Y-axis, set the **Align** parameter to `Bottom`.

The next time you scale the axes of the Time Scope window, the window appears as follows.
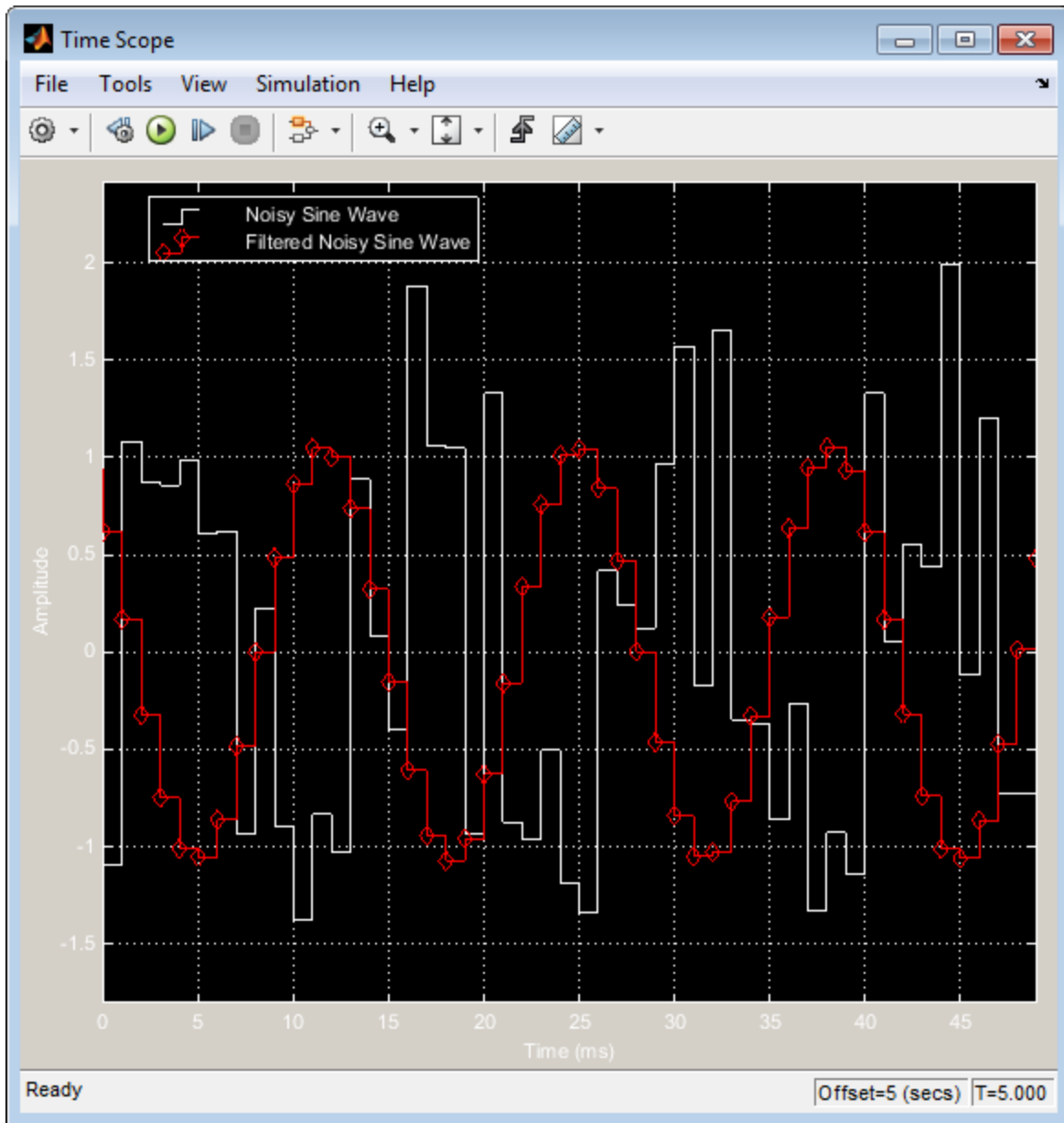
Use the Zoom Tools

The zoom tools allow you to zoom in simultaneously in the directions of both the *x*- and *y*-axes , or in either direction individually. For example, to zoom in on the signal between 5010 ms and 5020 ms, you can use the **Zoom X** option.

- To activate the **Zoom X** tool, select **Tools** > **Zoom X**, or press the corresponding toolbar button ( ). The Time Scope indicates that the **Zoom X** tool is active by depressing the toolbar button and placing a check mark next to the **Tools** > **Zoom X** menu option.

- To zoom in on the region between 5010 ms and 5020 ms, in the Time Scope window, click and drag your cursor from the 10 ms mark to the 20 ms mark.

- While zoomed in, to activate the **Pan** tool, select **Tools** > **Pan**, or press the corresponding toolbar button ( ).

- To zoom out of the Time Scope window, right-click inside the window, and select **Zoom Out**. Alternatively, you can return to the original view of your signal by right-clicking inside the Time Scope window and selecting **Reset to Original View**.

Manage Multiple Time Scopes

The Time Scope block provides tools to help you manage multiple Time Scope blocks in your models. The model used throughout this tutorial, `ex_timescope_tut`, contains two Time Scope blocks, labeled `Time Scope` and `Time Scope1`. The following sections discuss the tools you can use to manage these Time Scope blocks.

Open All Time Scope Windows

When you have multiple windows open on your desktop, finding the one you need can be difficult. The Time Scope block offers a **View** > **Bring All Time Scopes Forward** menu option to help you manage your Time Scope windows. Selecting this option brings all Time Scope windows into view. If a Time Scope window is not currently open, use this menu option to open the window and bring it into view.

To try this menu option in the `ex_timescope_tut` model, open the Time Scope window, and close the Time Scope1 window. From the **View** menu of the Time Scope window, select **Bring All Time Scopes Forward**. The Time Scope1 window opens, along with the already active Time Scope window. If you have any Time Scope blocks in other open Simulink models, then these also come into view.

Open Time Scope Windows at Simulation Start

When you have multiple Time Scope blocks in your model, you may not want all Time Scope windows to automatically open when you start simulation. You can control whether or not the Time Scope window opens at simulation start by selecting **File** > **Open at Start of Simulation** from the Time Scope window. When you select this option, the Time Scope GUI opens automatically when you start the simulation. When you do not select this option, you must manually open the scope window by double-clicking the corresponding Time Scope block in your model.

Find the Right Time Scope Block in Your Model

Sometimes, you have multiple Time Scope blocks in your model and need to find the location of one that corresponds to the active Time Scope window. In such cases, you can use the **View** > **Highlight Simulink Block** menu option or the corresponding toolbar button ( ). When you do so, the model window becomes your active window, and the corresponding Time Scope block flashes three times in the model window. This option can help you locate Time Scope blocks in your model and determine to which signals they are attached.

To try this feature, open the Time Scope window, and on the simulation toolbar, click the Highlight Simulink Block button. Doing so opens the `ex_timescope_tut` model. The Time Scope block flashes three times in the model window, allowing you to see where in your model the block of interest is located.

Docking Time Scope Windows in the Scopes Group Container

When you have multiple Time Scope blocks in your model you may want to see them in the same window and compare them side-by-side. In such cases, you can select the Dock Time Scope button ( ) at the top-right corner of the Time Scope window for the Time Scope block.
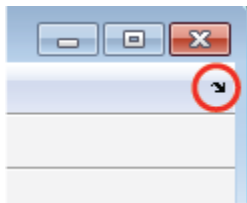
The Time Scope window now appears in the Scopes group container. Next, press the Dock Time Scope button at the top-right corner of the Time Scope window for the Time Scope1 block.

By default, the Scopes group container is situated above the MATLAB Command Window. However, you can undock the Scopes group container by pressing the Show Actions button (⊙) at the top-right corner of the container and selecting **Undock**. The Scopes group container is now independent from the MATLAB Command Window.
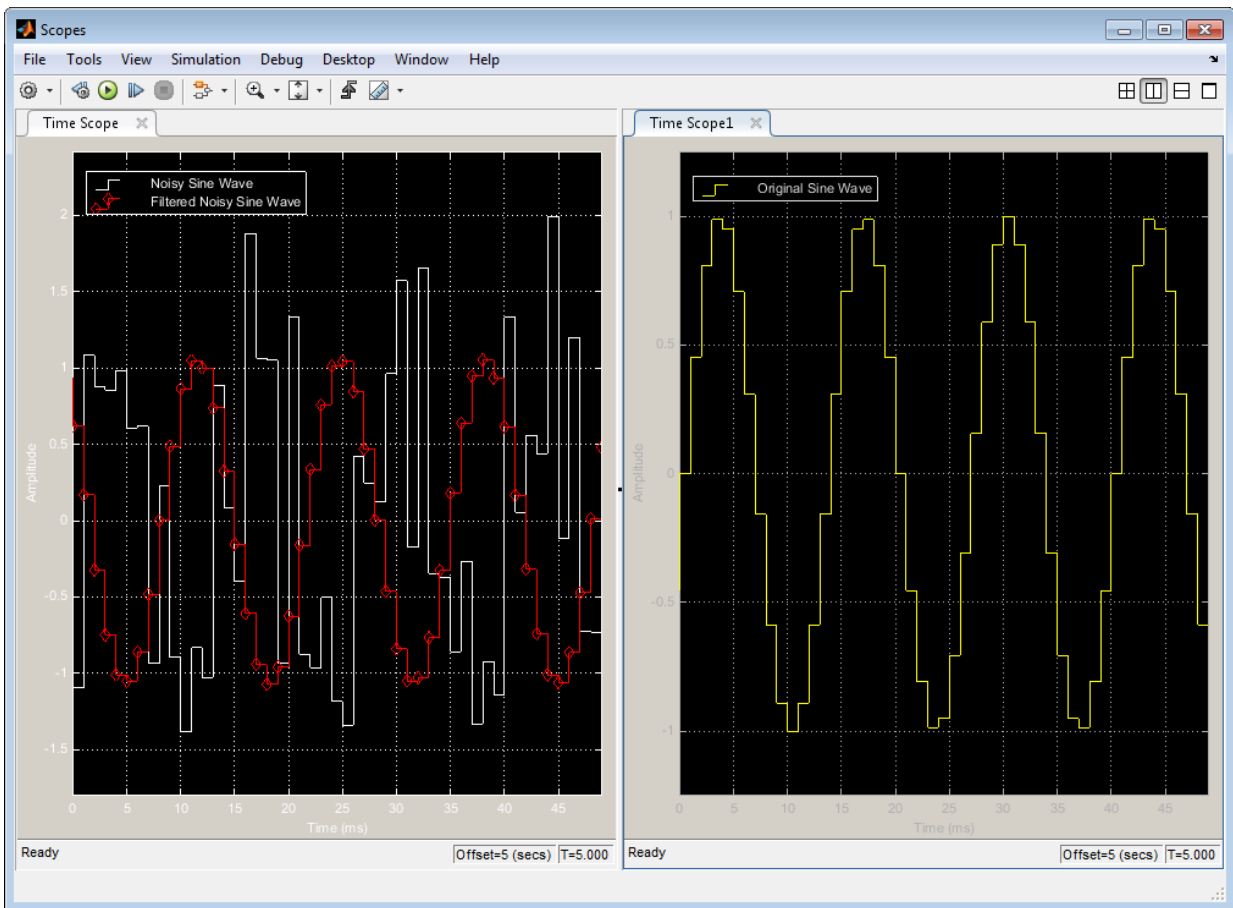
Once docked, the Scopes group container displays the toolbar and menu bar of the Time Scope window. If you open additional instances of Time Scope, a new Time Scope window appears in the Scopes group container.

You can undock any instance of Time Scope by pressing the corresponding Undock button (⬈) in the title bar of each docked instance. If you close the Scopes group container, all docked instances of Time Scope close but the Simulink model continues to run.

Close All Time Scope Windows

If you save your model with Time Scope windows open, those windows will reopen the next time you open the model. Reopening the Time Scope windows when you open your model can increase the amount of time it takes your model to load. If you are working with a large model, or a model containing multiple Time Scopes, consider closing all Time Scope windows before you save and close that model. To do so, use the **File** > **Close All Time Scope Windows** menu option.

To use this menu option in the `ex_timescope_tut` model, open the Time Scope or Time Scope1 window, and select **File** > **Close All Time Scope Windows**. Both the Time Scope and Time Scope1 windows close. If you now save and close the model, the Time Scope windows do not automatically open the next time you open the model. You can open Time Scope windows at any time by double-clicking a Time Scope block in your model. Alternatively, you can choose to automatically open the Time Scope windows at simulation start. To do so, from the Time Scope window, select **File** > **Open at Start of Simulation**.

Display Frequency-Domain Data in Spectrum Analyzer

You can use DSP System Toolbox blocks to work with signals in both the time and frequency domain. To display frequency-domain signals, you can use blocks from the Sinks library, such as the Vector Scope, Spectrum Analyzer, Matrix Viewer, and Waterfall Scope blocks.

With the `Spectrum Analyzer` block, you can display the frequency spectra of time-domain input data. In contrast to the Vector Scope block, the Spectrum Analyzer block computes the Fast Fourier Transform (FFT) of the input signal internally, transforming the signal into the frequency domain.

This example shows how you can use a Spectrum Analyzer block to display the frequency content of two frame-based signals simultaneously:

1 At the MATLAB command prompt, type `ex_spectrumanalyzer_tut`.

The Spectrum Analyzer example opens, and the variables, `Fs` and `mtlb`, are loaded into the MATLAB workspace.
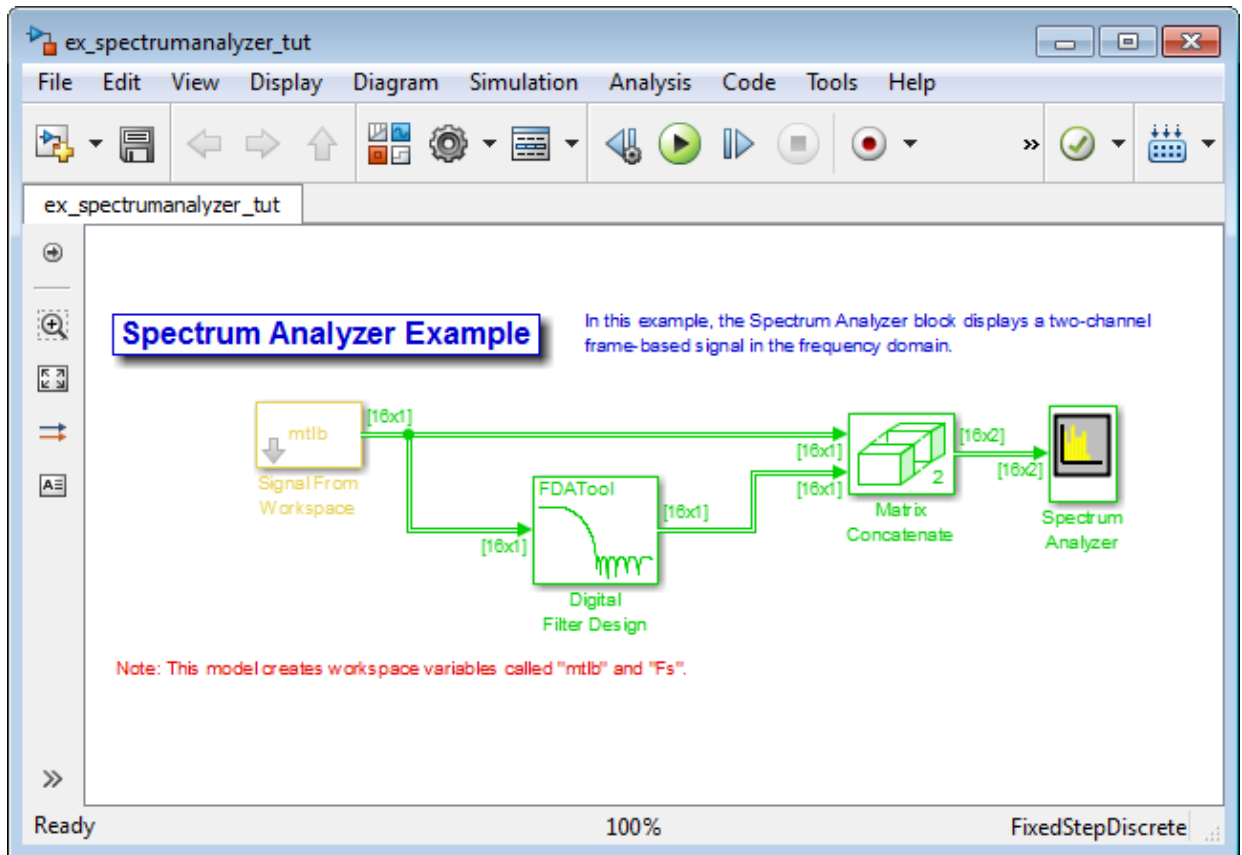
2   Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **Signal** = `mtlb`
- **Sample time** = `1`
- **Samples per frame** = `16`
- **Form output after final data value** = `Cyclic Repetition`

Based on these parameters, the Signal From Workspace block repeatedly outputs the input signal, `mtlb`, as a frame-based signal with a sample period of 1 second.

3 Create two distinct signals to send to the Spectrum Analyzer block. Use the Digital Filter Design block to filter the input signal, using the default parameters.
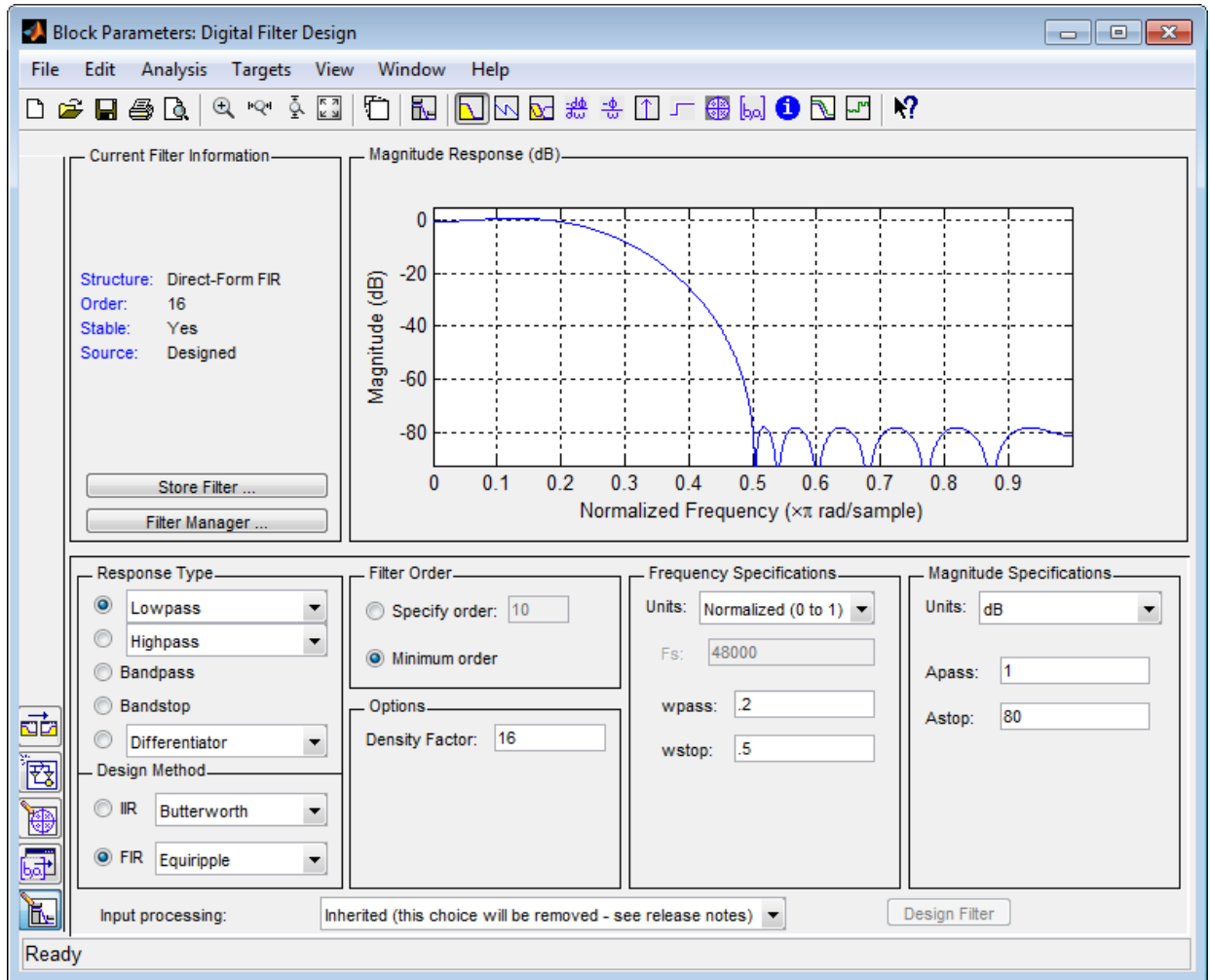
4 Double-click the Matrix Concatenate block. Set the block parameters as follows, and then click **OK**:

- **Number of inputs** = 2
- **Mode** = `Multidimensional array`

- **Concatenate dimension** = 2

The Matrix Concatenate block combines the two signals so that each column corresponds to a different signal.

5 Double-click the Spectrum Analyzer block. The Spectrum Analyzer figure appears. In the menu, select **View** > **Spectrum Settings**. The Spectrum Settings panel opens.

- Expand the **Main options** pane, if it is not already expanded.

  - Set **Type** to Power.
  - Select the **Full frequency span** check box.
  - Set **RBW (Hz)** to 5.91e-3.

- Expand the **Trace options** pane, if it is not already expanded.

  - Set **Units** to dBW.
  - Set **Averages** to 2.

- Expand the **Window options** pane, if it is not already expanded.

  - Set **Overlap (%)** to 50.
  - Set **Window** to Hann.

Based on these parameters, the Spectrum Analyzer uses 128 samples from each input channel to calculate a new windowed data segment, as shown in the following equation.

$$D = \frac{NENBW \times F_s}{RBW} = \frac{1.512 \times 1 Hz}{11.8125 \times 10^{-3} Hz} = 128 samples$$

There are also 128 frequency points in the FFT. Also, because **Overlap (%)** is set to 50, there is a buffer overlap length of 64 samples in each spectral estimate, as shown in the following equation.

$$O_L = \frac{O_P}{100} \times L = \frac{50}{100} \times 128 = 64 samples$$

Every time the scope updates the display, 64 points are plotted for each channel. At 16 samples per frame, Spectrum Analyzer waits for 3 frames or 48 samples before displaying the first power spectral estimate.

2-101

6 Fit all the calculated data points into the display. In the Spectrum Analyzer menu, select **Tools** > **Automatically Scale Axes Limits**.

7 In the Spectrum Analyzer menu, select **View** > **Configuration Properties**. Then, select the **Show legend** check box.

8 Run the model. The Spectrum Analyzer block computes the FFT of each of the input signals. It then displays the power spectra of the frequency-domain signals in the Spectrum Analyzer window.
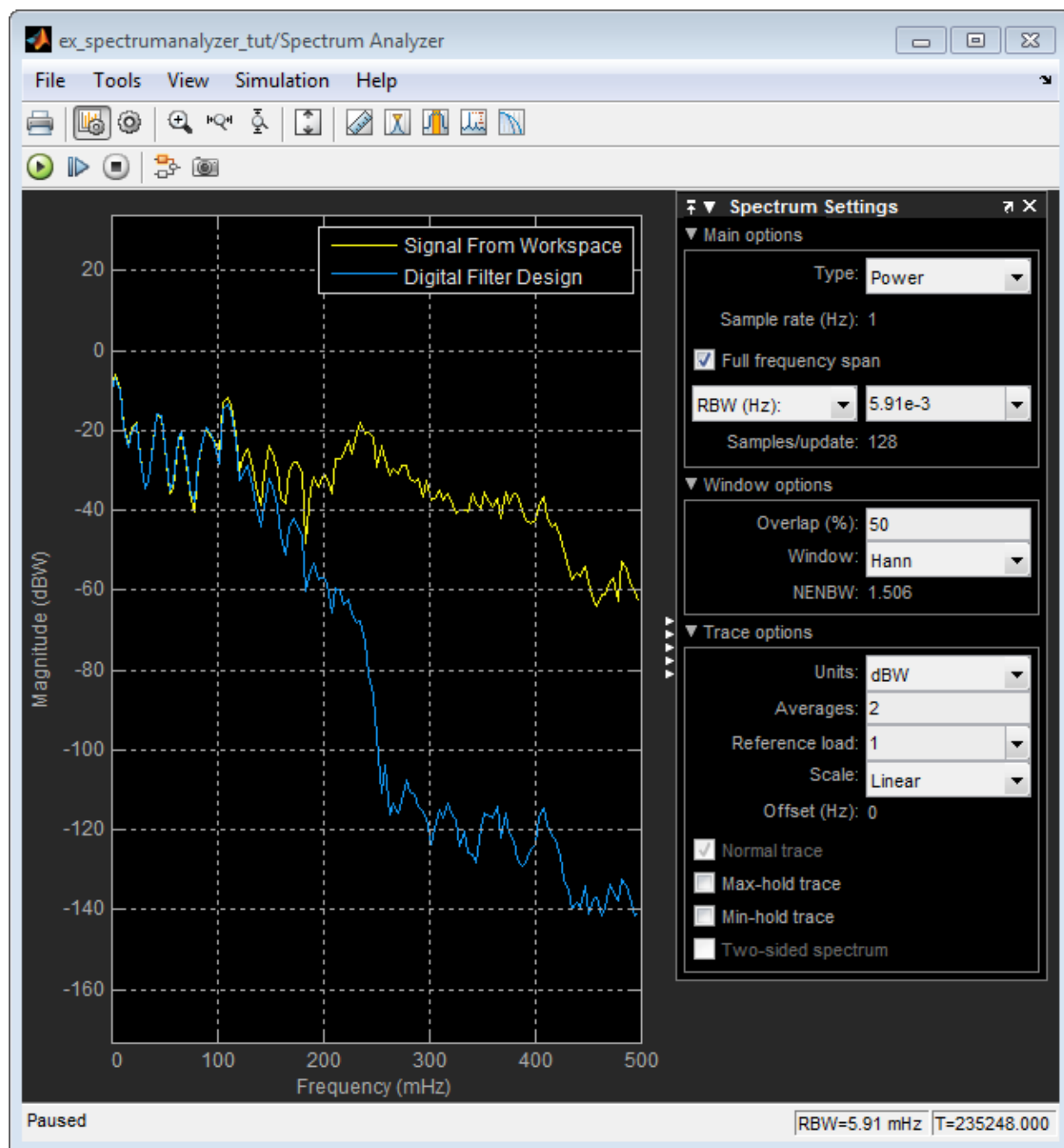
The power spectrum of the first input signal, from column one, is the yellow line. The power spectrum of the second input signal, from column two, is the blue line.

Visualize Central Limit Theorem in Array Plot

In this section...

Display a Uniform Distribution

This example shows how to use and configure the dsp.ArrayPlot to visualize the Central Limit Theorem. This theorem states that the mean of a large number of independent random variables with finite mean and variance exhibits a normal distribution.

First, generate uniformly distributed random variables in MATLAB using the `rand` function. Find their distributions using the `histogram` function. At the MATLAB command line, type:

```
numsamples  = 1e4;
numbins     = 20;
r = rand(numsamples,1);
hst = histogram(r,numbins);
```

Create a new Array Plot object.

```
hap3 = dsp.ArrayPlot;
```

Configure the properties of the Array Plot object to plot a histogram.

```
hap3 = dsp.ArrayPlot;
hap3.XOffset = 0;
hap3.SampleIncrement = 1/numbins;
hap3.PlotType = 'Stem';
hap3.YLimits = [0, max(hst)+1];
```

Call the step method to plot the uniform distribution.

```
step(hap3,hst');
```

The following Array Plot figure appears, showing a uniform distribution.

Processing
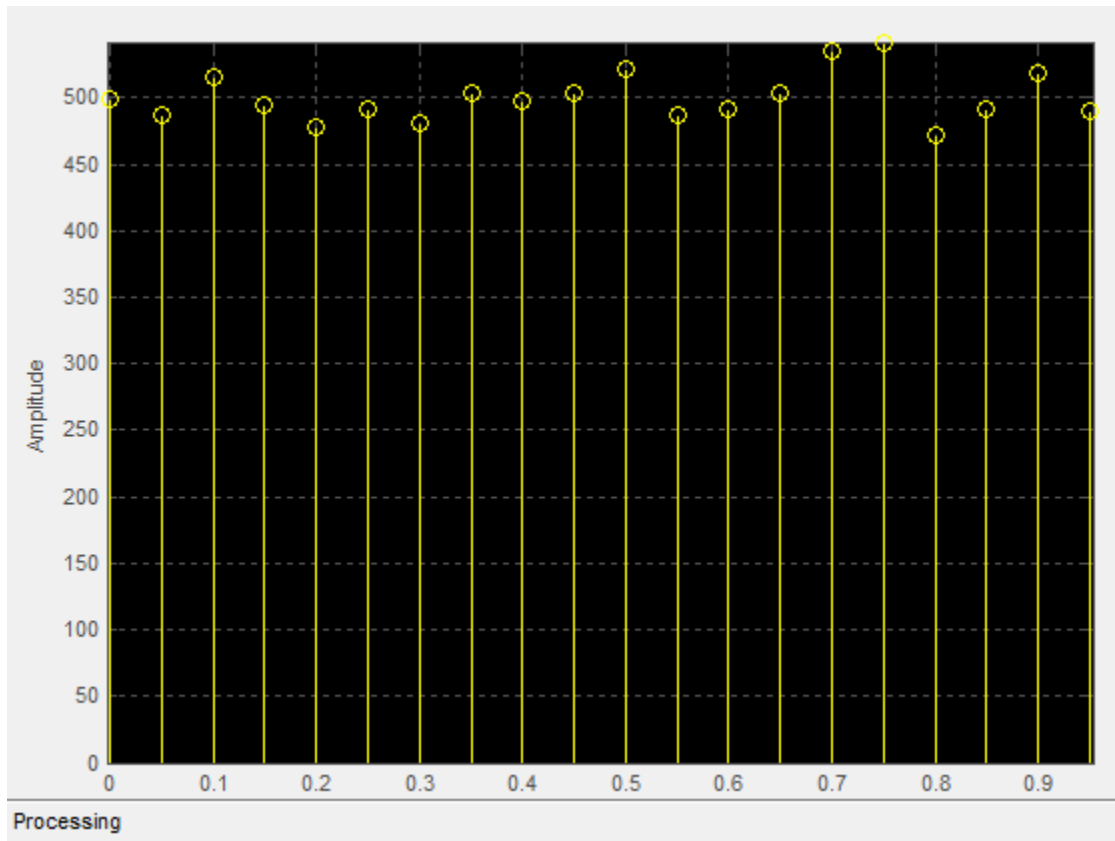
Display the Sum of Many Uniform Distributions

Next, calculate the mean of multiple uniformly distributed random variables. As the number of random variables increases, the distribution more closely resembles a normal curve. Run the release method to let property values and input characteristics change. At the MATLAB command line, type:

```
release(hap3);
```

Change the configuration of the Array Plot properties for the display of a distribution function.

```
numbins    = 201;
```

```
numtrials  = 100;
r = zeros(numsamples,1);
hap3.SampleIncrement = 1/numbins;
hap3.PlotType = 'Stairs';
```

Call the step method repeatedly to plot the uniform distribution.

```
for ii = 1:numtrials
    r = rand(numsamples,1)+r;
    hst = histogram(r/ii,0:1/numbins:1);
    hap3.YLimits = [min(hst)-1, max(hst)+1];
    step(hap3,hst');
    pause(0.1);
end
```

When the simulation has finished, the Array Plot figure displays a bell curve, indicating a distribution that is close to normal.

Processing
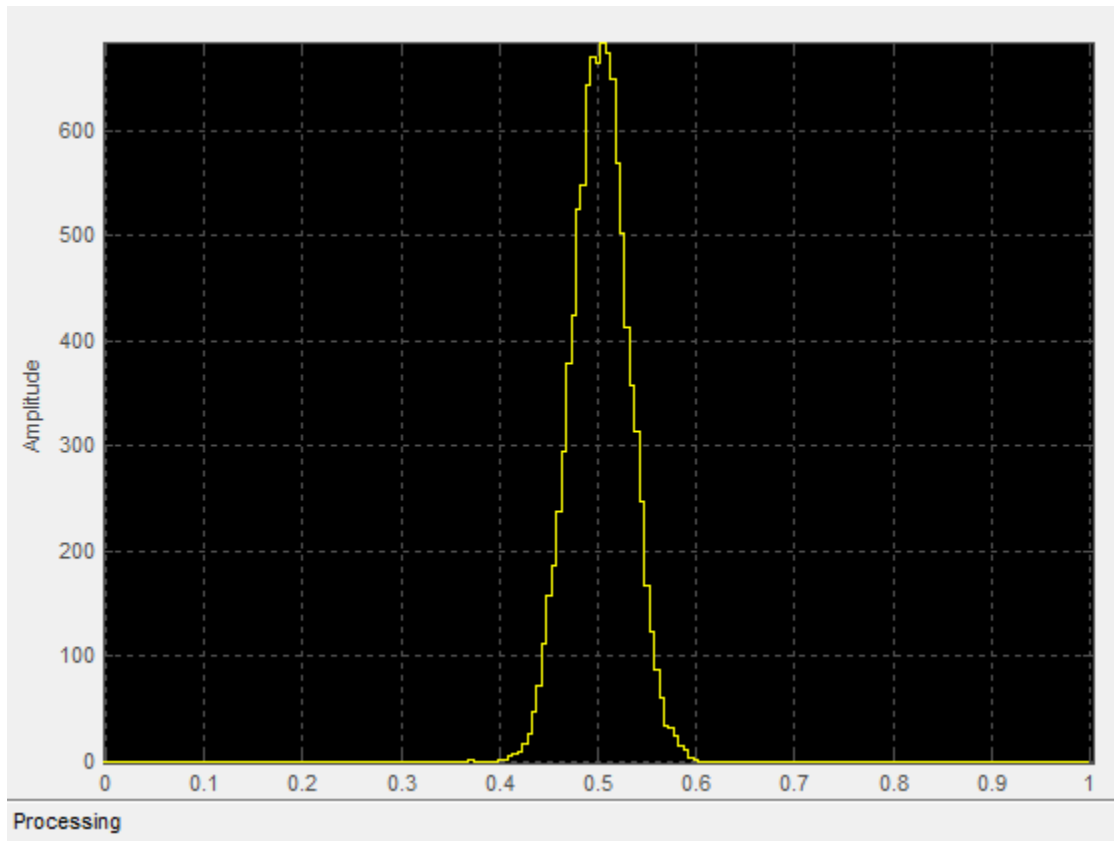
Inspect Your Data by Zooming

The zoom tools allow you to zoom in simultaneously in the directions of both the *x*- and *y*-axes or in either direction individually. For example, to zoom in on the distribution between 0.3 and 0.7, you can use the **Zoom X** option.

- To activate the **Zoom X** tool, select **Tools** > **Zoom X**, or press the corresponding toolbar button ( ). You can determine if the **Zoom X** tool is active by looking for an indented toolbar button or a check mark next to the **Tools** > **Zoom X** menu option.
- Next, zoom in on the region between 0.3 and 0.7. In the Array Plot window, click on the 0.3-second mark, and drag to the 0.7-second mark. The display reflects this new *x*-axis setting, as shown in the following figure.

Processing
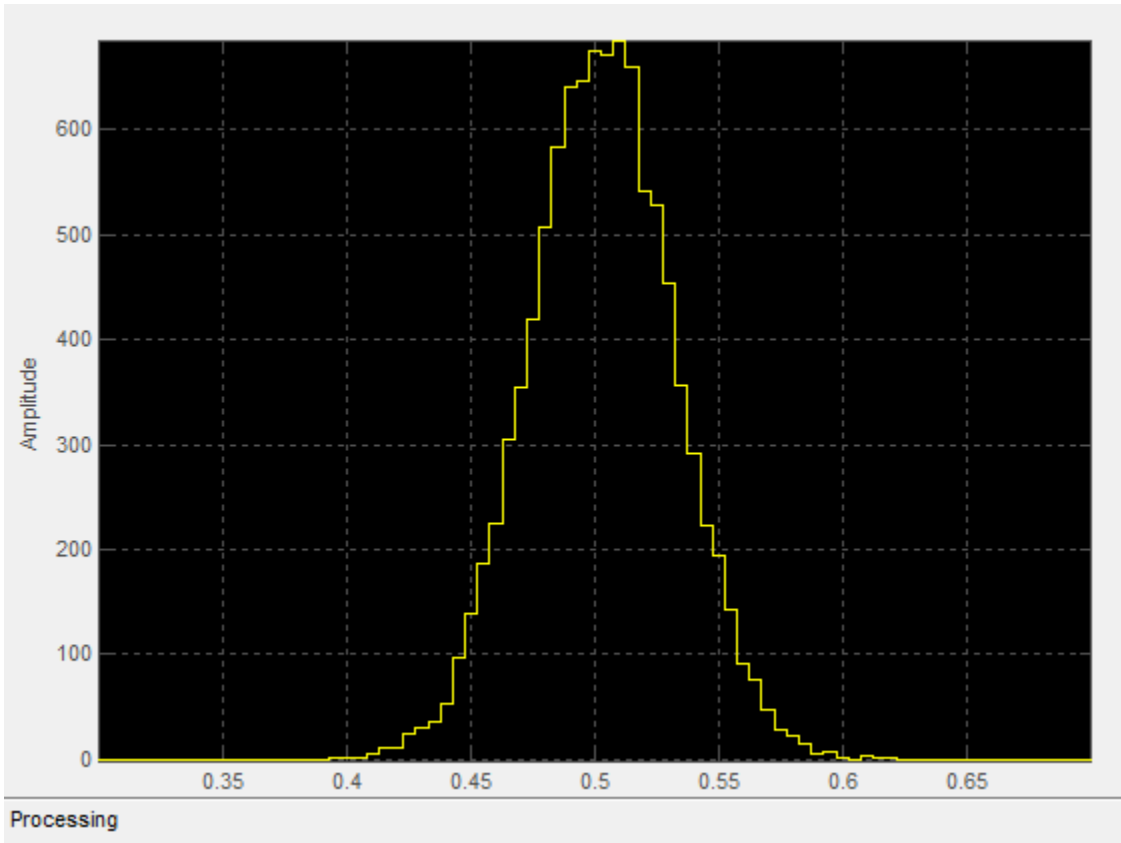
Display Multiple Signals in the Time Scope

In this section...
"Multiple Signal Input" on page 2-110
"Multiple Time Offsets" on page 2-112
"Multiple Displays" on page 2-113

Multiple Signal Input

You can configure the Time Scope block to show multiple signals within the same display or on separate displays. By default, the signals appear as different-colored lines on the same display. The signals can have different dimensions, sample rates, and data types. Each signal can be either real or complex valued. You can set the number of input ports on the Time Scope block in the following ways:

- Right-click the Time Scope block in your model to bring up the context menu. Point your cursor to the **Signals & Ports** > **Number of Input Ports** item on the context menu. You can then select the number of input ports for the Time Scope block. If the desired number of input signals is 1, 2, or 3, then click on the appropriate value. To configure a Time Scope block to have more than three input ports, select **More**, and enter a number for the **Number of input ports** parameter.

- Open the Time Scope window by double-clicking the Time Scope block in your model. In the scope menu, select **File** > **Number of Input Ports**.

- Open the Time Scope window by double-clicking the Time Scope block in your model. In the scope menu, select **View** > **Configuration Properties** and set the **Number of input ports** on the **Main** tab.

An input signal may contain multiple channels, depending on its dimensions. Multiple channels of data are always shown as different-colored lines on the same display.

Multiple Signal Names

By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal named `Signal1` would have the following default names in the channel legend: `Signal1:1`, `Signal1:2`. In the following example, there is one 3-channel input signal

and one 4-channel input signal to the scope block, one signal named `Signal1` and one unnamed signal coming from a block named `Block2`.
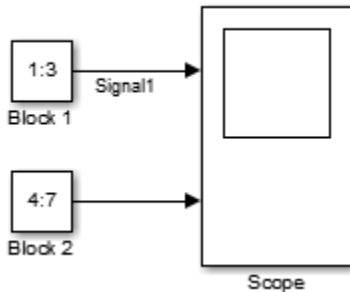
To see all the signal names, run the simulation and show the legend. To show the legend, select **View** > **Configuration Properties**, click the **Display** tab, and select the **Show Legend** check box. The following legend appears in the display.
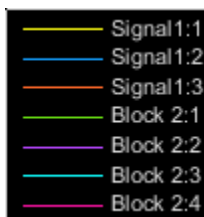
Note: Continuous signals appear as straight lines in the legend. Discrete signals appear as step-shaped lines.

The scope does not display signal names that were labeled within an unmasked subsystem. You must label all input signals to the scope block that originate from an unmasked subsystem.

Multiple Signal Colors

By default, the scope has a black axes background and chooses line colors for each channel in the same manner as the Simulink `Scope` block. When the scope axes

background is black, it assigns each channel of each input signal a line color in the order shown above.

If there are more than 7 channels, then the scope repeats this order to assign line colors to the remaining channels. To choose line colors for each channel in the same manner as the MATLAB `plot` function, change the axes background color to any color except black. To change the axes background color to white, select **View** > **Style**, click the

Axes background color button (), and select white from the color palette. Run the simulation again. The following legend appears in the display.
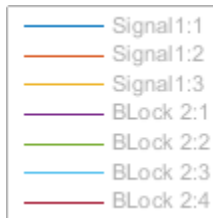
When the scope axes background is not black, it assigns each channel of each input signal a line color in order shown above.If there are more than 7 channels, then the scope repeats this order to assign line colors to the remaining channels. To manually modify any line color, select **View** > **Style** to open the Style dialog box. Next to **Properties for line**, select the signal name whose color you want to change. Then, next to **Line**, click

the Line color button () and select any color from the palette.

Multiple Time Offsets

You can offset all channels of an input signal by the same number of seconds or offset each channel independently. To offset all channels equally, select **View** > **Configuration Properties**, and specify a scalar value for the **Time display offset** parameter on the **Main** pane. To offset each channel independently, specify a vector of offset values. When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

- When *N* is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.
- When *N* is less than the number of input channels, the scope applies the values you specify in the offset vector to the first *N* input channels. The scope does not offset the remaining channels.

- When *N* is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

Multiple Displays

You can display multiple channels of data on different displays in the scope window. In the scope toolbar, select **View** > **Layout**, or select the Layout button (⊞) in the dropdown below the Configuration Properties button.

Note: The **Layout** menu item and button are not available when the scope is in snapshot mode.

This feature allows you to tile the window into a number of separate displays, up to a grid of 4 rows and 4 columns. For example, if there are three inputs to the scope, you can display the signals in separate displays by selecting row 3, column 1, as shown in the following figure.
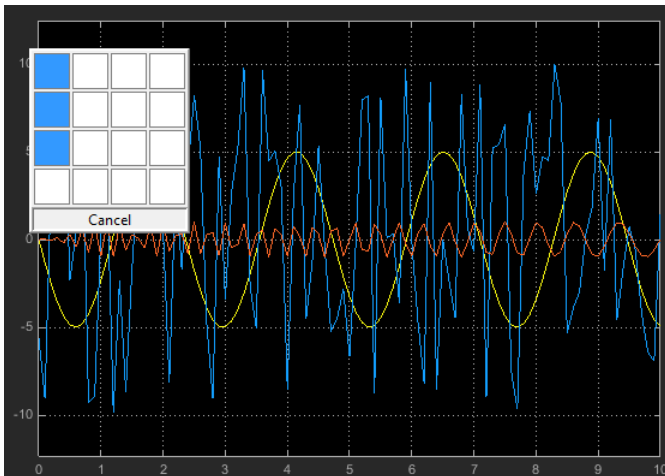
After you select row 3, column 1, the scope window is partitioned into three separate displays, as shown in the following figure.
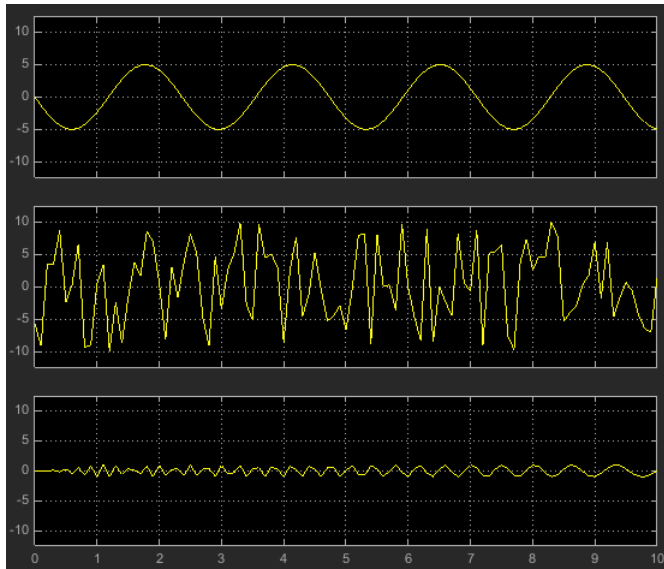
When you use the Layout option to tile the window into multiple displays, the display highlighted in yellow is referred to as the *active display*. The scope dialog boxes reference the active display.

3

Data and Signal Management

Learn concepts such as sample- and frame-based processing, sample rate, delay and latency.

Sample- and Frame-Based Concepts

In this section...
"Sample- and Frame-Based Signals" on page 3-2
"Model Sample- and Frame-Based Signals in MATLAB and Simulink" on page 3-3
"What Is Sample-Based Processing?" on page 3-3
"What Is Frame-Based Processing?" on page 3-4

Sample- and Frame-Based Signals

Sample-based signals are the most basic type of signal and are the easiest to construct from a real-world (physical) signal. You can create a sample-based signal by sampling a physical signal at a given sample rate, and outputting each individual sample as it is received. In general, most Digital-to-Analog converters output sample-based signals.

You can create frame-based signals from sample-based signals. When you buffer a batch of N samples, you create a frame of data. You can then output sequential frames of data at a rate that is $1/N$ times the sample rate of the original sample-based signal. The rate at which you output the frames of data is also known as the *frame rate* of the signal.

Frame-based data is a common format in real-time systems. Data acquisition hardware often operates by accumulating a large number of signal samples at a high rate. The hardware then propagates those samples to the real-time system as a block of data. Doing so maximizes the efficiency of the system by distributing the fixed process overhead across many samples. The faster data acquisition is suspended by slower interrupt processes after each frame is acquired, rather than after each individual sample. See "Benefits of Frame-Based Processing" on page 3-6 for more information.

DSP System Toolbox Source Blocks	Create Sample-Based Signals	Create Frame-Based Signals
Chirp	X	X
Constant	X	X
Colored Noise	X	X
Discrete Impulse	X	X
From Multimedia File	X	X

DSP System Toolbox Source Blocks	Create Sample-Based Signals	Create Frame-Based Signals
Identity Matrix	X	
Multiphase Clock	X	X
N-Sample Enable	X	X
Random Source	X	
Signal From Workspace	X	X
Sine Wave	X	X
UDP Receive	X	

Model Sample- and Frame-Based Signals in MATLAB and Simulink

When you process signals using DSP System Toolbox software, you can do so in either a sample- or frame-based manner. When you are working with blocks in Simulink, you can specify, on a block-by-block basis, which type of processing the block performs. In most cases, you specify the processing mode by setting the **Input processing** parameter. When you are using System objects in MATLAB, only frame-based processing is available. The following table shows the common parameter settings you can use to perform sample- and frame-based processing in MATLAB and Simulink.

	Sample-Based Processing	Frame-Based Processing
Simulink — Blocks	**Input processing** = `Elements as channels (sample based)`	**Input processing** = `Columns as channels (frame based)`

What Is Sample-Based Processing?

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.

When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an *M*-by-*N* matrix as multichannel signal with *M*N* independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

What Is Frame-Based Processing?

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.

3 samples per channel

2 channels per frame

When you configure a block to perform frame-based processing, the block interprets an *M*-by-1 vector as a single-channel signal containing *M* samples per frame. Similarly, the block interprets an *M*-by-*N* matrix as a multichannel signal with *N* independent channels and *M* samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.

Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the "Frame-based processing changes" section of the *DSP System Toolbox Release Notes*.

Benefits of Frame-Based Processing

Frame-based processing is an established method of accelerating both real-time systems and model simulations.

Accelerate Real-Time Systems

Frame-based data is a common format in real-time systems. Data acquisition hardware often operates by accumulating a large number of signal samples at a high rate, and then propagating those samples to the real-time system as a block of data. This type of propagation maximizes the efficiency of the system by distributing the fixed process overhead across many samples; the faster data acquisition is suspended by slower interrupt processes after each frame is acquired, rather than after each individual sample is acquired.

The following figure illustrates how frame-based processing increases throughput. The thin blocks each represent the time elapsed during acquisition of a sample. The thicker blocks each represent the time elapsed during the interrupt service routine (ISR) that reads the data from the hardware.

In this example, the frame-based operation acquires a frame of 16 samples between each ISR. Thus, the frame-based throughput rate is many times higher than the sample-based alternative.

Sample-based operation
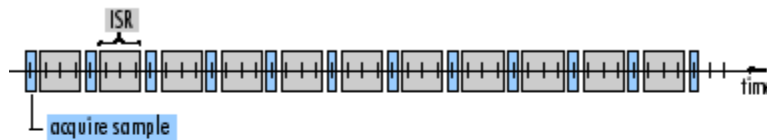
Frame-based operation

Be aware that frame-based processing introduces a certain amount of latency into a process due to the inherent lag in buffering the initial frame. In many instances,

however, you can select frame sizes that improve throughput without creating unacceptable latencies. For more information, see "Delay and Latency" on page 3-60.

Accelerate Model Simulations

The simulation of your model also benefits from frame-based processing. In this case, you reduce the overhead of block-to-block communications by propagating frames of data rather than individual samples.

More About

- "Inspect Sample and Frame Rates in Simulink" on page 3-8
- "Convert Sample and Frame Rates in Simulink" on page 3-19

Inspect Sample and Frame Rates in Simulink

In this section...
"Sample Rate and Frame Rate Concepts" on page 3-8
"Inspect Signals Using the Probe Block" on page 3-9
"Inspect Signals Using Color Coding" on page 3-13

Sample Rate and Frame Rate Concepts

Sample rates and frame rates are important issues in most signal processing models. This is especially true with systems that incorporate rate conversions. Fortunately, in most cases when you build a Simulink model, you only need to set sample rates for the source blocks. Simulink automatically computes the appropriate sample rates for the blocks that are connected to the source blocks. Nevertheless, it is important to become familiar with the sample rate and frame rate concepts as they apply to Simulink models.

The *input frame period* (T_{fi}) of a frame signal is the time interval between consecutive vector or matrix inputs to a block. Similarly, the *output frame period* (T_{fo}) is the time interval at which the block updates the frame vector or matrix value at the output port.

In contrast, the sample period, T_s, is the time interval between individual samples in a frame, this value is shorter than the frame period when the frame size is greater than 1. The sample period of a frame signal is the quotient of the frame period and the frame size, M:

$$T_s = T_f \ / \ M$$

More specifically, the sample periods of inputs (T_{si}) and outputs (T_{so}) are related to their respective frame periods by

$$T_{si} = T_{fi} \ / \ M_i$$

$$T_{so} = T_{fo} \ / \ M_o$$

where M_i and M_o are the input and output frame sizes, respectively.

The illustration below shows a single-channel, frame signal with a frame size (M_i) of 4 and a frame period (T_{fi}) of 1. The sample period, T_{si}, is therefore 1/4, or 0.25 second.
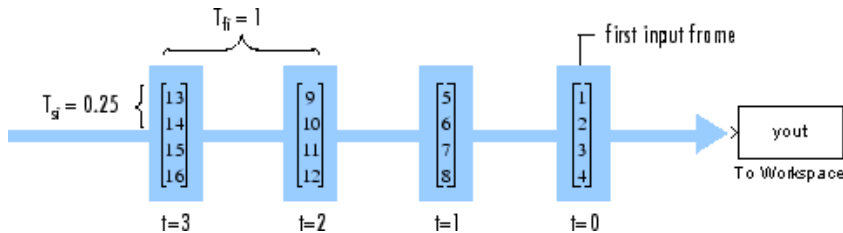
The frame rate of a signal is the reciprocal of the frame period. For instance, the input frame rate would be $1/T_{fi}$. Similarly, the output frame rate would be $1/T_{fo}$.

The sample rate of a signal is the reciprocal of the sample period. For instance, the sample rate would be $1/T_s$.

In most cases, the sequence sample period T_{si} is most important, while the frame rate is simply a consequence of the frame size that you choose for the signal. For a sequence with a given sample period, a larger frame size corresponds to a slower frame rate, and vice versa.

The block decides whether to process the signal one sample at a time or one frame at a time depending on the settings in the block dialog box. For example, a `Biquad` filter block with **Input processing** parameter set to `Columns as channels (frame based)` treats a 3-by-2 input signal as a two-frame signal with three samples in each frame. If **Input processing** parameter is set to `Elements as channels (sample based)`, the 3-by-2 input signal is treated as a six-channel signal with one sample in each channel.

Inspect Signals Using the Probe Block

You can use the `Probe` block to display the sample period or the frame period of a signal. The Probe block displays the label `Ts`, the sample period or frame period of the sequence, followed by a two-element vector. The left element is the period of the signal being measured. The right element is the signal's sample time offset, which is usually 0.

Note Simulink offers the ability to shift the sample time of a signal by an arbitrary value, which is equivalent to shifting the signal's phase by a fractional sample period. However, sample-time offsets are rarely used in signal processing systems, and DSP System Toolbox blocks do not support them.

Display the Sample Period of a Signal Using the Probe Block

1 At the MATLAB command prompt, type `ex_probe_tut1`.

The Probe Example 1 model opens. Double-click the `Signal From Workspace` block. Note that the **Samples per frame** parameter is set to 1.
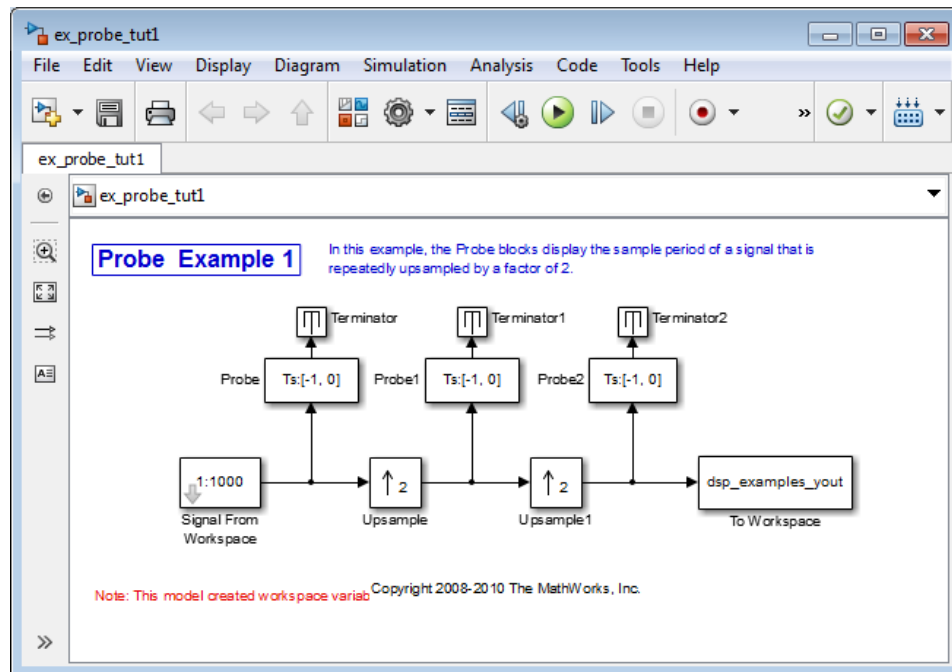
2 Run the model.

The figure below illustrates how the Probe blocks display the sample period of the signal before and after each upsample operation.
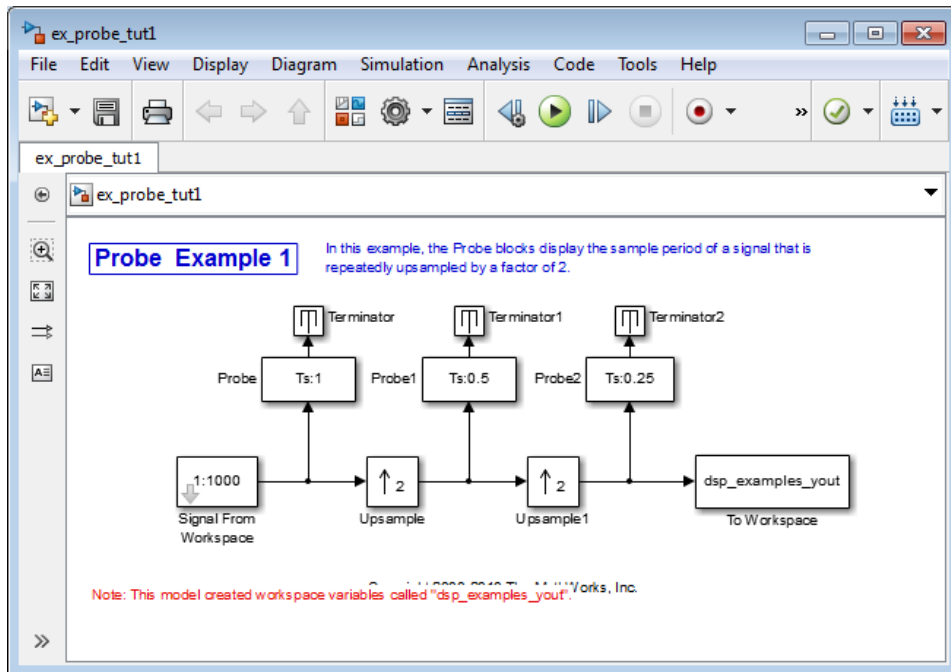
As displayed by the Probe blocks, the output from the Signal From Workspace block is a signal with a sample period of 1 second. The output from the first Upsample block has a sample period of 0.5 second, and the output from the second Upsample block has a sample period of 0.25 second.

Display the Frame Period of a Signal Using the Probe Block

1 At the MATLAB command prompt, type `ex_probe_tut2`.

The Probe Example 2 model opens. Double-click the `Signal From Workspace` block. Note that the **Samples per frame** parameter is set to `16`. Each frame in the signal contains `16` samples.
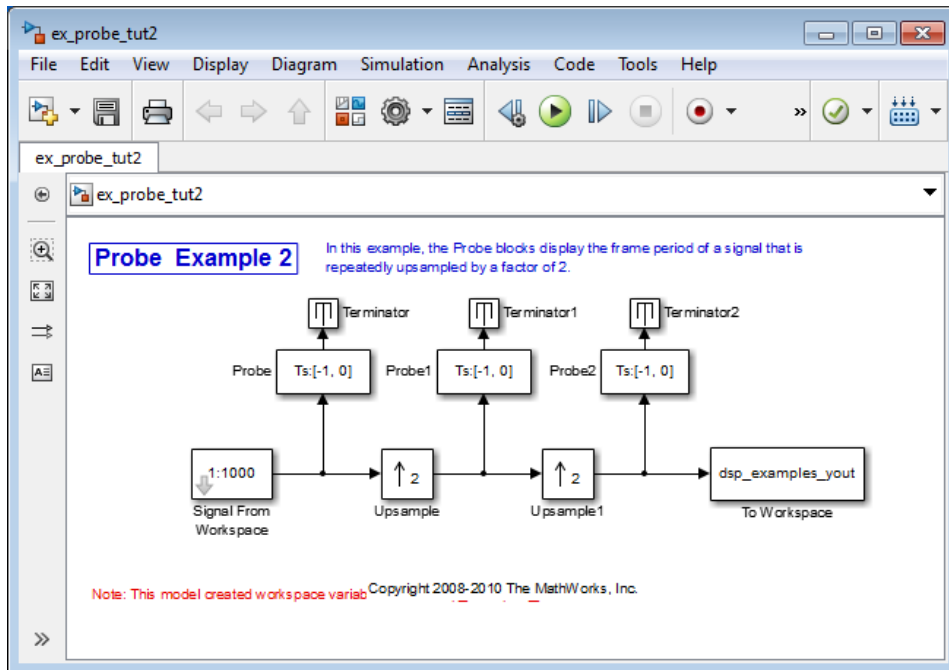
2   Run the model.

The figure below illustrates how the Probe blocks display the frame period of the signal before and after each upsample operation.
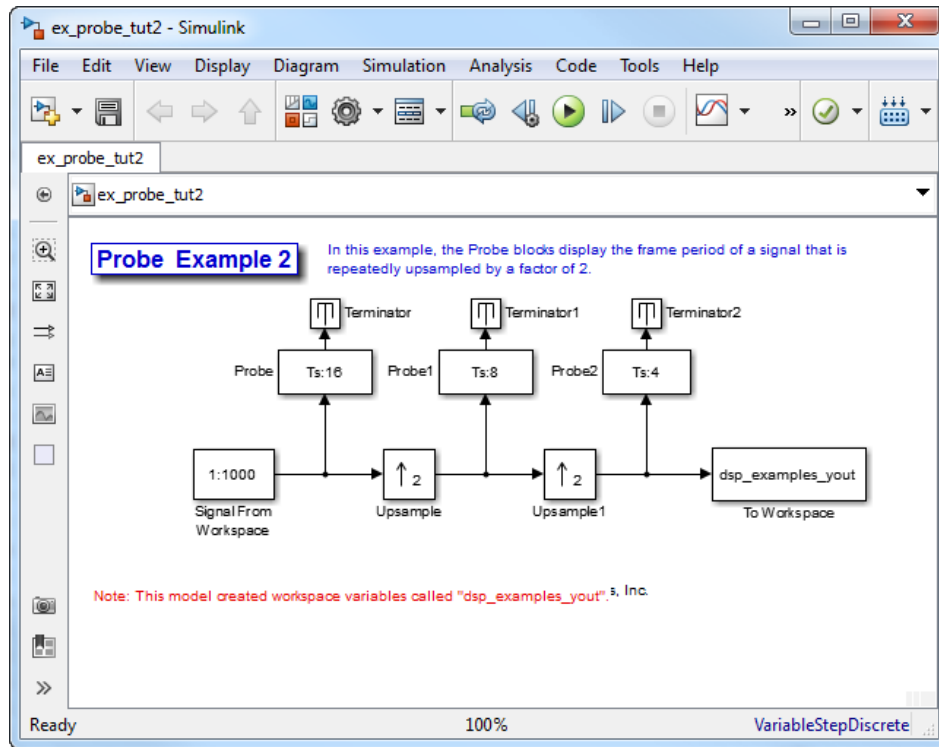
As displayed by the Probe blocks, the output from the Signal From Workspace block has a frame period of 16 seconds. The output from the first Upsample block has a frame period of 8 seconds, and the output from the second Upsample block has a frame period of 4 seconds.

Note that the sample rate conversion is implemented through a change in the frame period rather than the frame size.

Inspect Signals Using Color Coding

View the Sample Rate of a Signal Using the Sample Time Color Coding

1   At the MATLAB command prompt, type ex_color_tut1.

The Sample Time Color Example 1 model opens. Double-click the `Signal From Workspace` block. Note that the **Samples per frame** parameter is set to `1`.
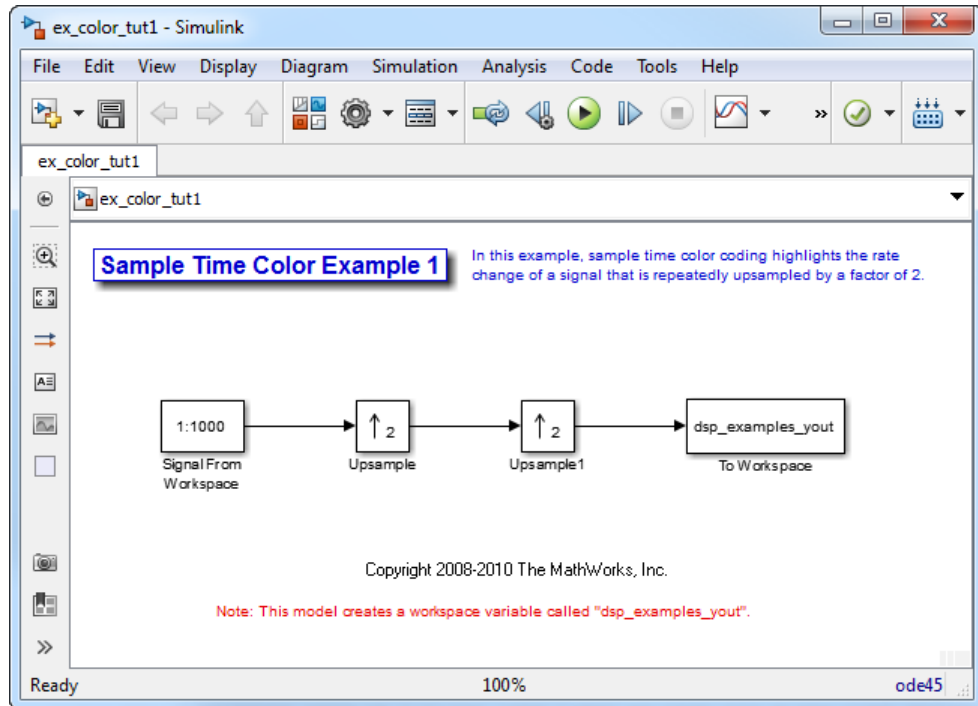
2   From the **Display** menu, point to **Sample Time**, and select **Colors**.

This selection turns on sample time color coding. Simulink now assigns each sample rate a different color.

3   Run the model.

The model should now look similar to the following figure:
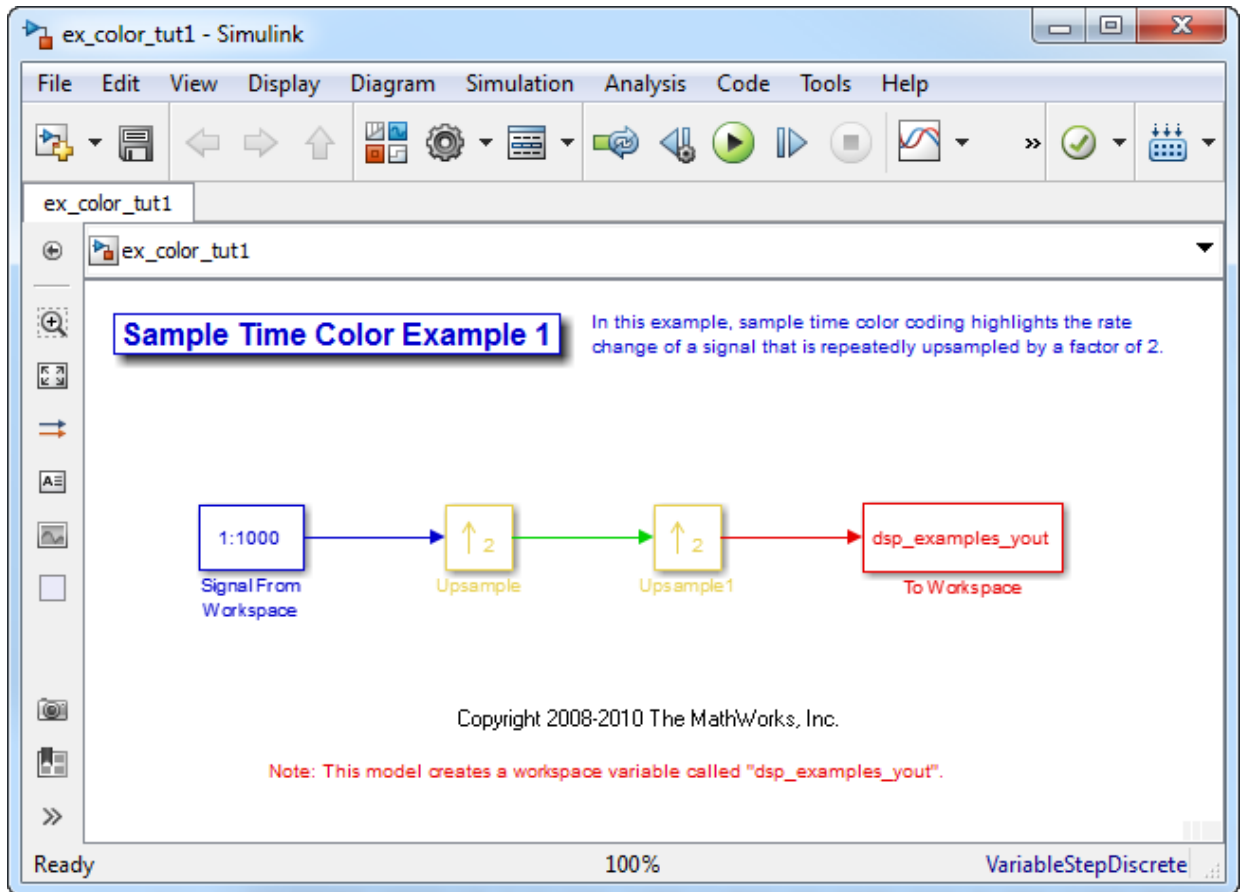
Every signal in this model has a different sample rate. Therefore, each signal is assigned a different color.

View the Frame Rate of a Signal Using the Sample Time Color Coding

1   At the MATLAB command prompt, type `ex_color_tut2`.

The Sample Time Color Example 2 model opens. Double-click the `Signal From Workspace` block. Note that the **Samples per frame** parameter is set to `16`. Each frame in the signal contains `16` samples.
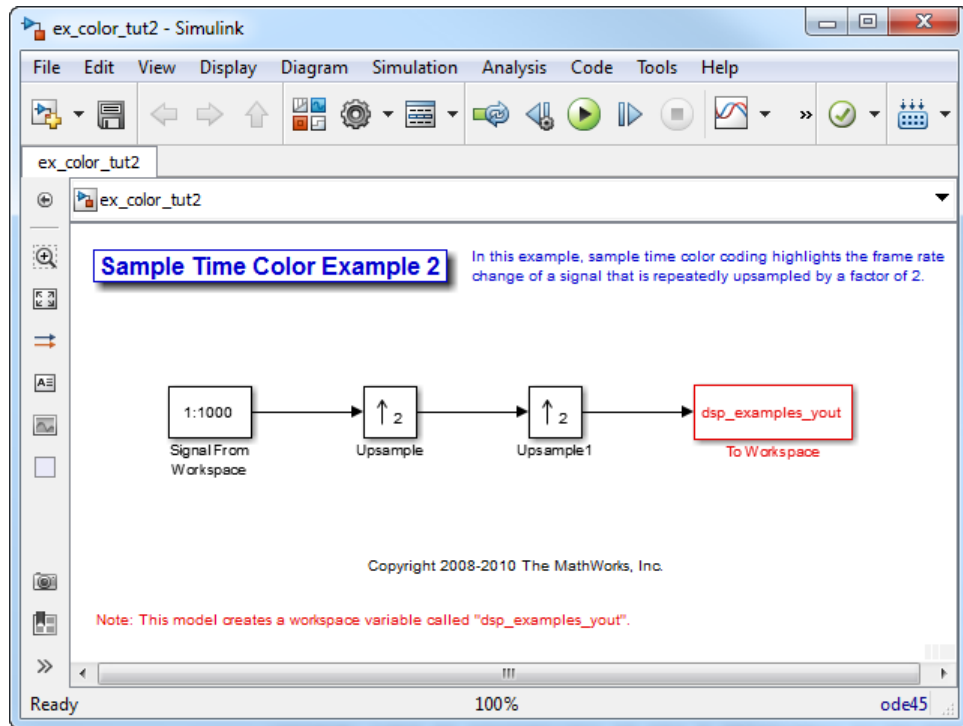
2 To turn on sample time color coding, from the **Display** menu, point to **Sample Time**, and select **Colors**.

Simulink now assigns each frame rate a different color.

3 Run the model.

The model should now look similar to the following figure:
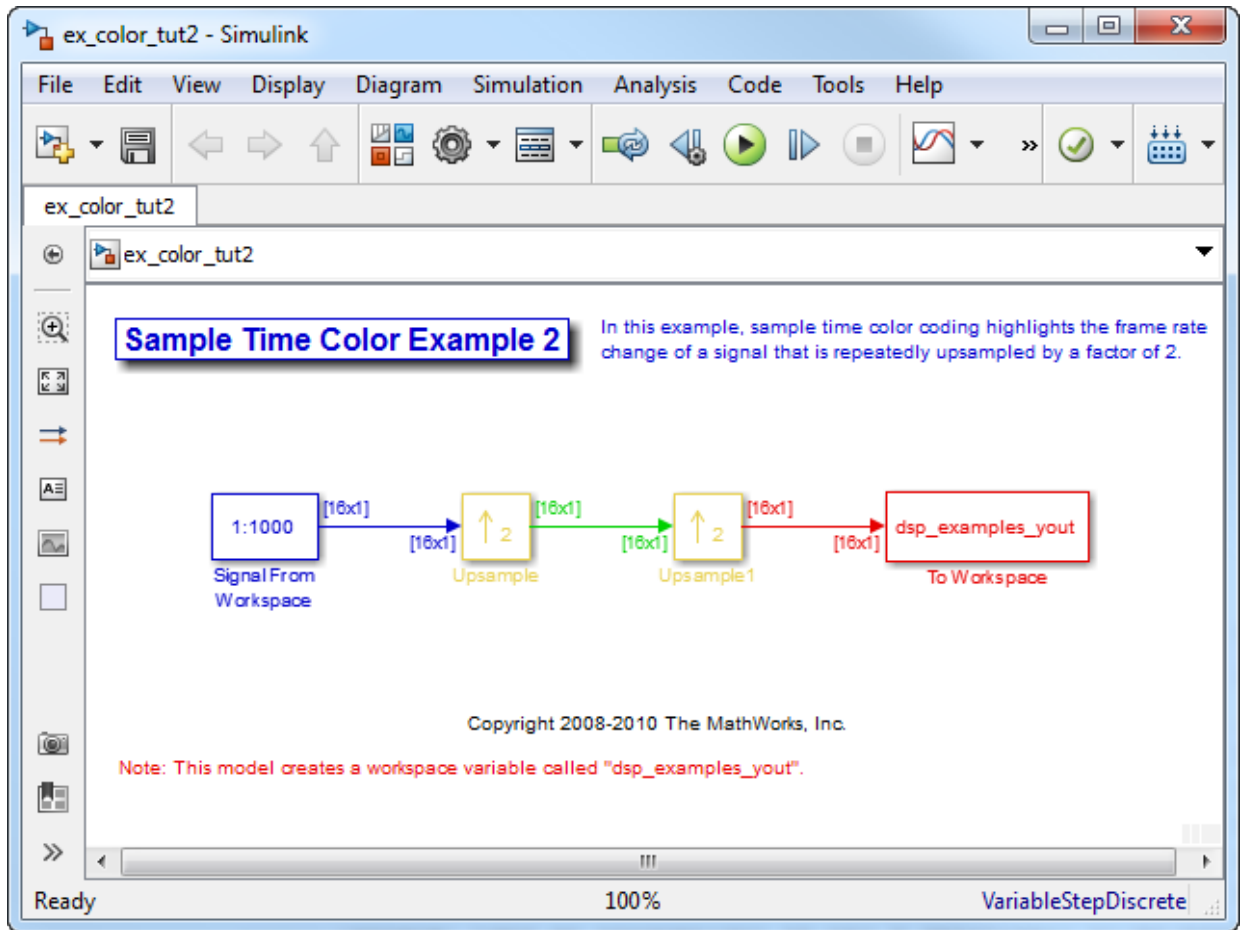
Because the **Rate options** parameter in the Upsample blocks is set to `Allow multirate processing`, each Upsample block changes the frame rate. Therefore, each frame signal in the model is assigned a different color.

4   Double-click on each Upsample block and change the **Rate options** parameter to `Enforce single-rate processing`.

5   Run the model.

Every signal is coded with the same color. Therefore, every signal in the model now has the same frame rate.
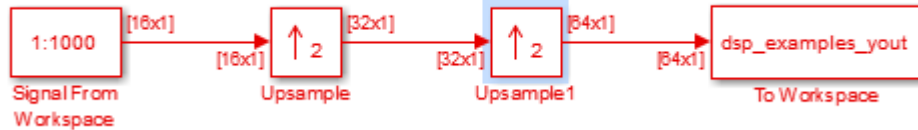
For more information about sample time color coding, see "View Sample Time Information" in the Simulink documentation.

More About

- "Convert Sample and Frame Rates in Simulink" on page 3-19
- "Sample- and Frame-Based Concepts" on page 3-2

Convert Sample and Frame Rates in Simulink

In this section...

Rate Conversion Blocks

There are two common types of operations that impact the frame and sample rates of a signal: direct rate conversion and frame rebuffering. Direct rate conversions, such as upsampling and downsampling, can be implemented by altering either the frame rate or the frame size of a signal. Frame rebuffering which is used to alter the frame size of a signal in order to improve simulation throughput, usually changes either the sample rate or frame rate of the signal as well.

The following table lists the principal rate conversion blocks in DSP System Toolbox software. Blocks marked with an asterisk (*) offer the option of changing the rate by either adjusting the frame size or frame rate.

Block	Library
Downsample *	Signal Operations
Dyadic Analysis Filter Bank	Filtering / Multirate Filters
Dyadic Synthesis Filter Bank	Filtering / Multirate Filters
FIR Decimation *	Filtering / Multirate Filters
FIR Interpolation *	Filtering / Multirate Filters
FIR Rate Conversion	Filtering / Multirate Filters
Repeat *	Signal Operations
Upsample *	Signal Operations

Direct Rate Conversion

Rate conversion blocks accept an input signal at one sample rate, and propagate the same signal at a new sample rate. Several of these blocks contain a **Rate options** parameter offering two options for multirate versus single-rate processing:

- `Enforce single-rate processing`: When you select this option, the block maintains the input sample rate.

- `Allow multirate processing`: When you select this option, the block downsamples the signal such that the output sample rate is K times slower than the input sample rate.

Note: When a Simulink model contains signals with various frame rates, the model is called *multirate*. You can find a discussion of multirate models in "Excess Algorithmic Delay (Tasking Latency)" on page 3-68. Also see "Time-Based Scheduling and Code Generation" in the Simulink Coder documentation.

Rate Conversion by Frame-Rate Adjustment

One way to change the sample rate of a signal, $1/T_{so}$, is to change the output frame rate ($T_{fo} \neq T_{fi}$), while keeping the frame size constant ($M_o = M_i$). Note that the sample rate of a signal is defined as $1/T_{so} = M_o/T_{fo}$:

1   At the MATLAB command prompt, type `ex_downsample_tut1`.

The Downsample Example T1 model opens.

2 From the **Display** menu, point to **Signals & Ports**, and select **Signal Dimensions**.

When you run the model, the dimensions of the signals appear next to the lines connecting the blocks.

3 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

4 Set the block parameters as follows:

- **Sample time** = `0.125`
- **Samples per frame** = `8`

Based on these parameters, the Signal From Workspace block outputs a signal with a sample period of 0.125 second and a frame size of 8.

5 Save these parameters and close the dialog box by clicking **OK**.

6 Double-click the Downsample block. The **Function Block Parameters: Downsample** dialog box opens.

7 Set the **Rate options** parameter to `Allow multirate processing`, and then click **OK**.

The Downsample block is configured to downsample the signal by changing the frame rate rather than the frame size.

8 Run the model.

After the simulation, the model should look similar to the following figure.
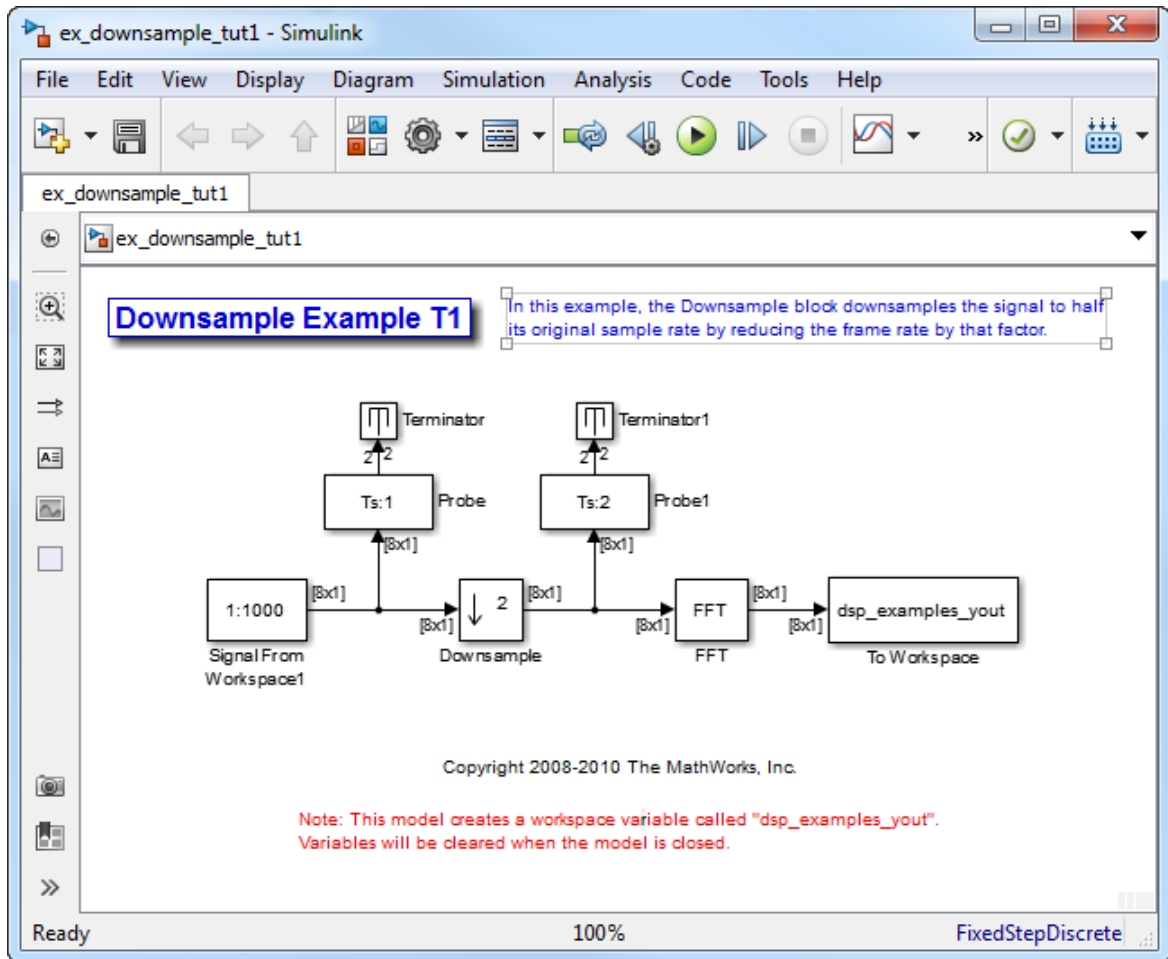
Because $T_{fi} = M_i \times T_{si}$, the input frame period, T_{fi}, is $T_{fi} = 8 \times 0.125 = 1$ second. This value is displayed by the first Probe block. Therefore the input frame rate, $1 / T_{fi}$, is also 1 frame per second.

The second Probe block in the model verifies that the output from the Downsample block has a frame period, T_{fo}, of 2 seconds, twice the frame period of the input.

However, because the frame rate of the output, $1/T_{fo}$, is 0.5 frames per second, the Downsample block actually downsampled the original signal to half its original rate. As a result, the output sample period, $T_{so} = T_{fo} / M_o$, is doubled to 0.25 second without any change to the frame size. The signal dimensions in the model confirm that the frame size did not change.

Rate Conversion by Frame-Size Adjustment

One way to change the sample rate of a signal is by changing the frame size (that is $M_o \neq M_i$), but keep the frame rate constant ($T_{fo} = T_{fi}$). Note that the sample rate of a signal is defined as $1/T_{so} = M_o/T_{fo}$:

1 At the MATLAB command prompt, type `ex_downsample_tut2`.

The Downsample Example T2 model opens.

2   From the **Display** menu, point to **Signals & Ports**, and select **Signal Dimensions**.

When you run the model, the dimensions of the signals appear next to the lines connecting the blocks.

3   Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

4   Set the block parameters as follows:

- **Sample time** = `0.125`
- **Samples per frame** = `8`

Based on these parameters, the Signal From Workspace block outputs a signal with a sample period of 0.125 second and a frame size of 8.

5 Save these parameters and close the dialog box by clicking **OK**.

6 Double-click the Downsample block. The **Function Block Parameters: Downsample** dialog box opens.

7 Set the **Rate options** parameter to `Enforce single-rate processing`, and then click **OK**.

The Downsample block is configured to downsample the signal by changing the frame size rather than the frame rate.

8 Run the model.

After the simulation, the model should look similar to the following figure.
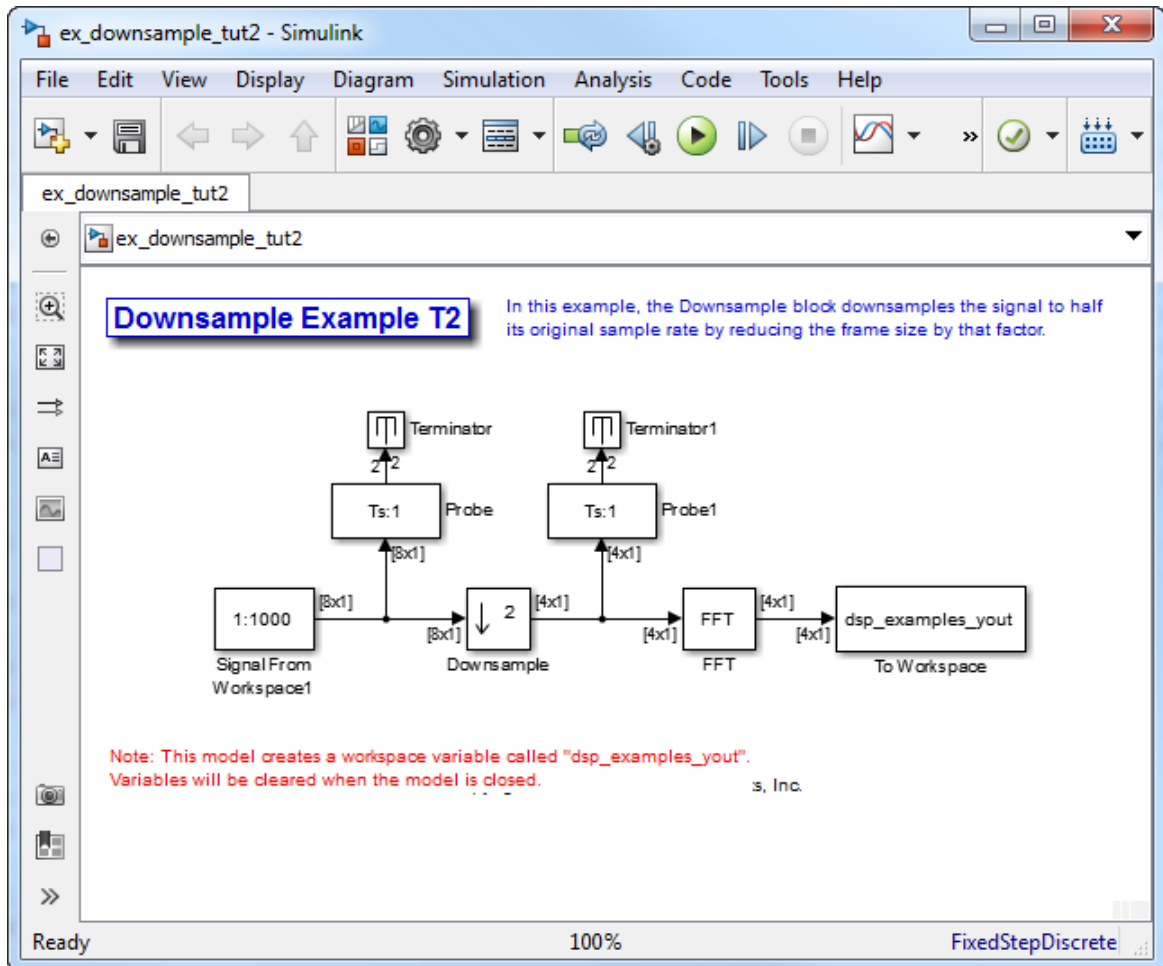
Because $T_{fi} = M_i \times T_{si}$, the input frame period, T_{fi}, is $T_{fi} = 8 \times 0.125 = 1$ second. This value is displayed by the first Probe block. Therefore the input frame rate, $1/T_{fi}$, is also 1 frame per second.

The Downsample block downsampled the input signal to half its original frame size. The signal dimensions of the output of the Downsample block confirm that the

downsampled output has a frame size of 4, half the frame size of the input. As a result, the sample period of the output, $T_{so} = T_{fo} / M_o$ is 0.25 second. This process occurred without any change to the frame rate ($T_{fi} = T_{fo}$).

Avoid Unintended Rate Conversion

It is important to be aware of where rate conversions occur in a model. In a few cases, unintentional rate conversions can produce misleading results:

1 At the MATLAB command prompt, type `ex_vectorscope_tut1`.

   The Vector Scope Example model opens.

2 Double-click the upper Sine Wave block. The **Source Block Parameters: Sine Wave** dialog box opens.

3 Set the block parameters as follows:

   - **Frequency (Hz)** = 1
   - **Sample time** = 0.1
   - **Samples per frame** = 128

   Based on the **Sample time** and the **Samples per frame** parameters, the Sine Wave outputs a sinusoid with a frame period of 128*0.1 or 12.8 seconds.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the lower Sine Wave block.

6 Set the block parameters as follows, and then click **OK**:

   - **Frequency (Hz)** = 2
   - **Sample time** = 0.1
   - **Samples per frame** = 128

   Based on the **Sample time** and the **Samples per frame** parameters, the Sine Wave outputs a sinusoid with a frame period of 128*0.1 or 12.8 seconds.

7 Double-click the Magnitude FFT block. The **Function Block Parameters: Magnitude FFT** dialog box opens.

8 Select the **Inherit FFT length from input dimensions** check box, and then click **OK**.

This setting instructs the block to use the input frame size (128) as the FFT length (which is also the output size).

9 Double-click the Vector Scope block. The **Sink Block Parameters: Vector Scope** dialog box opens.

10 Set the block parameters as follows, and then click **OK**:

- Click the **Scope Properties** tab.
- **Input domain** = Frequency
- Click the **Axis Properties** tab.
- **Minimum Y-limit** = -10
- **Maximum Y-limit** = 40

11 Run the model.

The model should now look similar to the following figure. Note that the signal leaving the Magnitude FFT block is 128-by-1.
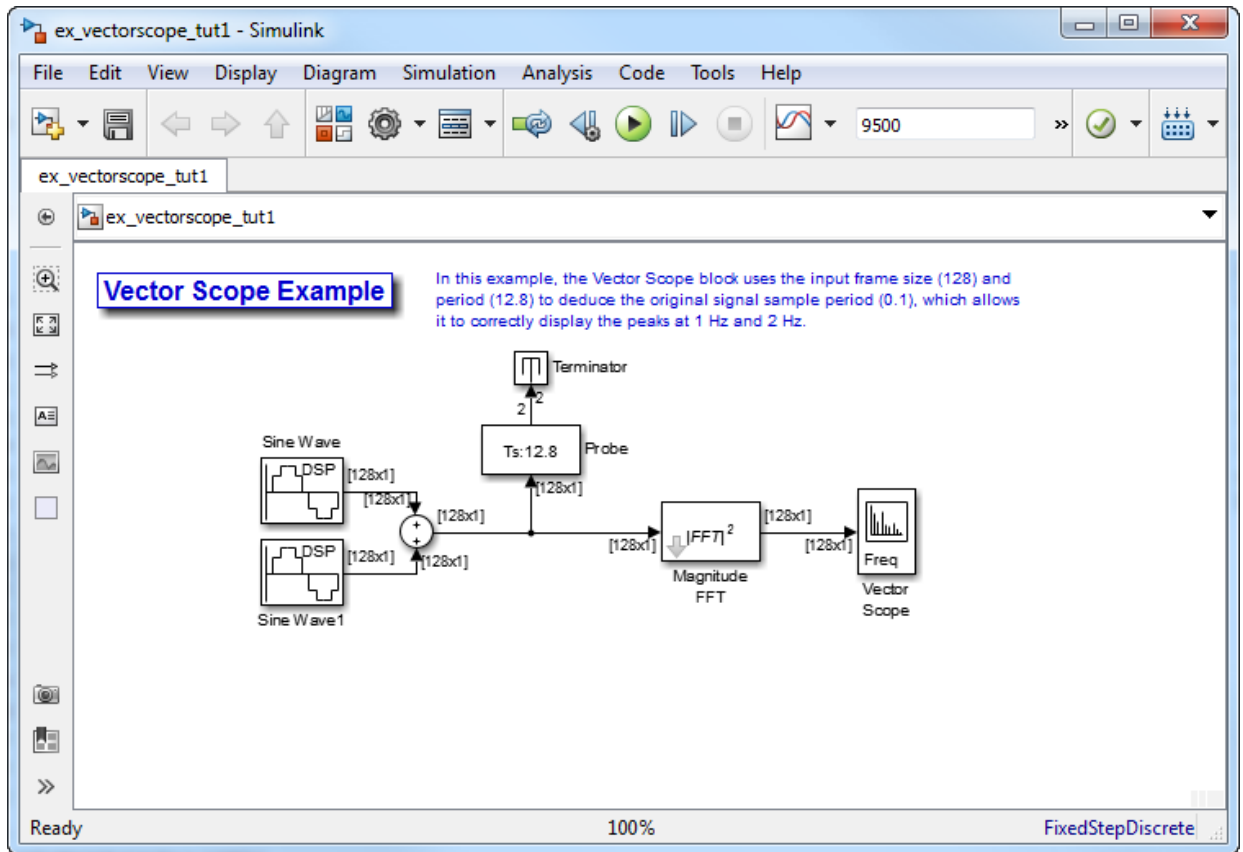
The **Vector Scope** window displays the magnitude FFT of a signal composed of two sine waves, with frequencies of 1 Hz and 2 Hz.
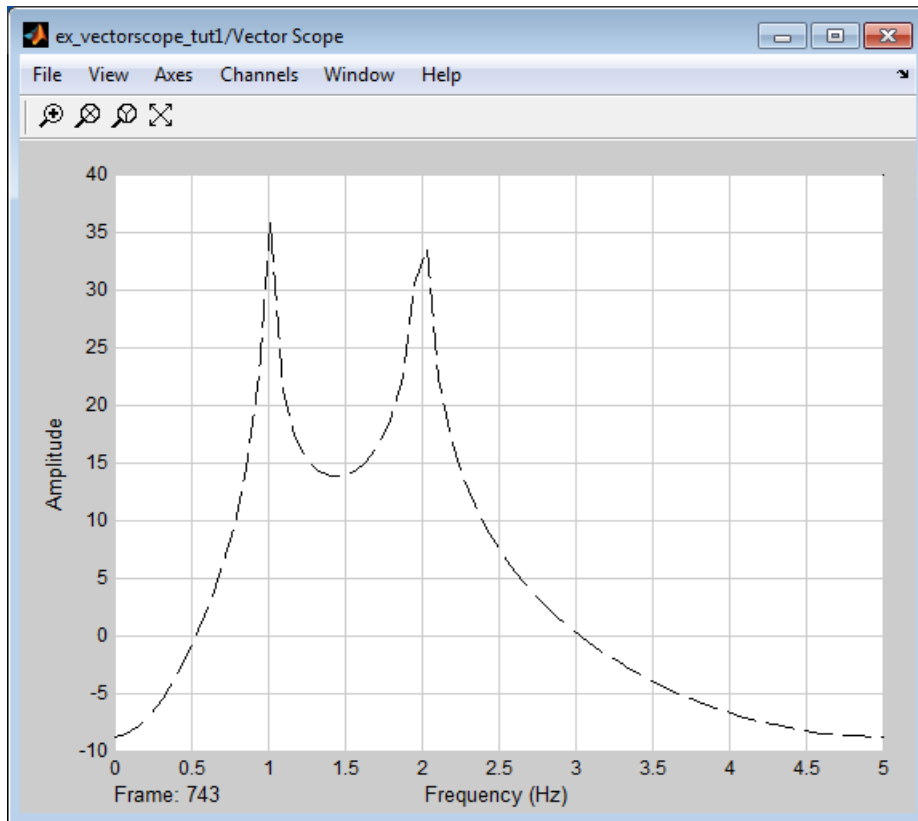
The Vector Scope block uses the input frame size (128) and period (12.8) to deduce the original signal's sample period (0.1), which allows it to correctly display the peaks at 1 Hz and 2 Hz.

12 Double-click the Magnitude FFT block. The **Function Block Parameters: Magnitude FFT** dialog box opens.

13 Set the block parameters as follows:

- Clear the **Inherit FFT length from input dimensions** check box.
- Set the **FFT length** parameter to 256.

Based on these parameters, the Magnitude FFT block zero-pads the length-128 input frame to a length of 256 before performing the FFT.

14 Run the model.

The model should now look similar to the following figure. Note that the signal leaving the Magnitude FFT block is 256-by-1.
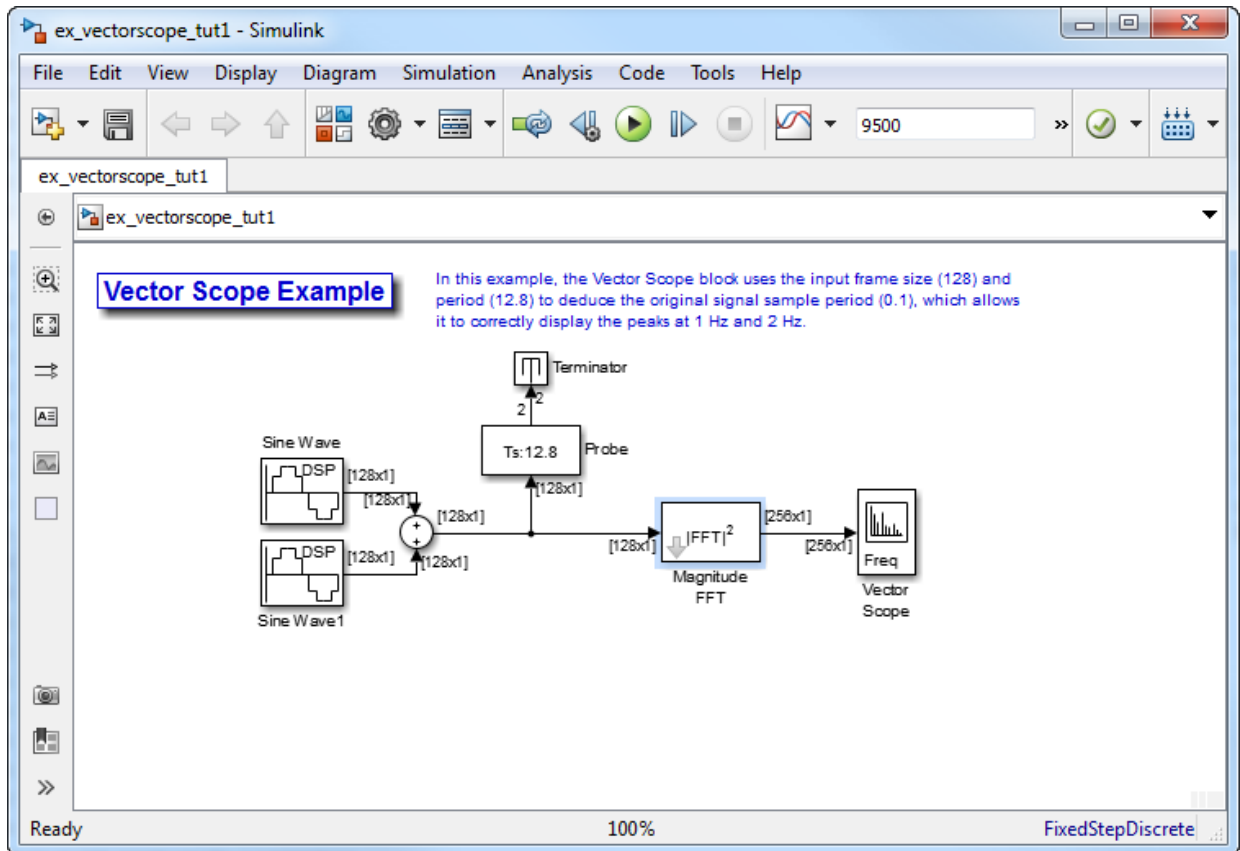
The **Vector Scope** window displays the magnitude FFT of a signal composed of two sine waves, with frequencies of 2 Hz and 4 Hz.
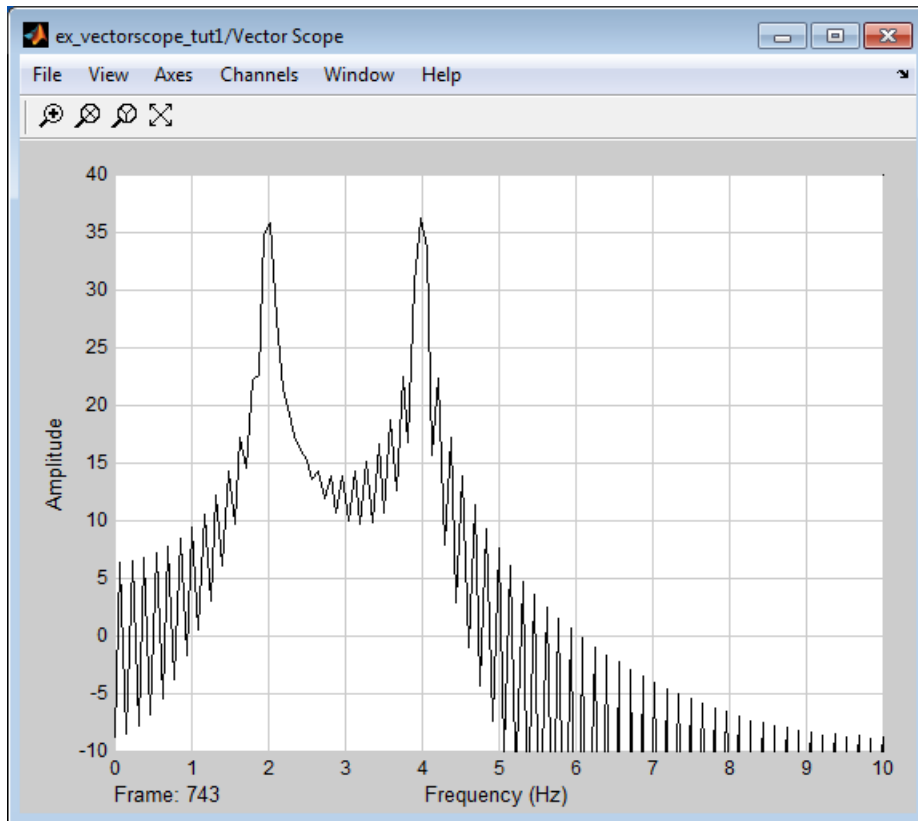
In this case, based on the input frame size (256) and frame period (12.8), the Vector Scope block incorrectly calculates the original signal's sample period to be (12.8/256) or 0.05 second. As a result, the spectral peaks appear incorrectly at 2 Hz and 4 Hz rather than 1 Hz and 2 Hz.

The source of the error described above is unintended rate conversion. The zero-pad operation performed by the Magnitude FFT block halves the sample period of the sequence by appending 128 zeros to each frame. To calculate the spectral peaks correctly, the Vector Scope block needs to know the sample period of the original signal.

15 To correct for the unintended rate conversion, double-click the Vector Scope block.

16 Set the block parameters as follows:

- Click the **Axis Properties** tab.
- Clear the **Inherit sample time from input** check box.
- Set the **Sample time of original time series** parameter to the actual sample period of `0.1`.

17 Run the model.

The Vector Scope block now accurately plots the spectral peaks at 1 Hz and 2 Hz.

In general, when you zero-pad or overlap buffers, you are changing the sample period of the signal. If you keep this in mind, you can anticipate and correct problems such as unintended rate conversion.

Frame Rebuffering Blocks

There are two common types of operations that impact the frame and sample rates of a signal: direct rate conversion and frame rebuffering. Direct rate conversions, such as upsampling and downsampling, can be implemented by altering either the frame rate or the frame size of a signal. Frame rebuffering, which is used alter the frame size of a signal in order to improve simulation throughput, usually changes either the sample rate or frame rate of the signal as well.

Sometimes you might need to rebuffer a signal to a new frame size at some point in a model. For example, your data acquisition hardware may internally buffer the sampled signal to a frame size that is not optimal for the signal processing algorithm in the model. In this case, you would want to rebuffer the signal to a frame size more appropriate for the intended operations without introducing any change to the data or sample rate.

The following table lists the principal DSP System Toolbox buffering blocks.

Block	Library
Buffer	Signal Management/ Buffers
Delay Line	Signal Management/ Buffers
Unbuffer	Signal Management/ Buffers
Variable Selector	Signal Management/ Indexing

Blocks for Frame Rebuffering with Preservation of the Signal

Buffering operations provide another mechanism for rate changes in signal processing models. The purpose of many buffering operations is to adjust the frame size of the signal, M, without altering the signal's sample rate T_s. This usually results in a change to the signal's frame rate, T_f, according to the following equation:

$$T_f = MT_s$$

However, the equation above is only true if no samples are added or deleted from the original signal. Therefore, the equation above does not apply to buffering operations that generate overlapping frames, that only partially unbuffer frames, or that alter the data sequence by adding or deleting samples.

There are two blocks in the Buffers library that can be used to change a signal's frame size without altering the signal itself:

- `Buffer` — redistributes signal samples to a larger or smaller frame size
- `Unbuffer` — unbuffers a signal with frame size M and frame period T_f to a signal with frame size `1` and frame period T_s

The Buffer block preserves the signal's data and sample period only when its **Buffer overlap** parameter is set to `0`. The output frame period, T_{fo}, is

$$T_{fo} = \frac{M_o T_{fi}}{M_i}$$

where T_{fi} is the input frame period, M_i is the input frame size, and M_o is the output frame size specified by the **Output buffer size (per channel)** parameter.

The Unbuffer block unbuffers a frame signal and always preserves the signal's data and sample period

$$T_{so} = T_{fi} / M_i$$

where T_{fi} and M_i are the period and size, respectively, of the frame signal.

Both the Buffer and Unbuffer blocks preserve the sample period of the sequence in the conversion ($T_{so} = T_{si}$).

Blocks for Frame Rebuffering with Alteration of the Signal

Some forms of buffering alter the signal's data or sample period in addition to adjusting the frame size. This type of buffering is desirable when you want to create sliding windows by overlapping consecutive frames of a signal, or select a subset of samples from each input frame for processing.

The blocks that alter a signal while adjusting its frame size are listed below. In this list, T_{si} is the input sequence sample period, and T_{fi} and T_{fo} are the input and output frame periods, respectively:

- The Buffer block adds duplicate samples to a sequence when the **Buffer overlap** parameter, L, is set to a nonzero value. The output frame period is related to the input sample period by

$$T_{fo} = (M_o - L)T_{si}$$

  where M_o is the output frame size specified by the **Output buffer size (per channel)** parameter. As a result, the new output sample period is

$$T_{so} = \frac{(M_o - L)T_{si}}{M_o}$$

- The Delay Line block adds duplicate samples to the sequence when the **Delay line size** parameter, M_o, is greater than 1. The output and input frame periods are the same, $T_{fo} = T_{fi} = T_{si}$, and the new output sample period is

$$T_{so} = \frac{T_{si}}{M_o}$$

- The Variable Selector block can remove, add, and/or rearrange samples in the input frame when **Select** is set to Rows. The output and input frame periods are the same, $T_{fo} = T_{fi}$, and the new output sample period is

$$T_{so} = \frac{M_i T_{si}}{M_o}$$

where M_o is the length of the block's output, determined by the **Elements** vector.

In all of these cases, the sample period of the output sequence is *not* equal to the sample period of the input sequence.

Buffer Signals by Preserving the Sample Period

In the following example, a signal with a sample period of 0.125 second is rebuffered from a frame size of 8 to a frame size of 16. This rebuffering process doubles the frame period from 1 to 2 seconds, but does not change the sample period of the signal ($T_{so} = T_{si} = 0.125$). The process also does not add or delete samples from the original signal:

1   At the MATLAB command prompt, type `ex_buffer_tut1`.

The Buffer Example T1 model opens.

2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the parameters as follows:

- **Signal** = 1:1000
- **Sample time** = 0.125
- **Samples per frame** = 8
- **Form output after final data value** = Setting to zero

Based on these parameters, the Signal from Workspace block outputs a signal with a sample period of 0.125 second. Each output frame contains eight samples.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.

6 Set the parameters as follows, and then click **OK**:

- **Output buffer size (per channel)** = 16
- **Buffer overlap** = 0
- **Initial conditions** = 0

Based on these parameters, the Buffer block rebuffers the signal from a frame size of 8 to a frame size of 16.

7 Run the model.

The following figure shows the model after simulation.
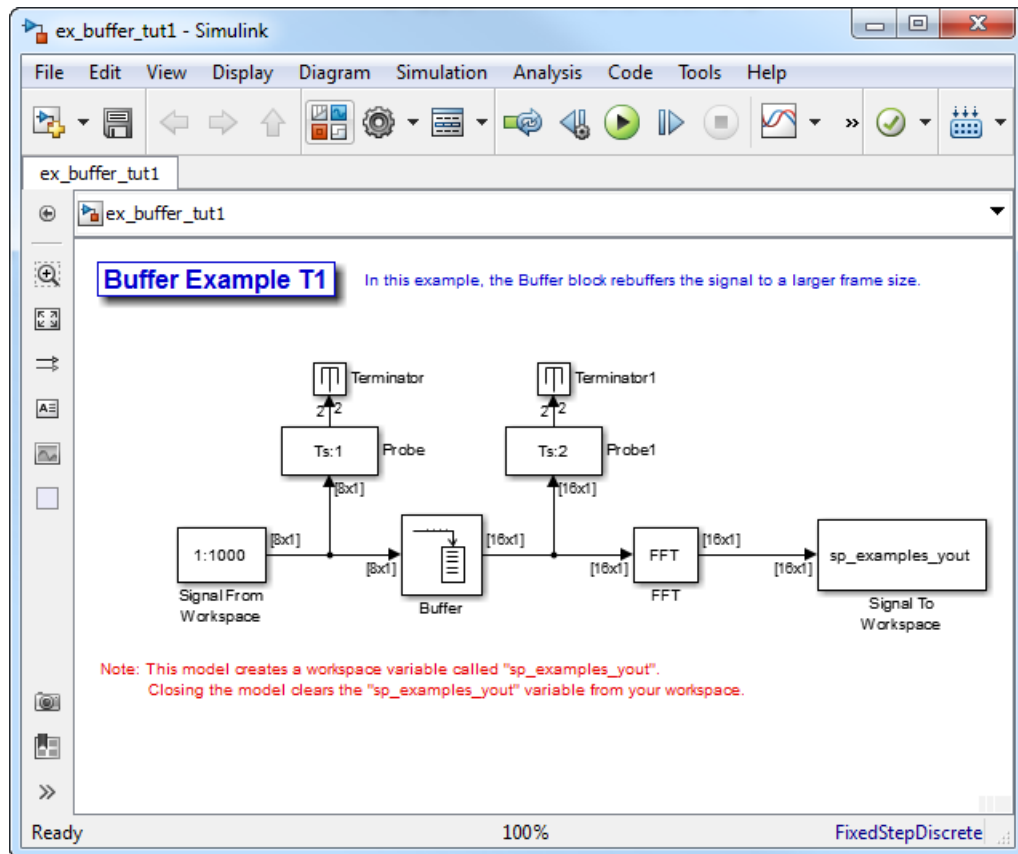
Note that the input to the Buffer block has a frame size of 8 and the output of the block has a frame size of 16. As shown by the Probe blocks, the rebuffering process doubles the frame period from 1 to 2 seconds.

Buffer Signals by Altering the Sample Period

Some forms of buffering alter the signal's data or sample period in addition to adjusting the frame size. In the following example, a signal with a sample period of 0.125 second is rebuffered from a frame size of 8 to a frame size of 16 with a buffer overlap of 4:

1 At the MATLAB command prompt, type `ex_buffer_tut2`.

The Buffer Example T2 model opens.

2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the parameters as follows:

- **Signal** = 1:1000
- **Sample time** = 0.125
- **Samples per frame** = 8
- **Form output after final data value** = Setting to zero

Based on these parameters, the Signal from Workspace block outputs a signal with a sample period of 0.125 second. Each output frame contains eight samples.

4   Save these parameters and close the dialog box by clicking **OK**.

5   Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.

6   Set the parameters as follows, and then click **OK**:

- **Output buffer size (per channel)** = 16
- **Buffer overlap** = 4
- **Initial conditions** = 0

Based on these parameters, the Buffer block rebuffers the signal from a frame size of 8 to a frame size of 16. Also, after the initial output, the first four samples of each output frame are made up of the last four samples from the previous output frame.

7   Run the model.

The following figure shows the model after the simulation has stopped.
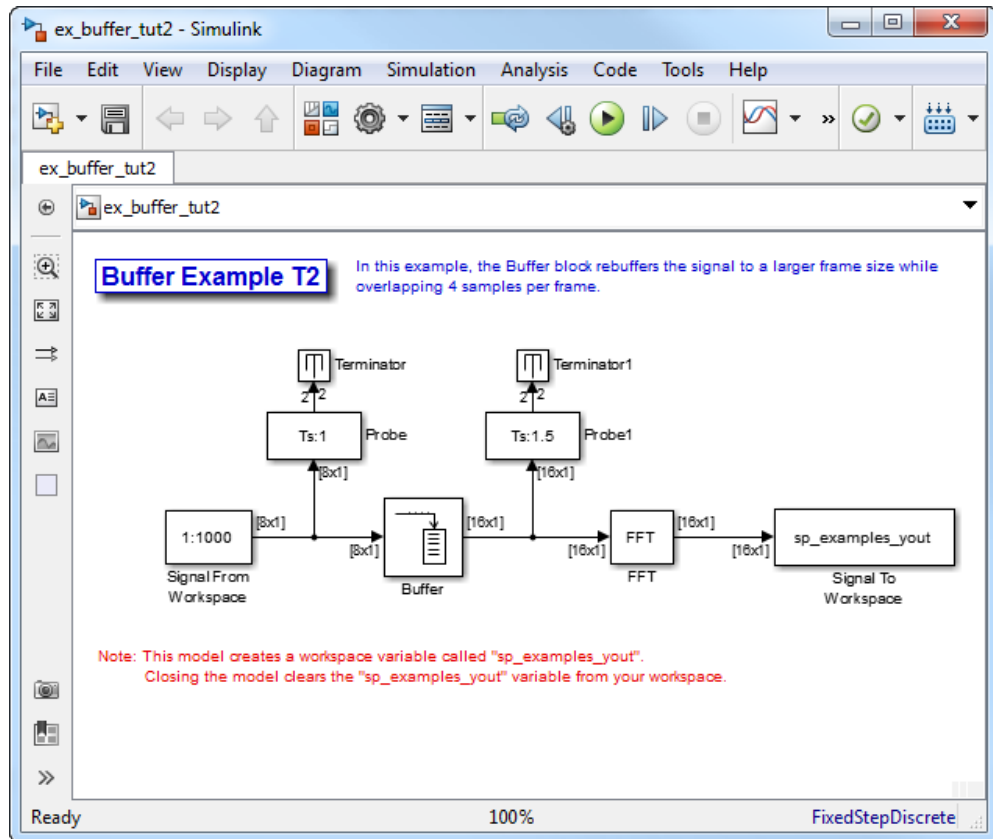
Note that the input to the Buffer block has a frame size of 8 and the output of the block has a frame size of 16. The relation for the output frame period for the Buffer block is

$$T_{fo} = (M_o - L)T_{si}$$

T_{fo} is (16-4)*0.125, or 1.5 seconds, as confirmed by the second **Probe** block. The sample period of the signal at the output of the Buffer block is no longer 0.125 second. It is now $T_{so} = T_{fo} / M_o = 1.5 / 16 = 0.0938$ second. Thus, both the signal's data and the signal's sample period have been altered by the buffering operation.

More About

Buffering and Frame-Based Processing

In this section...
"Buffer Input into Frames" on page 3-45
"Buffer Signals into Frames with Overlap" on page 3-48
"Buffer Frame Inputs into Other Frame Inputs" on page 3-52
"Buffer Delay and Initial Conditions" on page 3-55
"Unbuffer Frame Signals into Sample Signals" on page 3-55

Buffer Input into Frames

Multichannel signals of frame size 1 can be buffered into multichannel signals of frame size L using the Buffer block. L is greater than 1.

The following figure is a graphical representation of a signal with frame size 1 being converted into a signal of frame size L by the Buffer block.
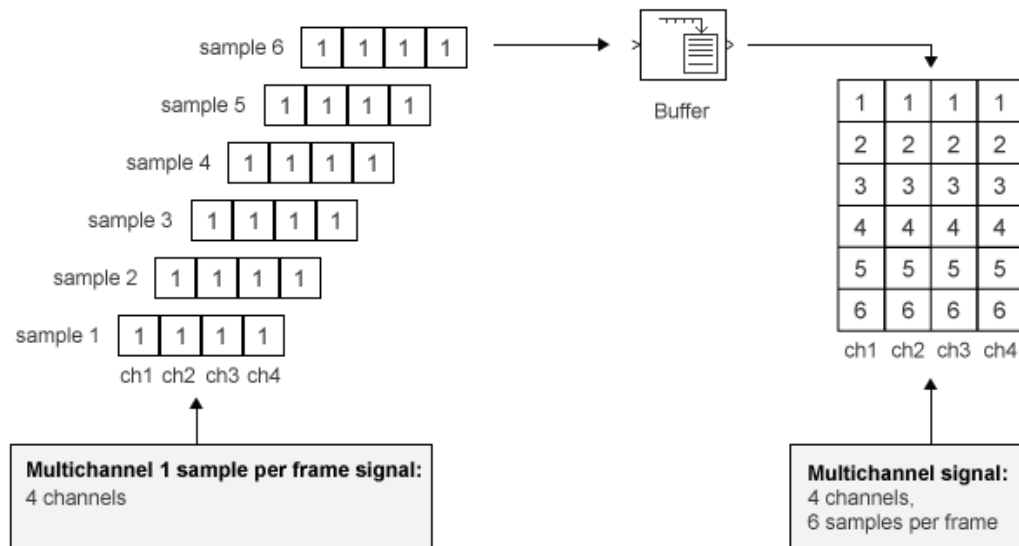
In the following example, a two-channel 1 sample per frame signal is buffered into a two-channel 1 sample per frame signal using a Buffer block:

1   At the MATLAB command prompt, type `ex_buffer_tut`.

     The Buffer Example model opens.
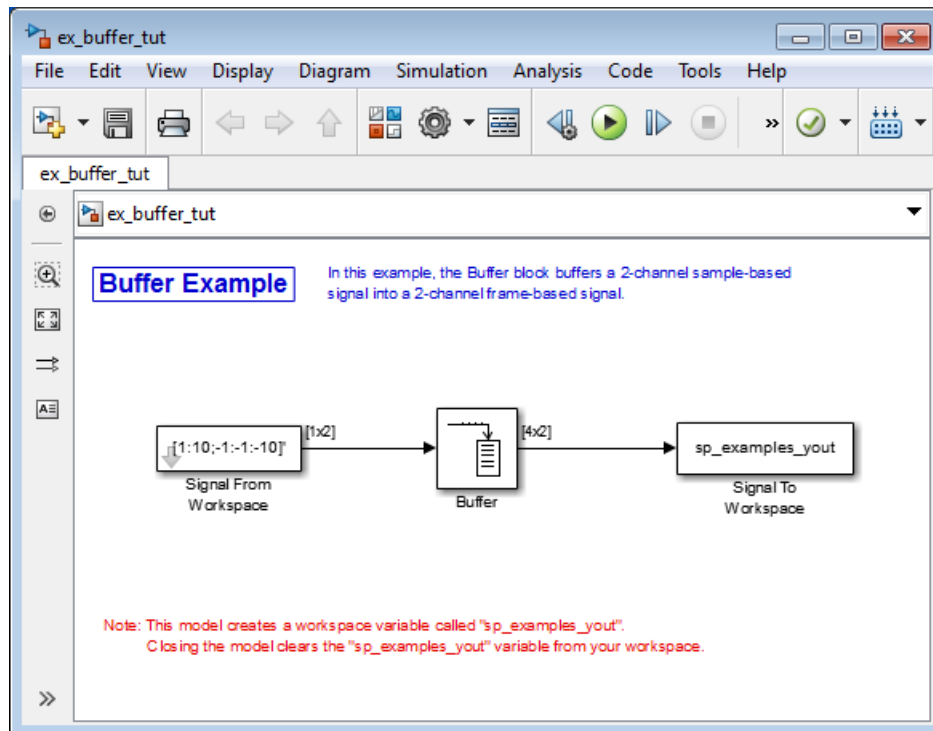
2   Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3   Set the parameters as follows:

     - **Signal** = `[1:10;-1:-1:-10]'`
     - **Sample time** = `1`
     - **Samples per frame** = `1`
     - **Form output after final data value** = `Setting to zero`

     Based on these parameters, the Signal from Workspace block outputs a signal with a frame length of 1 and a sample period of 1 second. Because you set the **Samples per**

frame parameter setting to 1, the Signal From Workspace block outputs one two-channel sample at each sample time.

4   Save these parameters and close the dialog box by clicking **OK**.

5   Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.

6   Set the parameters as follows:

   · **Output buffer size (per channel)** = 4

   · **Buffer overlap** = 0

   · **Initial conditions** = 0

   Because you set the **Output buffer size** parameter to 4, the Buffer block outputs a frame signal with frame size 4.

7   Run the model.

   The figure below is a graphical interpretation of the model behavior during simulation.
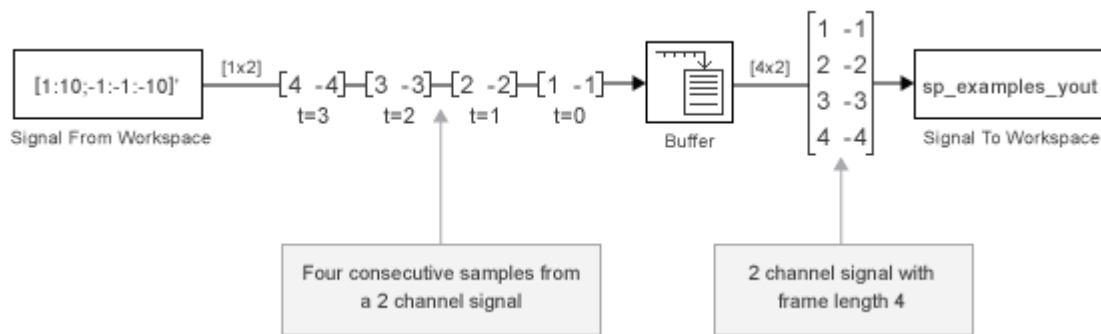
Note: Alternatively, you can set the **Samples per frame** parameter of the Signal From Workspace block to 4 and create the same signal shown above without using a Buffer block. The Signal From Workspace block performs the buffering internally, in order to output a two-channel frame.

Buffer Signals into Frames with Overlap

In some cases it is useful to work with data that represents overlapping sections of an original signal. For example, in estimating the power spectrum of a signal, it is often desirable to compute the FFT of overlapping sections of data. Overlapping buffers are also needed in computing statistics on a sliding window, or for adaptive filtering.

The **Buffer overlap** parameter of the Buffer block specifies the number of overlap points, L. In the overlap case ($L > 0$), the frame period for the output is $(M_o\text{-}L)*T_{si}$, where T_{si} is the input sample period and M_o is the **Buffer size**.

Note: Set the **Buffer overlap** parameter to a negative value to achieve output frame rates *slower* than in the nonoverlapping case. The output frame period is still $T_{si}*(M_o\text{-}L)$, but now with $L < 0$. Only the M_o newest inputs are included in the output buffers. The previous L inputs are discarded.

In the following example, a four-channel signal with frame length 1 and sample period 1 is buffered to a signal with frame size 3 and frame period 2. Because of the buffer overlap, the input sample period is not conserved, and the output sample period is 2/3:

1   At the MATLAB command prompt, type `ex_buffer_tut3`.

The Buffer Example T3 model opens.
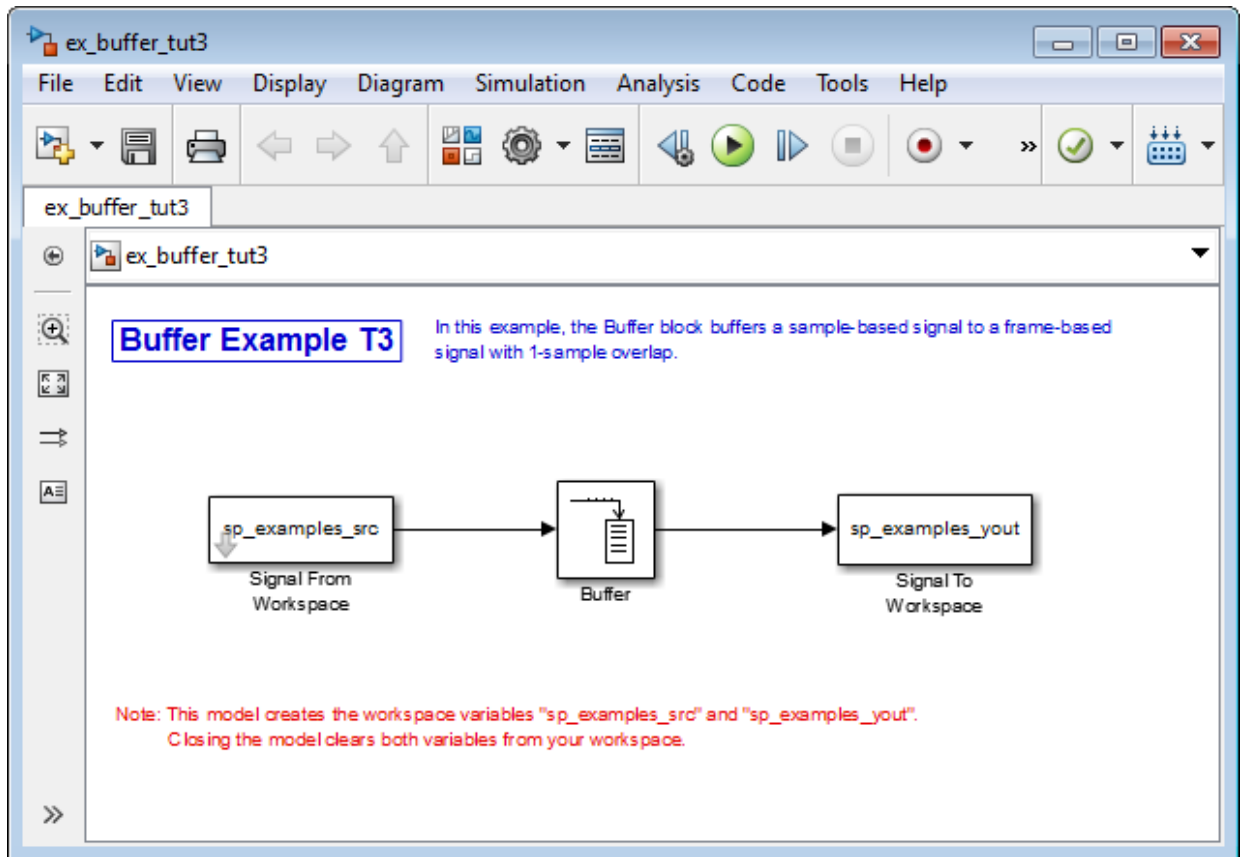
Also, the variable `sp_examples_src` is loaded into the MATLAB workspace. This variable is defined as follows:

```
sp_examples_src=[1 1 5 -1; 2 1 5 -2; 3 0 5 -3; 4 0 5 -4; 5 1 5 -5; 6 1 5 -6];
```

2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the block parameters as follows:

- **Signal** = sp_examples_src
- **Sample time** = 1
- **Samples per frame** = 1

• **Form output after final data value by** = `Setting to zero`

Based on these parameters, the Signal from Workspace block outputs a signal with a sample period of 1 second. Because you set the **Samples per frame** parameter setting to 1, the Signal From Workspace block outputs one four-channel sample at each sample time.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.

6 Set the block parameters as follows, and then click **OK**:

• **Output buffer size (per channel)** = 3

• **Buffer overlap** = 1

• **Initial conditions** = 0

Because you set the **Output buffer size** parameter to 3, the Buffer block outputs a signal with frame size 3. Also, because you set the **Buffer overlap** parameter to 1, the last sample from the previous output frame is the first sample in the next output frame.

7 Run the model.

The following figure is a graphical interpretation of the model's behavior during simulation.
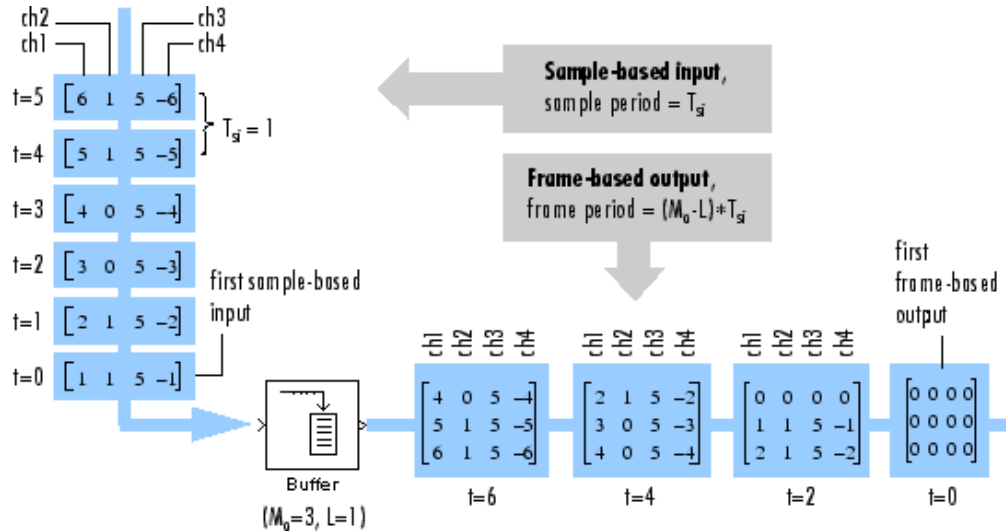
8 At the MATLAB command prompt, type `sp_examples_yout`.

The following is displayed in the MATLAB Command Window.

```
sp_examples_yout =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     1     1     5    -1
     2     1     5    -2
     2     1     5    -2
     3     0     5    -3
     4     0     5    -4
     4     0     5    -4
     5     1     5    -5
     6     1     5    -6
     6     1     5    -6
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

Notice that the inputs do not begin appearing at the output until the fifth row, the second row of the second frame. This is due to the block's latency.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-68 for general information about algorithmic delay. For instructions on how to calculate buffering delay, see "Buffer Delay and Initial Conditions" on page 3-55.

Buffer Frame Inputs into Other Frame Inputs

In the following example, a two-channel signal with frame size 4 is rebuffered to a signal with frame size 3 and frame period 2. Because of the overlap, the input sample period is not conserved, and the output sample period is 2/3:

1 At the MATLAB command prompt, type `ex_buffer_tut4`.

The Buffer Example T4 model opens.
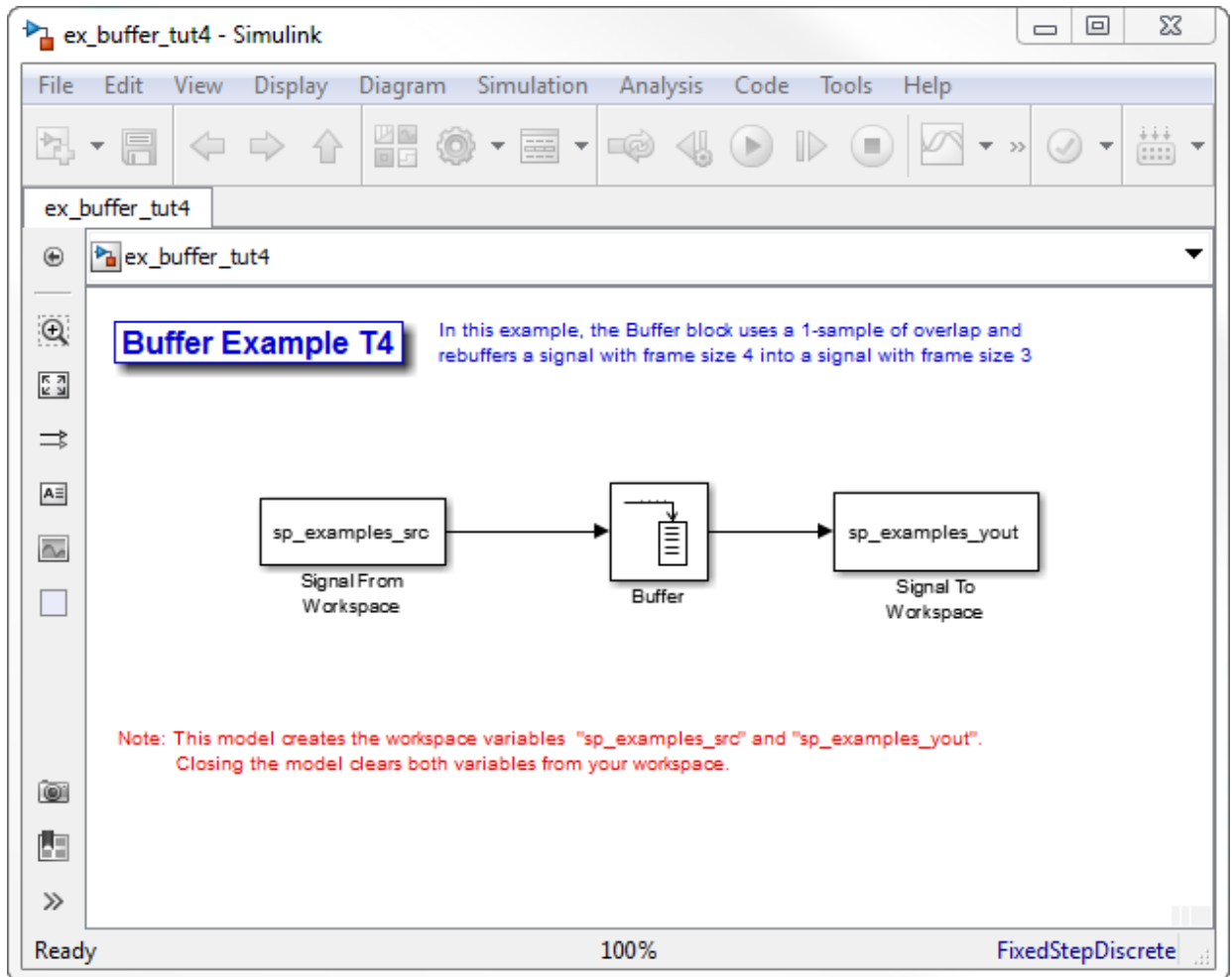
Also, the variable `sp_examples_src` is loaded into the MATLAB workspace. This variable is defined as

`sp_examples_src = [1 1; 2 1; 3 0; 4 0; 5 1; 6 1; 7 0; 8 0]`

2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the block parameters as follows:

- **Signal** = `sp_examples_src`
- **Sample time** = 1
- **Samples per frame** = 4

Based on these parameters, the Signal From Workspace block outputs a two-channel frame signal with a sample period of 1 second and a frame size of 4.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.

6 Set the block parameters as follows, and then click **OK**:

- **Output buffer size (per channel)** = 3
- **Buffer overlap** = 1
- **Initial conditions** = 0

Based on these parameters, the Buffer block outputs a two-channel frame signal with a frame size of 3.

7 Run the model.

The following figure is a graphical representation of the model's behavior during simulation.
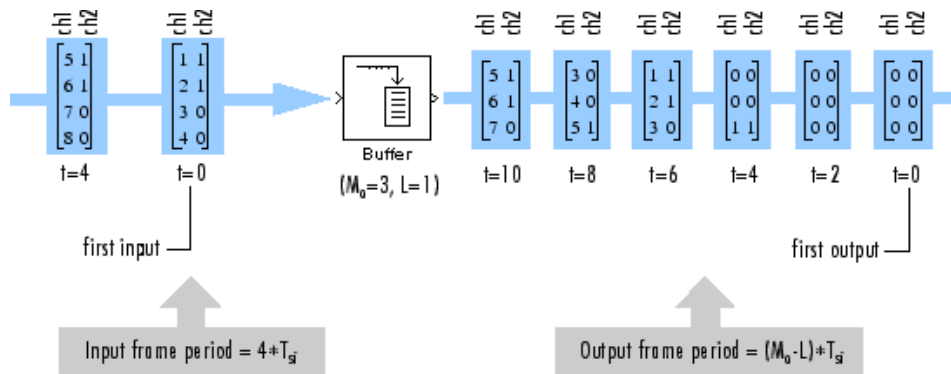
Note that the inputs do not begin appearing at the output until the last row of the third output matrix. This is due to the block's latency.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-68 for general information about algorithmic delay. For instructions on how to calculate buffering delay, and see "Buffer Delay and Initial Conditions" on page 3-55.

Buffer Delay and Initial Conditions

In the examples "Buffer Signals into Frames with Overlap" on page 3-48 and "Buffer Frame Inputs into Other Frame Inputs" on page 3-52, the input signal is delayed by a certain number of samples. The initial output samples correspond to the value specified for the **Initial condition** parameter. The initial condition is zero in both examples mentioned above.

Under most conditions, the Buffer and Unbuffer blocks have some amount of delay or latency. This latency depends on both the block parameter settings and the Simulink tasking mode. You can use the `rebuffer_delay` function to determine the length of the block's latency for any combination of frame size and overlap.

The syntax `rebuffer_delay(f,n,v)` returns the delay, in samples, introduced by the buffering and unbuffering blocks during multitasking operations, where `f` is the input frame size, `n` is the **Output buffer size** parameter setting, and `v` is the **Buffer overlap** parameter setting.

For example, you can calculate the delay for the model discussed in the "Buffer Frame Inputs into Other Frame Inputs" on page 3-52 using the following command at the MATLAB command line:

```
d = rebuffer_delay(4,3,1)
d = 8
```

This result agrees with the block's output in that example. Notice that this model was simulated in Simulink multitasking mode.

For more information about delay, see "Excess Algorithmic Delay (Tasking Latency)" on page 3-68. For delay information about a specific block, see the "Latency" section of the block reference page. For more information about the `rebuffer_delay` function, see `rebuffer_delay`.

Unbuffer Frame Signals into Sample Signals

You can unbuffer multichannel signals of frame length greater than 1 into multichannel signals of frame length equal to 1 using the `Unbuffer` block. The Unbuffer block

performs the inverse operation of the Buffer block's buffering process, where signals with frame length 1 are buffered into a signal with frame length greater than 1. The Unbuffer block generates an N-channel output containing one sample per frame from an N-channel input containing multiple channels per frame. The first row in each input matrix is always the first output.

The following figure is a graphical representation of this process.
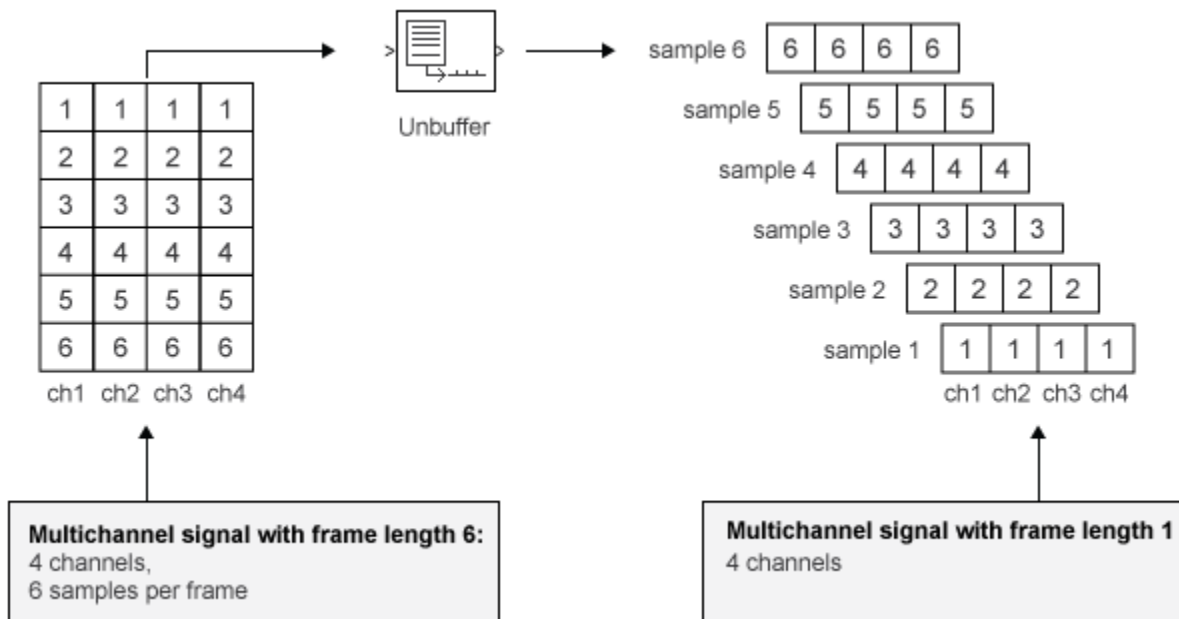
The sample period of the output, T_{so}, is related to the input frame period, T_{fi}, by the input frame size, M_i.

$$T_{so} = T_{fi} / M_i$$

The Unbuffer block always preserves the signal's sample period ($T_{so} = T_{si}$). See "Convert Sample and Frame Rates in Simulink" on page 3-19 for more information about rate conversions.

In the following example, a two-channel signal with four samples per frame is unbuffered into a two-channel signal with one sample per frame:

1 At the MATLAB command prompt, type `ex_unbuffer_tut`.

The Unbuffer Example model opens.
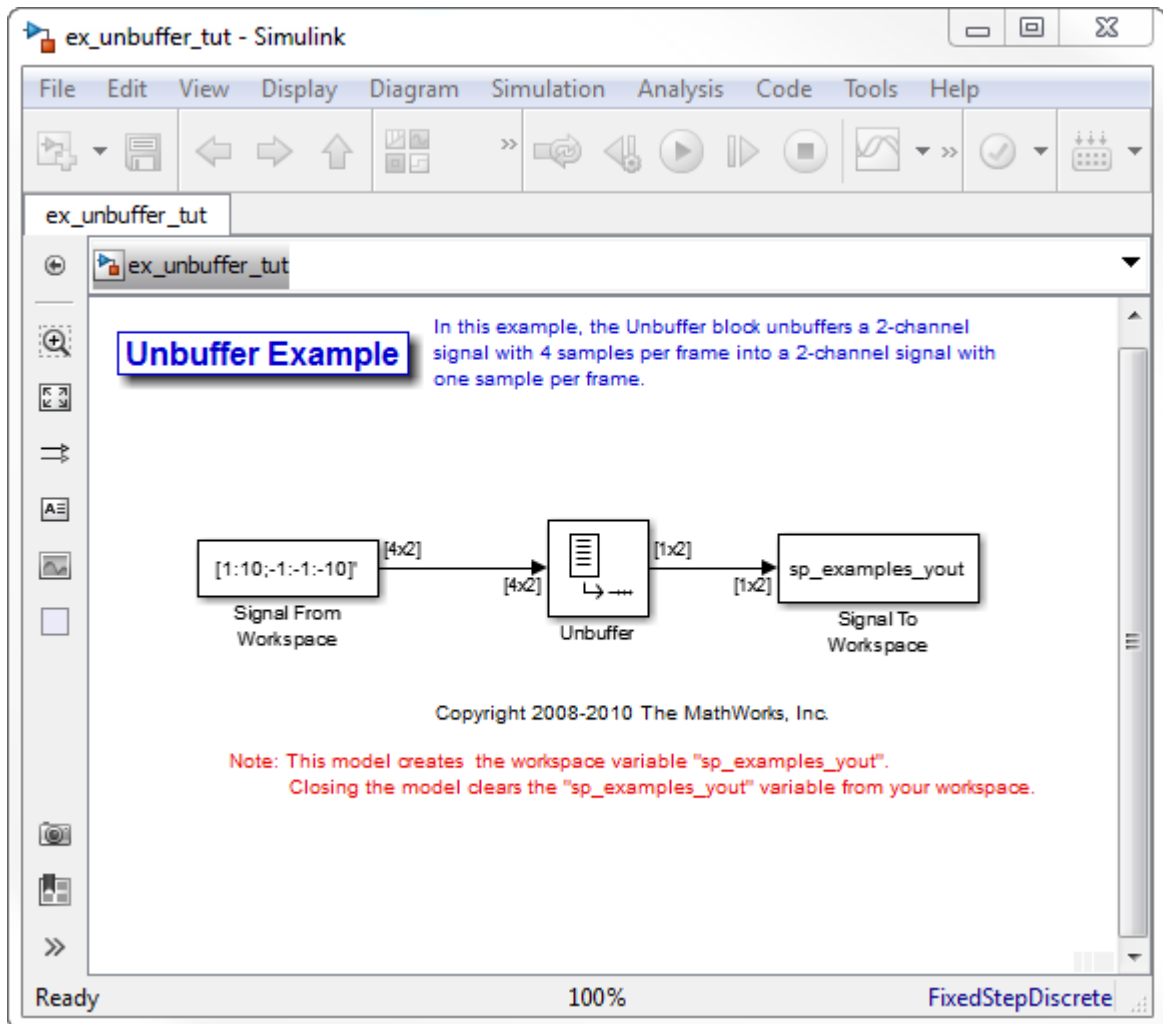
2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the block parameters as follows:

- **Signal** = [1:10;-1:-1:-10]'
- **Sample time** = 1
- **Samples per frame** = 4
- **Form output after final data value by** = Setting to zero

Based on these parameters, the Signal From Workspace block outputs a two-channel signal with frame size 4.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Unbuffer block. The **Function Block Parameters: Unbuffer** dialog box opens.

6 Set the **Initial conditions** parameter to 0, and then click **OK**.

The Unbuffer block unbuffers a two-channel signal with four samples per frame into a two-channel signal with one sample per frame.

7 Run the model.

The following figures is a graphical representation of what happens during the model simulation.
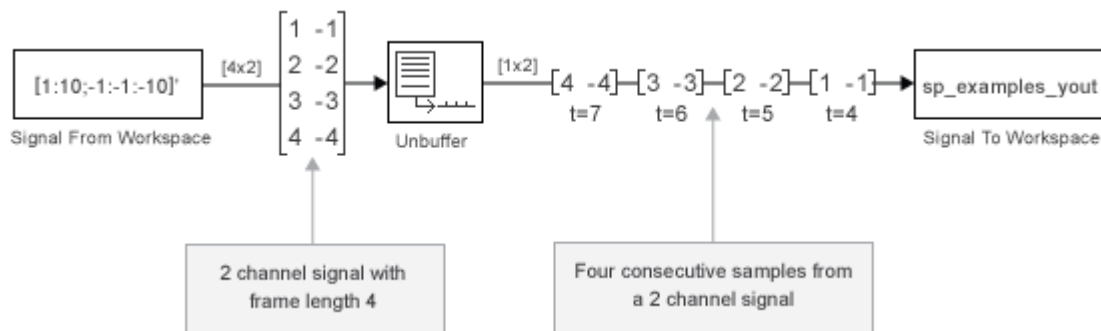
Note: The Unbuffer block generates initial conditions not shown in the figure below with the value specified by the **Initial conditions** parameter. See the Unbuffer reference page for information about the number of initial conditions that appear in the output.

8 At the MATLAB command prompt, type sp_examples_yout.

The following is a portion of the output.

```
sp_examples_yout(:,:,1) =

     O     O


sp_examples_yout(:,:,2) =

     O     O


sp_examples_yout(:,:,3) =

     O     O


sp_examples_yout(:,:,4) =

     O     O


sp_examples_yout(:,:,5) =

     1    -1


sp_examples_yout(:,:,6) =

     2    -2


sp_examples_yout(:,:,7) =

     3    -3
```

The Unbuffer block unbuffers the signal into a two-channel signal. Each page of the output matrix represents a different sample time.

More About

- "Sample- and Frame-Based Concepts" on page 3-2
- "Delay and Latency" on page 3-60

Delay and Latency

Computational Delay

The computational delay of a block or subsystem is related to the number of operations involved in executing that block or subsystem. For example, an FFT block operating on a 256-sample input requires Simulink software to perform a certain number of multiplications for each input frame. The actual amount of time that these operations consume depends heavily on the performance of both the computer hardware and underlying software layers, such as the MATLAB environment and the operating system. Therefore, computational delay for a particular model can vary from one computer platform to another.

The simulation time represented on a model's status bar, which can be accessed via the Simulink Digital Clock block, does not provide any information about computational delay. For example, according to the Simulink timer, the FFT mentioned above executes instantaneously, with no delay whatsoever. An input to the FFT block at simulation time t=25.0 is processed and output at simulation time t=25.0, regardless of the number of operations performed by the FFT algorithm. The Simulink timer reflects only algorithmic delay, not computational delay.

Reduce Computational Delay

There are a number of ways to reduce computational delay without actually running the simulation on faster hardware. To begin with, you should familiarize yourself with "Manual Performance Optimization" in the Simulink documentation, which describes some basic strategies. The following information discusses several options for improving performance.

A first step in improving performance is to analyze your model, and eliminate or simplify elements that are adding excessively to the computational load. Such elements might

include scope displays and data logging blocks that you had put in place for debugging purposes and no longer require. In addition to these model-specific adjustments, there are a number of more general steps you can take to improve the performance of any model:

- Use frame-based processing wherever possible. It is advantageous for the entire model to be frame based. See "Benefits of Frame-Based Processing" on page 3-6 for more information.

- Use the DSP Simulink model templates to tailor Simulink for digital signal processing modeling. For more information, see Configure the Simulink Environment for Signal Processing Models in DSP System Toolbox documentation.

- Turn off the Simulink status bar by deselecting the **Status bar** option in the **View** menu. Simulation speed will improve, but the time indicator will not be visible.

- Run your simulation from the MATLAB command line by typing

```
sim(gcs)
```

This method of starting a simulation can greatly increase the simulation speed, but also has several limitations:

- You cannot interact with the simulation (to tune parameters, for instance).
- You must press **Ctrl+C** to stop the simulation, or specify start and stop times.
- There are no graphics updates in MATLAB S-functions, which include blocks such as Vector Scope, etc.

- Use Simulink Coder code generation software to generate generic real-time (GRT) code targeted to your host platform, and run the model using the generated executable file. See the Simulink Coder documentation for more information.

Algorithmic Delay

Algorithmic delay is delay that is intrinsic to the algorithm of a block or subsystem and is independent of CPU speed. In this guide, the algorithmic delay of a block is referred to simply as the block's delay. It is generally expressed in terms of the number of samples by which a block's output lags behind the corresponding input. This delay is directly related to the time elapsed on the Simulink timer during that block's execution.

The algorithmic delay of a particular block may depend on both the block parameter settings and the general Simulink settings. To simplify matters, it is helpful to categorize a block's delay using the following categories:

The following topics explain the different categories of delay, and how the simulation and parameter settings can affect the level of delay that a particular block experiences.

Zero Algorithmic Delay

The FFT block is an example of a component that has no algorithmic delay. The Simulink timer does not record any passage of time while the block computes the FFT of the input, and the transformed data is available at the output in the same time step that the input is received. There are many other blocks that have zero algorithmic delay, such as the blocks in the Matrices and Linear Algebra libraries. Each of those blocks processes its input and generates its output in a single time step.

The Normalization block is an example of a block with zero algorithmic delay:

1   At the MATLAB command prompt, type `ex_normalization_tut`.

    The Normalization Example T1 model opens.
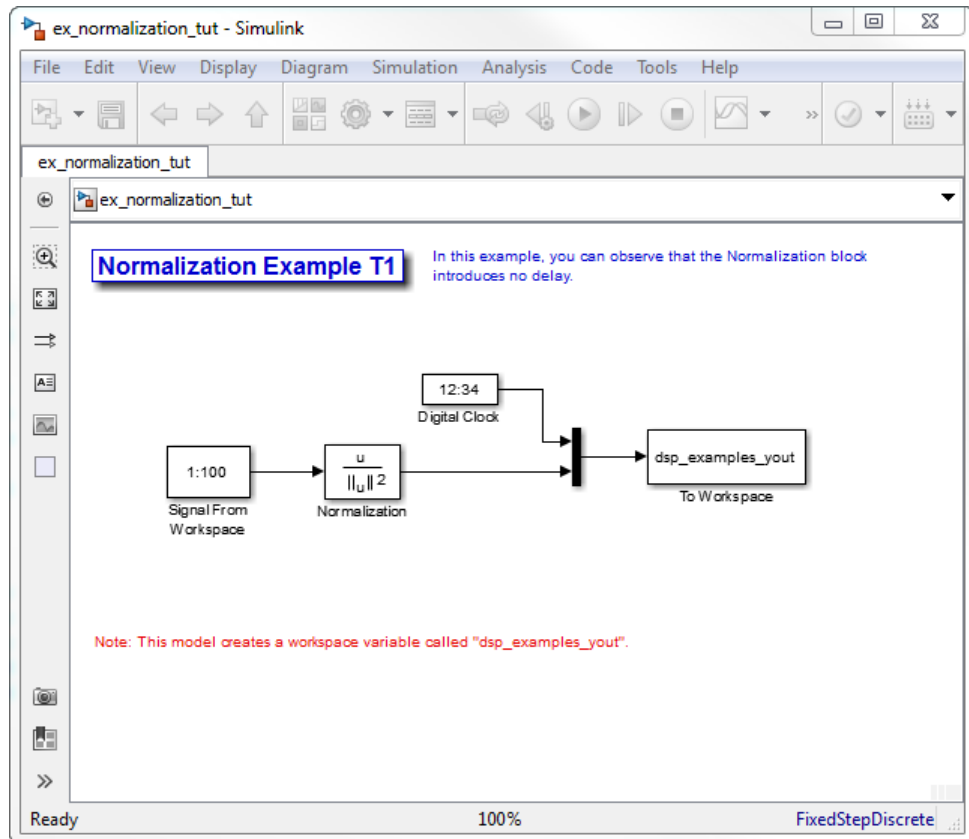
2   Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3   Set the block parameters as follows:

  · **Signal** = 1:100
  · **Sample time** = 1/4
  · **Samples per frame** = 4

4   Save these parameters and close the dialog box by clicking **OK**.

5   Run the model.

The model prepends the current value of the Simulink timer output from the Digital Clock block to each output frame.

The Signal From Workspace block generates a new frame containing four samples once every second ($T_{fo} = \Pi*4$). The first few output frames are:

```
(t=0)  [ 1  2  3  4]'
(t=1)  [ 5  6  7  8]'
(t=2)  [ 9 10 11 12]'
(t=3)  [13 14 15 16]'
(t=4)  [17 18 19 20]'
```

6 At the MATLAB command prompt, type `squeeze(dsp_examples_yout)'`.

The normalized output, `dsp_examples_yout`, is converted to an easier-to-read matrix format. The result, `ans`, is shown in the following figure:

```
ans =

         0    0.0333    0.0667    0.1000    0.1333
    1.0000    0.0287    0.0345    0.0402    0.0460
    2.0000    0.0202    0.0224    0.0247    0.0269
    3.0000    0.0154    0.0165    0.0177    0.0189
    4.0000    0.0124    0.0131    0.0138    0.0146
    5.0000    0.0103    0.0108    0.0113    0.0118
```

The first column of `ans` is the Simulink time provided by the Digital Clock block. You can see that the squared 2-norm of the first input,

```
[1 2 3 4]' ./ sum([1 2 3 4]'.^2)
```

appears in the first row of the output (at time $t=0$), the same time step that the input was received by the block. This indicates that the Normalization block has zero algorithmic delay.

Zero Algorithmic Delay and Algebraic Loops

When several blocks with zero algorithmic delay are connected in a feedback loop, Simulink may report an algebraic loop error and performance may generally suffer. You can prevent algebraic loops by injecting at least one sample of delay into a feedback loop , for example, by including a Delay block with **Delay** > 0. For more information, see "Algebraic Loops" in the Simulink documentation.

Basic Algorithmic Delay

The Variable Integer Delay block is an example of a block with algorithmic delay. In the following example, you use this block to demonstrate this concept:

1 At the MATLAB command prompt, type `ex_variableintegerdelay_tut`.

The Variable Integer Delay Example T1 opens.
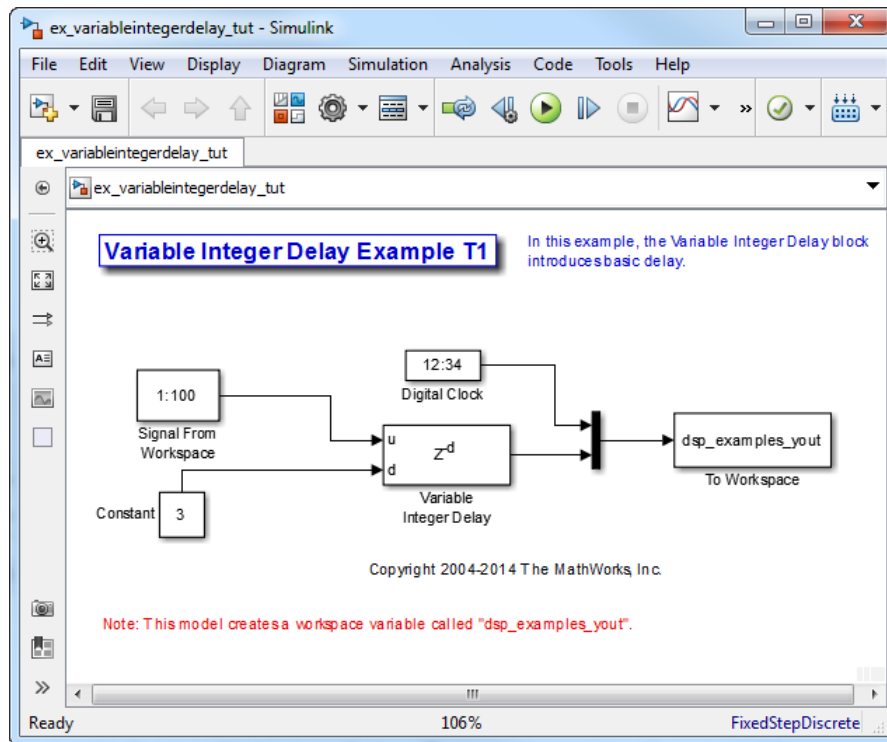
2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the block parameters as follows:

- **Signal** = 1:100
- **Sample time** = 1
- **Samples per frame** = 1

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Constant block. The **Source Block Parameters: Constant** dialog box opens.

6 Set the block parameters as follows:

- **Constant value** = 3
- **Interpret vector parameters as 1–D** = Clear this check box
- **Sample time** = 1

Click **OK** to save these parameters and close the dialog box.

The input to the Delay port of the Variable Integer Delay block specifies the number of sample periods that should elapse before an input to the In port is released to the output. This value represents the block's algorithmic delay. In this example, since the input to the Delay port is 3, and the sample period at the In and Delay ports is 1, then the sample that arrives at the block's In port at time t=0 is released to the output at time t=3.

7 Double-click the Variable Integer Delay block. The **Function Block Parameters: Variable Integer Delay** dialog box opens.

8 Set the **Initial conditions** parameter to -1, and then click **OK**.

9 From the **Display** menu, point to **Signals & Ports**, and select **Signal Dimensions** and **Wide Nonscalar Lines**.

10 Run the model.

The model should look similar to the following figure.
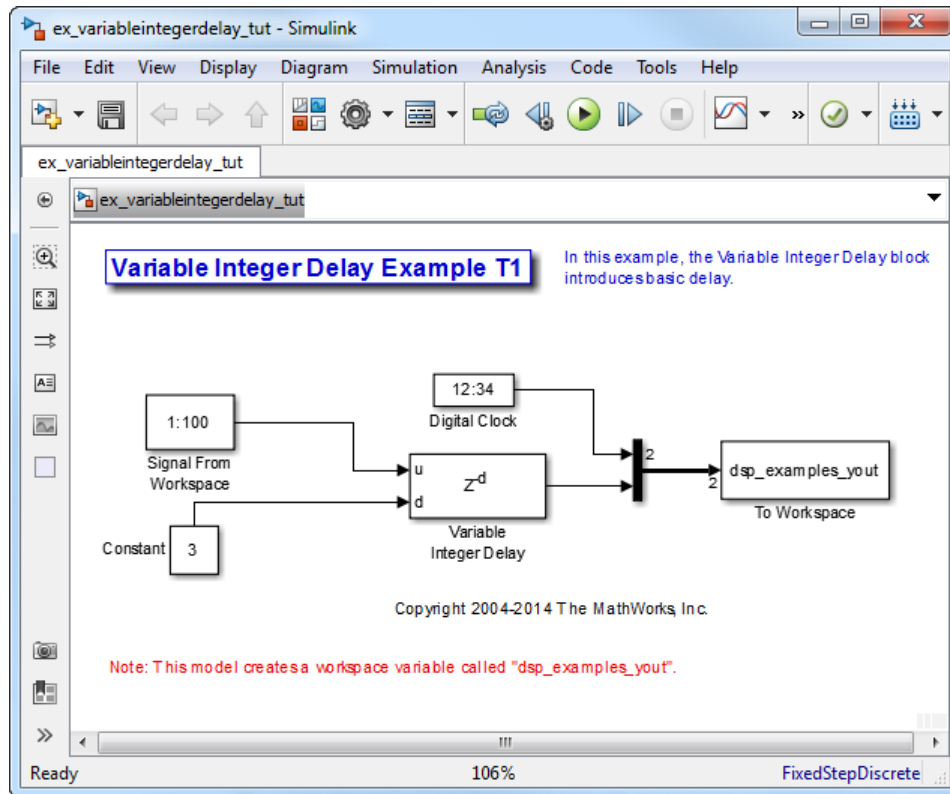
11 At the MATLAB command prompt, type `dsp_examples_yout`

The output is shown below:

```
dsp_examples_yout =

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

The first column is the Simulink time provided by the Digital Clock block. The second column is the delayed input. As expected, the input to the block at *t*=0 is

delayed three samples and appears as the fourth output sample, at *t*=3. You can also see that the first three outputs from the Variable Integer Delay block inherit the value of the block's **Initial conditions** parameter, -1. This period of time, from the start of the simulation until the first input is propagated to the output, is sometimes called the *initial delay* of the block.

Many DSP System Toolbox blocks have some degree of fixed or adjustable algorithmic delay. These include any blocks whose algorithms rely on delay or storage elements, such as filters or buffers. Often, but not always, such blocks provide an **Initial conditions** parameter that allows you to specify the output values generated by the block during the initial delay. In other cases, the initial conditions are internally set to 0.

Consult the block reference pages for the delay characteristics of specific DSP System Toolbox blocks.

Excess Algorithmic Delay (Tasking Latency)

Under certain conditions, Simulink may force a block to delay inputs longer than is strictly required by the block's algorithm. This excess algorithmic delay is called tasking latency, because it arises from synchronization requirements of the Simulink tasking mode. A block's overall algorithmic delay is the sum of its basic delay and tasking latency.

Algorithmic delay = Basic algorithmic delay + Tasking latency

The tasking latency for a particular block may be dependent on the following block and model characteristics:

- "Simulink Tasking Mode" on page 3-68
- "Block Rate Type" on page 3-69
- "Model Rate Type" on page 3-69
- "Block Input Processing Mode" on page 3-69

Simulink Tasking Mode

Simulink has two tasking modes:

- Single-tasking
- Multitasking

To select a mode, from the **Simulation** menu, select **Model Configuration Parameters**. In the **Select** pane, click **Solver**. From the **Type** list, select Fixed-step.

From the **Tasking mode for periodic sample times** list, choose `SingleTasking` or `MultiTasking`. If, from the **Tasking mode for periodic sample times** list you select `Auto`, the simulation runs in single-tasking mode if the model is single-rate, or multitasking mode if the model is multirate.

Note: Many multirate blocks have reduced latency in the Simulink single-tasking mode. Check the "Latency" section of a multirate block's reference page for details. Also see "Time-Based Scheduling and Code Generation" in the Simulink Coder documentation.

Block Rate Type

A block is called single-rate when all of its input and output ports operate at the same frame rate. A block is called multirate when at least one input or output port has a different frame rate than the others.

Many blocks are permanently single-rate. This means that all input and output ports always have the same frame rate. For other blocks, the block parameter settings determine whether the block is single-rate or multirate. Only multirate blocks are subject to tasking latency.

Note: Simulink may report an algebraic loop error if it detects a feedback loop composed entirely of multirate blocks. To break such an algebraic loop, insert a single-rate block with nonzero delay, such as a Unit Delay block. See the Simulink documentation for more information about "Algebraic Loops".

Model Rate Type

When all ports of all blocks in a model operate at a single frame rate, the model is called single-rate. When the model contains blocks with differing frame rates, or at least one multirate block, the model is called multirate. Note that Simulink prevents a single-rate model from running in multitasking mode by generating an error.

Block Input Processing Mode

Many blocks can operate in either sample-based or frame-based processing modes. To choose, you can set the **Input processing** parameter of the block to `Columns as channels (frame based)` or `Elements as channels (sample based)`.

Predict Tasking Latency

The specific amount of tasking latency created by a particular combination of block parameter and simulation settings is discussed in the "Latency" section of a block's reference page. In this topic, you use the `Upsample` block's reference page to predict the tasking latency of a model:

1   At the MATLAB command prompt, type `ex_upsample_tut1`.

The Upsample Example T1 model opens.
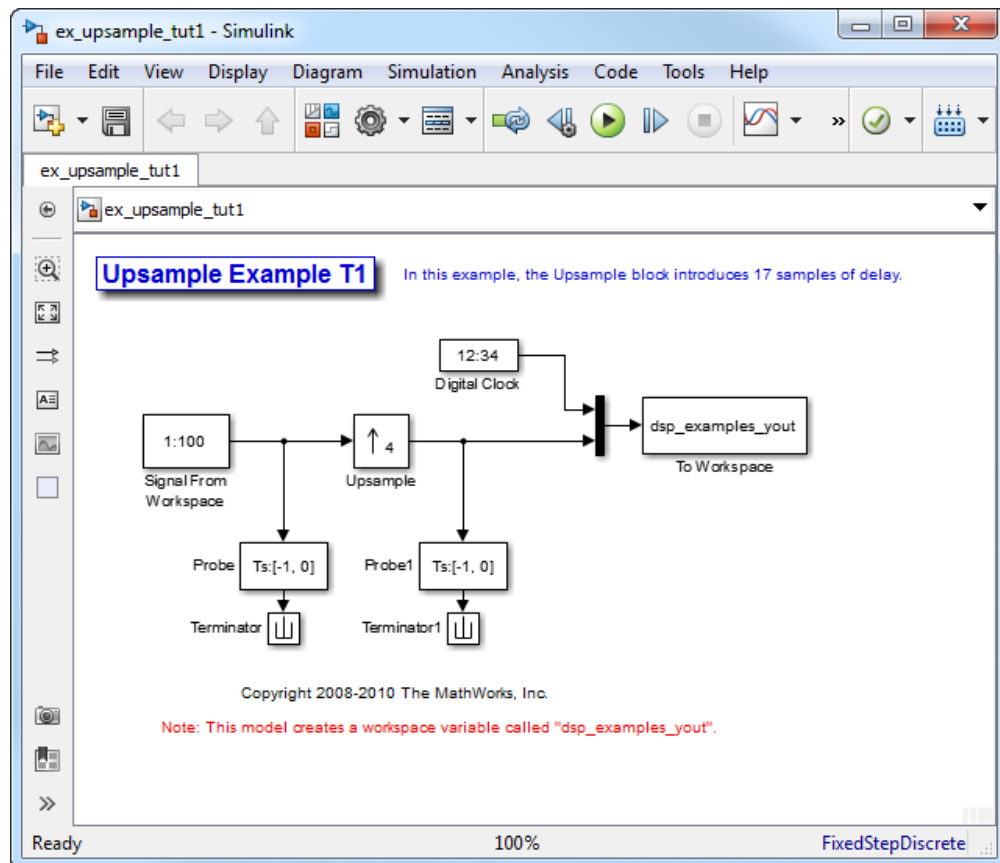
2   From the **Simulation** menu, select **Model Configuration Parameters**.

3 In the **Solver** pane, from the **Type** list, select `Fixed-step`. From the **Solver** list, select `discrete (no continuous states)`.

4 From the **Tasking mode for periodic sample times** list, select `MultiTasking`, and then click **OK**.

Most multirate blocks experience tasking latency only in the Simulink multitasking mode.

5 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

6 Set the block parameters as follows, and then click **OK**:

- **Signal** = `1:100`
- **Sample time** = `1/4`
- **Samples per frame** = `4`
- **Form output after final data value by** = `Setting to zero`

7 Double-click the Upsample block. The **Function Block Parameters: Upsample** dialog box opens.

8 Set the block parameters as follows, and then click **OK**:

- **Upsample factor, L** = `4`
- **Sample offset (0 to L-1)** = `0`
- **Input processing** = `Columns as channels (frame based)`
- **Rate options** = `Allow multirate processing`
- **Initial condition** = `-1`

The **Rate options** parameter makes the model multirate, since the input and output frame rates will not be equal.

9 Double-click the Digital Clock block. The **Source Block Parameters: Digital Clock** dialog box opens.

10 Set the **Sample time** parameter to `0.25`, and then click **OK**.

This matches the sample period of the Upsample block's output.

11 Run the model.

The model should now look similar to the following figure.
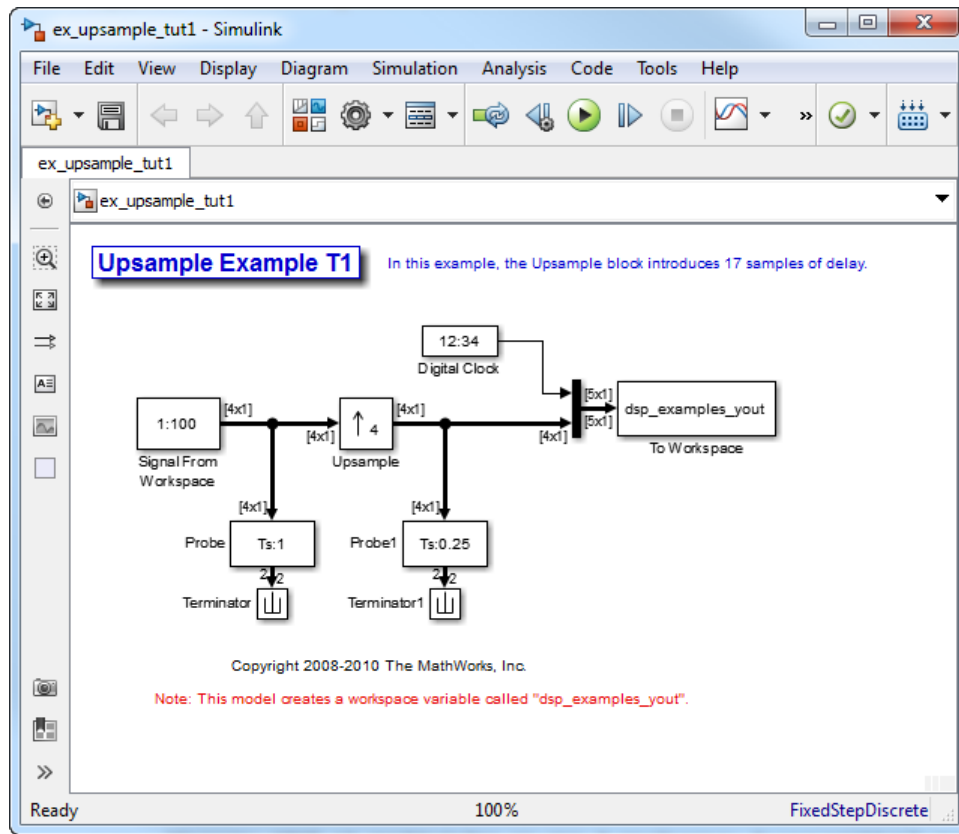
The model prepends the current value of the Simulink timer, from the Digital Clock block, to each output frame.

In the example, the Signal From Workspace block generates a new frame containing four samples once every second ($T_{fo} = \Pi*4$). The first few output frames are:

```
(t=0)   [ 1  2  3  4]
(t=1)   [ 5  6  7  8]
(t=2)   [ 9 10 11 12]
(t=3)   [13 14 15 16]
(t=4)   [17 18 19 20]
```

The Upsample block upsamples the input by a factor of 4, inserting three zeros between each input sample. The change in rates is confirmed by the Probe blocks in the model, which show a decrease in the frame period from $T_{fi} = 1$ to $T_{fo} = 0.25$.

12 At the MATLAB command prompt, type `squeeze(dsp_examples_yout)'`.

The output from the simulation is displayed in a matrix format. The first few samples of the result, `ans`, are:

```
ans =
```

0	-1.0000	0	0	0	1st output frame
0.2500	-1.0000	0	0	0	
0.5000	-1.0000	0	0	0	
0.7500	-1.0000	0	0	0	
1.0000	1.0000	0	0	0	5th output frame
1.2500	2.0000	0	0	0	
1.5000	3.0000	0	0	0	
1.7500	4.0000	0	0	0	
2.0000	5.0000	0	0	0	
time					

"Latency and Initial Conditions" in the Upsample block's reference page indicates that when Simulink is in multitasking mode, the first sample of the block's input appears in the output as sample M_iL+D+1, where M_i is the input frame size, L is the **Upsample factor**, and D is the **Sample offset**. This formula predicts that the first input in this example should appear as output sample 17 (that is, 4*4+0+1).

The first column of the output is the Simulink time provided by the Digital Clock block. The four values to the right of each time are the values in the output frame at that time. You can see that the first sample in each of the first four output frames inherits the value of the Upsample block's **Initial conditions** parameter. As a result of the tasking latency, the first input value appears as the first sample of the 5th output frame (at t=1). This is sample 17.

Now try running the model in single-tasking mode.

13 From the **Simulation** menu, select **Model Configuration Parameters**.

14 In the **Solver** pane, from the **Type** list, select `Fixed-step`. From the **Solver** list, select `Discrete (no continuous states)`.

15 From the **Tasking mode for periodic sample times** list, select `SingleTasking`.

16 Run the model.

The model now runs in single-tasking mode.

17 At the MATLAB command prompt, type `squeeze(dsp_examples_yout)'`.

The first few samples of the result, `ans`, are:

```
ans =
```

0	1.0000	0	0	0	1st output frame
0.2500	2.0000	0	0	0	
0.5000	3.0000	0	0	0	
0.7500	4.0000	0	0	0	
1.0000	5.0000	0	0	0	5th output frame
1.2500	6.0000	0	0	0	
1.5000	7.0000	0	0	0	
1.7500	8.0000	0	0	0	
2.0000	9.0000	0	0	0	

time

"Latency and Initial Conditions" in the Upsample block's reference page indicates that the block has zero latency for all multirate operations in the Simulink single-tasking mode.

The first column of the output is the Simulink time provided by the Digital Clock block. The four values to the right of each time are the values in the output frame at that time. The first input value appears as the first sample of the first output frame (at $t=0$). This is the expected behavior for the zero-latency condition. For the particular parameter settings used in this example, running `upsample_tut1` in single-tasking mode eliminates the 17-sample delay that is present when you run the model in multitasking mode.

You have now successfully used the Upsample block's reference page to predict the tasking latency of a model.

More About

- "Sample- and Frame-Based Concepts" on page 3-2
- "Buffering and Frame-Based Processing" on page 3-45

4

Filter Analysis, Design, and Implementation

Design a Filter in Fdesign — Process Overview

Process Flow Diagram and Filter Design Methodology

- "Exploring the Process Flow Diagram" on page 4-2
- "Select a Response" on page 4-4
- "Select a Specification" on page 4-4
- "Select an Algorithm" on page 4-6
- "Customize the Algorithm" on page 4-7
- "Design the Filter" on page 4-8
- "Design Analysis" on page 4-9
- "Realize or Apply the Filter to Input Data" on page 4-9

Note: You must minimally have the Signal Processing Toolbox™ installed to use `fdesign` and `design`. Some of the features described below may be unavailable if your installation does not additionally include the DSP System Toolbox license. The DSP System Toolbox significantly expands the functionality available for the specification, design, and analysis of filters. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

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.
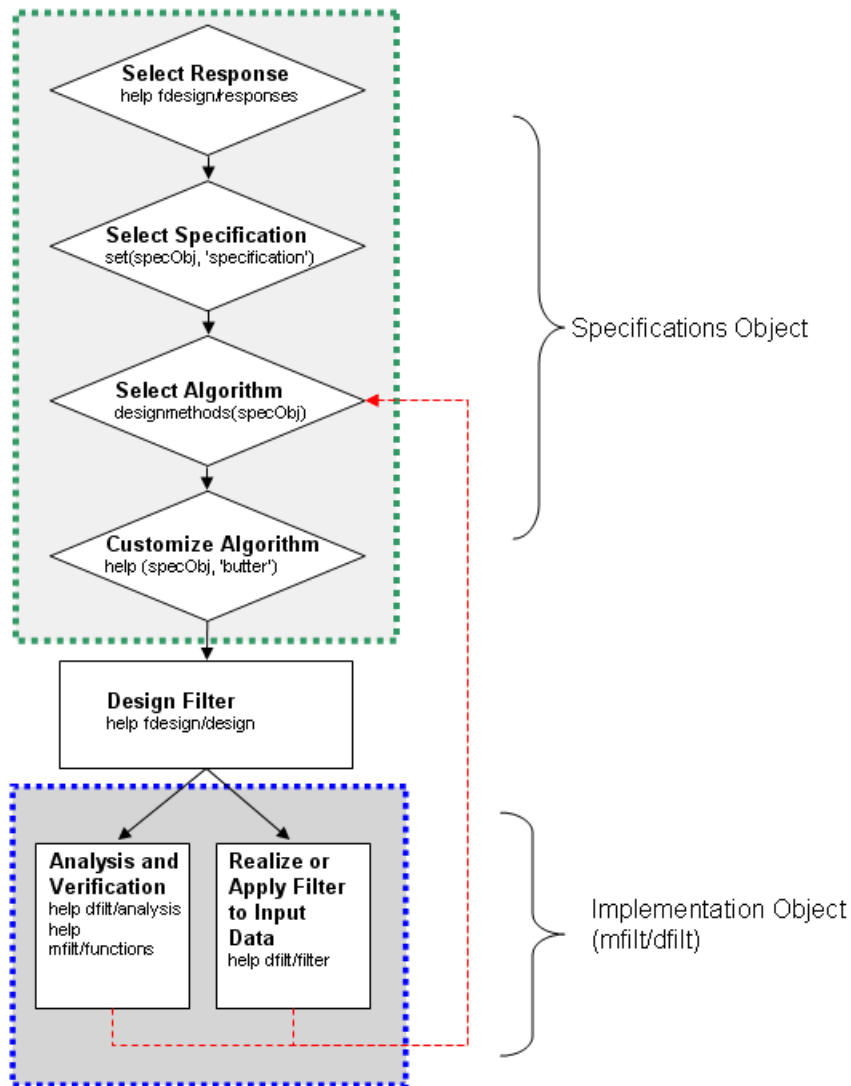
The first four steps of the filter design process relate to the filter Specifications Object, while the last two steps involve the filter Implementation Object. Both of these objects are discussed in more detail in the following sections. Step 5 - the design of the filter, is the transition step from 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 the software.

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

Select a Response

If you type:

```
help fdesign/responses
```
at the MATLAB command prompt, you see a list of all available filter responses. The responses marked with an asterisk require the DSP System Toolbox.

You must select a response to initiate the filter. In this example, a bandpass filter Specifications Object is created by typing the following:

```
d = fdesign.bandpass
```

Select a Specification

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

Note: A specification is not the same as the Specifications Object. A Specifications Object contains a specification as one of its properties.

When you select a filter response, there are a number of different specifications available. Each one contains a different combination of design parameters. After you create a filter Specifications Object, you can query the available specifications for that response. Specifications marked with an asterisk require the DSP System Toolbox.

```
>> 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.arbmag;
>> set(d,'specification')

ans =

    'N,F,A'
    'N,B,F,A'
```

The set command can be used to select one of the available specifications 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, `fdesign` returns the default specification for the response you chose in "Select a Response" on page 4-4, and provides default values for all design parameters included in the specification.

Select an Algorithm

The availability of algorithms depends the chosen filter response, the design parameters, and the availability of the DSP System Toolbox. 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`.

```
>> %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
```

However, for a specification of `'Fp,Fst,Ap,Ast'`, a number of algorithms are available. If the user has only the Signal Processing Toolbox installed, the following algorithms are available:

```
>>set (d, 'specification', 'Fp,Fst,Ap,Ast')
>>designmethods(d)

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


butter
cheby1
```

```
cheby2
ellip
equiripple
kaiserwin
```
If the user additionally has the DSP System Toolbox installed, the number of available algorithms for this response and specification string increases:

```
>>set(d,'specification','Fp,Fst,Ap,Ast')
>>designmethods(d)


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


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```
The user chooses a particular algorithm and implements the filter with the `design` function.

```
>>Hd=design(d,'butter');
```
The preceding code actually creates the filter, where `Hd` is the filter Implementation Object. This concept is discussed further in the next step.

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

Customize the Algorithm

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

```
help (d, 'algorithm-name')
```
where `d` is the Filter Specification Object, and `algorithm-name` is the name of the algorithm in single quotes, such as `'butter'` or `'cheby1'`.

The application of these customization options takes place while "Design the Filter" on page 4-8, because these options are the properties of the filter Implementation Object, not the Specification Object.

If you do not perform this step explicitly, the optimum algorithm structure is selected.

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
>> Hd = design(d);
```
where `Hd` is the Filter Object and `d` is the Specifications Object. This code creates a filter without specifying the algorithm. When the algorithm is not specified, the software selects the best available one.

To apply the algorithm chosen in "Select an Algorithm" on page 4-6, use the same `design` command, but specify the Butterworth algorithm as follows:

```
>> Hd = design(d, 'butter');
```
where `Hd` 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:

```
>> Hd = 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. A Filter Object is created only when `design` is called.

Design Analysis

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

- Frequency domain analysis — Includes the magnitude response, group delay, and pole-zero plots.
- 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 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. 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
```

Design a Filter in the Filterbuilder GUI

The Graphical Interface to Fdesign

Introduction to Filterbuilder

The `filterbuilder` function provides a graphical interface to the `fdesign` 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 filter for the desired response.

Note: `filterbuilder` requires the Signal Processing Toolbox. The functionality of `filterbuilder` is greatly expanded by the DSP System Toolbox. Some of the features described or displayed below are only available if the DSP System Toolbox is installed. You may verify your installation by typing `ver` at the command prompt.

Filterbuilder Design Process

The design process when using `filterbuilder` is similar to the process outlined in the section titled "Process Flow Diagram and Filter Design Methodology" 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 the software. If you have the DSP System Toolbox software installed, you have access to the full complement of filter responses.

Note: This step cannot be skipped because it is not automatically completed for you by the 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.

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 are 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.

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.
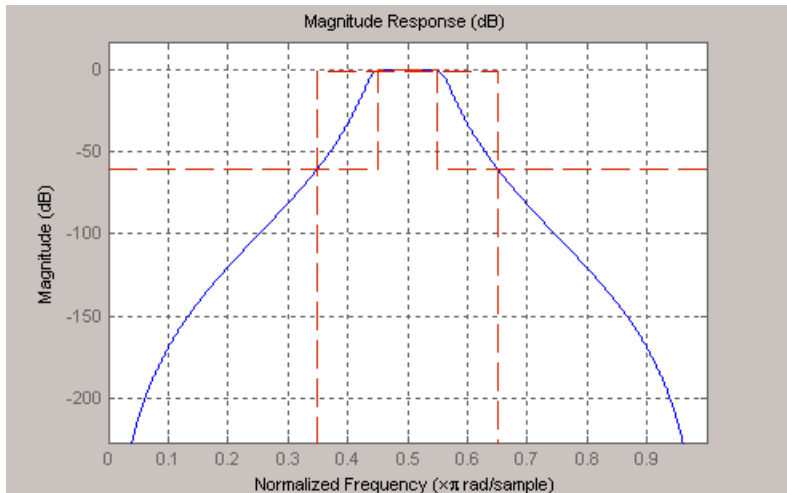
Realize or Apply the Filter to Input Data

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 the software. To filter your data, you must explicitly execute this step. In the **Filter Visualization Tool**, click OK and DSP System 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
```

Use FDATool with DSP System Toolbox Software

Design Advanced Filters in FDATool

Overview of FDATool Features

DSP System 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 additional 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 into 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.

Use FDATool with DSP System Toolbox Software

Adding DSP System 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.

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 `1000`.

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:
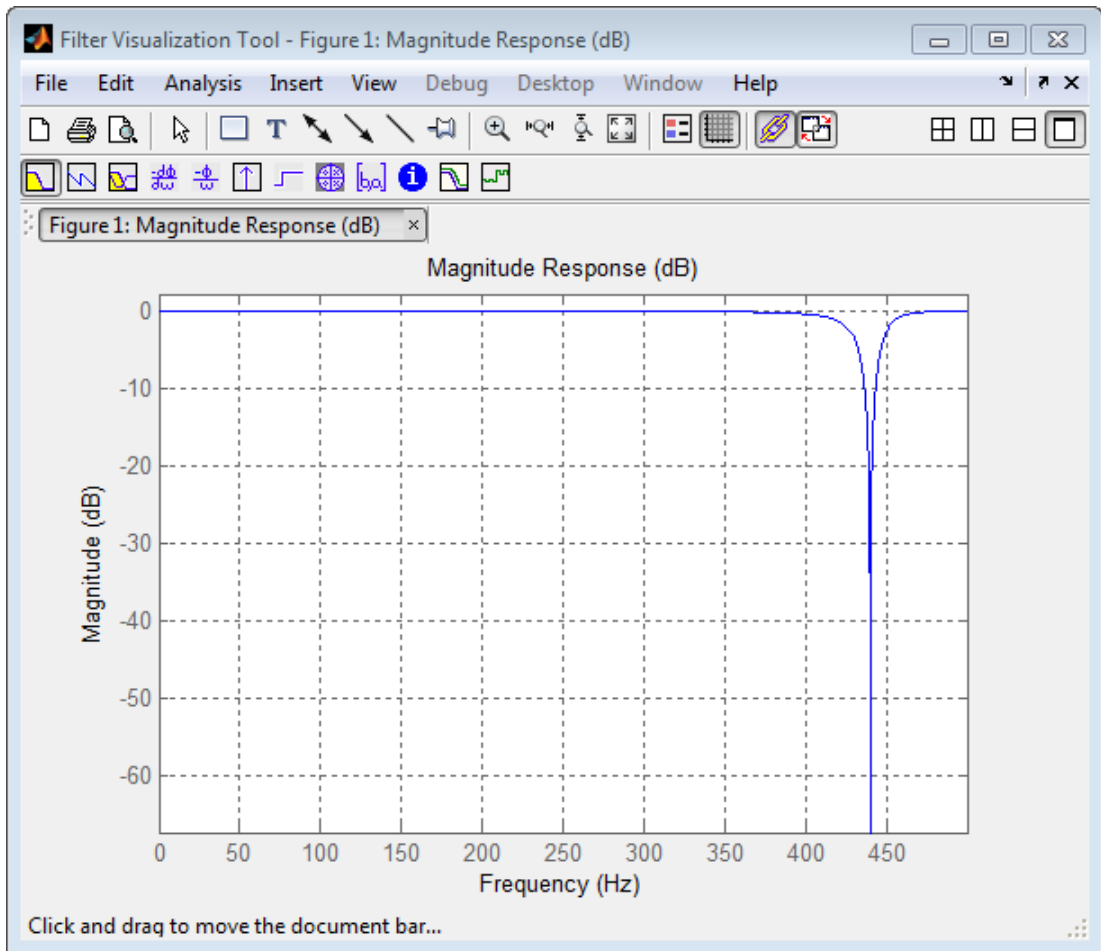
For more information about a design method, refer to the online Help system. For instance, to get further information about the **Q** setting for the notch filter in FDATool, enter

```
doc iirnotch
```

at the command line. 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.

Access the Quantization Features of FDATool

You use the quantization panel 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
```

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**.
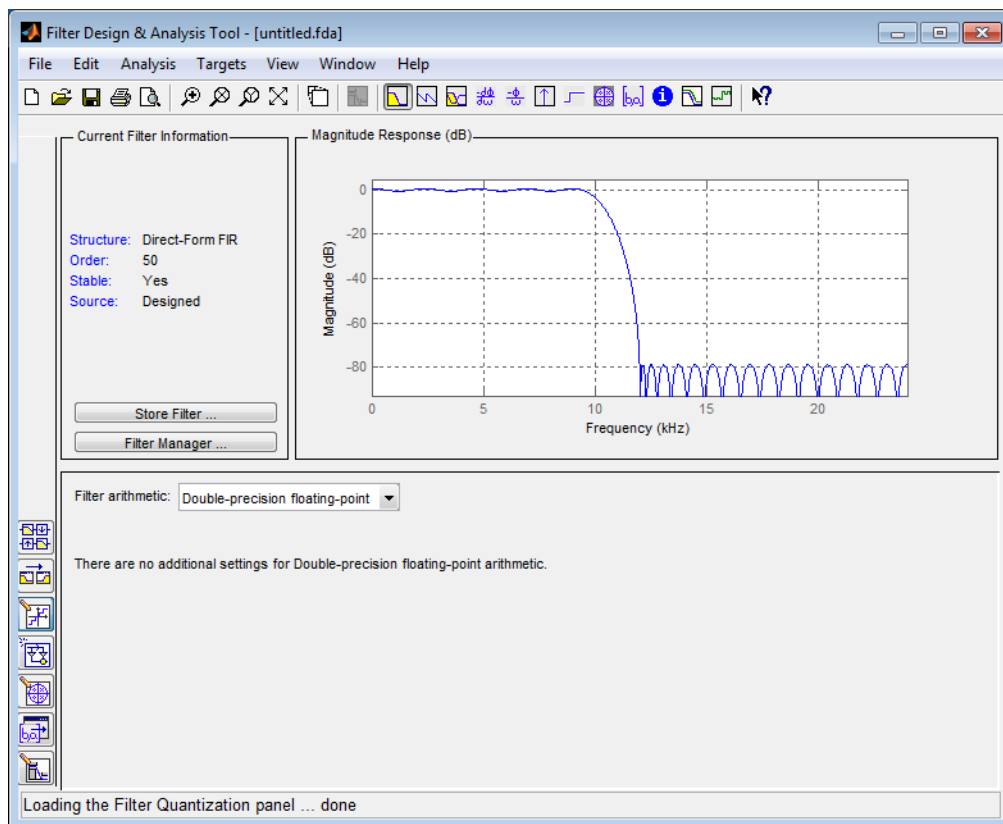
The **Filter arithmetic** option lets you quantize filters and investigate the effects of changing quantization settings. To enable the quantization settings in FDATool, select `Fixed-point` from the **Filter Arithmetic**.

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.

Quantize Filters in FDATool

Set 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.

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.
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.

Option Name	When Used	Description
		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.
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.

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

Option Name	When Used	Description
		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**.
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**.

Option Name	When Used	Description
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 below.

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`

Option	Equivalent Filter Property (Using Wildcard *)	Description
		(set overflowing values to the nearest representable value using modular arithmetic.
Filter Product (Multiply) Options		
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

Note: When you apply changes to the values in the Filter Internals pane, the plots for the **Magnitude response estimate** and **Round-off noise power spectrum** analyses update to reflect those changes. Other types of analyses are not affected by changes to the values in the Filter Internals pane.

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.

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.

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.

Analyze Filters with a Noise-Based Method

- "Analyze Filters with the Magnitude Response Estimate Method" on page 4-28
- "Compare the Estimated and Theoretical Magnitude Responses" on page 4-31
- "Select Quantized Filter Structures" on page 4-31
- "Convert the Structure of a Quantized Filter" on page 4-32
- "Convert Filters to Second-Order Sections Form" on page 4-32

Analyze Filters with 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 menu bar, 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.
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. Notice that the settings for the options reflect the defaults.

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 menu bar.

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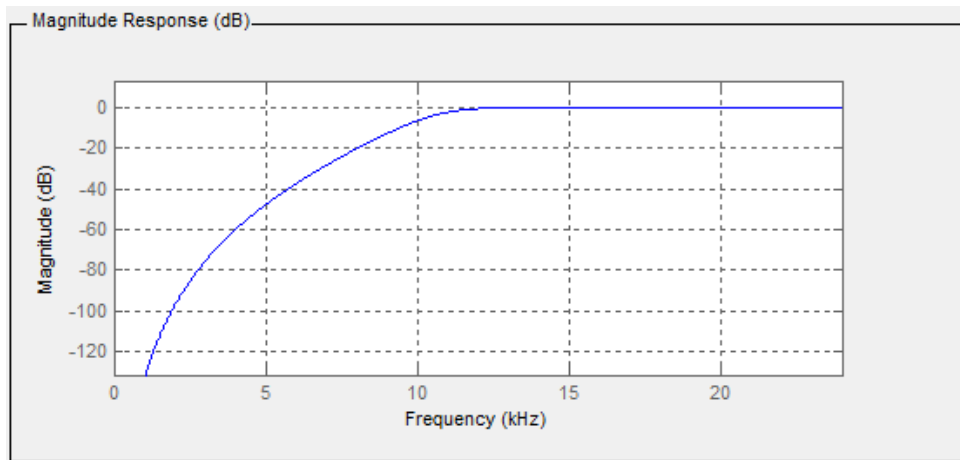
In the above figure you see the magnitude response as estimated by the analysis method.

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.

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.

Change Your Noise Analysis Parameters

In "Noise Method Applied to a Filter" on page 4-29, 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 "Analyze Filters with the Magnitude Response Estimate Method" on page 4-28.

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 menu bar.

   To give you access to the analysis parameters, the **Analysis Parameters** dialog box opens (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.

Compare 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.

Select 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 the Filter Structure" on page 15-21 in your Signal Processing Toolbox documentation.

Convert 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"
- "Direct Form I Transposed Filter Structure"
- "Direct Form II Filter Structure"
- "Direct Form I Filter Structure"
- "Direct Form Finite Impulse Response (FIR) Filter Structure"
- "Direct Form FIR Transposed Filter Structure"
- "Lattice Autoregressive Moving Average (ARMA) Filter Structure"
- `dfilt.calattice`
- `dfilt.calatticepc`
- "Direct Form Antisymmetric FIR Filter Structure (Any Order)"

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" for details about each of these structures.

Convert 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" on page 15-23 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.

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.

Scale Second-Order Section Filters

*   "Use the Reordering and Scaling Second-Order Sections Dialog Box" on page 4-33
*   "Scale an SOS Filter" on page 4-35

Use 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.
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.

Parameter	Description and Valid Value
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.

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.
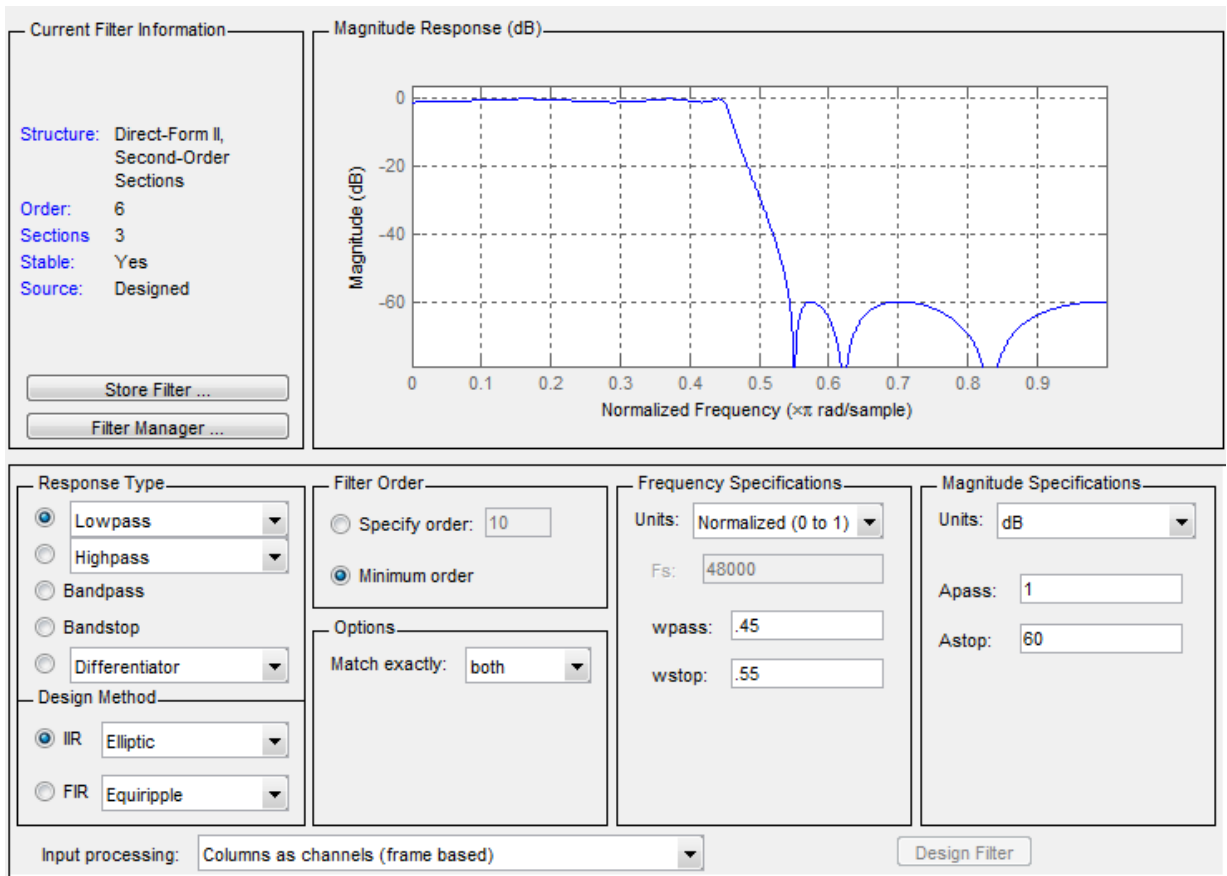
You kept the **Options** setting for **Match exactly** as both, meaning the filter design matches the specification for the passband and the stopband.

8 To switch to scaling the filter, select **Edit** > **Reorder and Scale Second-Order Sections** from the menu bar.

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.

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.

    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.

Reorder the Sections of Second-Order Section Filters

Reorder Filters Using FDATool

FDATool designs 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.
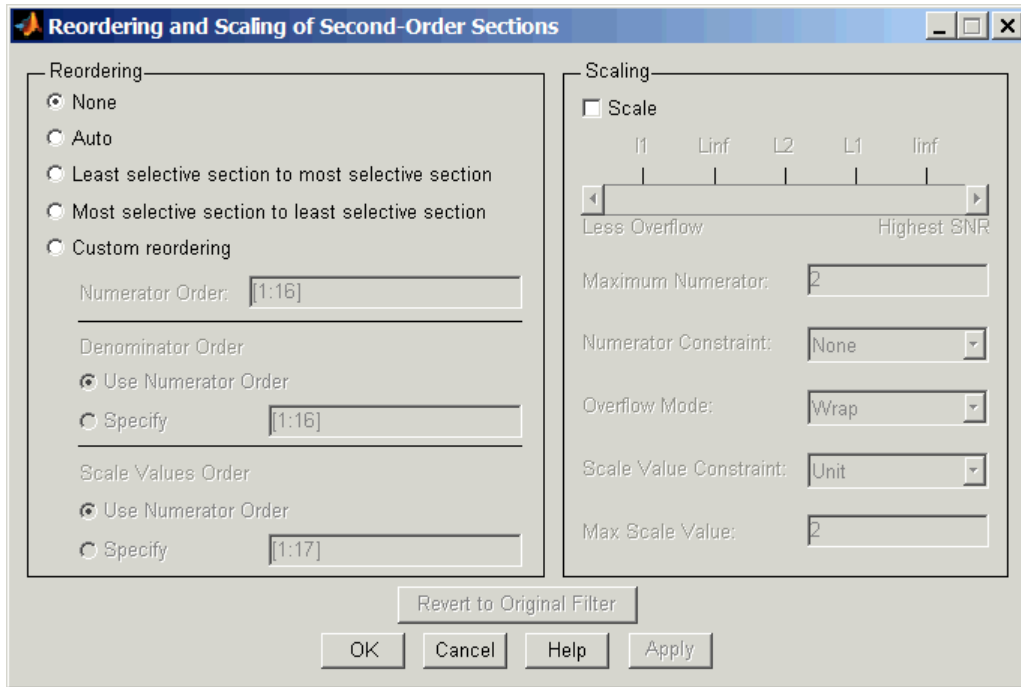
Controls on the Reordering and Scaling of Second-Order Sections dialog box

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 "Scale Second-Order Section Filters" on page 4-33 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.

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.

Control Option	Description
	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.
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].

Control Option	Description
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.

Reorder an SOS Filter

With FDATool open 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 designs 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.

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 Settings** from the menu bar so you see the sections of your filter displayed in FDATool.

4 In the **SOS View Settings** 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.
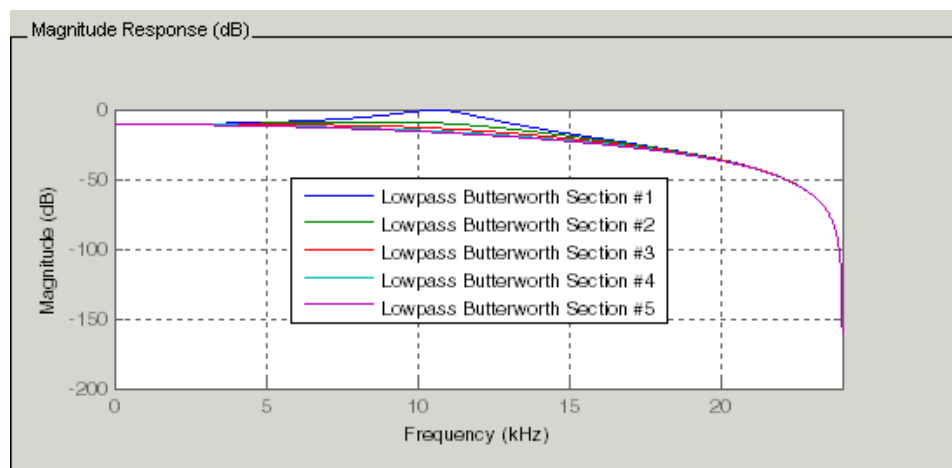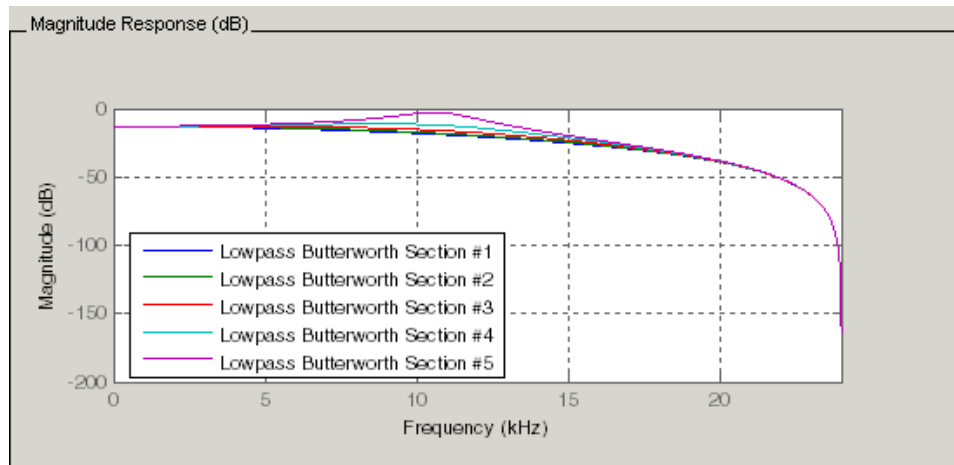
You see what reordering does, although the result is a bit subtle. Now try custom reordering the sections of your filter or using the most selective to least selective reordering option.

View SOS Filter Sections

- "Using the SOS View Dialog Box" on page 4-42
- "View the Sections of SOS Filters" on page 4-44

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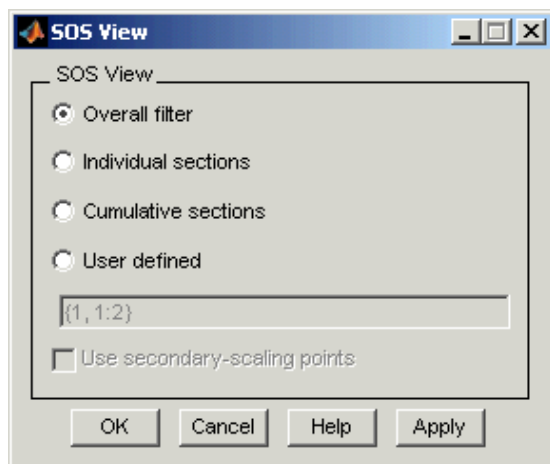
By default, SOS View shows the overall response of SOS filters. Options in the SOS View dialog box let you change the display. This table lists all the options and describes the effects of each.

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**.
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:

Option	Description
	• 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**.
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.

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.

   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.
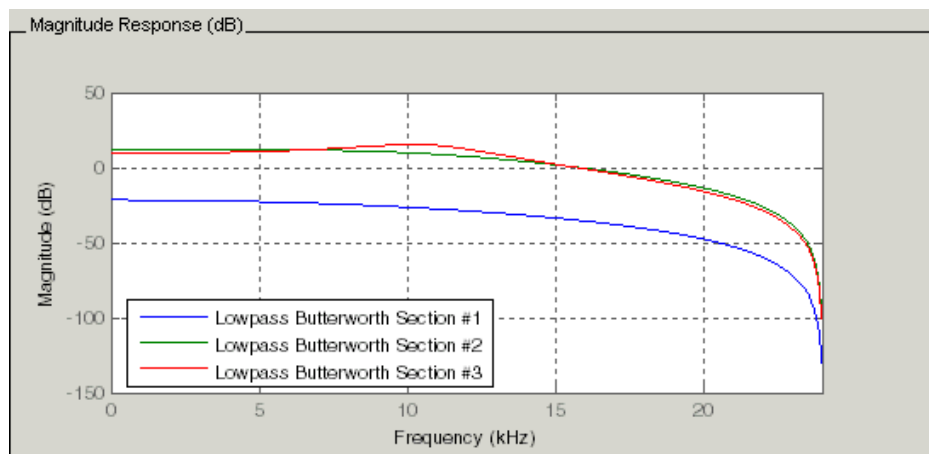
If you switch FDATool to display filter phase responses (by selecting **Analysis > Phase Response**), you see the phase response for each filter section in the analysis area.
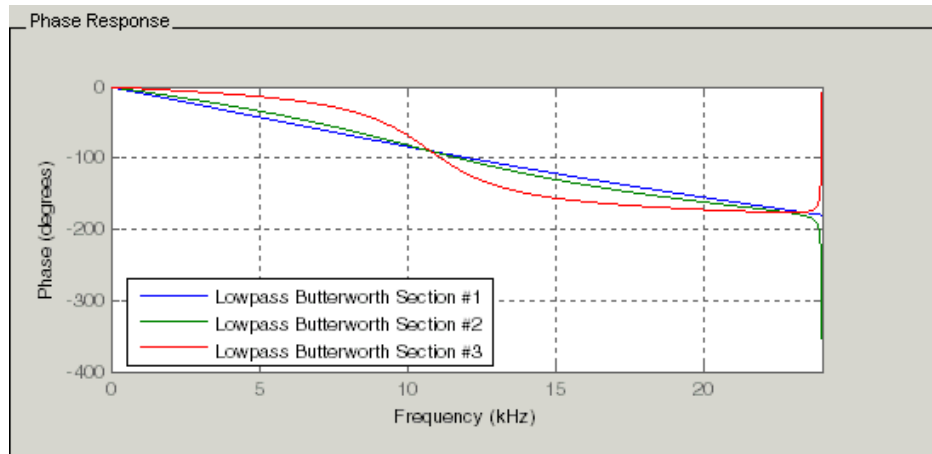
7   To define your own display of the sections, you use the **User defined** option and enter a vector of section indices to display. Now you see a display of the first section response, and the cumulative first, second, and third sections response:

 • 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.
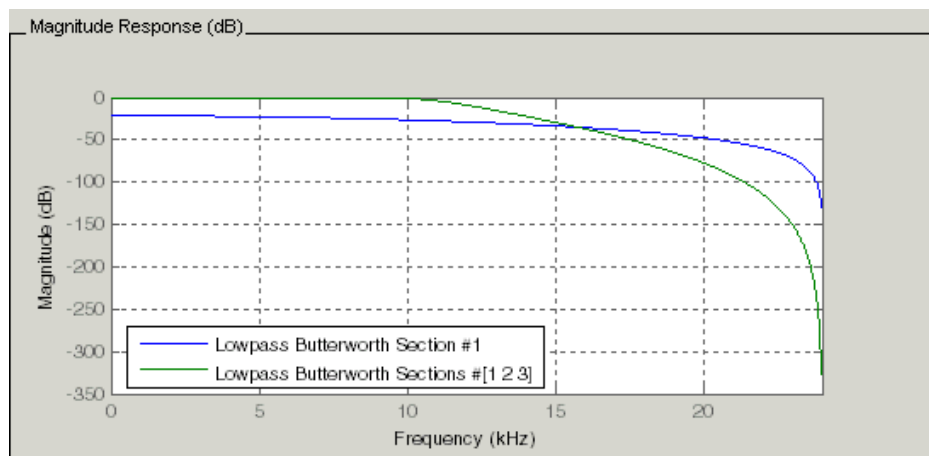
Import and Export Quantized Filters

- "Overview and Structures" on page 4-47
- "Import Quantized Filters" on page 4-48
- "To Export Quantized Filters" on page 4-50

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 "Importing a Filter Design" on page 15-35, and "Exporting a Filter Design" on page 15-23.

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

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 **File > Import Filter from Workspace** from the menu bar, or choose the **Import Filter from Workspace** icon in the side panel:

.

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

Export Coefficients, Objects, or System Objects to the Workspace

You can save the filter as filter coefficients variables, `dfilt` filter object variables, or System object variables.

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 From the **Export As** list, select one of the following options:

- Select `Coefficients` to save the filter coefficients.
- Select `Objects` to save the filter in a filter object.
- Select `System Objects` to save the filter in a filter System object.

The `System Objects` option does not appear in the drop-down list when the current filter structure is not supported by System objects.

4 Assign a variable name:

- For coefficients, assign variable names using the **Numerator** and **Denominator** options under **Variable Names**.
- For objects or System objects, assign the variable name in the **Discrete 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 **Export**.

Do not try to export the filter to a variable name that exists in your workspace without selecting **Overwrite existing variables**, in the previous step. If you do so, FDATool stops the export operation. The tool 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.
- Then select the **Overwrite existing variables** check box and click **Export**.

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.

Export Filter Coefficients 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 folder 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.

Export Filter Coefficients 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 dialog box is the standard Microsoft Windows save file dialog box.

5 Choose or enter a folder and filename for the text file, and click **OK**.

FDATool exports your quantized filter as a MAT-file with the specified name.

Generate MATLAB Code

You can generate MATLAB code using the **File** > **Generate MATLAB Code** menu. This menu has three options:

· **Filter Design Function**

This option creates MATLAB code that generates the DFILT/MFILT object currently designed in FDATool.

· **Filter Design Function (with System Objects)**

This option is similar to the previous option with the difference that a System object is generated instead of a DFILT/MFILT object. The option is disabled when the current filter is not supported by system objects.

· **Data Filtering Function (with System Objects)**

This option generates MATLAB code that filters input data with the current filter design. The MATLAB code is ready to be converted to C/C++ code using the codegen command. This option is disabled when the current filter is not supported by system objects.

Import XILINX Coefficient (.COE) Files

Import XILINX .COE Files into FDATool

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.

Transform Filters Using FDATool

- "Filter Transformation Capabilities of FDATool" on page 4-53
- "Original Filter Type" on page 4-54
- "Frequency Point to Transform" on page 4-57
- "Transformed Filter Type" on page 4-58
- "Specify Desired Frequency Location" on page 4-59

Filter Transformation Capabilities of FDATool

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, [icon], on the side bar, the **Transform Filter** panel opens in FDATool, as shown here.
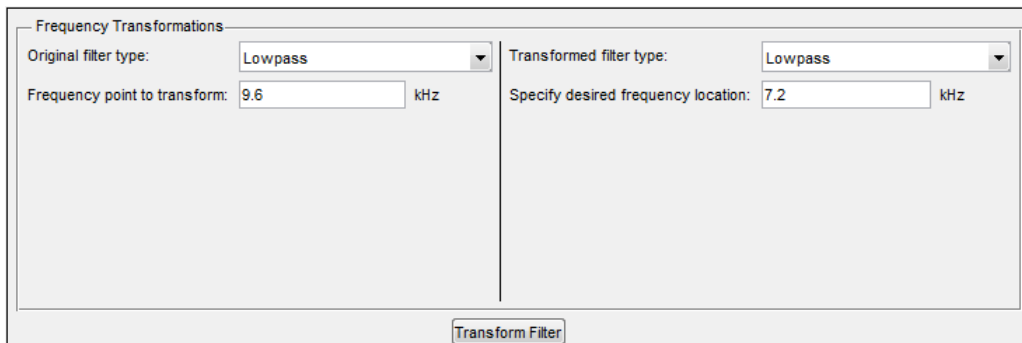
Your options for **Original filter type** refer to the type of your current filter to transform. If you select lowpass, you can transform your lowpass filter to another lowpass filter or to a highpass filter, or to numerous other filter formats, real and complex.

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**

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

Original Filter	Available Transformed Filter Types
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 FIR 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 4-93 and "Frequency Transformations for Complex Filters" on page 4-107.

Frequency Point to Transform

The frequency point you enter in this field identifies a magnitude response value (in dB) on the magnitude response curve.

When you enter frequency values in the **Specify desired frequency location** option, the frequency transformation tries to set the magnitude response of the transformed filter to the value identified by the frequency point you enter in this field.

While you can enter any location, generally you should specify a filter passband or stopband edge, or a value in the passband or stopband.

The **Frequency point to transform** sets the magnitude response at the values you enter in **Specify desired frequency location**. Specify a value that lies at either the edge of the stopband or the edge of the passband.

If, for example, you are creating a bandpass filter from a highpass filter, the transformation algorithm sets the magnitude response of the transformed filter at the **Specify desired frequency location** to be the same as the response at the **Frequency point to transform** value. Thus you get a bandpass filter whose response at the low and high frequency locations is the same. Notice that the passband between them is undefined. In the next two figures you see the original highpass filter and the transformed bandpass filter.

For more information about transforming filters, refer to "Digital Frequency Transformations" on page 4-85.

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 4-93 and "Frequency Transformations for Complex Filters" on page 4-107.

Specify Desired Frequency Location

The frequency point you enter in **Frequency point to transform** matched a magnitude response value. At each frequency you enter here, the transformation tries to make the magnitude response the same as the response identified by your **Frequency point to transform** value.

While you can enter any location, generally you should specify a filter passband or stopband edge, or a value in the passband or stopband.

For more information about transforming filters, refer to "Digital Frequency Transformations" on page 4-85.

Transform Filters

To transform the magnitude response of your filter, use the **Transform Filter** option on the side bar.

1 Design or import your filter into FDATool.

2 Click **Transform Filter**, , on the side bar.

   FDATool opens the **Transform Filter** panel in FDATool.

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

   When you select the type, whether is **lowpass**, **highpass**, **bandpass**, or **bandstop**, FDATool recognizes whether your filter form is FIR or IIR. Using both your filter type selection and the filter form, FDATool adjusts the entries on the **Transformed filter type** list to show only those that apply to your original filter.

4 Enter the frequency point to transform value in **Frequency point to transform**. Notice that the value you enter must be in kHz; for example, enter 0.1 for 100 Hz or 1.5 for 1500 Hz.

5 From the **Transformed filter type** list, select the type of filter you want to transform to.

Your filter type selection changes the options here.

- When you pick a lowpass or highpass filter type, you enter one value in **Specify desired frequency location**.

- When you pick a bandpass or bandstop filter type, you enter two values — one in **Specify desired low frequency location** and one in **Specify desired high frequency location**. Your values define the edges of the passband or stopband.

- When you pick a multiband filter type, you enter values as elements in a vector in **Specify a vector of 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 filters. 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.
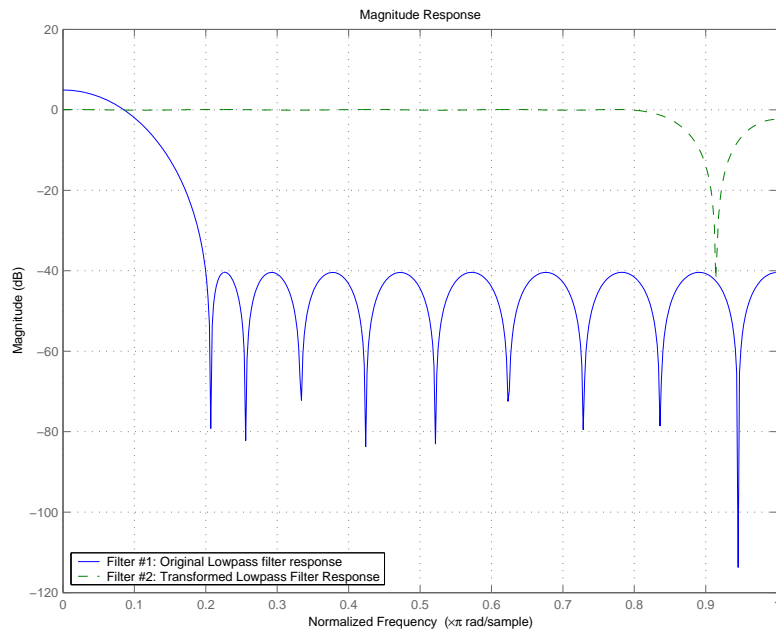
- To demonstrate the effects of selecting **Narrowband Highpass** or **Wideband Highpass**, the next figure presents the magnitude response curves for a source lowpass filter after it is transformed to both narrow- and wideband highpass filters. For comparison, the response of the original filter appears as well.
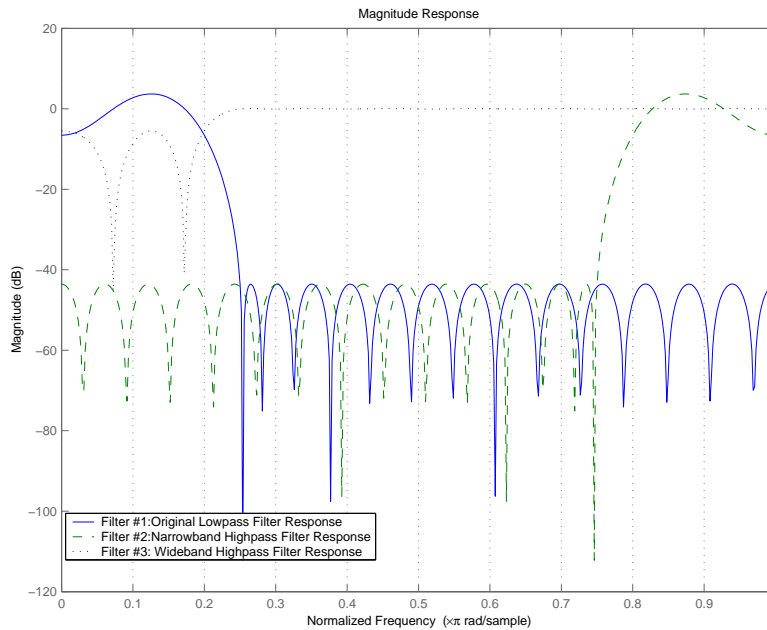
For the narrowband case, the transformation algorithm essentially reverses the magnitude response, like reflecting the curve around the *y*-axis, then translating the curve to the right until the origin lies at 1 on the *x*-axis. After reflecting and translating, the passband at high frequencies is the reverse of the passband of the original filter at low frequencies with the same rolloff and ripple characteristics.

Design Multirate Filters in FDATool

- "Introduction" on page 4-63

- "Switch FDATool to Multirate Filter Design Mode" on page 4-63

- "Controls on the Multirate Design Panel" on page 4-64

- "Quantize Multirate Filters" on page 4-71

- "Export Individual Phase Coefficients of a Polyphase Filter to the Workspace" on page 4-73

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.

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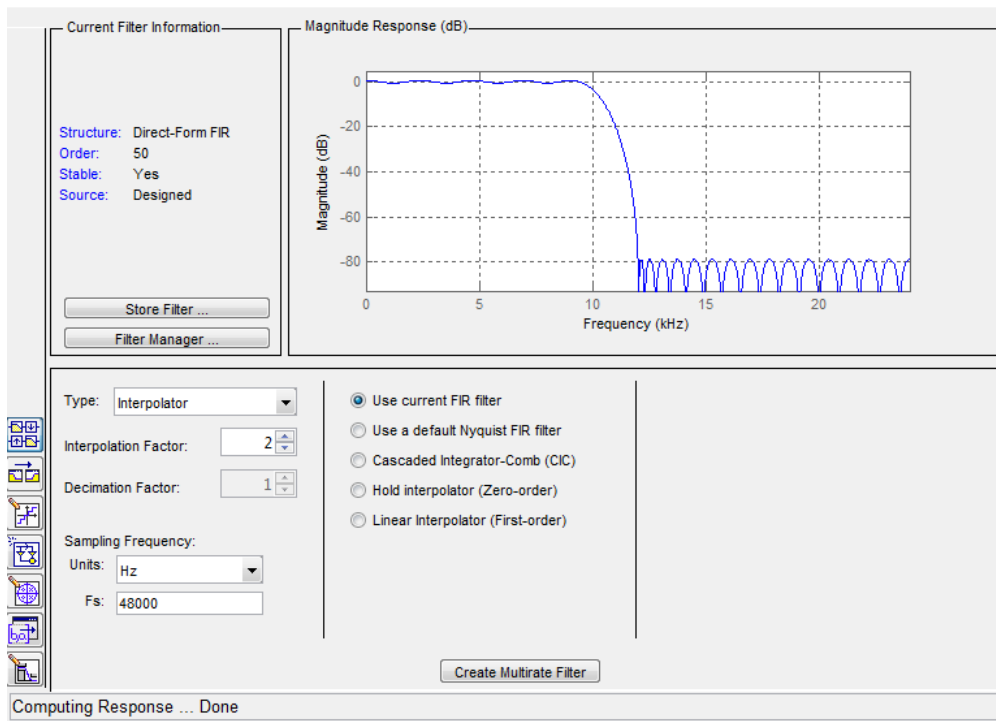
When the current filter in FDATool is not an FIR filter, the multirate filter design panel removes the **Use current FIR filter** option and selects the **Use default Nyquist FIR filter** option instead as the default setting.

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.

Select and Configure 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.

Option	Description
Decimation Factor	Use the up-down control arrows to specify the amount of decimation to apply to the signal. Factors range upwards from 2.
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**.

Design 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.
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.
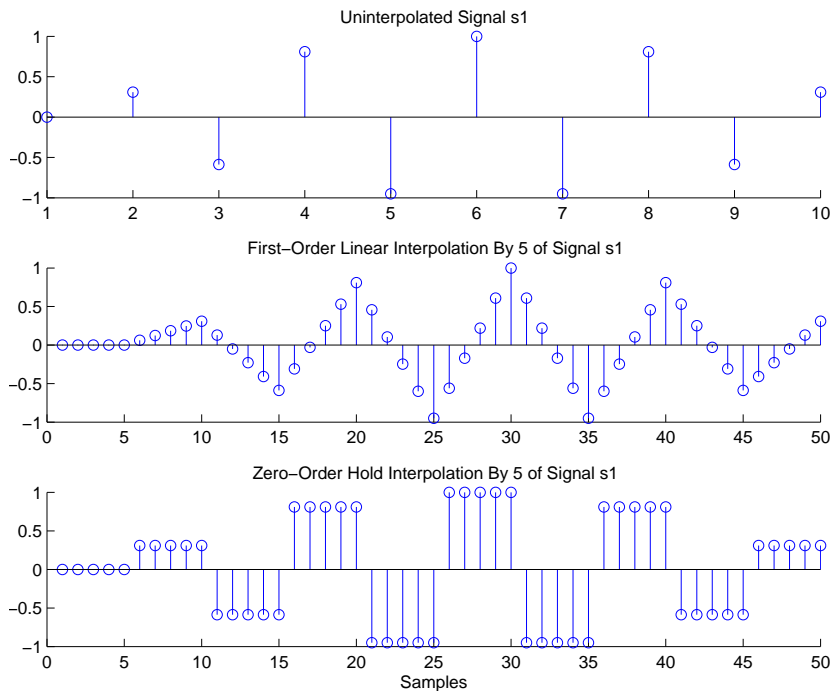
We used FDATool to create interpolators similar to the following code for the figure:

- 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.

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.

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.
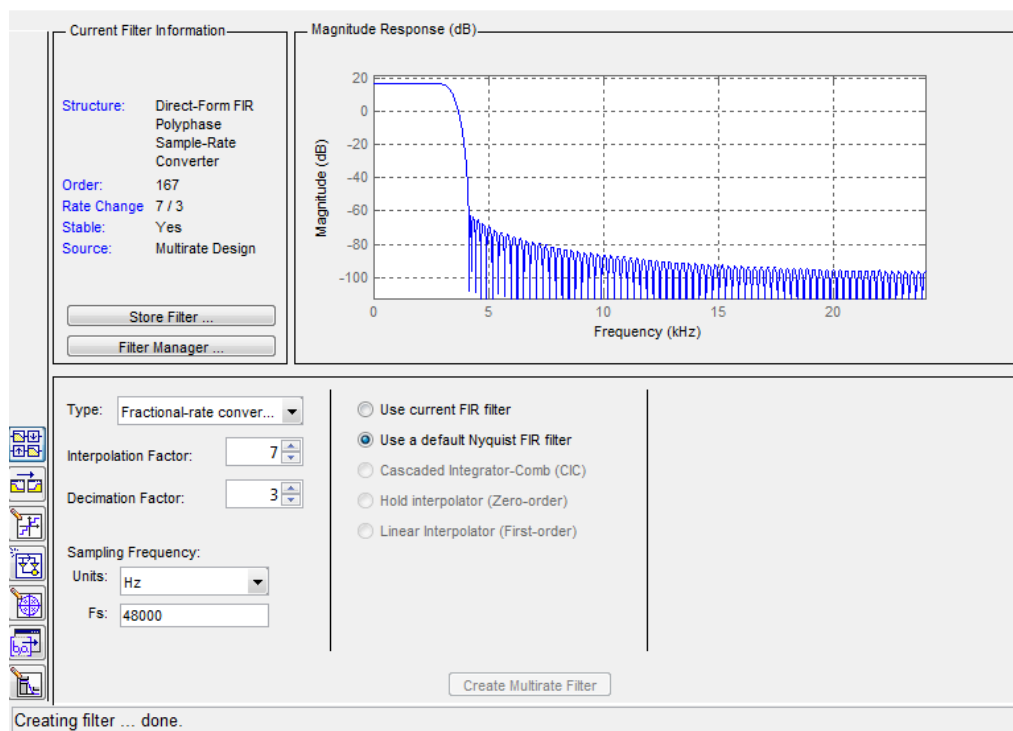
You can test the filter by exporting it to your workspace and using it to filter a signal. For information about exporting filters, refer to "Import and Export Quantized Filters" on page 4-47.

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**.

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.
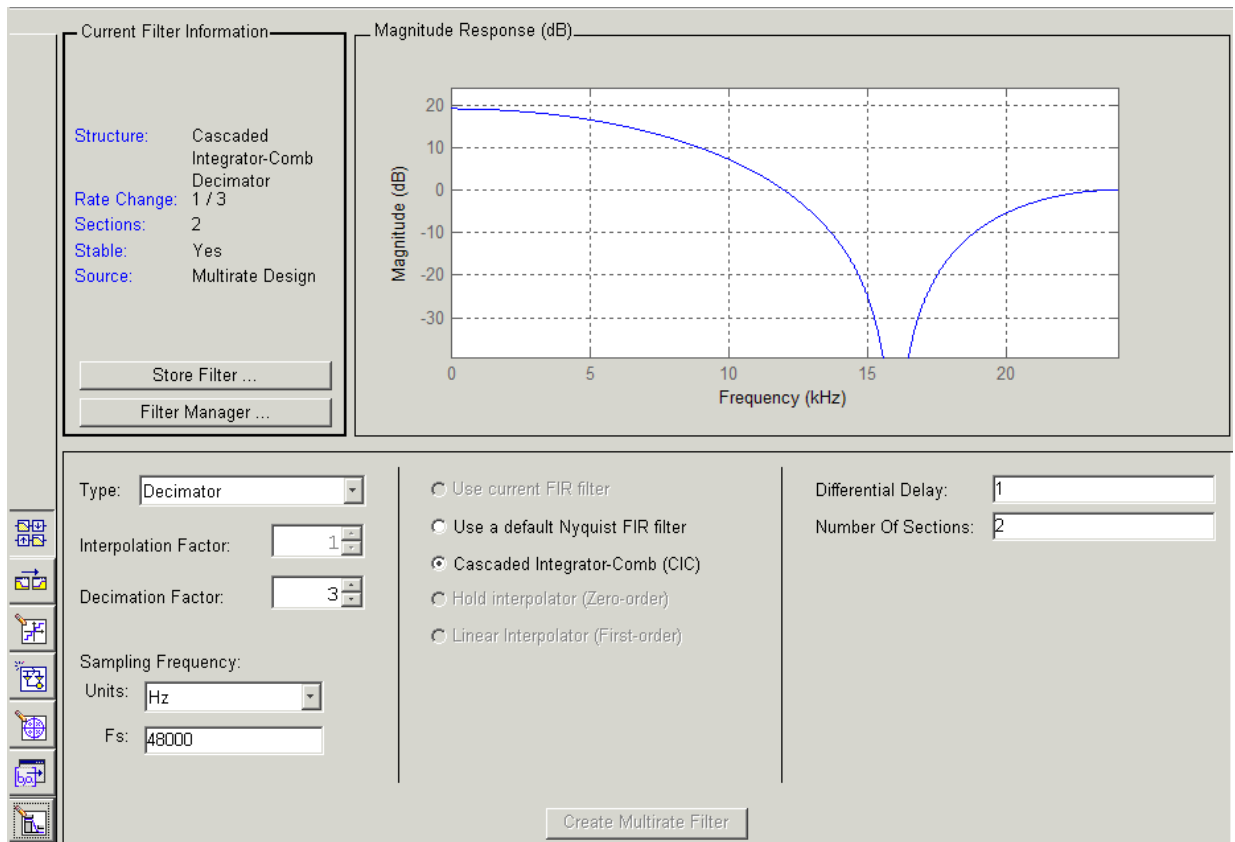
Designing other multirate filters follows the same pattern.

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**.

Quantize 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.

Quantize and Configure Multirate Filters

Follow these steps to convert your multirate filter to fixed-point arithmetic and set the fixed-point options.

1 Design or import your multirate filter and make sure it is the current filter in FDATool.

2 Click the **Set Quantization Parameters** button on the side bar.

3 From the **Filter Arithmetic** list on the Filter Arithmetic pane, select `Fixed-point`. If your filter is a CIC filter, the `Fixed-point` option is enabled by default and you do not set this option.

4 In the quantization panes, set the options for your filter. Set options for **Coefficients**, **Input/Output**, and **Filter Internals**.

5 Click **Apply**.

When 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.

Export 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.

Export 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.

Create a Matrix of Coefficients Using the polyphase Method

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.

Realize Filters as Simulink Subsystem Blocks

- "Introduction" on page 4-74
- "About the Realize Model Panel in FDATool" on page 4-74
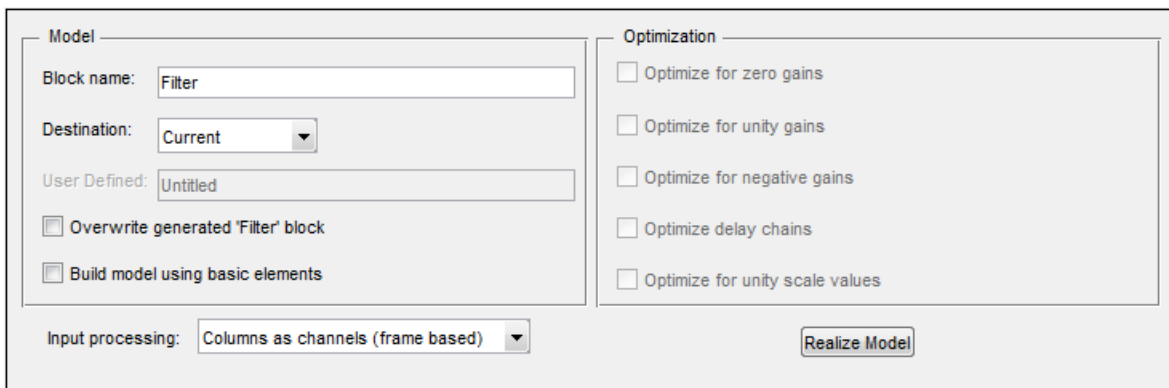
Introduction

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 digital filter blocks from the DSP System Toolbox library, or the Delay, Gain, and Sum blocks in Simulink. If you do not have a Fixed-Point Designer™ license, 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

To access to the Realize Model panel and the options for realizing your quantized filter as a Simulink subsystem block, switch FDATool to realize model mode by clicking 🔲 on the sidebar.

The following panel shows the options for configuring how FDATool implements your filter as a Simulink block.

For information on these parameters, see the descriptions on the Filter Realization Wizard block reference page.

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 the **Overwrite generated 'Filter' block** check box.

5 Select the **Build model using basic elements** check box to implement your filter as a subsystem block that consists of Sum, Gain, and Delay blocks.

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 negative 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
   - **Optimize for unity scale values** — removes all scale value multiplications by 1 from the filter structure

7 Click **Realize Model** to realize your quantized filter as a subsystem block according to the settings you selected.

If you double-click the filter block subsystem created by FDATool, you see the filter implementation in Simulink model form. Depending on the options you chose when you realized your filter, and the filter you started with, you might see one or more sections, or different architectures based on the form of your quantized filter. From this point on, the subsystem filter block acts like any other block that you use in Simulink models.

Supported Filter Structures

FDATool lets you realize discrete-time and multirate filters from the following forms:

Structure	Description
firdecim	Decimators based on FIR filters
firtdecim	Decimators based on transposed FIR filters
linearinterp	Linear interpolators
firinterp	Interpolators based on FIR filters
multirate polyphase	Multirate filters
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	

FIR Nyquist (L-th band) Filter Design

This example shows how to design lowpass FIR Nyquist filters. It also compares these filters with raised cosine and square root raised cosine filters. These filters are widely used in pulse-shaping for digital transmission systems. They also find application in interpolation/decimation and filter banks.

Magnitude Response Comparison

The plot shows the magnitude response of an equiripple Nyquist filter and a raised cosine filter. Both filters have an order of 60 and a rolloff-factor of 0.5. Because the equiripple filter has an optimal equiripple stopband, it has a larger stopband attenuation for the same filter order and transition width. The raised-cosine filter is obtained by truncating the analytical impulse response and it is not optimal in any sense.

```
NBand = 4;
N = 60;            % Filter order
R = 0.5;           % Rolloff factor
TW = R/(NBand/2);  % Transition Bandwidth
f1 = fdesign.nyquist(NBand,'N,TW',N,TW);
eq = design(f1,'equiripple','Zerophase',true,'SystemObject',true);
coeffs = rcosdesign(R,N/NBand,NBand,'normal');
coeffs = coeffs/max(abs(coeffs))/NBand;
rc     = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq,rc,'Color','white');
legend(fvt,'Equiripple NYQUIST design','Raised Cosine design');
```
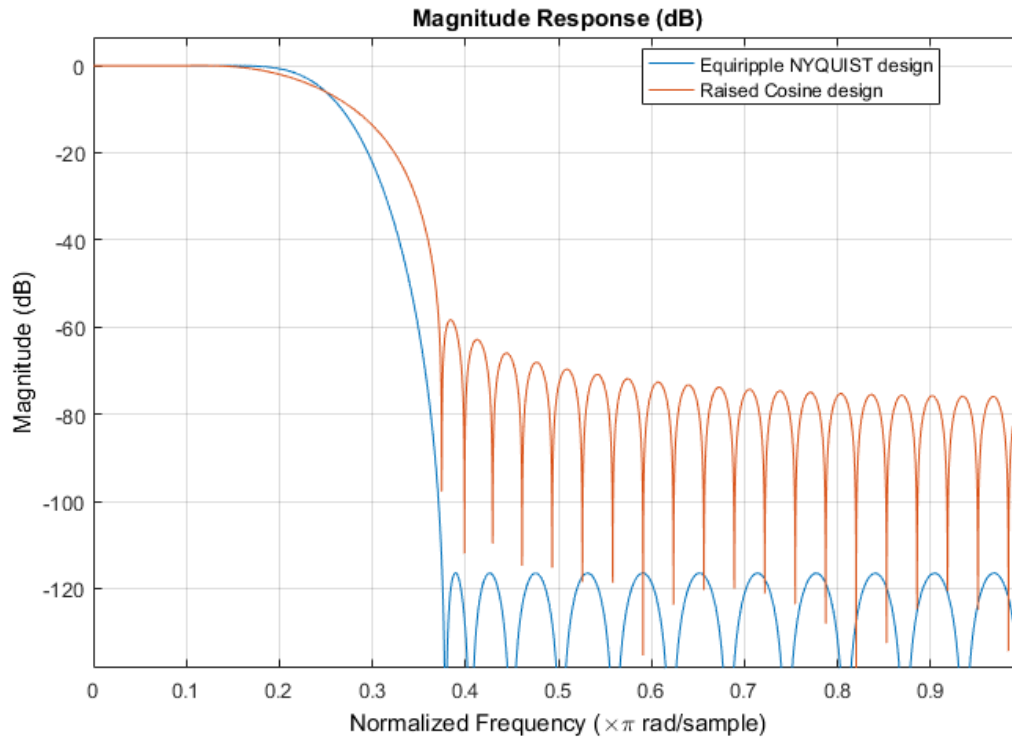
In fact, in this example it is necessary to increase the order of the raised-cosine design to about 1400 in order to attain similar attenuation.

Impulse Response Comparison

Here we compare the impulse responses. Notice that the impulse response in both cases is zero every 4th sample (except for the middle sample). Nyquist filters are also known as L-th band filters, because the cutoff frequency is Pi/L and the impulse response is zero every L-th sample. In this case we have 4th band filters.

```
f1.FilterOrder = 38;
eq1 = design(f1,'equiripple','Zerophase',true,'SystemObject',true);
coeffs = rcosdesign(R,f1.FilterOrder/NBand,NBand,'normal');
coeffs = coeffs/max(abs(coeffs))/NBand;
rc1 = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq1,rc1,'Color','white','Analysis','Impulse');
```

```
legend(fvt,'Equiripple NYQUIST','Raised Cosine');
title('Impulse response, Order=38, Rolloff = 0.5');
```
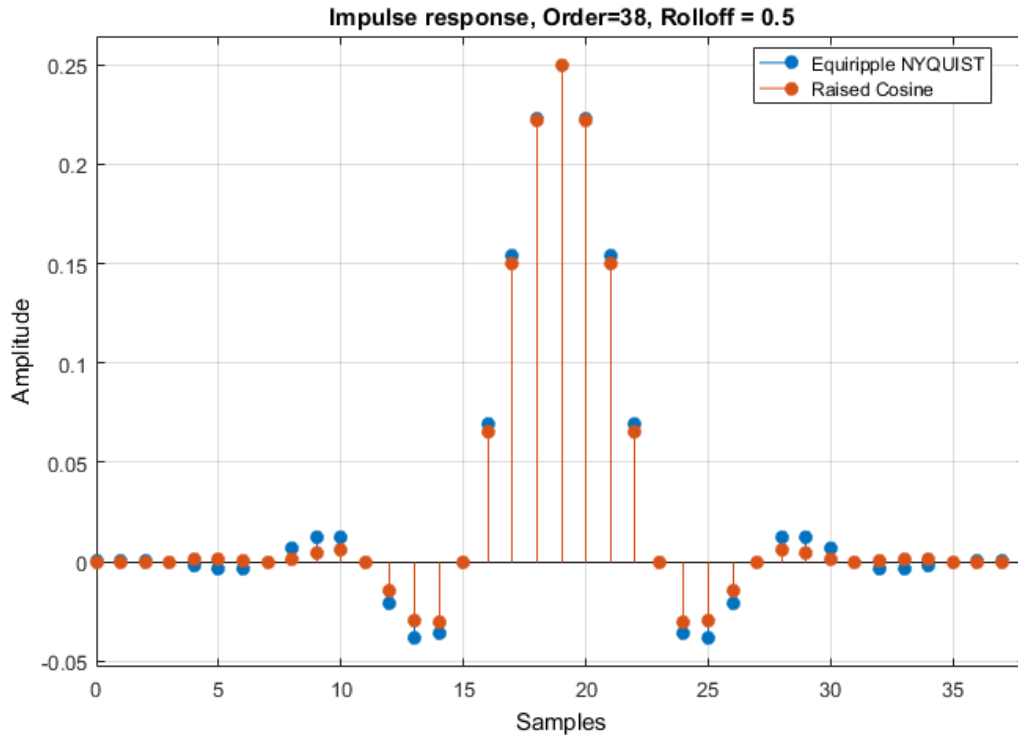

Nyquist Filters with a Sloped Stopband

Equiripple designs allow for control of the slope of the stopband of the filter. For example, the following designs have slopes of 0, 20, and 40 dB/(rad/sample)of attenuation:

```
f1.FilterOrder = 52;
f1.Band = 8;
f1.TransitionWidth = .05;
eq1 = design(f1,'equiripple','SystemObject',true);
eq2 = design(f1,'equiripple','StopbandShape','linear',...
    'StopbandDecay',20,'SystemObject',true);
eq3 = design(f1,'equiripple','StopbandShape','linear',...
    'StopbandDecay',40,'SystemObject',true);
fvt = fvtool(eq1,eq2,eq3,'Color','white');
```

```
legend(fvt,'Slope=0','Slope=20','Slope=40')
```
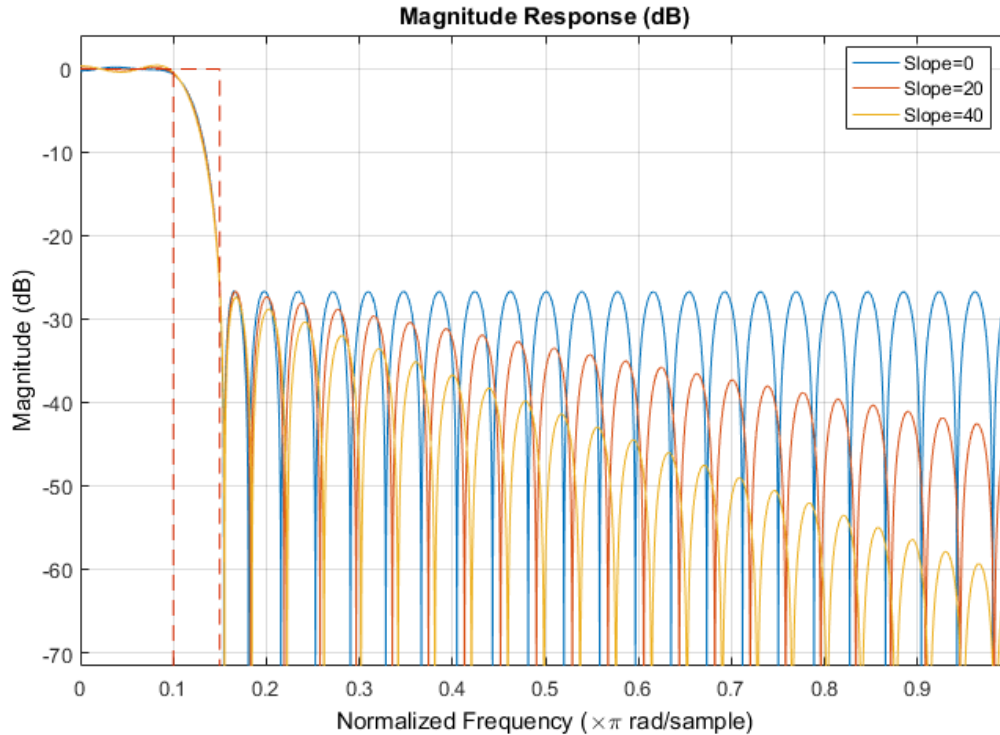

Minimum-Phase Design

We can design a minimum-phase spectral factor of the overall Nyquist filter (a square-root in the frequency domain). This spectral factor can be used in a similar manner to the square-root raised-cosine filter in matched filtering applications. A square-root of the filter is placed on the transmiter's end and the other square root is placed at the receiver's end.

```
f1.FilterOrder = 30;
f1.Band = NBand;
f1.TransitionWidth = TW;
eq1 = design(f1,'equiripple','Minphase',true,'SystemObject',true);
coeffs = rcosdesign(R,N/NBand,NBand);
coeffs = coeffs / max(coeffs) * (-1/(pi*NBand) * (pi*(R-1) - 4*R));
```

```
srrc   = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq1,srrc,'Color','white');
legend(fvt,'Minimum-phase equiripple design',...
    'Square-root raised-cosine design');
```
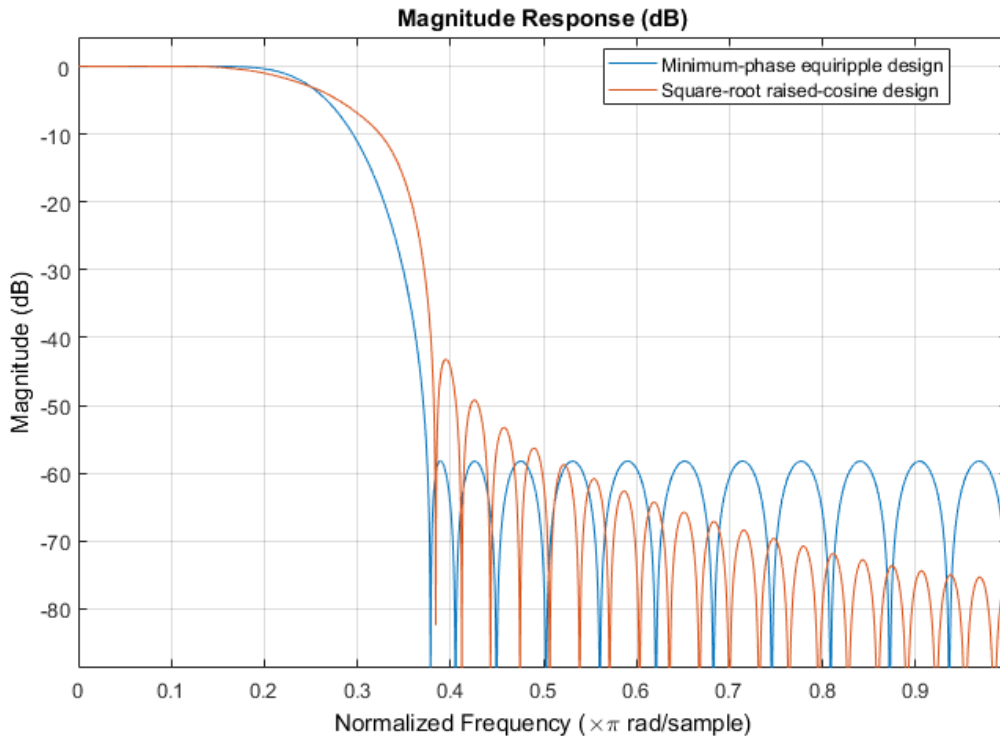

Decreasing the Rolloff Factor

The response of the raised-cosine filter improves as the rolloff factor decreases (shown here for rolloff = 0.2). This is because of the narrow main lobe of the frequency response of a rectangular window that is used in the truncation of the impulse response.

```
f1.FilterOrder = N;
f1.TransitionWidth = .1;
eq1 = design(f1,'equiripple','Zerophase',true,'SystemObject',true);
R = 0.2;
coeffs = rcosdesign(R,N/NBand,NBand,'normal');
coeffs = coeffs/max(abs(coeffs))/NBand;
```

```
rc1    = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq1,rc1,'Color','white');
legend(fvt,'NYQUIST equiripple design','Raised Cosine design');
```
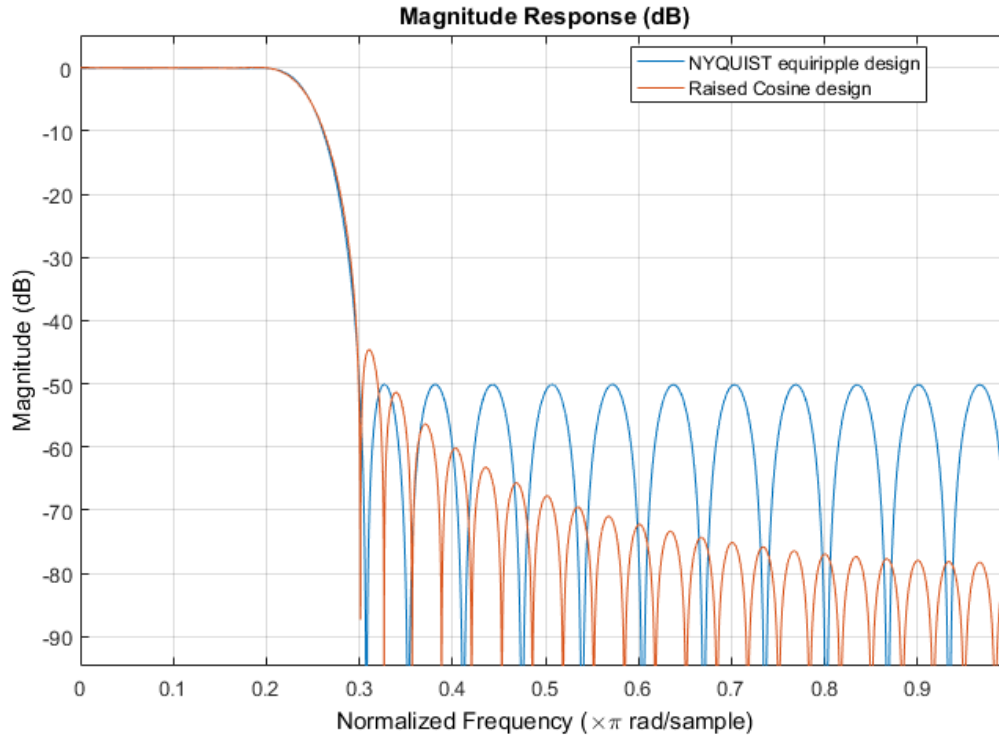

Windowed-Impulse-Response Nyquist Design

Nyquist filters can also be designed using the truncated-and-windowed impulse response method. This can be another alternative to the raised-cosine design. For example we can use the Kaiser window method to design a filter that meets the initial specs:

```
f1.TransitionWidth = TW;
kaiserFilt = design(f1,'kaiserwin','SystemObject',true);
```

The Kaiser window design requires the same order (60) as the equiripple design to meet the specs. (Remember that in contrast we required an extraordinary 1400th-order raised-cosine filter to meet the stopband spec.)

```
fvt = fvtool(eq,rc,kaiserFilt,'Color','white');
legend(fvt,'Equiripple design',...
    'Raised Cosine design','Kaiser window design');
```
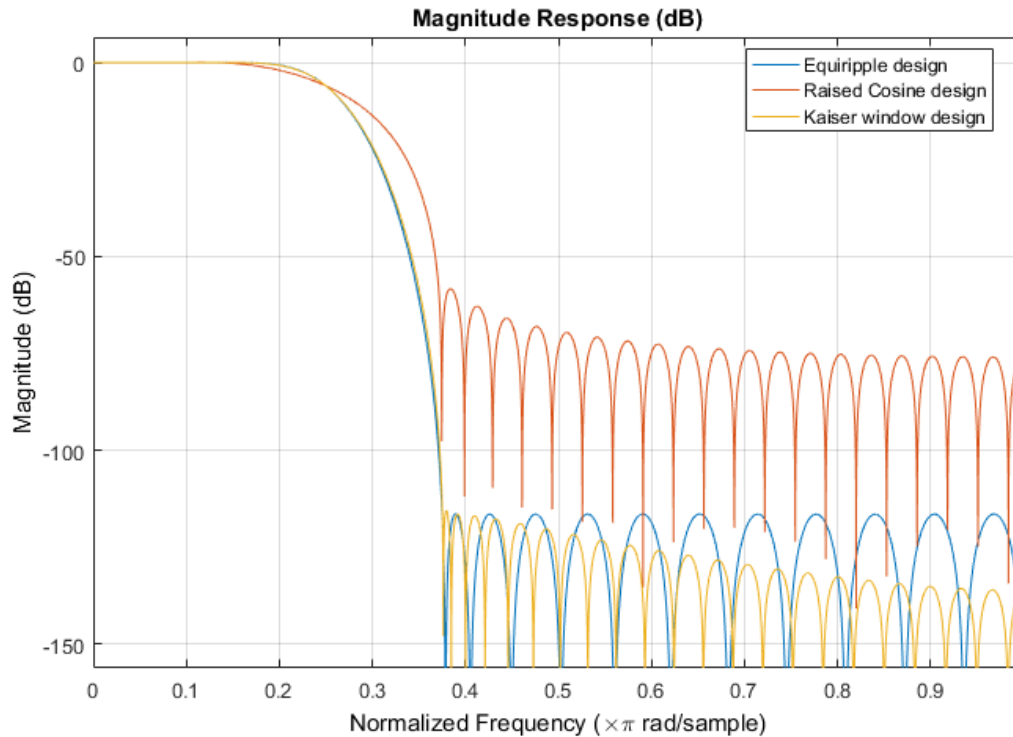

Nyquist Filters for Interpolation

Besides digital data transmission, Nyquist filters are attractive for interpolation purposes. The reason is that every L samples you have a zero sample (except for the middle sample) as mentioned before. There are two advantages to this, both are obvious by looking at the polyphase representation.

```
fm = fdesign.interpolator(4,'nyquist');
kaiserFilt = design(fm,'kaiserwin','SystemObject',true);
fvt = fvtool(kaiserFilt,'Color','white');
fvt.PolyphaseView = 'on';
```
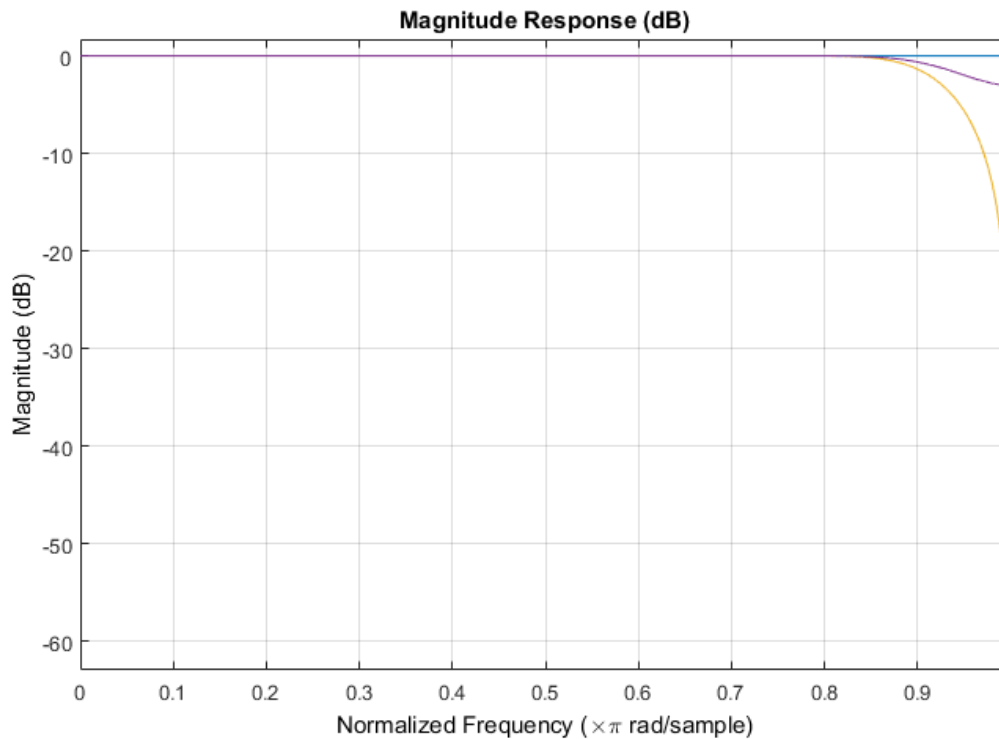
The polyphase subfilter #4 is an allpass filter, in fact it is a pure delay (select impulse response in FVTool, or look at the filter coefficients in FVTool), so that: 1. All of its multipliers are zero except for one, leading to an efficient implementation of that polyphase branch. 2. The input samples are passed through the interpolation filter without modification, even though the filter is not ideal.

Digital Frequency Transformations

Details and Methodology

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

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

$$\alpha_0 = 1$$

where

$H_o(z)$ — Transfer function of the prototype filter

$H_A(z)$ — Transfer function of the allpass mapping filter

$H_T(z)$ — Transfer function of the target filter

Let's look at a simple example of the transformation given by

$$H_T(z) = H_o(-z)$$

The target filter has its poles and zeroes flipped across the origin of the real and imaginary axes. For the real filter prototype, it gives a mirror effect against 0.5, which means that lowpass $H_o(z)$ gives rise to a real highpass $H_T(z)$. This is shown in the following figure for the prototype filter designed as a third-order halfband elliptic filter.

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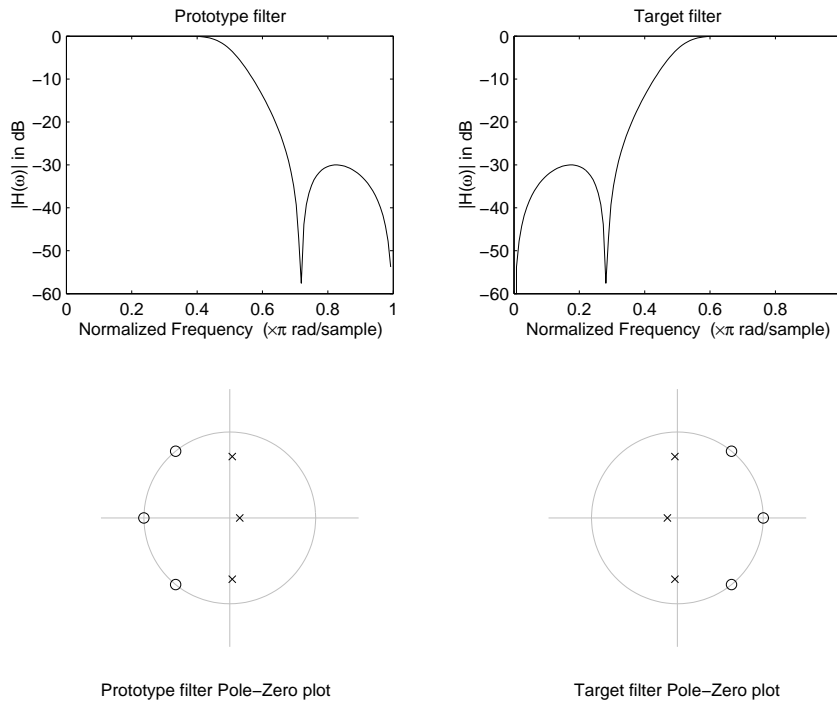
Example of a Simple Mirror Transformation

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.

4-87
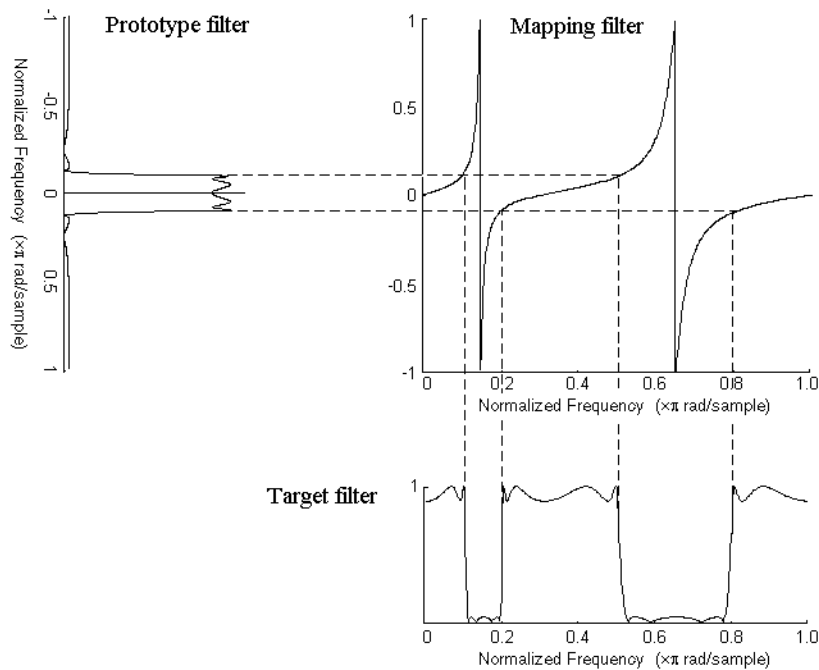
Graphical Interpretation of the Mapping Process

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].

Select 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.
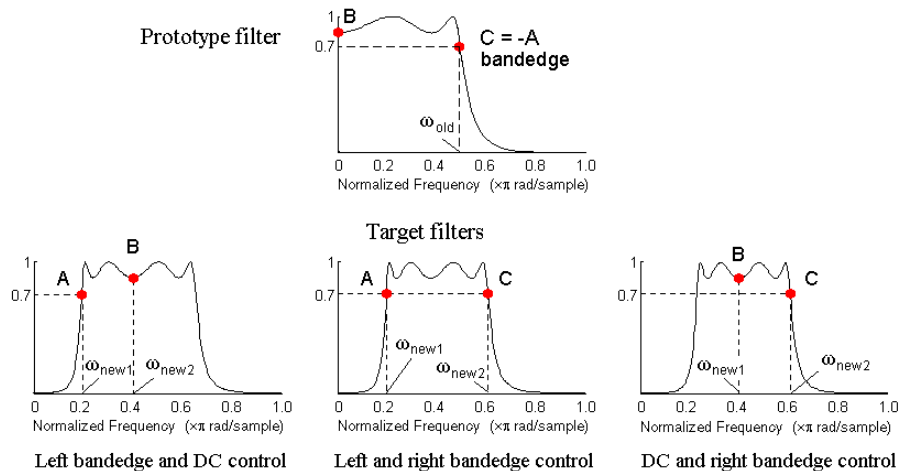
Feature Selection for Real Lowpass to Bandpass Transformation

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.
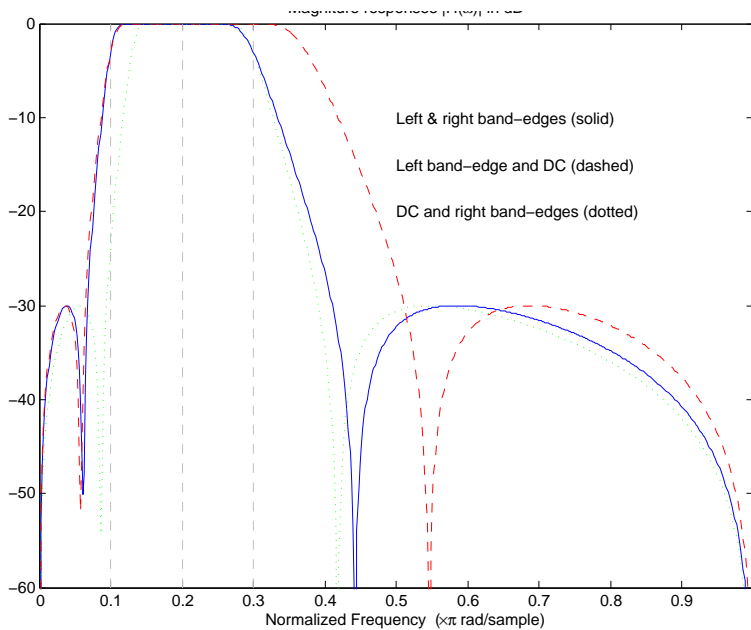
Result of Choosing Different Features

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 4-85. 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

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

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:

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

The root finding needs to be used on the bracketed expressions in order to find the poles and zeros of the target filter.

When the prototype filter is described in the numerator-denominator form:

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

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

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

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

$$
\alpha = \begin{cases} \dfrac{\cos\dfrac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\cos\dfrac{\pi}{2}\omega_{old}} & for & \left| \cos\dfrac{\pi}{2}(\omega_{old} - 2\omega_{new}) \right| < 1 \\[4ex] \dfrac{\sin\dfrac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\sin\dfrac{\pi}{2}\omega_{old}} & otherwise \end{cases}
$$

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);
```

Prototype filter

Target filter
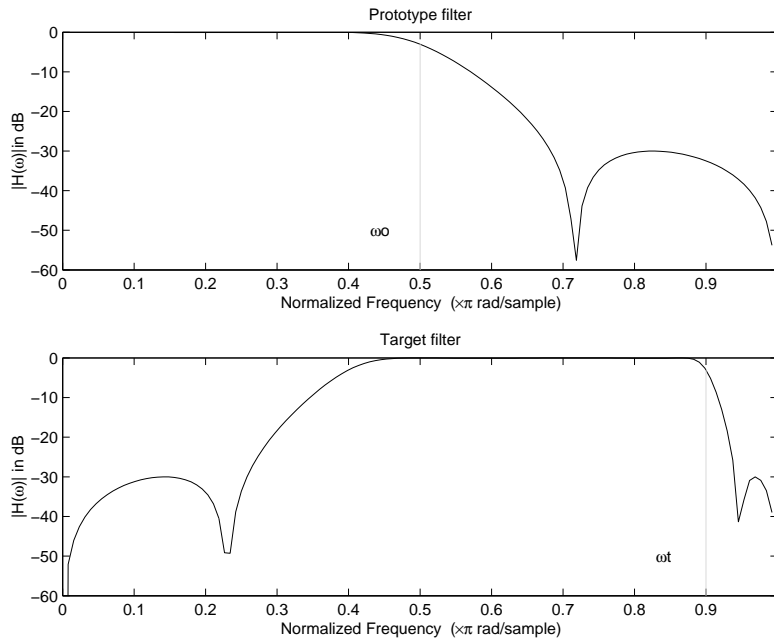
Example of Real Frequency Shift Mapping

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 a given by

$$\alpha = \frac{\sin\dfrac{\pi}{2}(\omega_{old} - \omega_{new})}{\sin\dfrac{\pi}{2}(\omega_{old} - \omega_{new})}$$

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);
```
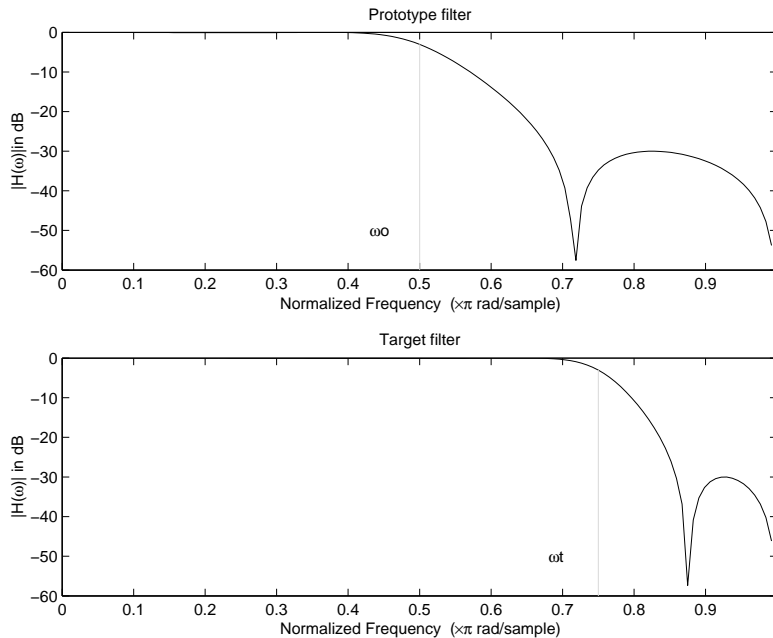
Example of Real Lowpass to Real Lowpass Mapping

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

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

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);
```
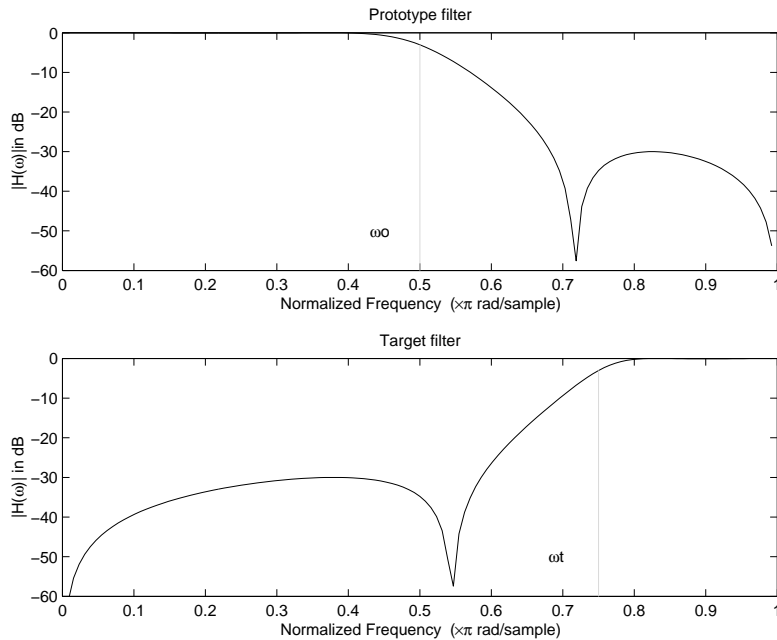
Example of Real Lowpass to Real Highpass Mapping

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

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

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

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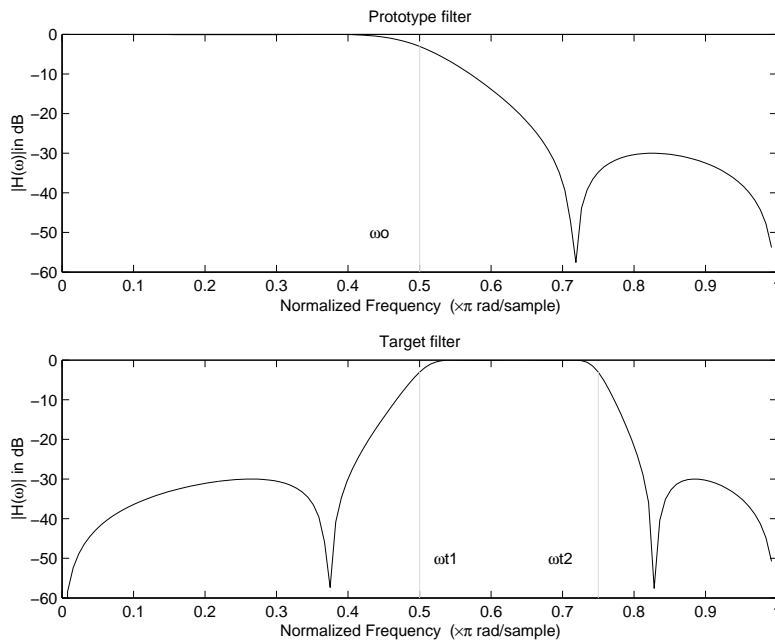
Example of Real Lowpass to Real Bandpass Mapping

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

$$\alpha = \frac{\cos\dfrac{\pi}{4}(2\omega_{old} + \omega_{new,2} - \omega_{new,1})}{\cos\dfrac{\pi}{4}(2\omega_{old} - \omega_{new,2} + \omega_{new,1})}$$

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

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 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]);
```
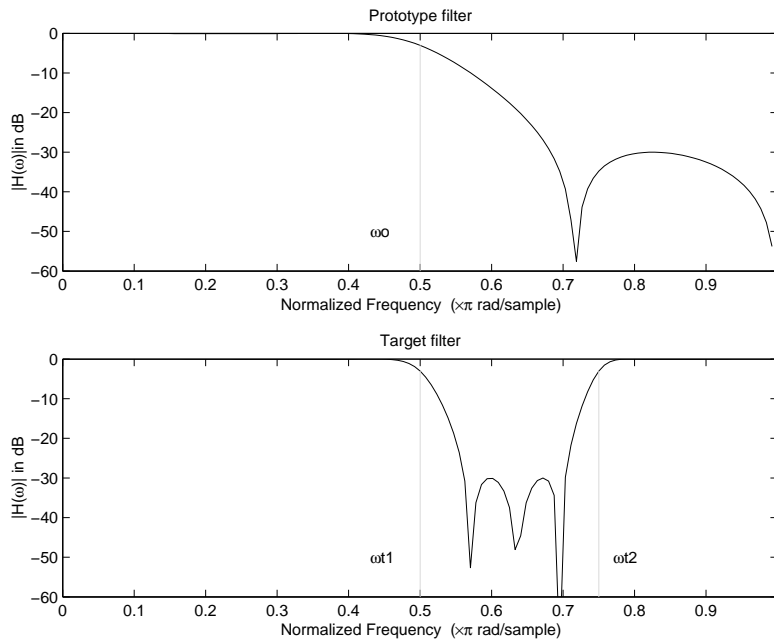
Example of Real Lowpass to Real Bandstop Mapping

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

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

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');
```

Prototype filter

Target filter
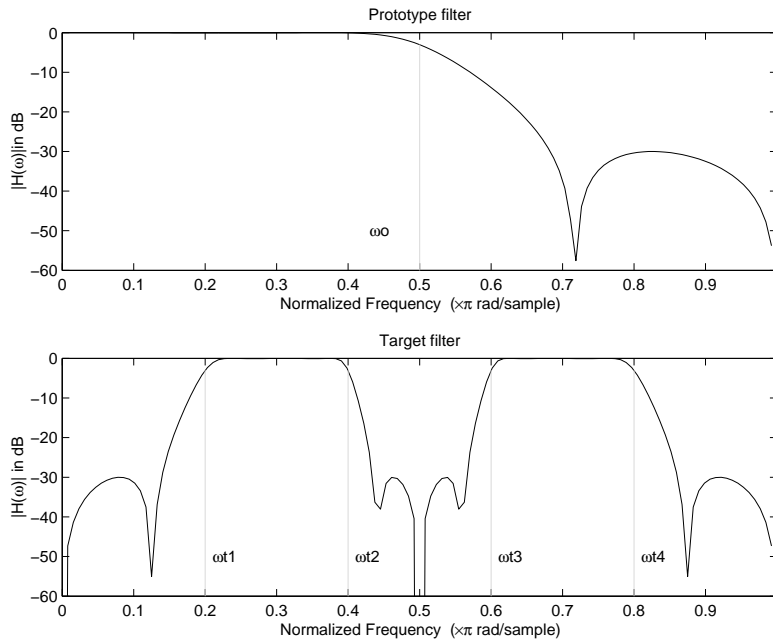
Example of Real Lowpass to Real Multiband Mapping

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 Nth-order multipoint frequency transformation the coefficients α are

$$\begin{cases} \sum_{i=1}^{N} \alpha_{N-i} z_{old,k} \cdot z_{new,k}^{i} - S \cdot z_{new,k}^{N-i} = -z_{old,k} - S \cdot z_{new,k} \\ z_{old,k} = e^{j\pi\omega_{old,k}} \\ z_{new,k} = e^{j\pi\omega_{new,k}} \\ k = 1,\ldots,N \end{cases}$$

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');
```
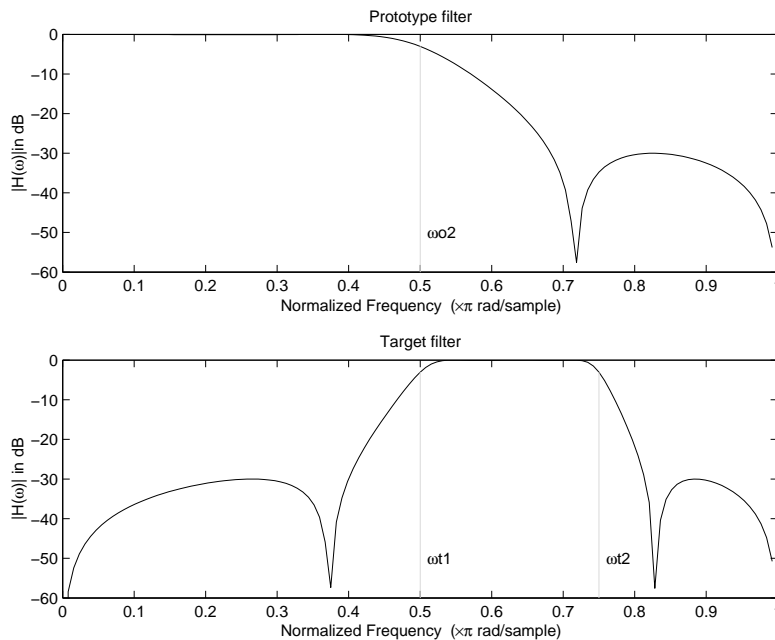
Example of Real Lowpass to Real Multipoint Mapping

Frequency Transformations for Complex Filters

- "Overview" on page 4-107
- "Complex Frequency Shift" on page 4-108
- "Real Lowpass to Complex Bandpass" on page 4-109
- "Real Lowpass to Complex Bandstop" on page 4-111
- "Real Lowpass to Complex Multiband" on page 4-112
- "Real Lowpass to Complex Multipoint" on page 4-114
- "Complex Bandpass to Complex Bandpass" on page 4-116

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

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

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);
```
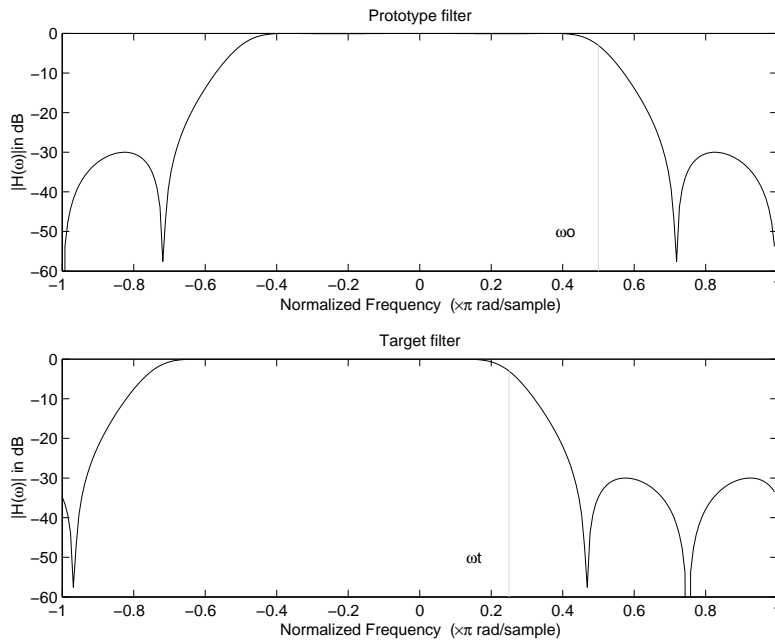
Example of Complex Frequency Shift Mapping

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

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

$$\beta = e^{-j\pi(\omega_{new} - \omega_{old})}$$

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]);
```
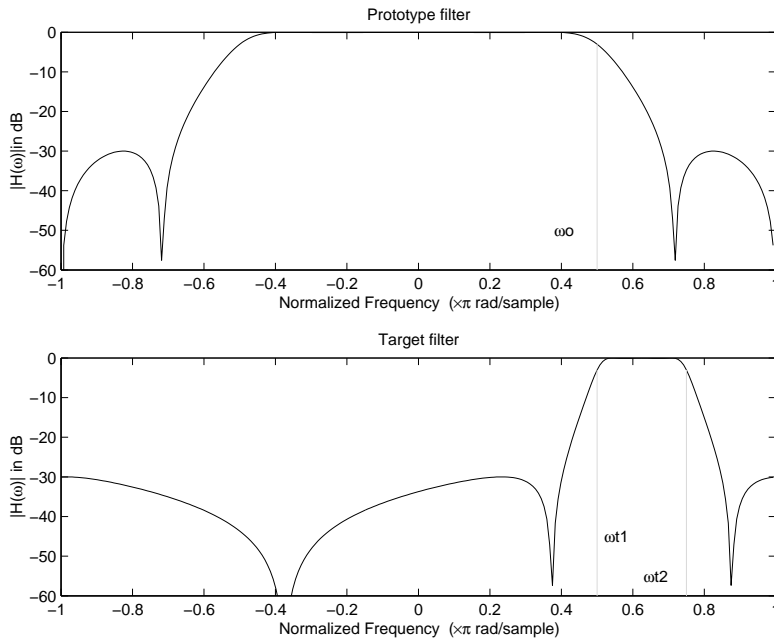

Example of Real Lowpass to Complex Bandpass Mapping

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

$$\alpha = \frac{\cos \pi (2\omega_{old} + v_{new,2} - v_{new,1})}{\cos \pi (2\omega_{old} - v_{new,2} + v_{new,1})}$$

$$\beta = e^{-j\pi(\omega_{new} - \omega_{old})}$$

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]);
```
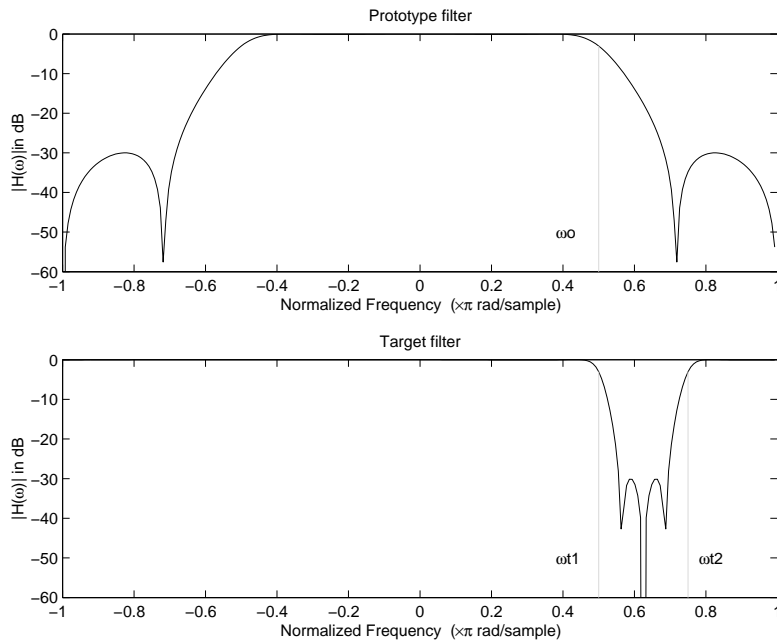
Example of Real Lowpass to Complex Bandstop Mapping

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 Nth-order complex allpass mapping filter is given by the following general transfer function form:

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

$$\alpha_0 = 1$$

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

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

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.

Prototype filter / Target filter
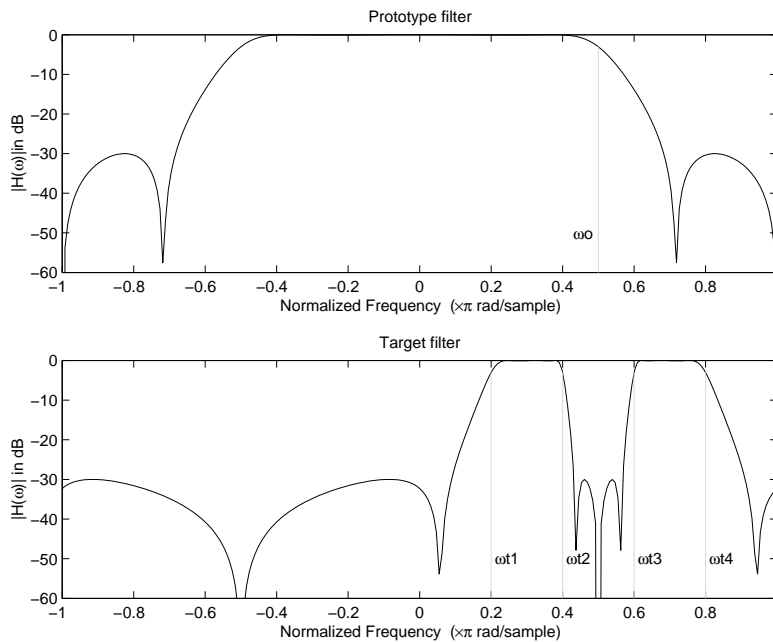
Example of Real Lowpass to Complex Multiband Mapping

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.

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

$$\alpha_0 = 1$$

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

$$\begin{cases} \displaystyle\sum_{i=1}^{N} \Re(\alpha_i) \cdot \left[\cos\beta_{1,k} - \cos\beta_{2,k} \right] + \Im(\alpha_i) \cdot \left[\sin\beta_{1,k} + \sin\beta_{2,k} \right] = \cos\beta_{3,k} \\[4mm] \displaystyle\sum_{i=1}^{N} \Re(\alpha_i) \cdot \left[\sin\beta_{1,k} - \sin\beta_{2,k} \right] - \Im(\alpha_i) \cdot \left[\cos\beta_{1,k} + \cos\beta_{2,k} \right] = \sin\beta_{3,k} \\[4mm] \beta_{1,k} = -\frac{\pi}{2} \left[\omega_{old,k} + \omega_{new,k}(N-k) \right] \\[4mm] \beta_{2,k} = -\frac{\pi}{2} \left[2\Delta C + \omega_{new,k} k \right] \\[4mm] \beta_{3,k} = -\frac{\pi}{2} \left[\omega_{old,k} + \omega_{new,k} N \right] \\[4mm] k = 1 \dots N \end{cases}$$

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

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]);
```
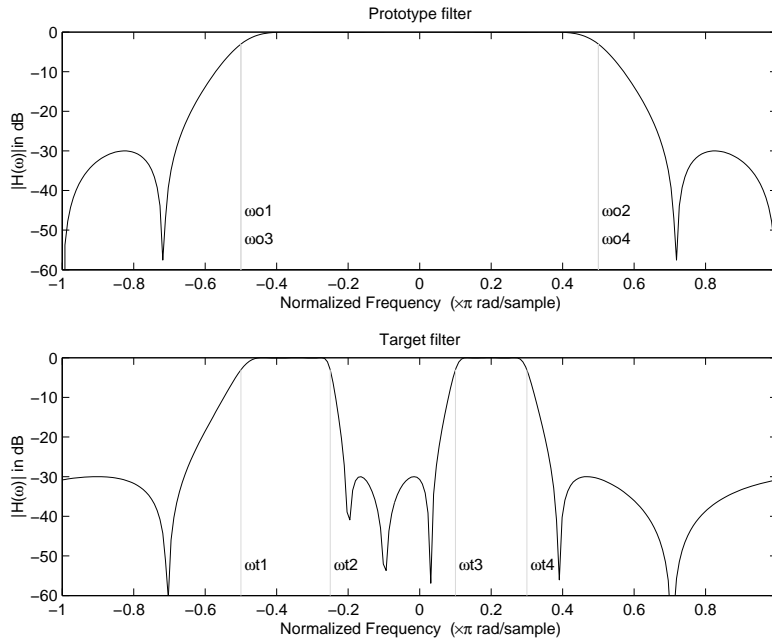

Example of Real Lowpass to Complex Multipoint Mapping

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

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

$$\alpha = e^{-j\pi(\omega_{old,2} - \omega_{old,1})}$$

$$\gamma = e^{-j\pi(\omega_{new,2} - \omega_{new,1})}$$

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]);
```

4-117
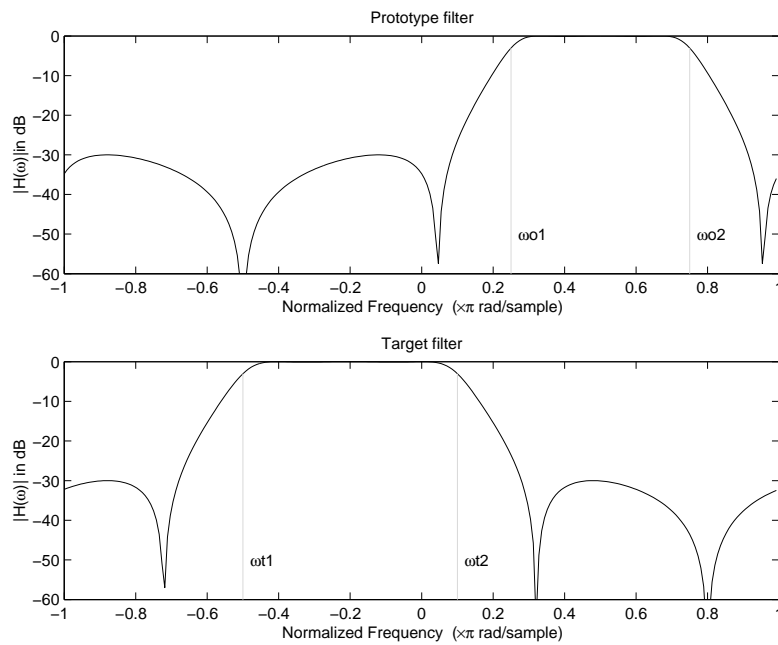
Example of Complex Bandpass to Complex Bandpass Mapping

Digital Filter Design Block

Overview of the Digital Filter Design Block

You can use the `Digital Filter Design` block to design and implement a digital filter. The filter you design can filter single-channel or multichannel signals. The Digital Filter Design block is ideal for simulating the numerical behavior of your filter on a floating-point system, such as a personal computer or DSP chip. You can use the Simulink Coder product to generate C code from your filter block. For more information on generating C code from models, see "Understanding C Code Generation" on page 10-2.

Filter Design and Analysis

You perform all filter design and analysis within the Filter Design and Analysis Tool (FDATool) GUI, which opens when you double-click the Digital Filter Design block. FDATool provides extensive filter design parameters and analysis tools such as pole-zero and impulse response plots.

Filter Implementation

Once you have designed your filter using FDATool, the block automatically realizes the filter using the filter structure you specify. You can then use the block to filter signals in your model. You can also fine-tune the filter by changing the filter specification parameters during a simulation. The outputs of the Digital Filter Design block numerically match the outputs of the DSP System Toolbox `filter` function and the MATLAB `filter` function.

Saving, Exporting, and Importing Filters

The Digital Filter Design block allows you to save the filters you design, export filters (to the MATLAB workspace, MAT-files, etc.), and import filters designed elsewhere.

To learn how to save your filter designs, see "Saving and Opening Filter Design Sessions" on page 15-33 in the Signal Processing Toolbox documentation. To learn how to import and export your filter designs, see "Import and Export Quantized Filters" on page 4-47.

Note: You can use the Digital Filter Design block to design and implement a filter. To implement a pre-designed filter, use the Discrete FIR Filter or Biquad Filter blocks. Both methods implement a filter design in the same manner and have the same behavior during simulation and code generation.

See the `Digital Filter Design` block reference page for more information. For information on choosing between the Digital Filter Design block and the Filter Realization Wizard, see "Select a Filter Design Block" on page 4-120.

Select a Filter Design Block

This section explains the similarities and differences between the Digital Filter Design and Filter Realization Wizard blocks.

Similarities

The Digital Filter Design block and Filter Realization Wizard are similar in the following ways:

- Filter design and analysis options — Both blocks use the Filter Design and Analysis Tool (FDATool) GUI for filter design and analysis.
- Output values — If the output of both blocks is double-precision floating point, single-precision floating point, or fixed point, the output values of both blocks numerically match the output of the `filter` method of the `dfilt` object.

Differences

The Digital Filter Design block and Filter Realization Wizard handle the following things differently:

- Supported filter structures — Both blocks support many of the same basic filter structures, but the Filter Realization Wizard supports more structures than the Digital Filter Design block. This is because the block can implement filters using Sum, Gain, and Delay blocks. See the `Filter Realization Wizard` and `Digital Filter Design` block reference pages for a list of all the structures they support.

- Data type support — The Filter Realization Wizard block supports single- and double-precision floating-point computation for all filter structures and fixed-point computation for some filter structures. The Digital Filter Design block only supports single- and double-precision floating-point computation.

- Block versus Wizard — The Digital Filter Design block is the filter itself, but the Filter Realization Wizard block just enables you to create new filters and put them in an existing model. Thus, the Filter Realization Wizard is not a block that processes data in your model, it is a wizard that generates filter blocks (or subsystems) which you can then use to process data in your model.

When to Use Each Block

The following are specific situations where only the Digital Filter Design block or the Filter Realization Wizard is appropriate.

- Digital Filter Design

  - Use to simulate single- and double-precision floating-point filters.

  - Use to generate highly optimized ANSI® C code that implements floating-point filters for embedded systems.

- Filter Realization Wizard

  - Use to simulate numerical behavior of fixed-point filters in a DSP chip, a field-programmable gate array (FPGA), or an application-specific integrated circuit (ASIC).

  - Use to simulate single- and double-precision floating-point filters with structures that the Digital Filter Design block does not support.

  - Use to visualize the filter structure, as the block can build the filter from Sum, Gain, and Delay blocks.

  - Use to rapidly generate multiple filter blocks.

See "Filter Realization Wizard" on page 4-129 and the `Filter Realization Wizard` block reference page for information.

Create a Lowpass Filter in Simulink

You can use the Digital Filter Design block to design and implement a digital FIR or IIR filter. In this topic, you use it to create an FIR lowpass filter:

1   Open Simulink and create a new model file.

2   From the DSP System Toolbox Filtering library, and then from the Filter Implementations library, click-and-drag a Digital Filter Design block into your model.

3   Double-click the Digital Filter Design block.

The Filter Design and Analysis Tool (FDATool) GUI opens.

4   Set the parameters as follows, and then click **OK**:

  · **Response Type** = `Lowpass`
  · **Design Method** = `FIR`, `Equiripple`
  · **Filter Order** = `Minimum order`
  · **Units** = `Normalized (0 to 1)`
  · **wpass** = `0.2`
  · **wstop** = `0.5`

5   Click **Design Filter** at the bottom of the GUI to design the filter.

Your Digital Filter Design block now represents a filter with the parameters you specified.

6   From the **Edit** menu, select **Convert Structure**.

The **Convert Structure** dialog box opens.

7   Select **Direct-Form FIR Transposed** and click **OK**.

8   Rename your block `Digital Filter Design - Lowpass`.

The Digital Filter Design block now represents a lowpass filter with a Direct-Form FIR Transposed structure. The filter passes all frequencies up to 20% of the Nyquist frequency (half the sampling frequency), and stops frequencies greater than or equal to 50% of the Nyquist frequency as defined by the **wpass** and **wstop** parameters. In the next topic, "Create a Highpass Filter in Simulink" on page 4-122, you use a `Digital Filter Design` block to create a highpass filter. For more information about implementing a pre-designed filter, see "Digital Filter Implementations" on page 4-141.

Create a Highpass Filter in Simulink

In this topic, you create a highpass filter using the Digital Filter Design block:

1 If the model you created in "Create a Lowpass Filter in Simulink" on page 4-121 is not open on your desktop, you can open an equivalent model by typing

`ex_filter_ex4`

at the MATLAB command prompt.

2 From the DSP System Toolbox Filtering library, and then from the Filter Implementations library, click-and-drag a second Digital Filter Design block into your model.

3 Double-click the Digital Filter Design block.

The Filter Design and Analysis Tool (FDATool) GUI opens.

4 Set the parameters as follows:

- **Response Type** = `Highpass`
- **Design Method** = `FIR`, `Equiripple`
- **Filter Order** = `Minimum order`
- **Units** = `Normalized (0 to 1)`
- **wstop** = `0.2`
- **wpass** = `0.5`

5 Click the **Design Filter** button at the bottom of the GUI to design the filter.

Your Digital Filter Design block now represents a filter with the parameters you specified.

6 In the **Edit** menu, select **Convert Structure**.

The **Convert Structure** dialog box opens.

7 Select **Direct-Form FIR Transposed** and click **OK**.

8 Rename your block `Digital Filter Design - Highpass`.

The block now implements a highpass filter with a direct form FIR transpose structure. The filter passes all frequencies greater than or equal to 50% of the Nyquist frequency (half the sampling frequency), and stops frequencies less than or equal to 20% of the Nyquist frequency as defined by the **wpass** and **wstop** parameters. This highpass filter is the opposite of the lowpass filter described in "Create a Lowpass Filter in Simulink" on page 4-121. The highpass filter passes the frequencies stopped by the lowpass filter, and stops the frequencies passed by the lowpass filter. In the next topic, "Filter High-Frequency Noise in Simulink" on page 4-124, you use these `Digital Filter`

`Design` blocks to create a model capable of removing high frequency noise from a signal. For more information about implementing a pre-designed filter, see "Digital Filter Implementations" on page 4-141.

Filter High-Frequency Noise in Simulink

In the previous topics, you used Digital Filter Design blocks to create FIR lowpass and highpass filters. In this topic, you use these blocks to build a model that removes high frequency noise from a signal. In this model, you use the highpass filter, which is excited using a uniform random signal, to create high-frequency noise. After you add this noise to a sine wave, you use the lowpass filter to filter out the high-frequency noise:

1   If the model you created in "Create a Highpass Filter in Simulink" on page 4-122 is not open on your desktop, you can open an equivalent model by typing

    `ex_filter_ex5`

    at the MATLAB command prompt.

2   Click-and-drag the following blocks into your model.

Block	Library	Quantity
Add	Simulink Math Operations library	1
Random Source	Sources	1
Sine Wave	Sources	1
Time Scope	Sinks	1

3   Set the parameters for these blocks as indicated in the following table. Leave the parameters not listed in the table at their default settings.

Parameter Settings for the Other Blocks

Block	Parameter Setting
Add	• **Icon shape** = `rectangular` • **List of signs** = ++
Random Source	• **Source type =** = `Uniform` • **Minimum** = 0 • **Maximum** = 4 • **Sample mode** = `Discrete`

Block	Parameter Setting
	• **Sample time** = 1/1000
	• **Samples per frame** = 50
Sine Wave	• **Frequency (Hz)** = 75
	• **Sample time** = 1/1000
	• **Samples per frame** = 50
Time Scope	• **File** > **Number of Input Ports** > **3**
	• **File** > **Configuration ...**
	• Open the **Visuals:Time Domain Options** dialog and set **Time span** = One frame period

4 Connect the blocks as shown in the following figure. You might need to resize some of the blocks to accomplish this task.
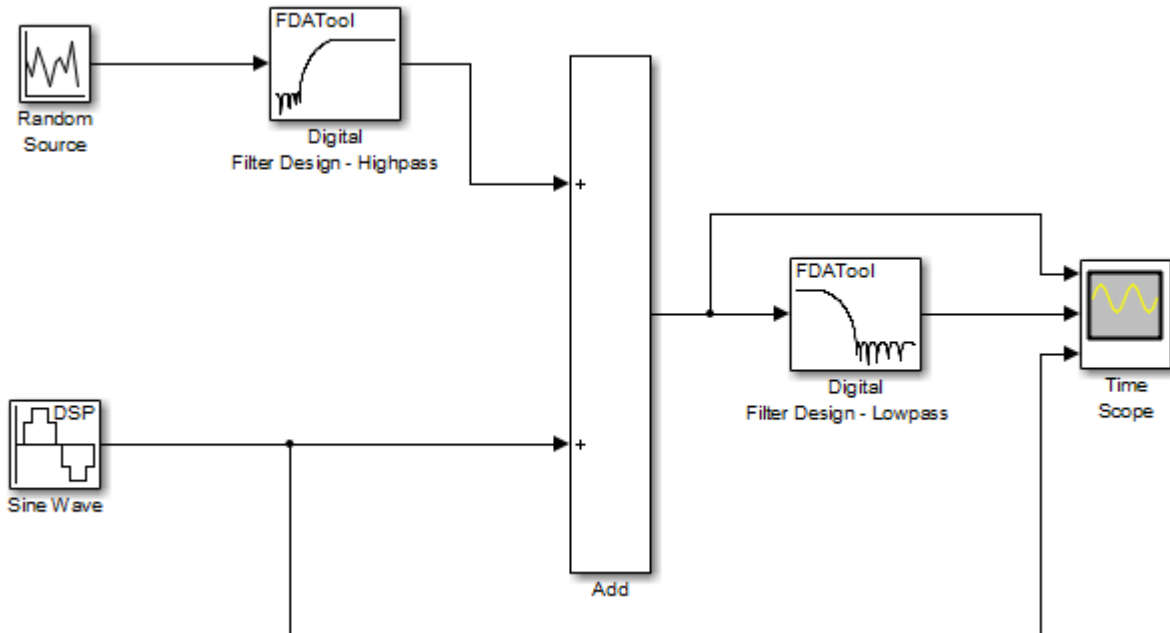
5 From the Simulation menu, select **Model Configuration Parameters**.

The **Configuration Parameters** dialog box opens.

6 In the **Solver** pane, set the parameters as follows, and then click **OK**:

- **Start time** = 0
- **Stop time** = 5
- **Type** = Fixed-step
- **Solver** = Discrete (no continuous states)

7 In the model window, from the **Simulation** menu, choose **Run**.

The model simulation begins and the scope displays the three input signals.

8 After simulation is complete, select **View** > **Legend** from the Time Scope menu. The legend appears in the Time Scope window. You can click-and-drag it anywhere on the scope display. To change the channel names, double-click inside the legend and replace the default channel names with the following:

- Add = Noisy Sine Wave
- Digital Filter Design – Lowpass = Filtered Noisy Sine Wave
- Sine Wave = Original Sine Wave

In the next step, you will set the color, style, and marker of each channel.

9 In the Time Scope window, select **View** > **Line Properties**, and set the following:

Line	Style	Marker	Color
Noisy Sine Wave	-	**None**	**Black**
Filtered Noisy Sine Wave	-	**diamond**	**Red**
Original Sine Wave	**None**	*****	**Blue**

10 The **Time Scope** display should now appear as follows:
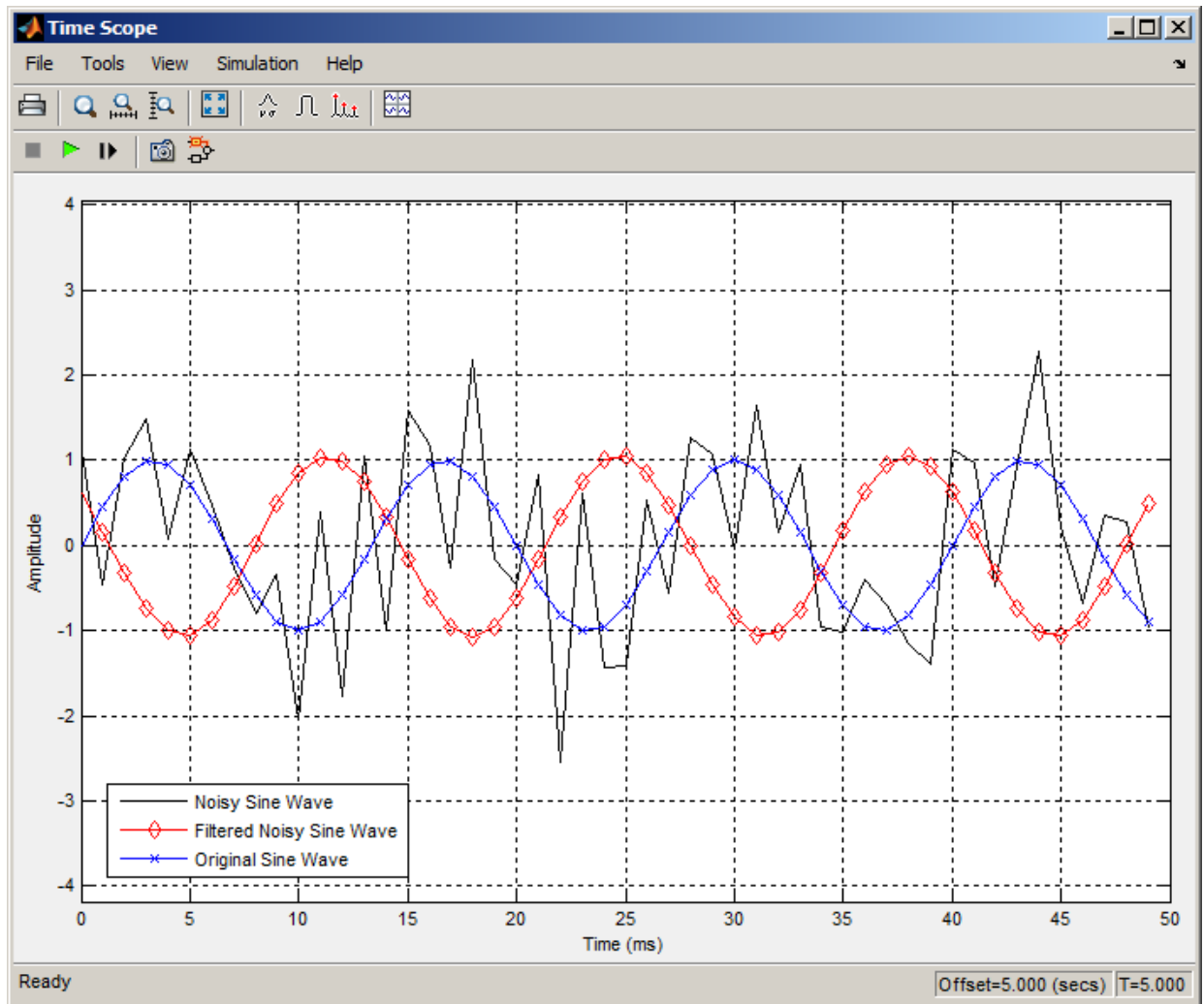
You can see that the lowpass filter filters out the high-frequency noise in the noisy sine wave.

You have now used Digital Filter Design blocks to build a model that removes high frequency noise from a signal. For more information about these blocks, see the `Digital Filter Design` block reference page. For information on another block capable of

designing and implementing filters, see "Filter Realization Wizard" on page 4-129. To learn how to save your filter designs, see "Saving and Opening Filter Design Sessions" on page 15-33 in the Signal Processing Toolbox documentation. To learn how to import and export your filter designs, see "Import and Export Quantized Filters" on page 4-47 in the DSP System Toolbox documentation.

Filter Realization Wizard

In this section...
"Overview of the Filter Realization Wizard" on page 4-129
"Design and Implement a Fixed-Point Filter in Simulink" on page 4-129
"Set the Filter Structure and Number of Filter Sections" on page 4-138
"Optimize the Filter Structure" on page 4-139

Overview of the Filter Realization Wizard

The Filter Realization Wizard is another DSP System Toolbox block that can be used to design and implement digital filters. You can use this tool to filter single-channel floating-point or fixed-point signals. Like the Digital Filter Design block, double-clicking a Filter Realization Wizard block opens FDATool. Unlike the Digital Filter Design block, the Filter Realization Wizard starts FDATool with the **Realize Model** panel selected. This panel is optimized for use with DSP System Toolbox software.

For more information, see the `Filter Realization Wizard` block reference page. For information on choosing between the Digital Filter Design block and the Filter Realization Wizard, see "Select a Filter Design Block" on page 4-120.

Design and Implement a Fixed-Point Filter in Simulink

In this section, a tutorial guides you through creating a fixed-point filter with the Filter Realization Wizard. You will use the Filter Realization Wizard to remove noise from a signal. This tutorial has the following parts:

- "Part 1 — Create a Signal with Added Noise" on page 4-129
- "Part 2 — Create a Fixed-Point Filter with the Filter Realization Wizard" on page 4-131
- "Part 3 — Build a Model to Filter a Signal" on page 4-135
- "Part 4 — Examine Filtering Results" on page 4-137

Part 1 — Create a Signal with Added Noise

In this section of the tutorial, you will create a signal with added noise. Later in the tutorial, you will filter this signal with a fixed-point filter that you design with the Filter Realization Wizard.

1 Type

```
load mtlb
soundsc(mtlb,Fs)
```

at the MATLAB command line. You should hear a voice say "MATLAB." This is the signal to which you will add noise.

2 Create a noise signal by typing

```
noise = cos(2*pi*3*Fs/8*(0:length(mtlb)-1)/Fs)';
```

at the command line. You can hear the noise signal by typing

```
soundsc(noise,Fs)
```

3 Add the noise to the original signal by typing

```
u = mtlb + noise;
```

at the command line.

4 Scale the signal with noise by typing

```
u = u/max(abs(u));
```

at the command line. You scale the signal to try to avoid overflows later on. You can hear the scaled signal with noise by typing

```
soundsc(u,Fs)
```

5 View the scaled signal with noise by typing

```
spectrogram(u,256,[],[],Fs);colorbar
```

at the command line.

The spectrogram appears as follows.
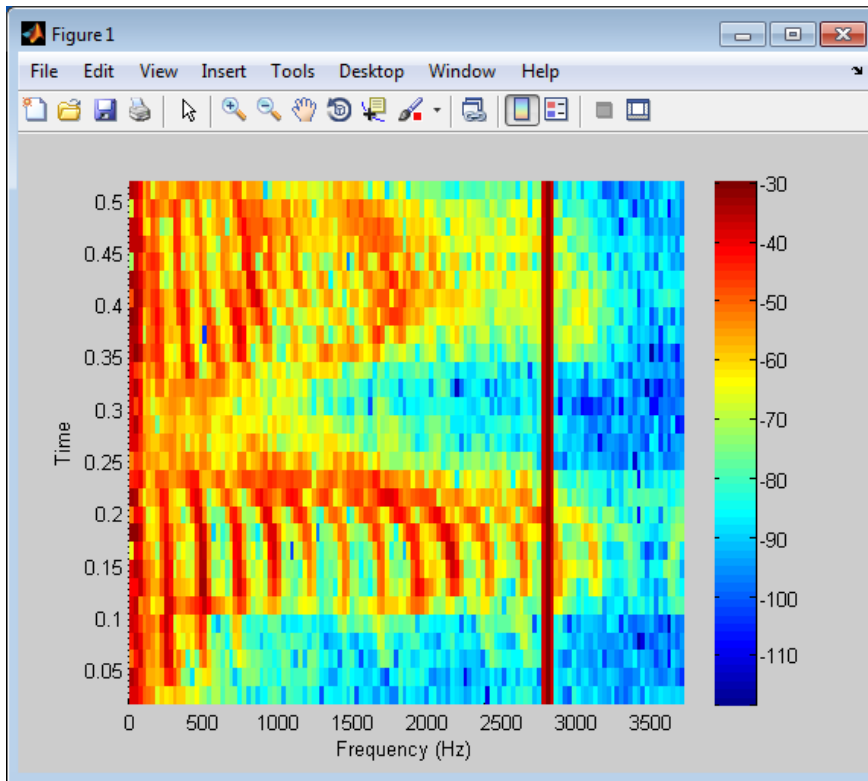
In the spectrogram, you can see the noise signal as a line at about 2800 Hz, which is equal to `3*Fs/8`.

Part 2 — Create a Fixed-Point Filter with the Filter Realization Wizard

Next you will create a fixed-point filter using the Filter Realization Wizard. You will create a filter that reduces the effects of the noise on the signal.

1 Open a new Simulink model, and drag-and-drop a Filter Realization Wizard block from the Filtering / Filter Implementations library into the model.

> **Note:** You do not have to place a Filter Realization Wizard block in a model in order to use it. You can open the GUI from within a library. However, for purposes of this tutorial, we will keep the Filter Realization Wizard block in the model.

2   Double-click the Filter Realization Wizard block in your model. The **Realize Model** panel of the Filter Design and Analysis Tool (FDATool) appears.

3   Click the Design Filter button (🔲) on the bottom left of FDATool. This brings forward the **Design filter** panel of the tool.

4   Set the following fields in the **Design filter** panel:

- Set **Design Method** to IIR -- Constrained Least Pth-norm
- Set **Fs** to Fs
- Set **Fpass** to 0.2*Fs
- Set **Fstop** to 0.25*Fs
- Set **Max pole radius** to 0.8
- Click the **Design Filter** button

The **Design filter** panel should now appear as follows.
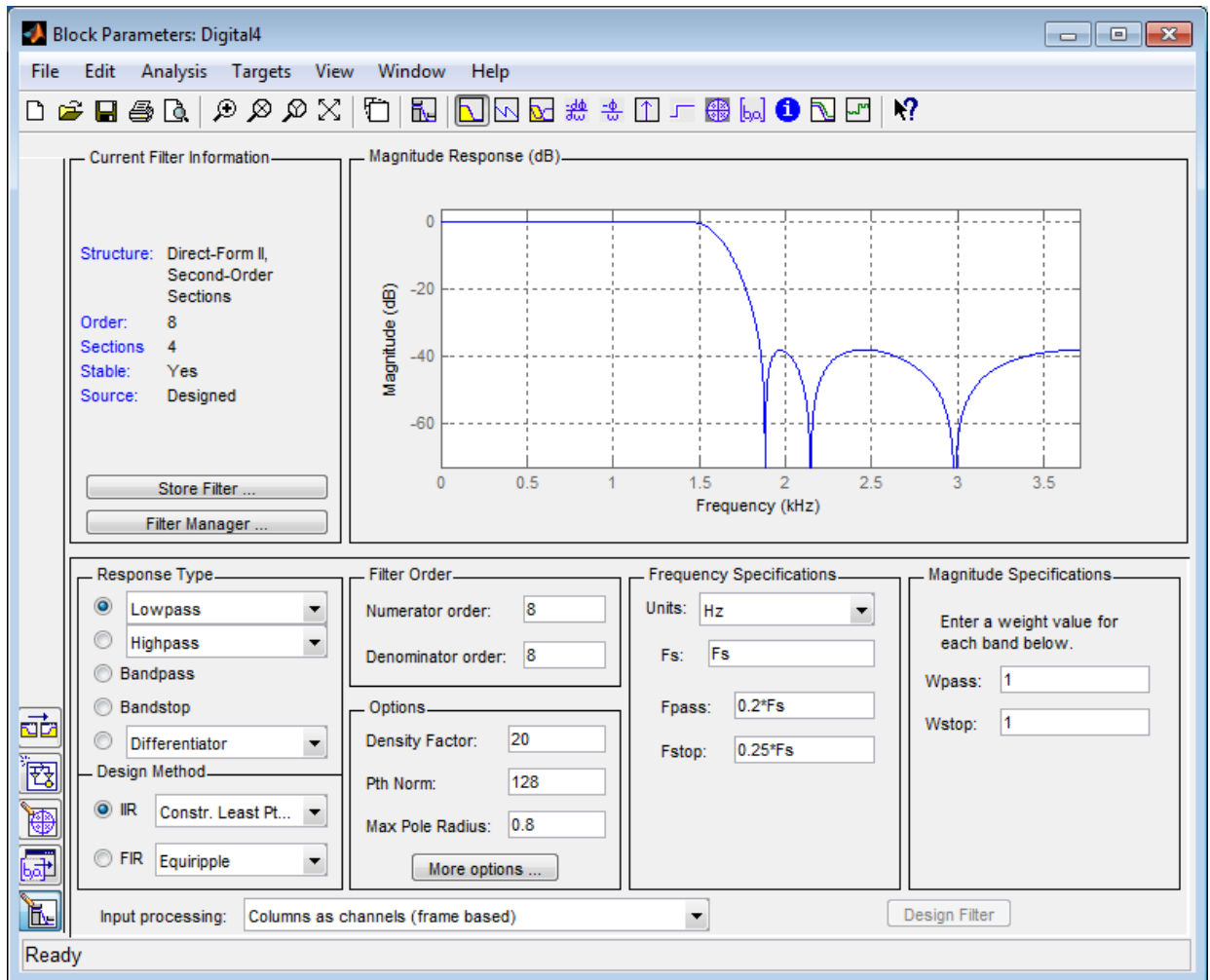
5   Click the **Set quantization parameters** button on the bottom left of FDATool
    ( ). This brings forward the **Set quantization parameters** panel of the tool.

6   Set the following fields in the **Set quantization parameters** panel:

    • Select Fixed-point for the **Filter arithmetic** parameter.

· Make sure the **Best precision fraction lengths** check box is selected on the **Coefficients** pane.

The **Set quantization parameters** panel should appear as follows.
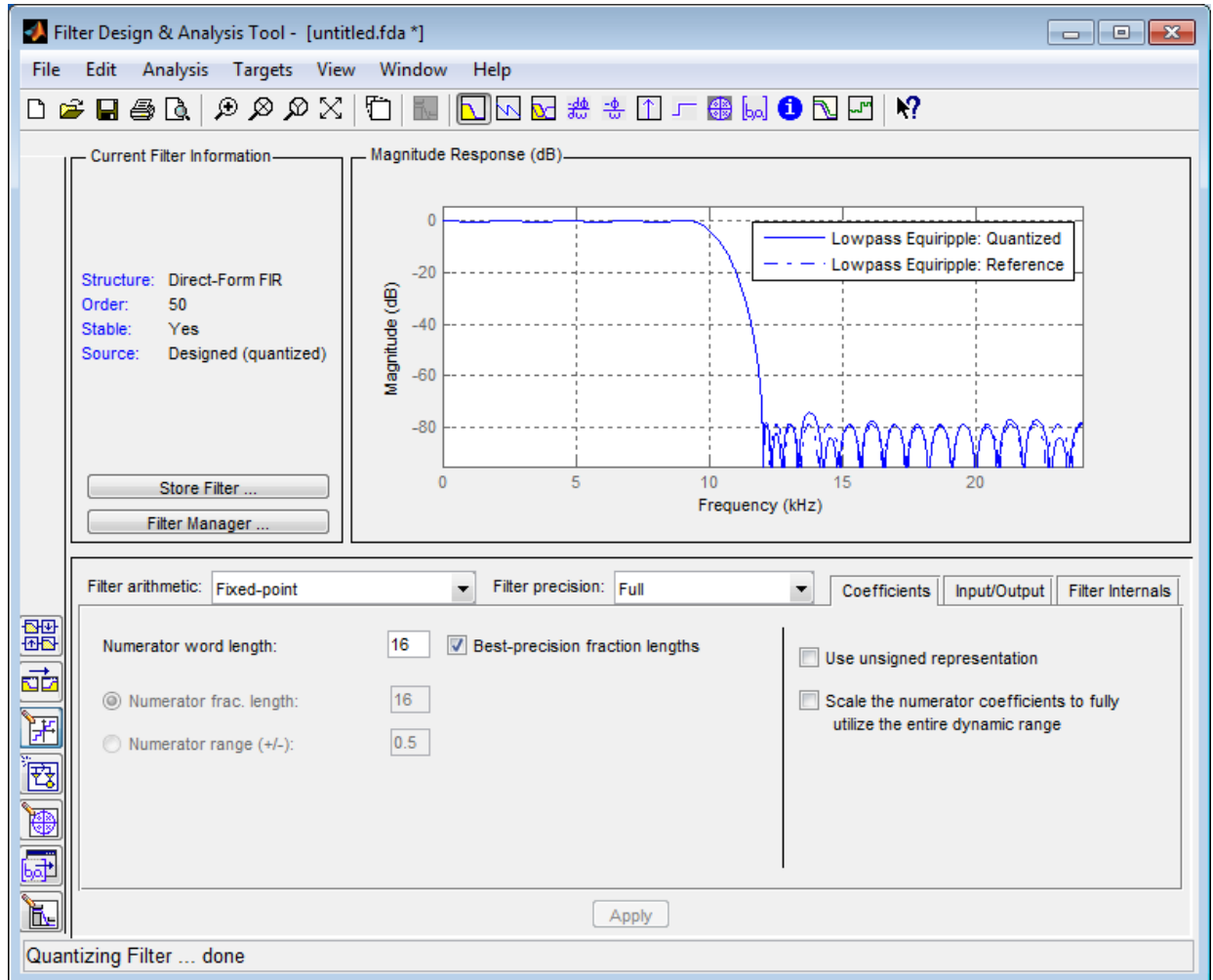
7 Click the Realize Model button on the left side of FDATool (⊞). This brings forward the **Realize Model** panel of the tool.

8 Select the **Build model using basic elements** check box, then click the **Realize Model** button on the bottom of FDATool. A subsystem block for the new filter appears in your model.
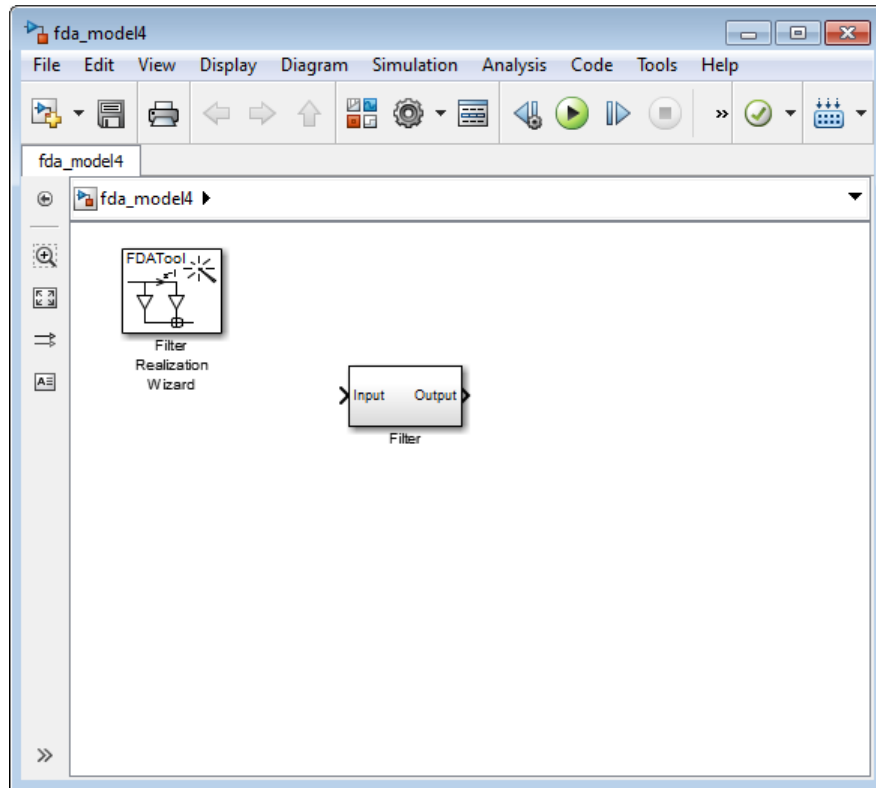
Note: You do not have to keep the Filter Realization Wizard block in the same model as the generated Filter block. However, for this tutorial, we will keep the blocks in the same model.

9 Double-click the `Filter` subsystem block in your model to view the filter implementation.

Part 3 — Build a Model to Filter a Signal

In this section of the tutorial, you will filter noise from a signal in your Simulink model.

1   Connect a `Signal From Workspace` block from the Sources library to the input port of your filter block.

2   Connect a `To Workspace` block from the Sinks library to the output port of your filter block. Your blocks should now be connected as follows.
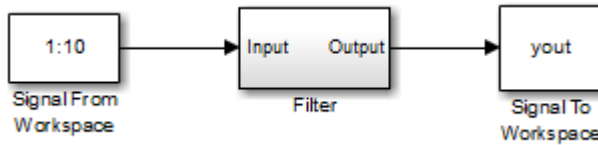
3   Open the Signal From Workspace block dialog box and set the **Signal** parameter to `u`. Click **OK** to save your changes and close the dialog box.

4   Open the **Model Configuration Parameters** dialog box from the **Simulation** menu of the model. In the **Solver** pane of the dialog, set the following fields:

   · **Stop time** = `length(u)-1`
   · **Type** = `Fixed-step`

   Click **OK** to save your changes and close the dialog box.

5   Run the model.

6   From the **Display** menu of the model, select **Signals & Ports** > **Port Data Types**. You can now see that the input to the Filter block is a signal of type `double` and the output of the Filter block has a data type of `sfix16_En11`.
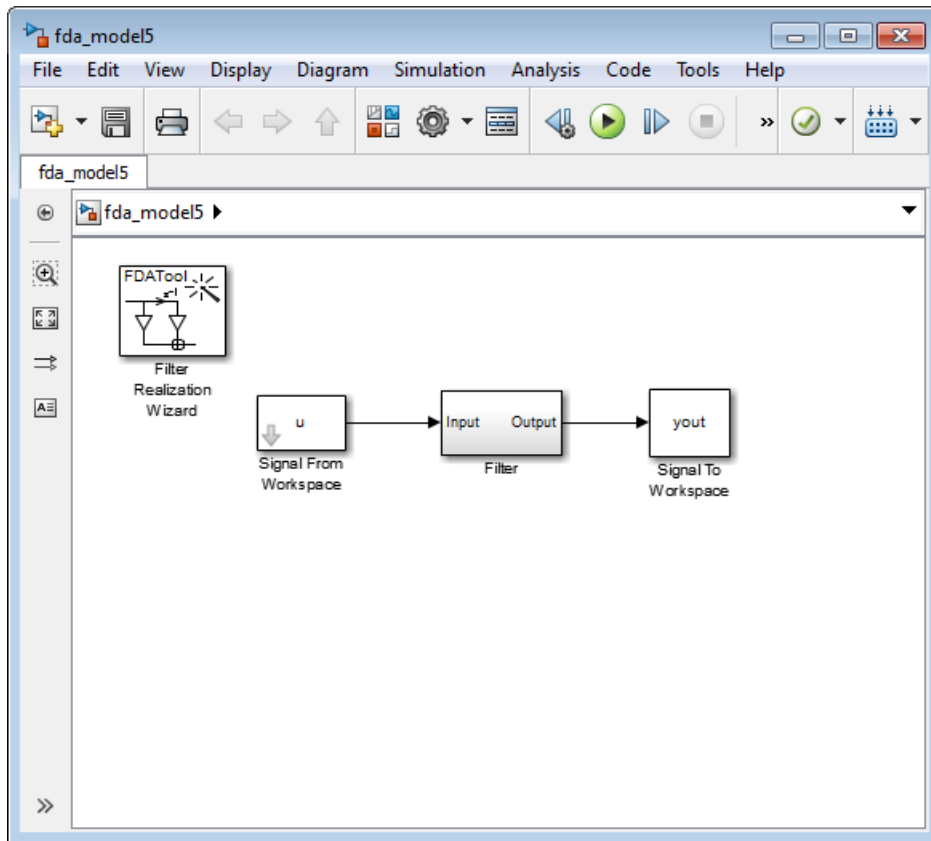
Part 4 — Examine Filtering Results

Now you can listen to and look at the results of the fixed-point filter you designed and implemented.

1   Type

```
soundsc(yout,Fs)
```

at the command line to hear the output of the filter. You should hear a voice say "MATLAB." The noise portion of the signal should be close to inaudible.

2   Type

```
figure
spectrogram(yout,256,[],[],Fs);colorbar
```

at the command line.
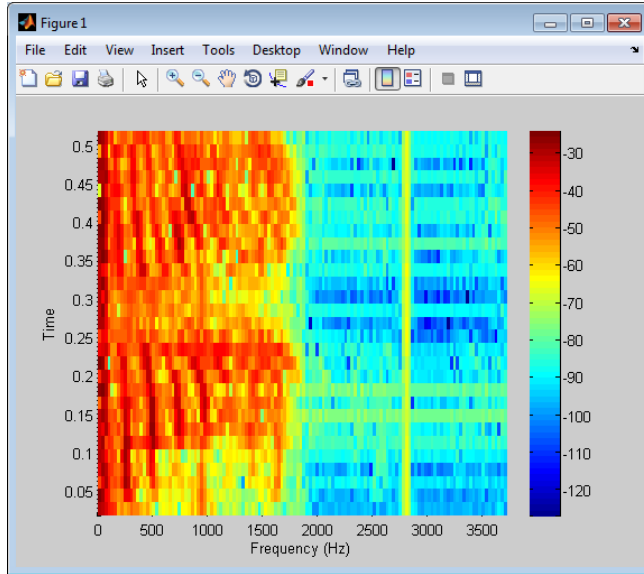
From the colorbars at the side of the input and output spectrograms, you can see that the noise has been reduced by about 40 dB.

Set the Filter Structure and Number of Filter Sections

The **Current Filter Information** region of FDATool shows the structure and the number of second-order sections in your filter.

Change the filter structure and number of filter sections of your filter as follows:

- Select **Convert Structure** from the **Edit** menu to open the **Convert Structure** dialog box. For details, see "Converting to a New Structure" in the Signal Processing Toolbox documentation.
- Select **Convert to Second-order Sections** from the **Edit** menu to open the **Convert to SOS** dialog box. For details, see "Converting to Second-Order Sections" in the Signal Processing Toolbox documentation.

Optimize the Filter Structure

The Filter Realization Wizard can implement a digital filter using either digital filter blocks from the DSP System Toolbox library or by creating a subsystem block that implements the filter using `Sum`, `Gain`, and `Delay` blocks. The following procedure shows you how to optimize the filter implementation:

1 Open the **Realize Model** pane of FDATool by clicking the Realize Model button in the lower-left corner of FDATool.

2 Select the desired optimizations in the **Optimization** region of the **Realize Model** pane. See the following descriptions and illustrations of each optimization option.
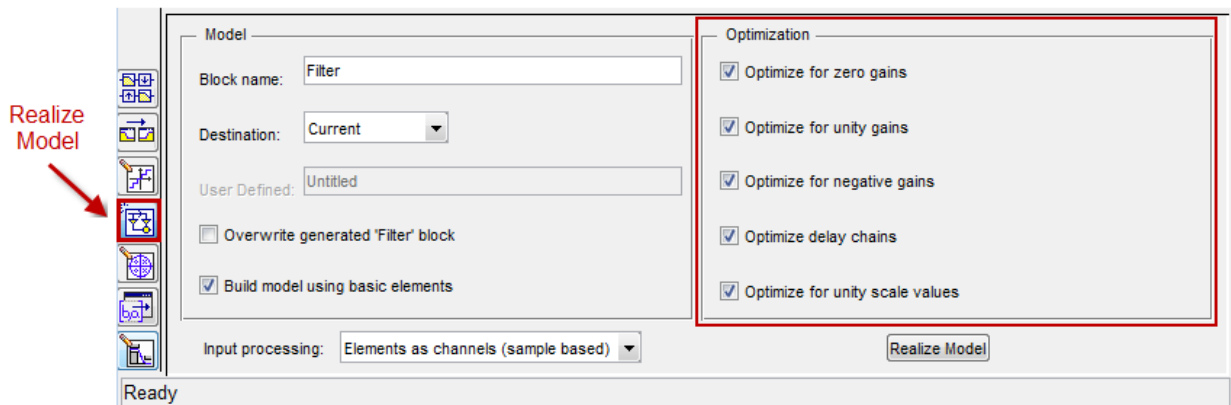
- **Optimize for zero gains** — Remove zero-gain paths.
- **Optimize for unity gains** — Substitute gains equal to one with a wire (short circuit).
- **Optimize for negative gains** — Substitute gains equal to -1 with a wire (short circuit), and change the corresponding sums to subtractions.

- **Optimize delay chains** — Substitute any delay chain made up of *n* unit delays with a single delay by *n*.
- **Optimize for unity scale values** — Remove all scale value multiplications by 1 from the filter structure.

The following diagram illustrates the results of each of these optimizations.
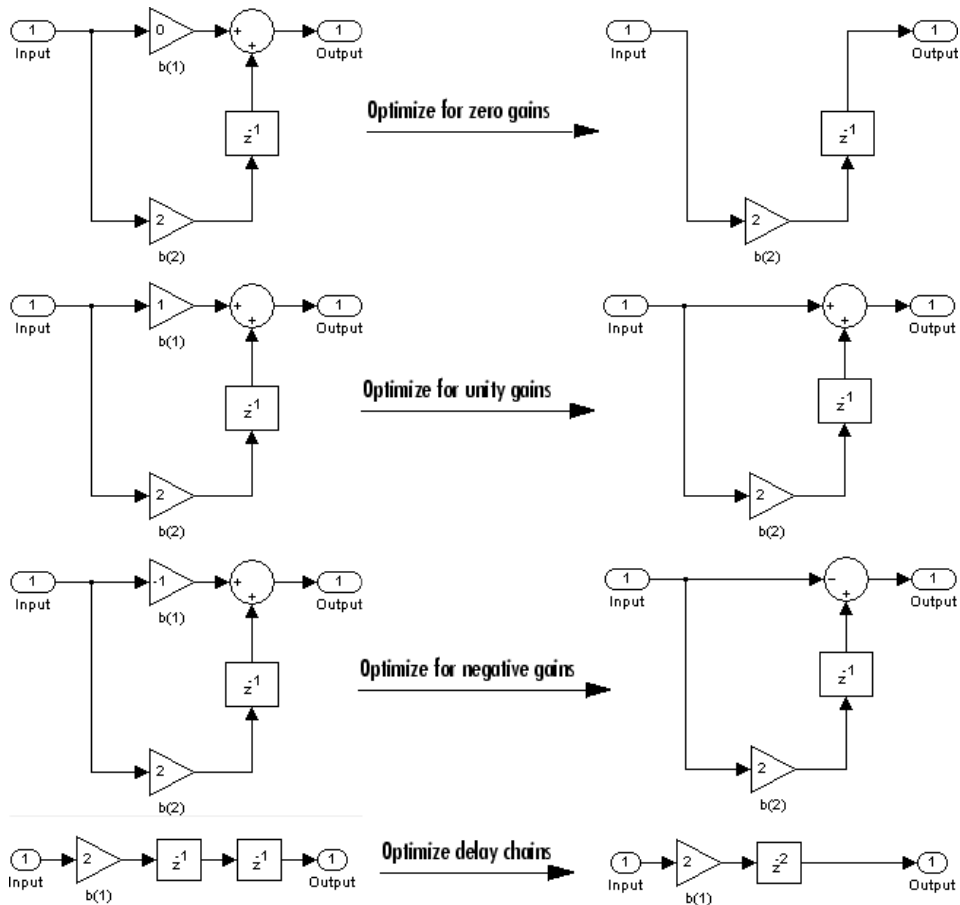
Digital Filter Implementations

In this section...

Using Digital Filter Blocks

DSP System Toolbox provides several blocks implementing digital filters, such as `Discrete FIR Filter` and `Biquad Filter`.

Use these blocks if you have already performed the design and analysis and know your desired filter coefficients. You can use these blocks to filter single-channel and multichannel signals, and to simulate floating-point and fixed-point filters. Then, you can use the Simulink Coder product to generate highly optimized C code from your filters.

To implement a filter, you must provide the following basic information about the filter:

- The desired filter structure
- The filter coefficients

Note: Use the Digital Filter Design block to design and implement a filter. Use the Discrete FIR Filter and Biquad Filter blocks to implement a pre-designed filter. Both methods implement a filter in the same manner and have the same behavior during simulation and code generation.

Implement a Lowpass Filter in Simulink

Use the `Discrete FIR Filter` block to implement a lowpass filter:

1 Define the lowpass filter coefficients in the MATLAB workspace by typing

```
lopassNum = [-0.0021 -0.0108 -0.0274 -0.0409 -0.0266 0.0374
0.1435 0.2465 0.2896 0.2465 0.1435 0.0374 -0.0266 -0.0409 -0.0274
-0.0108 -0.0021];
```

2 Open Simulink and create a new model file.

3 From the DSP System Toolbox Filtering>Filter Implementations library, click-and-drag a Discrete FIR Filter block into your model.

4 Double-click the Discrete FIR Filter block. Set the block parameters as follows, and then click **OK**:

- **Coefficient source** = `Dialog parameters`
- **Filter structure** = `Direct form transposed`
- **Coefficients** = `lopassNum`
- **Input processing** = `Columns as channels (frame based)`
- **Initial states** = `0`

Note that you can provide the filter coefficients in several ways:

- Type in a variable name from the MATLAB workspace, such as `lopassNum`.
- Type in filter design commands from Signal Processing Toolbox software or DSP System Toolbox software, such as `fir1(5, 0.2, 'low')`.
- Type in a vector of the filter coefficient values.

5 Rename your block Digital Filter - Lowpass.

The Discrete FIR Filter block in your model now represents a lowpass filter. In the next topic, "Implement a Highpass Filter in Simulink" on page 4-142, you use a Discrete FIR Filter block to implement a highpass filter. For more information about the Discrete FIR Filter block, see the `Discrete FIR Filter` block reference page. For more information about designing and implementing a new filter, see "Digital Filter Design Block" on page 4-119.

Implement a Highpass Filter in Simulink

In this topic, you implement a highpass filter using the `Discrete FIR Filter` block:

1 If the model you created in "Implement a Lowpass Filter in Simulink" on page 4-141 is not open on your desktop, you can open an equivalent model by typing

```
ex_filter_ex1
```

at the MATLAB command prompt.

2 Define the highpass filter coefficients in the MATLAB workspace by typing

```
hipassNum = [-0.0051 0.0181 -0.0069 -0.0283 -0.0061 ...
0.0549 0.0579 -0.0826 -0.2992 0.5946 -0.2992 -0.0826 ...
0.0579 0.0549 -0.0061 -0.0283 -0.0069 0.0181 -0.0051];
```

3 From the DSP System Toolbox Filtering library, and then from the Filter Implementations library, click-and-drag a Discrete FIR Filter block into your model.

4 Double-click the Discrete FIR Filter block. Set the block parameters as follows, and then click **OK**:

- **Coefficient source** = `Dialog parameters`
- **Filter structure** = `Direct form transposed`
- **Coefficients** = `hipassNum`
- **Input processing** = `Columns as channels (frame based)`
- **Initial states** = `0`

You can provide the filter coefficients in several ways:

- Type in a variable name from the MATLAB workspace, such as `hipassNum`.
- Type in filter design commands from Signal Processing Toolbox software or DSP System Toolbox software, such as `fir1(5, 0.2, 'low')`.
- Type in a vector of the filter coefficient values.

5 Rename your block Digital Filter - Highpass.

You have now successfully implemented a highpass filter. In the next topic, "Filter High-Frequency Noise in Simulink" on page 4-143, you use these Discrete FIR Filter blocks to create a model capable of removing high frequency noise from a signal. For more information about designing and implementing a new filter, see "Digital Filter Design Block" on page 4-119.

Filter High-Frequency Noise in Simulink

In the previous topics, you used `Discrete FIR Filter` blocks to implement lowpass and highpass filters. In this topic, you use these blocks to build a model that removes high frequency noise from a signal. In this model, you use the highpass filter, which is

excited using a uniform random signal, to create high-frequency noise. After you add this noise to a sine wave, you use the lowpass filter to filter out the high-frequency noise:

1   If the model you created in "Implement a Highpass Filter in Simulink" on page 4-142 is not open on your desktop, you can open an equivalent model by typing

```
ex_filter_ex2
```

at the MATLAB command prompt.

2   If you have not already done so, define the lowpass and highpass filter coefficients in the MATLAB workspace by typing

```
lopassNum = [-0.0021 -0.0108 -0.0274 -0.0409 -0.0266 ...
0.0374 0.1435 0.2465 0.2896 0.2465 0.1435 0.0374 ...
-0.0266 -0.0409 -0.0274 -0.0108 -0.0021];
hipassNum = [-0.0051 0.0181 -0.0069 -0.0283 -0.0061 ...
0.0549 0.0579 -0.0826 -0.2992 0.5946 -0.2992 -0.0826 ...
0.0579 0.0549 -0.0061 -0.0283 -0.0069 0.0181 -0.0051];
```

3   Click-and-drag the following blocks into your model file.

Block	Library	Quantity
Add	Simulink / Math Operations library	1
Random Source	Sources	1
Sine Wave	Sources	1
Time Scope	Sinks	1

4   Set the parameters for the rest of the blocks as indicated in the following table. For any parameters not listed in the table, leave them at their default settings.

Block	Parameter Setting
Add	• **Icon shape** = `rectangular` • **List of signs** = ++
Random Source	• **Source type** = `Uniform` • **Minimum** = 0 • **Maximum** = 4 • **Sample mode** = `Discrete` • **Sample time** = `1/1000`

Block	Parameter Setting
	• **Samples per frame** = 50
Sine Wave	• **Frequency (Hz)** = 75
	• **Sample time** = 1/1000
	• **Samples per frame** = 50
Time Scope	• **File** > **Number of Input Ports** > **3**
	• **File** > **Configuration ...**
	• Open the **Visuals:Time Domain Options** dialog and set **Time span** = One frame period

5 Connect the blocks as shown in the following figure. You may need to resize some of your blocks to accomplish this task.
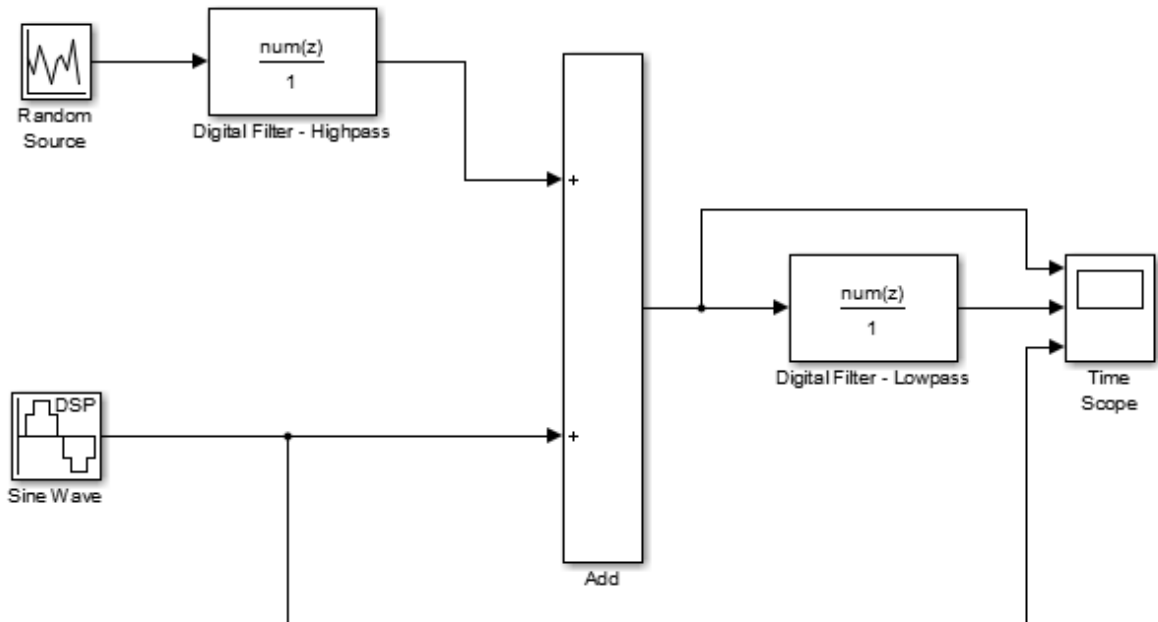
6 From the Simulation menu, select **Model Configuration Parameters**.

The **Configuration Parameters** dialog box opens.

7 In the **Solver** pane, set the parameters as follows, and then click **OK**:

- **Start time** = 0
- **Stop time** = 5
- **Type** = Fixed-step
- **Solver** = discrete (no continuous states)

8 In the model window, from the **Simulation** menu, choose **Run**.

The model simulation begins and the Scope displays the three input signals.

9 After simulation is complete, select **View** > **Legend** from the Time Scope menu. The legend appears in the Time Scope window. You can click-and-drag it anywhere on the scope display. To change the channel names, double-click inside the legend and replace the current numbered channel names with the following:

- Add = Noisy Sine Wave
- Digital Filter – Lowpass = Filtered Noisy Sine Wave
- Sine Wave = Original Sine Wave

In the next step, you will set the color, style, and marker of each channel.

10 In the Time Scope window, select **View** > **Line Properties**, and set the following:

Line	Style	Marker	Color
Noisy Sine Wave	-	**None**	**Black**
Filtered Noisy Sine Wave	-	**diamond**	**Red**
Original Sine Wave	**None**	*****	**Blue**

11 The **Time Scope** display should now appear as follows:

You can see that the lowpass filter filters out the high-frequency noise in the noisy sine wave.
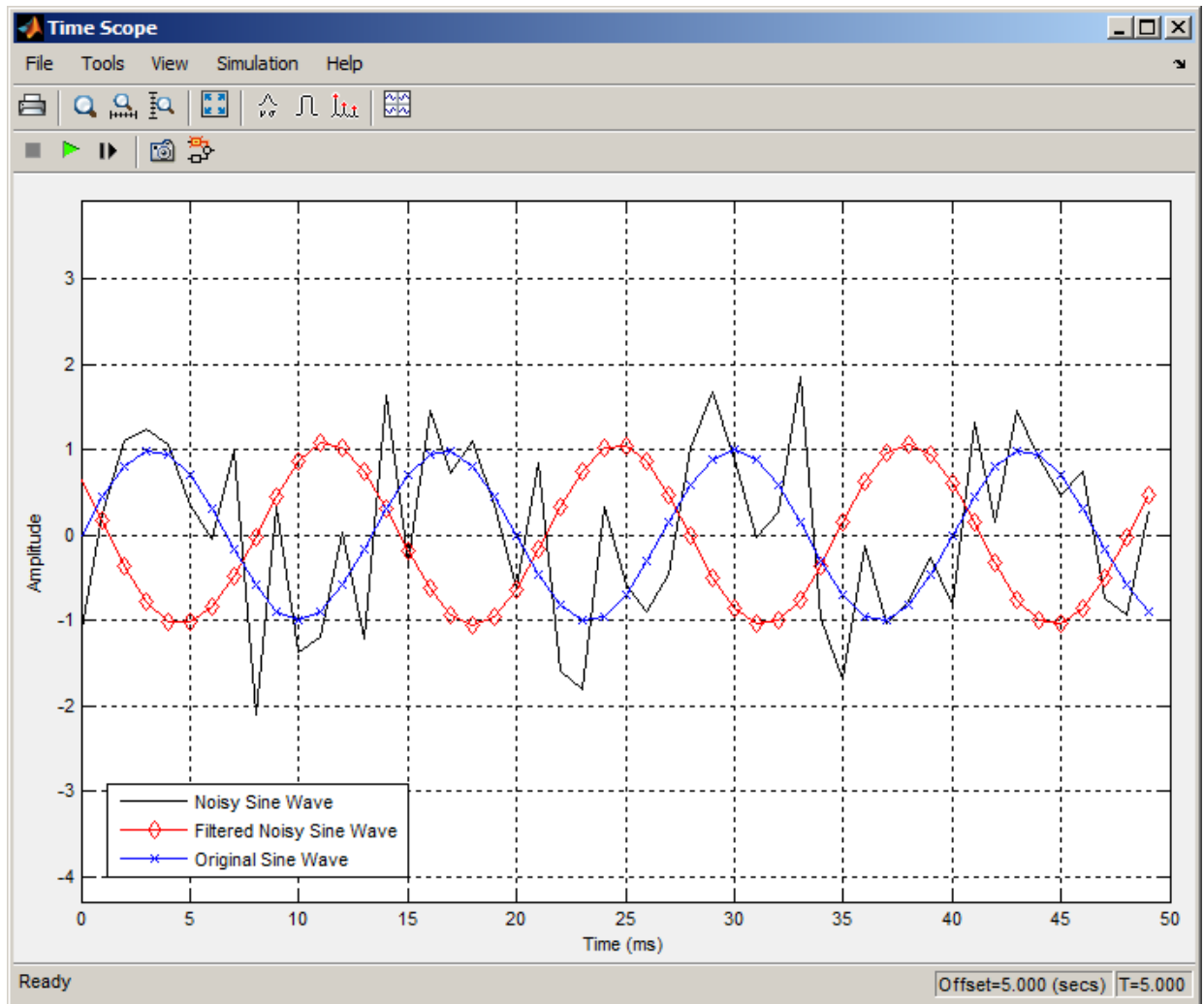
You have now used Discrete FIR Filter blocks to build a model that removes high frequency noise from a signal. For more information about designing and implementing a new filter, see "Digital Filter Design Block" on page 4-119.

Specify Static Filters

You can specify a static filter using the `Discrete FIR Filter` or `Biquad Filter` block. To do so, set the **Coefficient source** parameter to `Dialog parameters`.

For the Discrete FIR Filter, set the **Coefficients** parameter to a row vector of numerator coefficients. If you set **Filter structure** to `Lattice MA`, the **Coefficients** parameter represents reflection coefficients.

For the Biquad Filter, set the **SOS matrix (Mx6)** to an *M*-by-6 matrix, where *M* is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and denominator coefficients of the corresponding section in the filter. Set **Scale values** to a scalar or vector of *M*+1 scale values used between SOS stages.

Tuning the Filter Coefficient Values During Simulation

To change the static filter coefficients during simulation, double-click the block, type in the new filter coefficients, and click **OK**. You cannot change the filter order, so you cannot change the number of elements in the matrix of filter coefficients.

Specify Time-Varying Filters

Time-varying filters are filters whose coefficients change with time. You can specify a time-varying filter that changes once per frame. You can filter multiple channels with each filter. However, you cannot apply different filters to each channel; all channels use the same filter.

To specify a time-varying filter using a Biquad Filter block or a Discrete FIR Filter block:

1   Set the **Coefficient source** parameter to `Input port(s)`, which enables extra block input ports for the time-varying filter coefficients.

- The Discrete FIR Filter block has a `Num` port for the numerator coefficients.

- The Biquad Filter block has `Num` and `Den` ports rather than a single port for the SOS matrix. Separate ports enable you to use different fraction lengths for numerator and denominator coefficients. The scale values port, `g`, is optional. You can disable the `g` port by setting **Scale values mode** to `Assume all are unity and optimize`.

2   Provide matrices of filter coefficients to the block input ports.

- For Discrete FIR Filter block, the number of filter taps, *N*, cannot vary over time. The input coefficients must be in a 1-by-*N* vector.

- For Biquad Filter block, the number of filter sections, *N*, cannot vary over time. The numerator coefficients input, `Num`, must be a 3-by-*N* matrix. The denominator input coefficients, `Den`, must be a 2-by-*N* matrix. The scale values input, `g`, must be a 1-by-(*N*+1) vector.

Specify the SOS Matrix (Biquadratic Filter Coefficients)

Use the Biquad Filter block to specify a static biquadratic IIR filter (also known as a second-order section or SOS filter). Set the following parameters:

- **Filter structure** — `Direct form I`, or `Direct form I transposed`, or `Direct form II`, or `Direct form II transposed`

- **SOS matrix (Mx6)** *M*-by-6 SOS matrix

  The SOS matrix is an *M*-by-6 matrix, where *M* is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and denominator coefficients (b_{ik} and a_{ik}) of the corresponding section in the filter.

- **Scale values** Scalar or vector of *M*+1 scale values to be used between SOS stages

  If you enter a scalar, the value is used as the gain value before the first section of the second-order filter. The rest of the gain values are set to 1.

  If you enter a vector of *M*+1 values, each value is used for a separate section of the filter. For example, the first element is the first gain value, the second element is the second gain value, and so on.
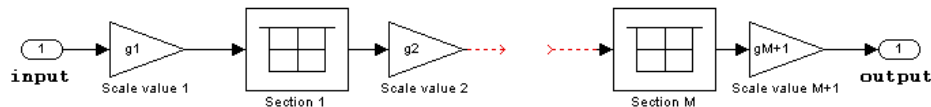
You can use the `ss2sos` and `tf2sos` functions from Signal Processing Toolbox software to convert a state-space or transfer function description of your filter into the second-order section description used by this block.

$$
\begin{bmatrix}
b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\
b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
b_{0M} & b_{1M} & b_{2M} & a_{0M} & a_{1M} & a_{2M}
\end{bmatrix}
$$

The block normalizes each row by a_{1i} to ensure a value of 1 for the zero-delay denominator coefficients.

Removing High-Frequency Noise from an ECG Signal

This examples shows you how to filter an ECG signal that has high-freqquency noise, and remove the noise by low-pass filtering.

```
% Create one period of ECG signal
x = ecg(500).';
y = sgolayfilt(x,0,5);
Fs = 1000;
[M,N] = size(y);

% Initialize scopes
TS = dsp.TimeScope('SampleRate',Fs,...
                   'TimeSpan',1.5,...
                   'YLimits',[-1 1],...
                   'ShowGrid',true,...
                   'NumInputPorts',2,...
                   'LayoutDimensions',[2 1],...
                   'Title','Noisy and Filtered Signals');

% Design lowpass filter
Fpass  = 200;
Fstop = 400;
Dpass = 0.05;
Dstop = 0.0001;
F     = [0 Fpass Fstop Fs/2]/(Fs/2);
A     = [1 1    0     0];
D     = [Dpass Dstop];
b = firgr('minorder', F, A, D);
LP = dsp.FIRFilter('Numerator',b);

% Design Highpass Filter
Fstop = 200;
Fpass = 400;
Dstop = 0.0001;
Dpass = 0.05;
F = [0 Fstop Fpass Fs/2]/(Fs/2); % Frequency vector
A = [0 0    1     1]; % Amplitude vector
D = [Dstop   Dpass];  % Deviation (ripple) vector
b  = firgr('minord', F, A, D);
HP = dsp.FIRFilter('Numerator', b);

% Stream
tic;
```

```matlab
while toc < 30
    x = .1 * randn(M,N);
    highFreqNoise = step(HP,x);
    noisySignal   = y + highFreqNoise;
    filteredSignal = step(LP,noisySignal);
    step(TS,noisySignal,filteredSignal);
end

% Finalize
release(TS)
```

5
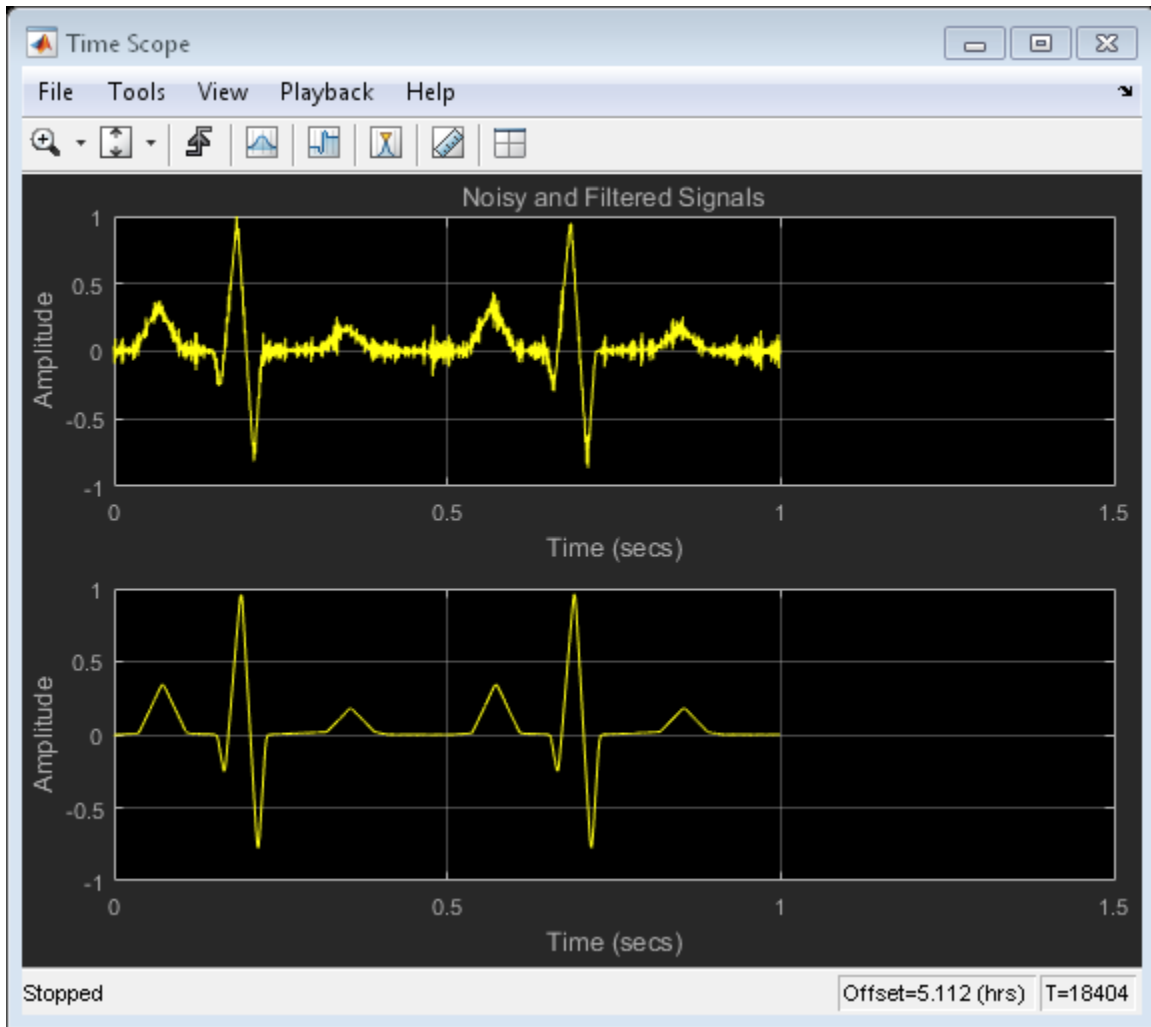
Adaptive Filters

Learn how to design and implement adaptive filters.

Overview of Adaptive Filters and Applications

In this section...
"Introduction to Adaptive Filtering" on page 5-2
"Adaptive Filtering Methodology" on page 5-2
"Choosing an Adaptive Filter" on page 5-4
"System Identification" on page 5-5
"Inverse System Identification" on page 5-6
"Noise or Interference Cancellation" on page 5-7
"Prediction" on page 5-7

Introduction to 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.

For more detailed information about adaptive filters and adaptive filter theory, refer to the books listed in the "Selected Bibliography" on page 5-64.

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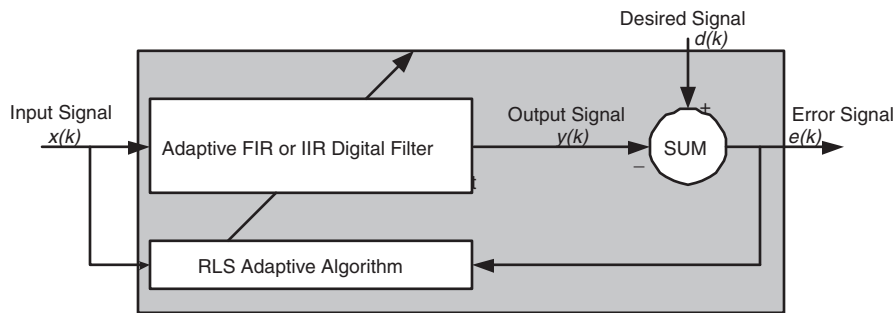
Block Diagram That Defines the Inputs and Output of a Generic RLS Adaptive Filter

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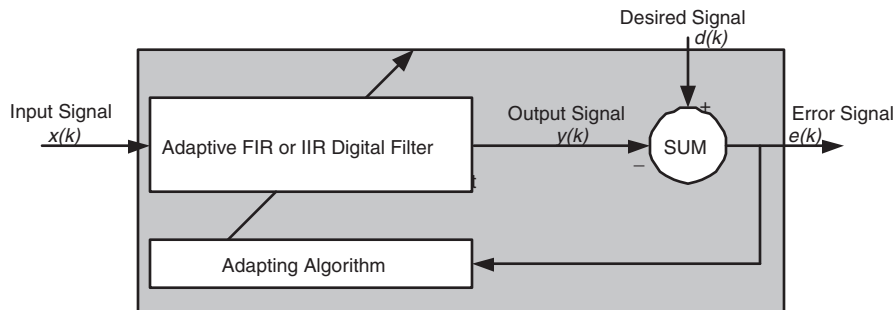
Block Diagram Defining General Adaptive Filter Algorithm Inputs and Outputs

DSP System 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 dsp.LMSFilter and dsp.FilteredXLMSFilter

- Adaptive filters based on recursive least squares (RLS) techniques, including sign-data, sign-error, and sign-sign. See dsp.RLSFilter.

- Adaptive filters based on lattice filters. See dsp.AdaptiveLatticeFilter.

- Adaptive filters that operate in the frequency domain. See dsp.FrequencyDomainAdaptiveFilter.

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.

5-3

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 DSP System Toolbox Software" on page 5-9 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 DSP System 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.
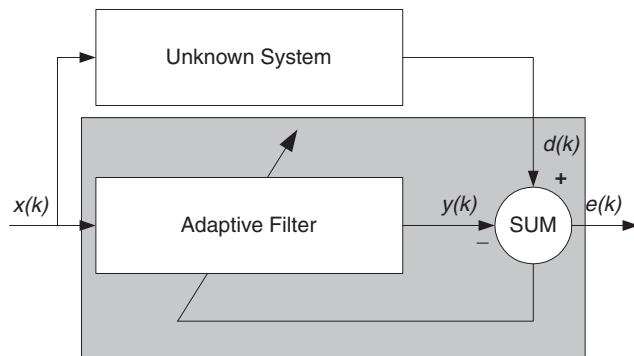
Using an Adaptive Filter to Identify an Unknown System

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.
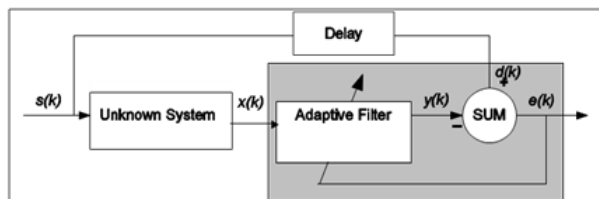
Determining an Inverse Response to an Unknown System

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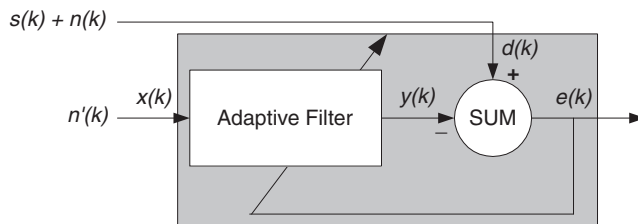
Using an Adaptive Filter to Remove Noise from an Unknown System

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.
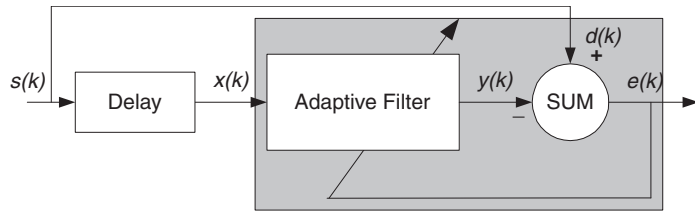
Predicting Future Values of a Periodic Signal

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 DSP System Toolbox Software

In this section...
"Overview of Adaptive Filtering in DSP System Toolbox Software" on page 5-9
"Algorithms" on page 5-9
"Using Adaptive Filter Objects" on page 5-11

Overview of Adaptive Filtering in DSP System Toolbox Software

DSP System 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 objects, 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
dsp.BlockLMSFilter	Block LMS FIR adaptive filter algorithm
dsp.FilteredXLMSFilter	Filtered-x LMS FIR adaptive filter algorithm
dsp.LMSFilter	LMS FIR adaptive filter algorithm Normalized LMS FIR adaptive filter algorithm

Adaptive Filter Method	Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
	Sign-data LMS FIR adaptive filter algorithm
	Sign-error LMS FIR adaptive filter algorithm
	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
dsp.FastTransversalFilt	Fast transversal least-squares adaptation algorithm
	Sliding window FTF adaptation algorithm
dsp.RLSFilter	QR-decomposition RLS adaptation algorithm
	Householder RLS adaptation algorithm
	Householder SWRLS adaptation algorithm
	Recursive-least squares (RLS) adaptation algorithm
	Sliding window (SW) RLS 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
dsp.AffineProjectionFilt	Affine projection algorithm that uses direct matrix inversion
	Affine projection algorithm that uses recursive matrix updating
	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
dsp.FrequencyDomainAdapt	Frequency domain adaptation algorithm
	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
dsp.AdaptiveLatticeFilt	Gradient adaptive lattice filter adaptation algorithm
	Least squares lattice adaptation algorithm
	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 5-64.

Using Adaptive Filter Objects

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

- "LMS Adaptive Filters" on page 5-12
- "RLS Adaptive Filters" on page 5-30

LMS Adaptive Filters

In this section...

LMS Filter Introductory Examples

This section provides introductory examples using some of the least mean squares (LMS) adaptive filter functionality in the toolbox.

The toolbox provides dsp.LMSFilter, which is a System object that uses LMS algorithms to search for the optimal solution to the adaptive filter. The `dsp.LMSFilter` object supports these algorithms:

- The LMS algorithm, which solves the Wiener-Hopf equation and finds the filter coefficients for an adaptive filter

- The normalized variation of the LMS algorithm

- The sign-data variation of the LMS algorithm, where the correction to the filter weights at each iteration depends on the sign of the input $x(k)$

- The sign-error variation of the LMS algorithm, where the correction applied to the current filter weights for each successive iteration depends on the sign of the error, $e(k)$

- The sign-sign variation of the LMS algorithm, where 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 `fircband` examples.

```
[b,err,res]=fircband(12,[O 0.4 0.5 1], [1 1 0 0], [1 0.2],...
{'w' 'c'});
fvtool(b,1);
```

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 to identify this filter in a system identification role.

To review the general model for system ID mode, look at "System Identification" on page 5-5 for the layout.
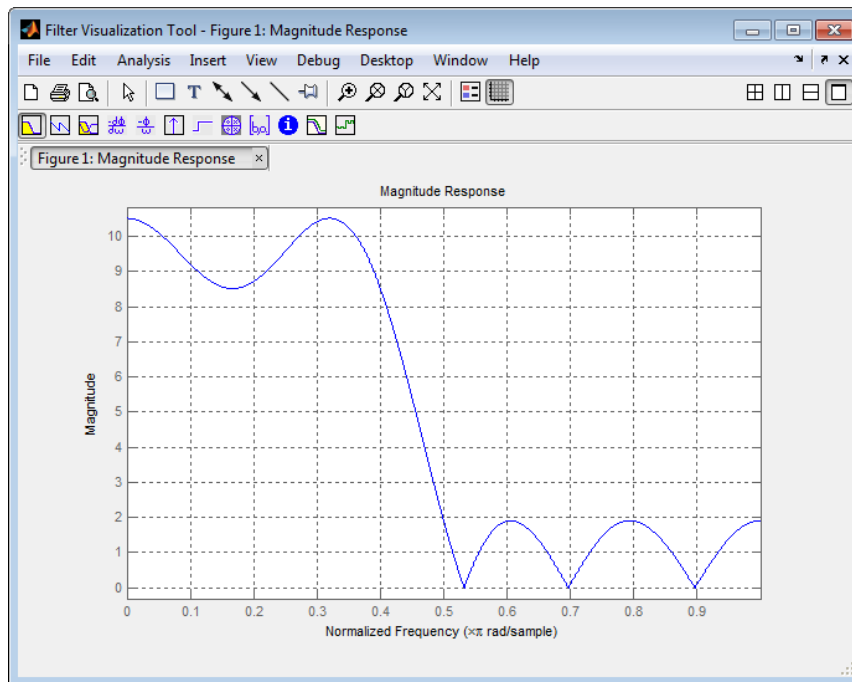
For the sign variations of the LMS algorithm, the examples use noise cancellation as the demonstration application, as opposed to the system identification application used in the LMS examples.

System Identification Using the LMS Algorithm

To use the adaptive filter functions in the toolbox you need to provide three things:

- The adaptive LMS algorithm to use. You can select the algorithm of your choice by setting the `Method` property of dsp.LMSFilter to the desired algorithm.
- An unknown system or process to adapt to. In this example, the filter designed by `fircband` 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(250,1);
```

The input is broadband noise. For the unknown system filter, use `fircband` to create a twelfth-order lowpass filter:

```
[b,err,res] = fircband(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. Set the `InitialConditions` property of `dsp.LMSFilter` to the desired initial values of the 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;
lms = dsp.LMSFilter(13,'StepSize',mu,'WeightsOutputPort',true);
```

Finally, using the `dsp.LMSFilter` object `lms`, 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 `fircband` to the coefficients found by `dsp.LMSFilter`.

```
[y,e,w] = step(lms,x,d);;
stem([b.' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual Filter Weights','Estimated Filter Weights',...
        'Location','NorthEast')
```
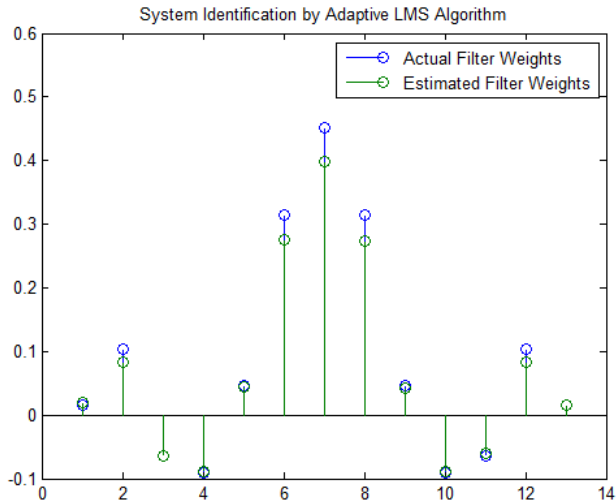

As an experiment, try changing the step size to 0.2. Repeating the example with `mu = 0.2` results in the following stem plot. The estimated weights fail to approximate the actual weights closely.

```
mu = 0.2;
lms = dsp.LMSFilter(13,'StepSize',mu,'WeightsOutputPort',true);
[y,e,w] = step(lms,x,d);
stem([b.' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual Filter Weights','Estimated Filter Weights',...
        'Location','NorthEast')
```

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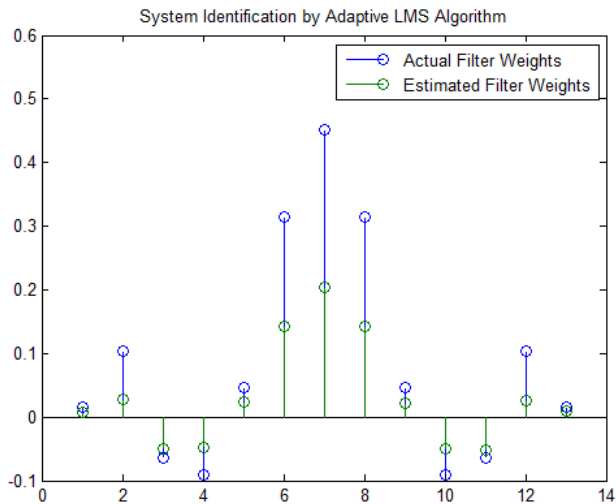
Since this may be because you did not iterate over the LMS algorithm enough times, try using 1000 samples. With 1000 samples, the stem plot, shown in the next figure, looks much better, albeit at the expense of much more computation. Clearly you should take care to select the step size with both the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

```
for index = 1:4
  x = 0.1*randn(250,1);
  d = filter(b,1,x);
  [y,e,w] = step(lms,x,d);
end
stem([b.' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual Filter Weights','Estimated Filter Weights',...
       'Location','NorthEast')
```

System Identification by Adaptive LMS Algorithm
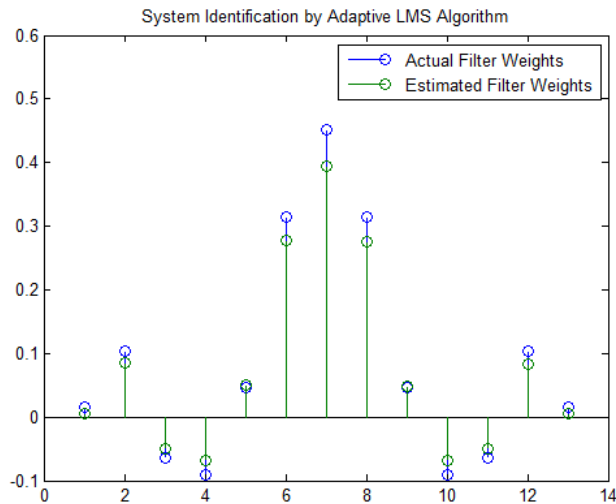
System Identification Using the Normalized LMS Algorithm

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 normalized LMS algorithm example, you used `fircband` to create the filter that you would identify. So you can compare the results, you use the same filter, and set the `Method` property on dsp.LMSFilter to `'Normalized LMS'`. 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(500,1);
[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.

```
lms = dsp.LMSFilter(13,'StepSize',mu,'Method',...
    'Normalized LMS','WeightsOutputPort',true);
```

You use the preceding code to initialize the normalized LMS algorithm. For more information about the optional input arguments, refer to dsp.LMSFilter.

Running the system identification process is a matter of using the `dsp.LMSFilter` object 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,w] = step(lms,x,d);
stem([b.' w])
title('System Identification by Normalized LMS Algorithm')
legend('Actual Filter Weights','Estimated Filter Weights',...
        'Location','NorthEast')
```

As shown in the following stem plot (a convenient way to compare the estimated and actual filter coefficients), the two are nearly identical.
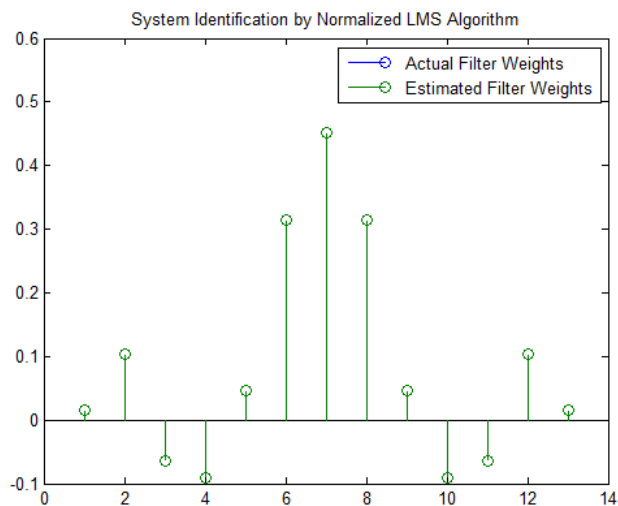
If you compare the convergence performance of the regular LMS algorithm to the normalized LMS variant, you see the normalized version adapts in far fewer iterations to a result almost as good as the nonnormalized version.

```
lms_normalized = dsp.LMSFilter(13,'StepSize',mu,...
    'Method','Normalized LMS','WeightsOutputPort',true);
lms_nonnormalized = dsp.LMSFilter(13,'StepSize',mu,...
    'Method','LMS','WeightsOutputPort',true);
[~,e1,~] = step(lms_normalized,x,d);
[~,e2,~] = step(lms_nonnormalized,x,d);
plot([e1,e2]);
title('Comparing the LMS and NLMS Conversion Performance');
legend('NLMS Derived Filter Weights', ...
       'LMS Derived Filter Weights','Location', 'NorthEast');
```
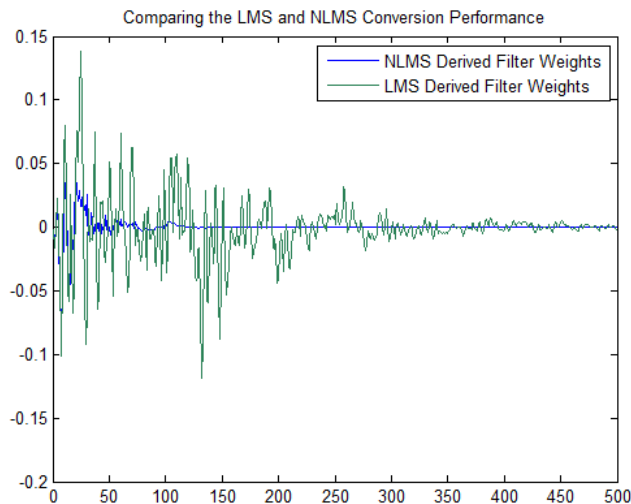

Noise Cancellation Using the Sign-Data LMS Algorithm

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

$$w(k+1) = w(k) + \mu e(k)\,\text{sgn}\big[x(k)\big],$$

$$\text{sgn}\big[x(k)\big] = \begin{cases} 1, x(k) > 0 \\ 0, x(k) = 0 \\ -1, x(k) < 0 \end{cases}$$

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, set dsp.LMSFilter `Method` property to `'Sign-Data LMS'`. This example requires two input data sets:

- Data containing a signal corrupted by noise. In Using an Adaptive Filter to Remove Noise from an Unknown System, 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) 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(1000,1);
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 `dsp.LMSFilter` object for processing, set the weight initial conditions (`InitialConditions`) and mu (`StepSize`) 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 "System Identification Using the LMS Algorithm" on page 5-13, 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 `dsp.LMSFilter` prepared, construct the LMS filter object, run the adaptation, and view the results.

```
lms = dsp.LMSFilter(12,'Method','Sign-Data LMS',...
    'StepSize',mu,'InitialConditions',coeffs);
[~,e] = step(lms,noise,d);
L = 200;
plot(0:L-1,signal(1:L),0:L-1,e(1:L));
title('Noise Cancellation by the Sign-Data Algorithm');
legend('Actual Signal','Result of Noise Cancellation',...
        'Location','NorthEast');
```

When `dsp.LMSFilter` 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 `InitialConditions`, `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.
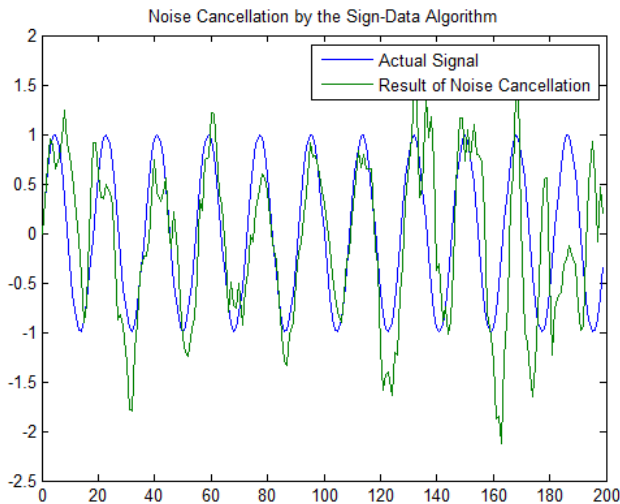
Noise Cancellation Using Sign-Error LMS Algorithm

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

$$w(k+1) = w(k) + \mu \, \text{sgn}\big[e(k)\big]\big[x(k)\big],$$

$$\text{sgn}\big[e(k)\big] = \begin{cases} 1, e(k) > 0 \\ 0, e(k) = 0 \\ -1, e(k) < 0 \end{cases}$$

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 (*μ<< 1*) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, the dsp.LMSFilter object requires two input data sets:

- Data containing a signal corrupted by noise. In Using an Adaptive Filter to Remove Noise from an Unknown System, 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) 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(1000,1);
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 `dsp.LMSFilter` object for processing, set the weight initial conditions (`InitialConditions`) and mu (`StepSize`) 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 "System Identification Using the LMS Algorithm" on page 5-13, 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 `dsp.LMSFilter` prepared, run the adaptation and view the results.

```
lms = dsp.LMSFilter(12,'Method','Sign-Error LMS',...
   'StepSize',mu,'InitialConditions',coeffs);
[~,e] = step(lms,noise,d);
L = 200;
plot(0:199,signal(1:200),0:199,e(1:200));
title('Noise Cancellation Performance by the Sign-Error LMS Algorithm');
legend('Actual Signal','Error After Noise Reduction',...
       'Location','NorthEast')
```

When the sign-error LMS algorithm 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 the weight initial conditions (`InitialConditions`) and mu (`StepSize`), or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.
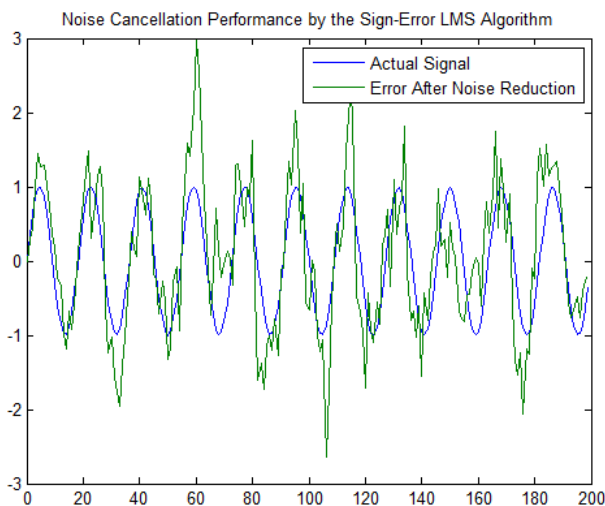
Noise Cancellation Using Sign-Sign LMS Algorithm

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 "Noise Cancellation Using the Sign-Data LMS Algorithm" on page 5-19.

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

$$w(k+1) = w(k) + \mu \operatorname{sgn}\big[e(k)\big]\operatorname{sgn}\big[x(k)\big],$$

$$\operatorname{sgn}\big[z(k)\big] = \begin{cases} 1, z(k) > 0 \\ 0, z(k) = 0 \\ -1, z(k) < 0 \end{cases}$$

where

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

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, dsp.LMSFilter object requires two input data sets:

- Data containing a signal corrupted by noise. In Using an Adaptive Filter to Remove Noise from an Unknown System, 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) 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(1000,1);
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 `dsp.LMSFilter` object for processing, set the weight initial conditions (`InitialConditions`) and mu (`StepSize`) 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 "System Identification Using the LMS Algorithm" on page 5-13, 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 `dsp.LMSFilter` prepared, run the adaptation and view the results.

```
lms = dsp.LMSFilter(12,'Method','Sign-Sign LMS',...
   'StepSize',mu,'InitialConditions',coeffs);
[~,e] = step(lms,noise,d);
L = 200;
plot(0:199,signal(1:200),0:199,e(1:200));
title('Noise Cancellation Performance by the Sign-Error LMS Algorithm');
legend('Actual Signal','Error After Noise Reduction',...
       'Location','NorthEast')
```

When `dsp.LMSFilter` 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 the weight initial conditions (`InitialConditions`) and mu (`StepSize`), or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.
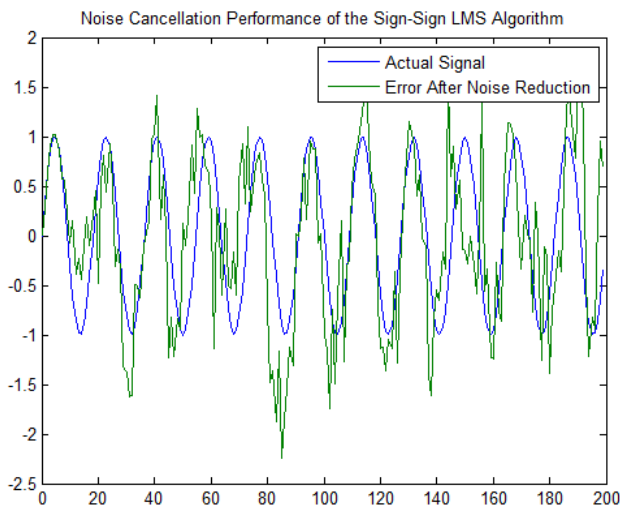
As an aside, the sign-sign LMS algorithm is part of the international CCITT standard for 32 Kb/s ADPCM telephony.

RLS Adaptive Filters

In this section...

Compare RLS and LMS Adaptive Filter Algorithms

This section provides an introductory example that uses the RLS adaptive filter System object dsp.RLSFIlter.

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.

Inverse System Identification Using dsp.RLSFilter

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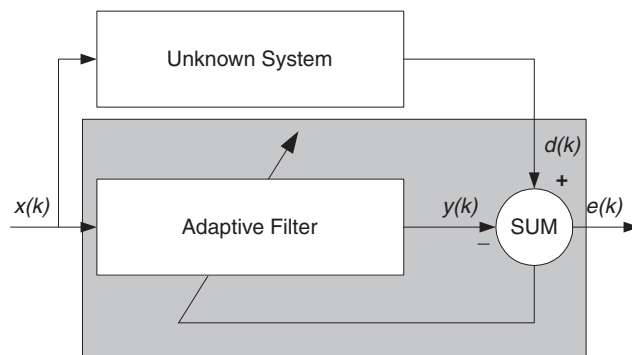
Cascading the adaptive filter with the unknown filter causes the adaptive filter to converge to a solution that is the inverse of the unknown system.

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(3000,1);
```

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(12,1);
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 RLS algorithm, instatiate a `dsp.RLSFilter` object and set its `Length`, `ForgettingFactor`, and `InitialInverserCovariance` properties.

For more information about the input conditions to prepare the RLS algorithm object, refer to dsp.RLSFilter.

```
p0 = 2 * eye(13);
lambda = 0.99;
```

```
rls = dsp.RLSFilter(13,'ForgettingFactor',lambda,...
   'InitialInverseCovariance',pO);
```

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] = step(rls,xdata,d);
```

where y returns the filtered output and e contains the error signal as the filter adapts to find the inverse of the unknown system. To view the estimated coefficient of the RLS filter, type rls.Coefficients.

The next figure presents the results of the adaptation. In the figure, the magnitude response curves for the unknown and adapted filters show. As a reminder, the unknown filter was a lowpass filter with cutoff at 0.55, on the normalized frequency scale from 0 to 1.
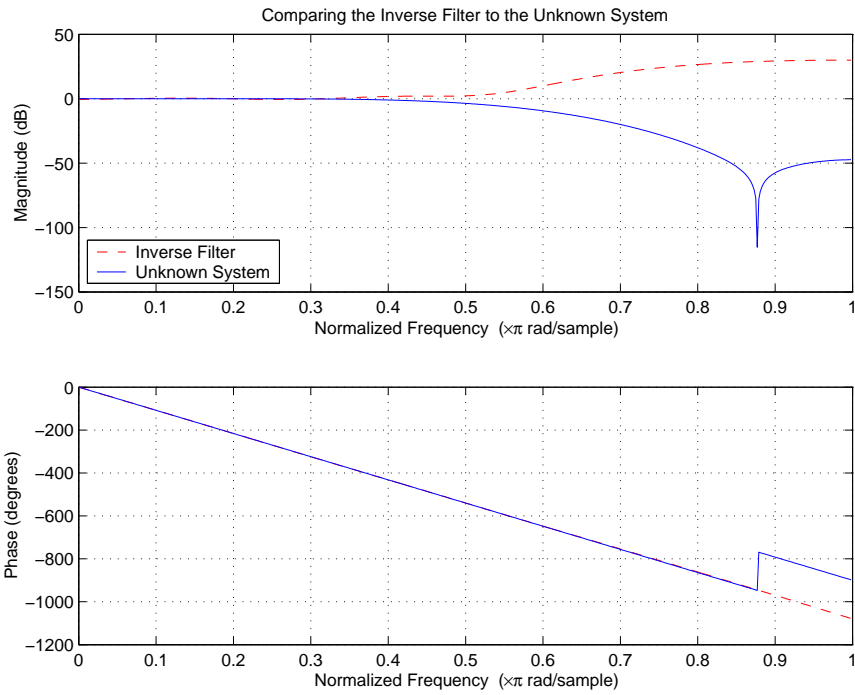
Viewed alone (refer to the following figure), the inverse system looks like a fair compensator for the unknown lowpass filter — a high pass filter with linear phase.

Inverse Filter Resulting from RLS Adaptation
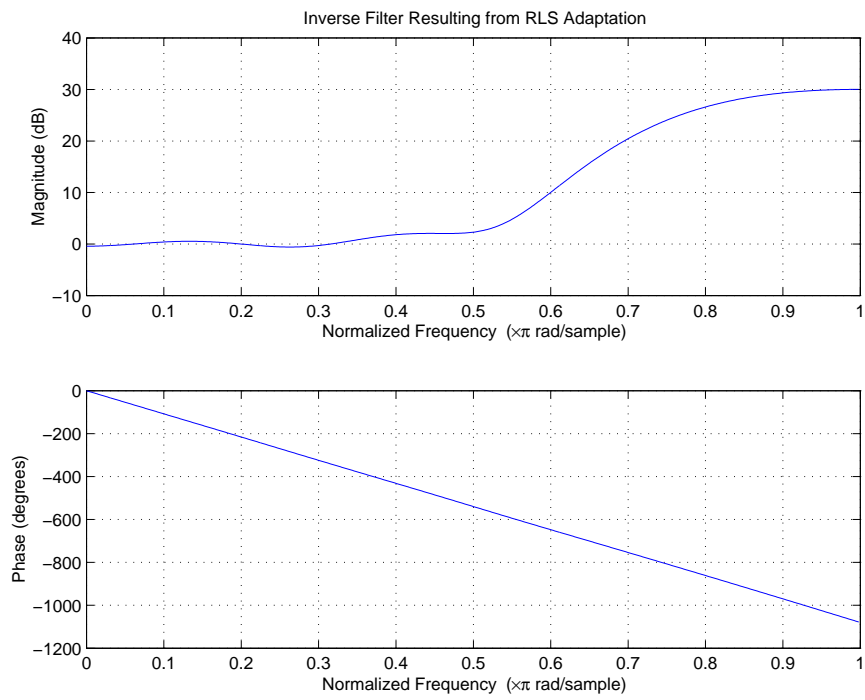
Adaptive Noise Cancellation Using RLS Adaptive Filtering

This example shows how to use an RLS filter to extract useful information from a noisy signal. The information bearing signal is a sine wave that is corrupted by additive white gaussian noise.

The adaptive noise cancellation system assumes the use of two microphones. A primary microphone picks up the noisy input signal, while a secondary microphone receives noise that is uncorrelated to the information bearing signal, but is correlated to the noise picked up by the primary microphone.

Note: This example is equivalent to the Simulink® model 'rlsdemo' provided.

Reference: S.Haykin, "Adaptive Filter Theory", 3rd Edition, Prentice Hall, N.J., 1996.

The model illustrates the ability of the Adaptive RLS filter to extract useful information from a noisy signal.

```
priv_drawrlsdemo
axis off
```
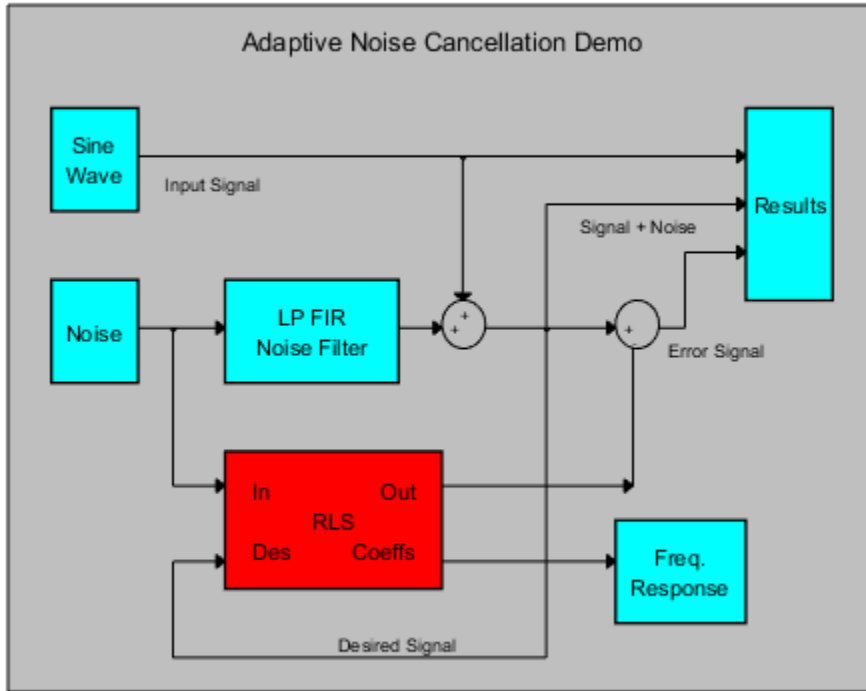
The information bearing signal is a sine wave of 0.055 cycles/sample.

```
signal = sin(2*pi*0.055*(0:1000-1)');
Hs = dsp.SignalSource(signal,'SamplesPerFrame',100,...
    'SignalEndAction','Cyclic repetition');

plot(0:199,signal(1:200));
grid; axis([0 200 -2 2]);
title('The information bearing signal');
```
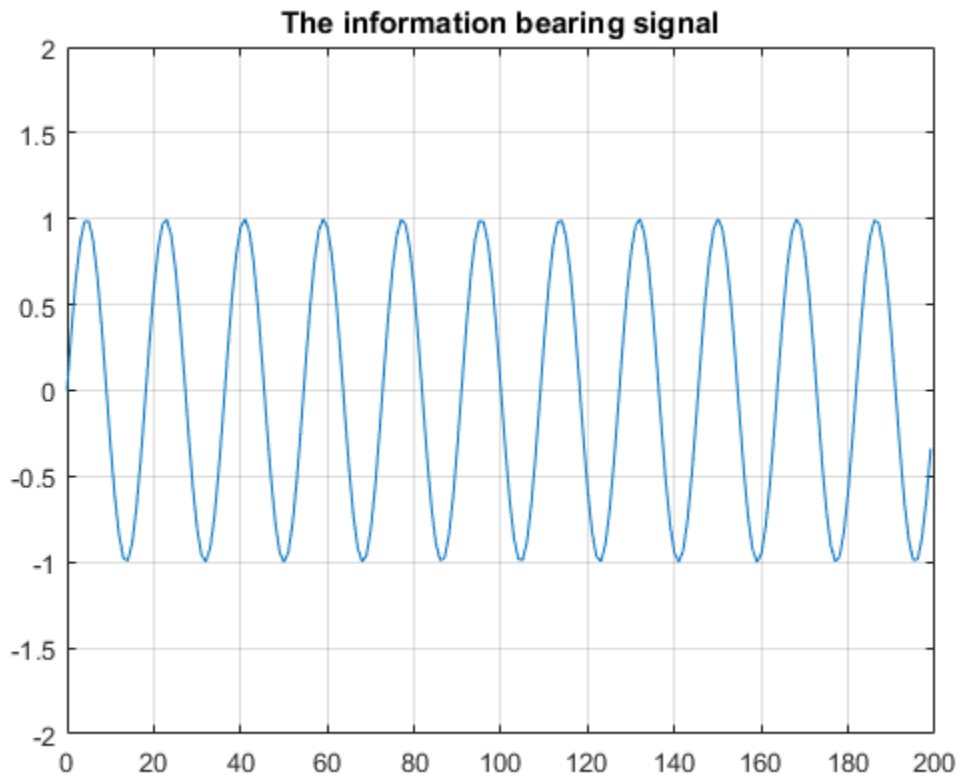
The noise picked up by the secondary microphone is the input for the RLS adaptive filter. The noise that corrupts the sine wave is a lowpass filtered version of (correlated to) this noise. The sum of the filtered noise and the information bearing signal is the desired signal for the adaptive filter.

```
nvar  = 1.0;                     % Noise variance
noise = randn(1000,1)*nvar;   % White noise
Hn = dsp.SignalSource(noise,'SamplesPerFrame',100,...
    'SignalEndAction','Cyclic repetition');

plot(0:999,noise);
title('Noise picked up by the secondary microphone');
grid; axis([0 1000 -4 4]);
```
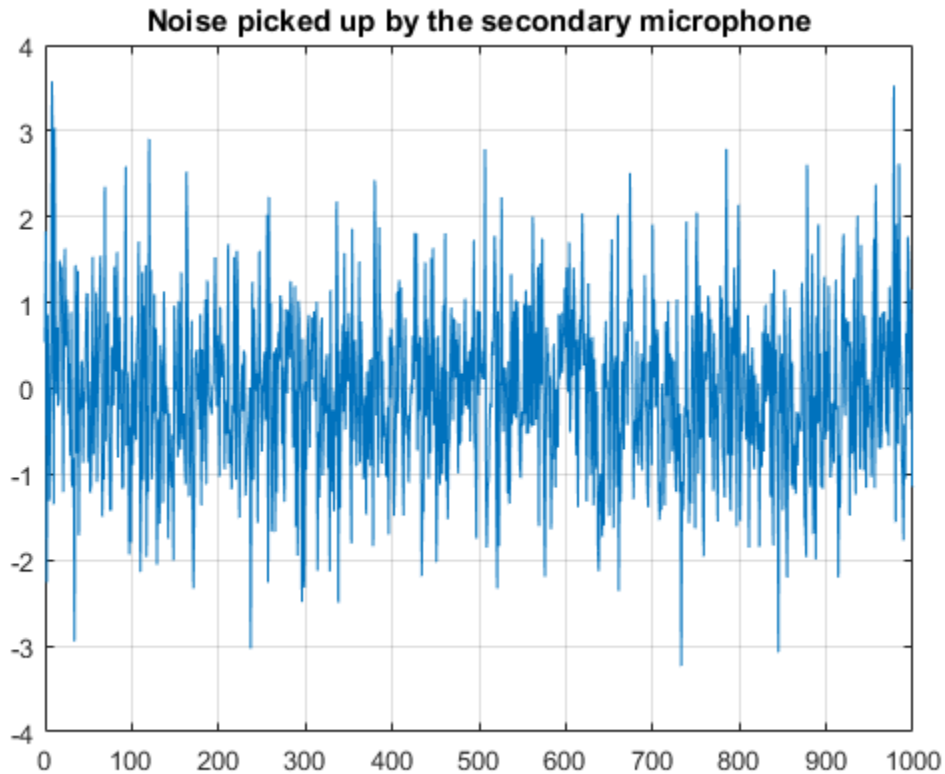
The noise corrupting the information bearing signal is a filtered version of 'noise'. Initialize the filter that operates on the noise.

```
Hd = dsp.FIRFilter('Numerator',fir1(31,0.5));% Low pass FIR filter
```

Set and initialize RLS adaptive filter parameters and values:

```
M     = 32;                 % Filter order
delta = 0.1;                % Initial input covariance estimate
P0    = (1/delta)*eye(M,M); % Initial setting for the P matrix
Hadapt = dsp.RLSFilter(M,'InitialInverseCovariance',P0);
```

Running the RLS adaptive filter for 1000 iterations. As the adaptive filter converges, the filtered noise should be completely subtracted from the "signal + noise". Also the error, 'e', should contain only the original signal.

```
Hts = dsp.TimeScope('TimeSpan',1000,'YLimits',[-2,2]);
for k = 1:10
    n = step(Hn); % Noise
    s = step(Hs);
    d = step(Hd,n) + s;
    [y,e]  = step(Hadapt,n,d);
    step(Hts,[s,e]);
end
```
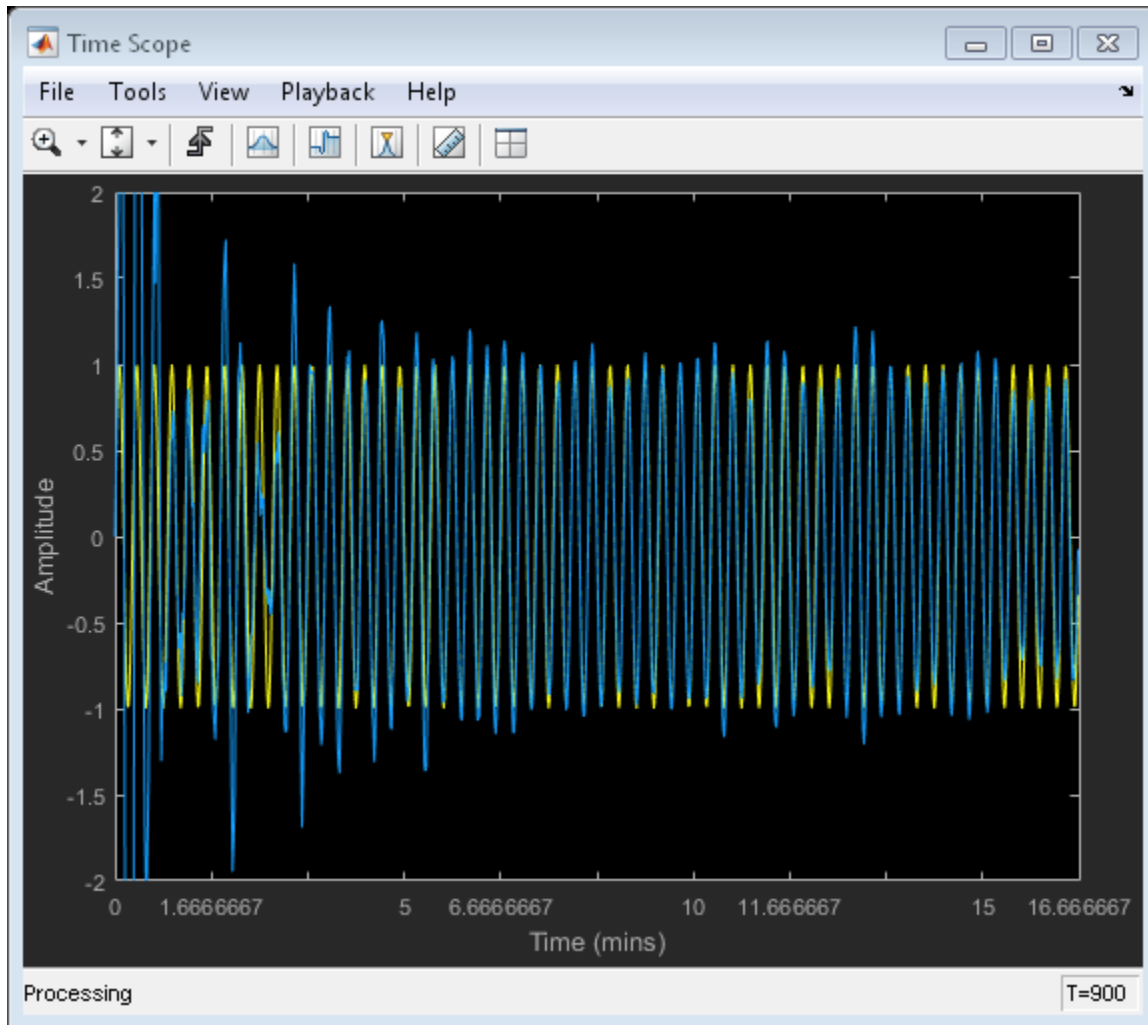
The plot shows the convergence of the adaptive filter response to the response of the FIR filter.

```
H  = abs(freqz(Hadapt.Coefficients,1,64));
H1 = abs(freqz(Hd.Numerator,1,64));

wf = linspace(0,1,64);
```

```
plot(wf,H,wf,H1);
xlabel('Normalized Frequency  (\times\pi rad/sample)');
ylabel('Magnitude');
legend('Adaptive Filter Response','Required Filter Response');
grid;
axis([O 1 O 2]);
```

Signal Enhancement Using LMS and Normalized LMS

This example 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) and normalized LMS. Both of these algorithms are available with the dsp.LMSFilter System object.

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;
lms = dsp.LMSFilter(L,'Method','LMS',...
   'WeightOutputPort',true);
nlms = dsp.LMSFilter(L,'Method','Normalized LMS',...
   'WeightOutputPort',true);
```

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(lms,x)
[mumaxnlms,mumaxmsenlms] = maxstep(nlms);

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.

```
lms.StepSize  = mumaxmselms/30;
nlms.StepSize = mumaxmsenlms/20;
```

If you know the step size to use, you can set the step size value when you create the object. For example,

```
lms = dsp.LMSFilter(L,'Method','LMS','StepSize',0.2);
```

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,wlms] = step(lms,v2,x);
[ynlms,enlms,wnlms] = step(nlms,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 filter — to compare the performance of the various techniques.

```
plot(n(900:end),[ew(900:end), elms(900:end),enlms(900:end)]);
legend('Wiener filter denoised sinusoid',...
    'LMS denoised sinusoid','NLMS denoised sinusoid');
xlabel('Time index (n)');
ylabel('Amplitude');
```
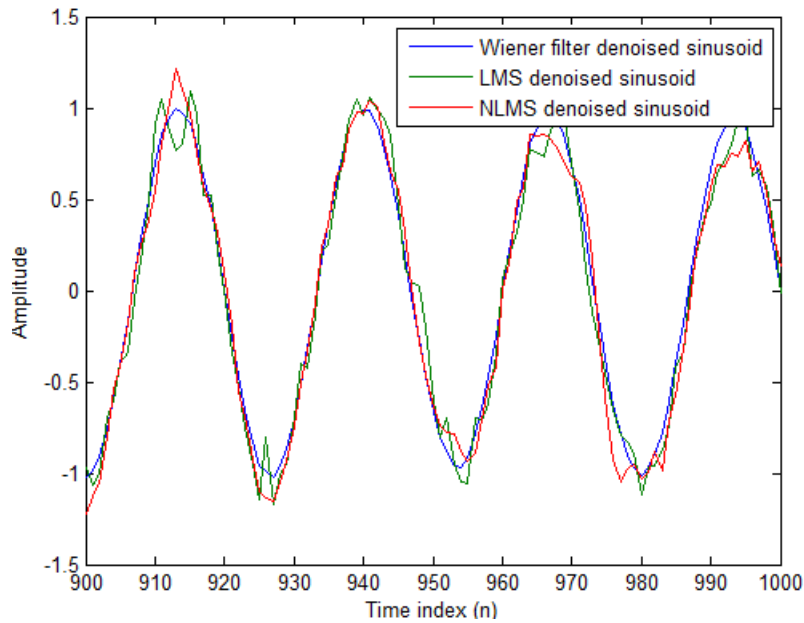
As a reference point, include the noisy signal as a dotted line in the plot.

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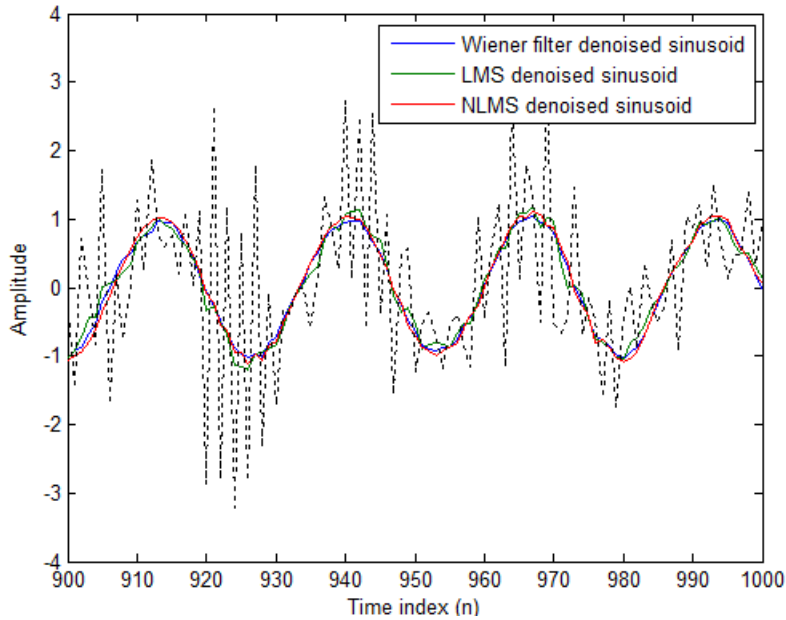
Compare the Final Coefficients

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.' wlms wnlms]

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

You can reset the internal filter states at any time by calling the `reset` method on the filter object.

For instance, the following successive calls produce the same output after resetting the object.

```
[ylms,elms,wlms] = step(lms,v2,x);
[ynlms,enlms,wnlms] = step(nlms,v2,x);
```

If you do not reset the filter object, the filter uses the final states and coefficients from the previous run as the initial conditions for the next run and set of data.

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(lms);
reset(nlms);
M = 10; % Decimation factor
mselms = msesim(lms,v2,x,M);
```

```
msenlms = msesim(nlms,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.
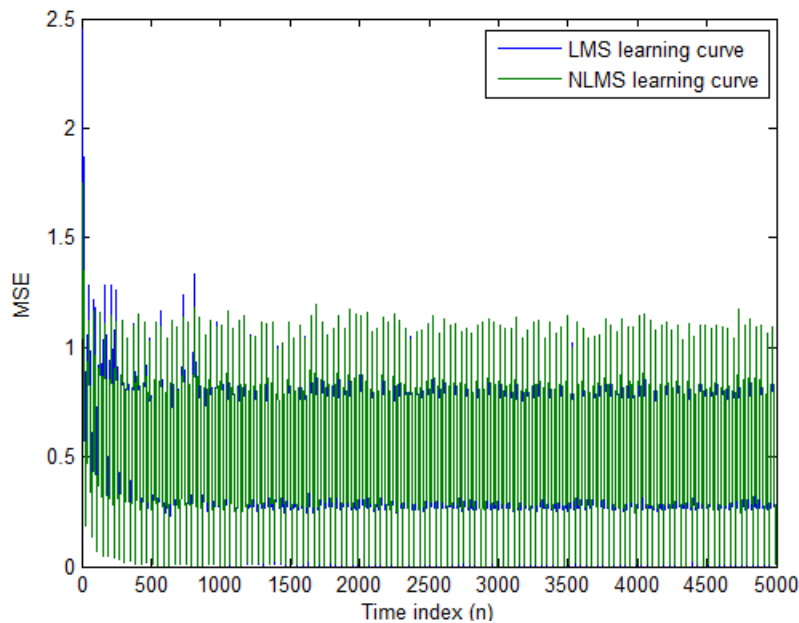
Compute the Theoretical Learning Curves

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(lms);
```

```
[mmselms,emselms,meanwlms,pmselms] = msepred(lms,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');
```
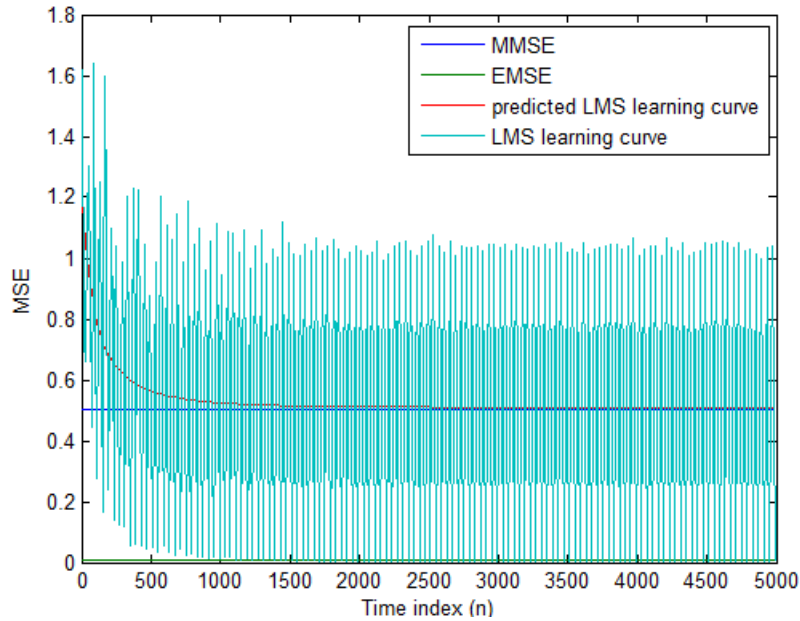
Adaptive Filters in Simulink

In this section...
"Create an Acoustic Environment in Simulink" on page 5-52
"LMS Filter Configuration for Adaptive Noise Cancellation" on page 5-54
"Modify Adaptive Filter Parameters During Model Simulation" on page 5-59
"Adaptive Filtering Examples" on page 5-63

Create an Acoustic Environment in Simulink

Adaptive filters are filters whose coefficients or weights change over time to adapt to the statistics of a signal. They are used in a variety of fields including communications, controls, radar, sonar, seismology, and biomedical engineering.

In this topic, you learn how to create an acoustic environment that simulates both white noise and colored noise added to an input signal. You later use this environment to build a model capable of adaptive noise cancellation using adaptive filtering:

1   At the MATLAB command line, type dspanc.

The DSP System Toolbox Acoustic Noise Cancellation example opens.
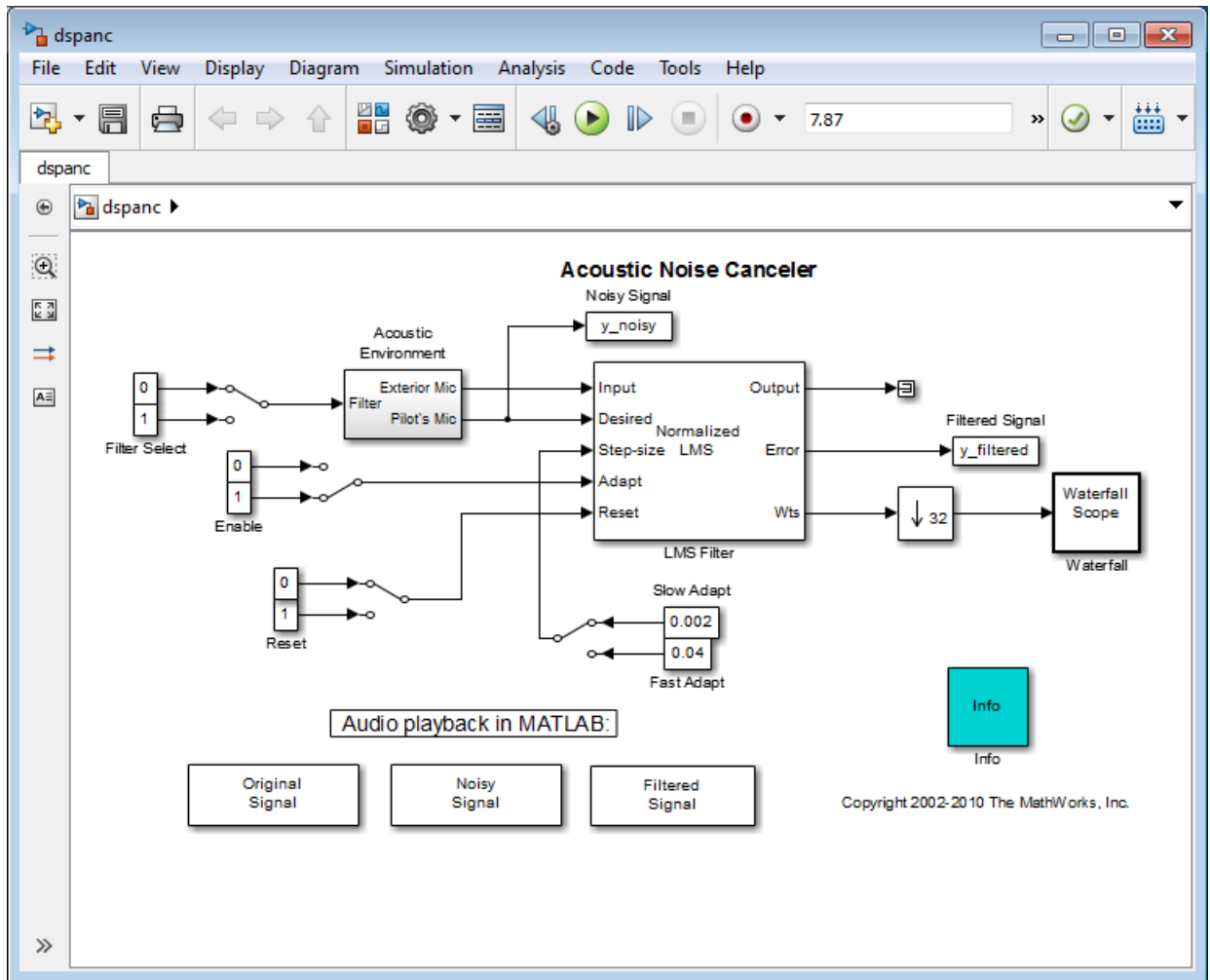
2   Copy and paste the subsystem called Acoustic Environment into a new model.

3   Double-click the Acoustic Environment subsystem.

Gaussian noise is used to create the signal sent to the Exterior Mic output port. If the input to the Filter port changes from 0 to 1, the Digital Filter block changes from a lowpass filter to a bandpass filter. The filtered noise output from the Digital Filter

block is added to signal coming from a `.wav` file to produce the signal sent to the Pilot's Mic output port.

You have now created an acoustic environment. In the following topics, you use this acoustic environment to produce a model capable of adaptive noise cancellation.

LMS Filter Configuration for Adaptive Noise Cancellation

In the previous topic, "Create an Acoustic Environment in Simulink" on page 5-52, you created a system that produced two output signals. The signal output at the Exterior Mic port is composed of white noise. The signal output at the Pilot's Mic port is composed of colored noise added to a signal from a `.wav` file. In this topic, you create an adaptive filter to remove the noise from the Pilot's Mic signal. This topic assumes that you are working on a Windows operating system:

1   If the model you created in "Create an Acoustic Environment in Simulink" on page 5-52 is not open on your desktop, you can open an equivalent model by typing

    `ex_adapt1_audio`

    at the MATLAB command prompt.

2   From the DSP System Toolbox Filtering library, and then from the Adaptive Filters library, click-and-drag an LMS Filter block into the model that contains the Acoustic Environment subsystem.

3   Double-click the LMS Filter block. Set the block parameters as follows, and then click **OK**:

    - **Algorithm** = `Normalized LMS`
    - **Filter length** = 40
    - **Step size (mu)** = `0.002`
    - **Leakage factor (0 to 1)** = 1

    The block uses the normalized LMS algorithm to calculate the forty filter coefficients. Setting the **Leakage factor (0 to 1)** parameter to 1 means that the current filter coefficient values depend on the filter's initial conditions and all of the previous input values.

4   Click-and-drag the following blocks into your model.

Block	Library	Quantity
Constant	Simulink/Sources	2
Manual Switch	Simulink/Signal Routing	1
Terminator	Simulink/Sinks	1
Downsample	Signal Operations	1
Audio Device Writer	Sinks	1
Waterfall Scope	Sinks	1

5 Connect the blocks so that your model resembles the following figure.
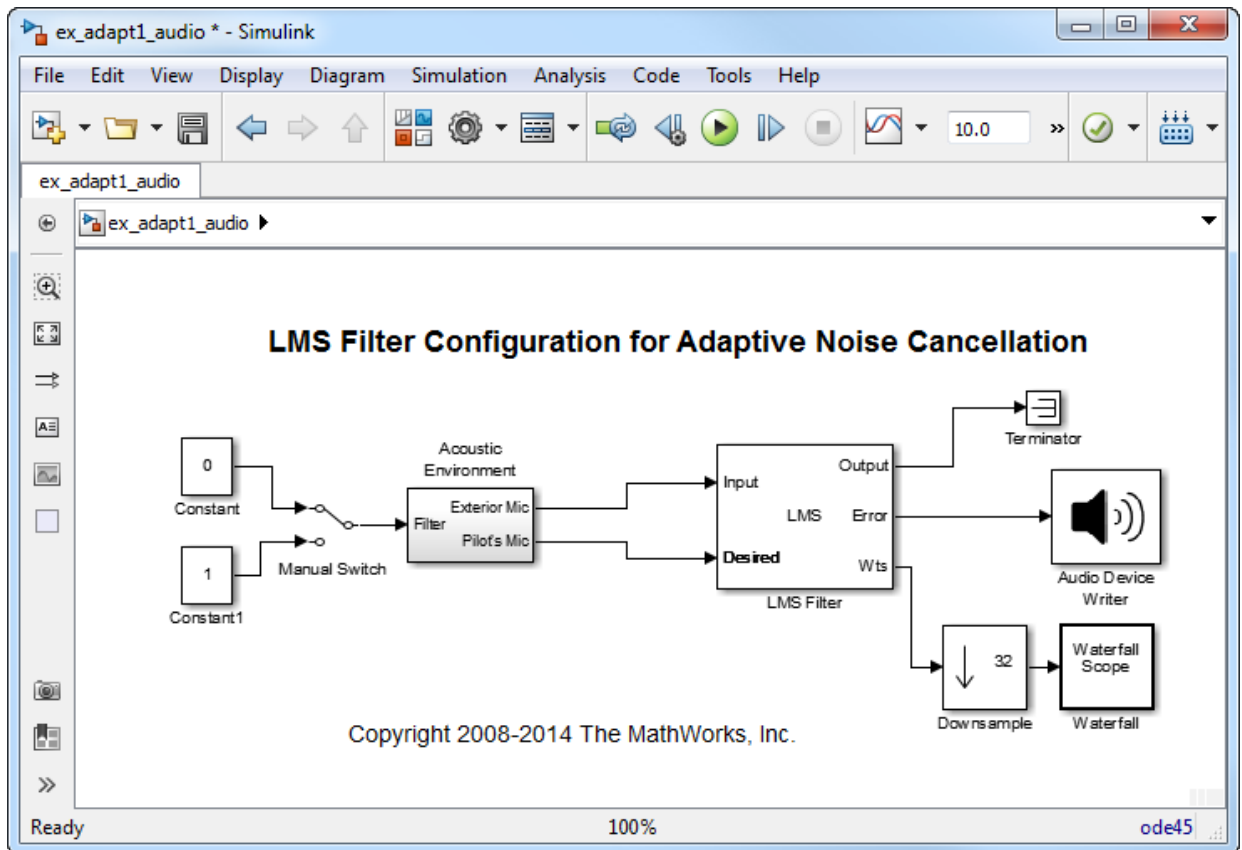
6 Double-click the Constant block. Set the **Constant value** parameter to 0 and then click **OK**.

7 Double-click the Downsample block. Set the **Downsample factor, K** parameter to 32. Click **OK**.

The filter weights are being updated so often that there is very little change from one update to the next. To see a more noticeable change, you need to downsample the output from the Wts port.

8 Double-click the Waterfall Scope block. The **Waterfall** scope window opens.

9 Click the **Scope** parameters button.
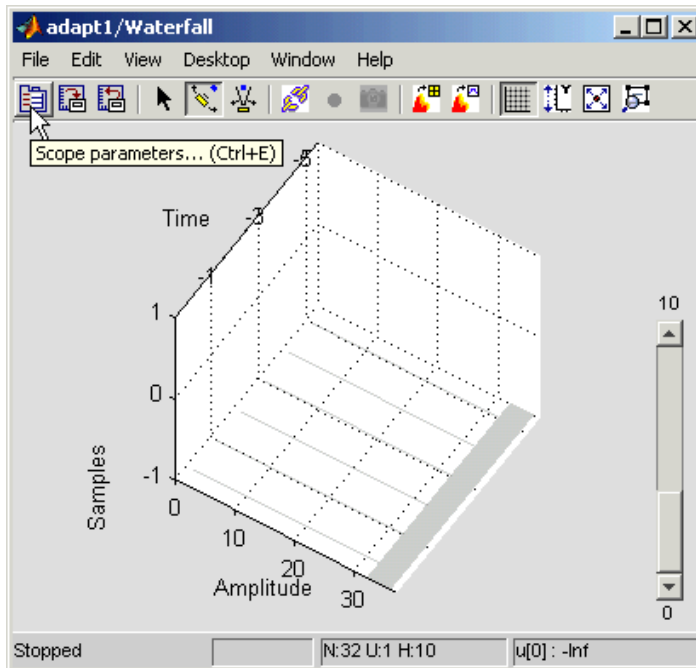
The **Parameters** window opens.
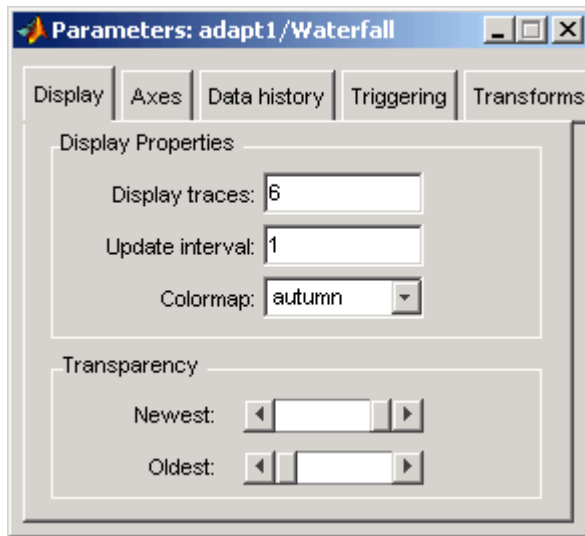
10　Click the **Axes** tab. Set the parameters as follows:

- **Y Min** = -0.188
- **Y Max** = 0.179

11　Click the **Data history** tab. Set the parameters as follows:

- **History traces** = 50
- **Data logging** = All visible

12　Close the **Parameters** window leaving all other parameters at their default values.

You might need to adjust the axes in the **Waterfall** scope window in order to view the plots.

13　Click the **Fit to view** button in the **Waterfall** scope window. Then, click-and-drag the axes until they resemble the following figure.
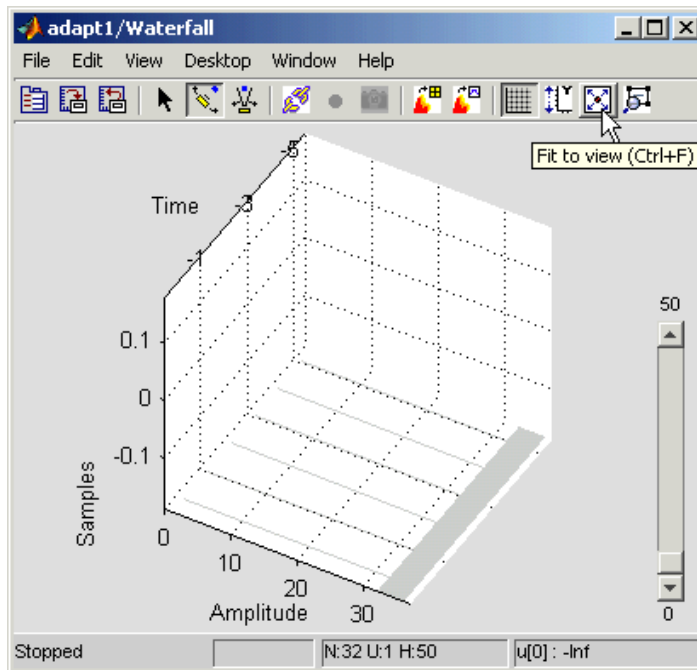
14 In the model window, from the **Simulation** menu, select **Model Configuration Parameters**. In the **Select** pane, click **Solver**. Set the parameters as follows, and then click **OK**:

- **Stop time** = `inf`
- **Type** = `Fixed-step`
- **Solver** = `Discrete (no continuous states)`

15 Run the simulation and view the results in the **Waterfall** scope window. You can also listen to the simulation using the speakers attached to your computer.

16 Experiment with changing the Manual Switch so that the input to the Acoustic Environment subsystem is either 0 or 1.

When the value is 0, the Gaussian noise in the signal is being filtered by a lowpass filter. When the value is 1, the noise is being filtered by a bandpass filter. The adaptive filter can remove the noise in both cases.

You have now created a model capable of adaptive noise cancellation. The adaptive filter in your model is able to filter out both low frequency noise and noise within a frequency range. In the next topic, "Modify Adaptive Filter Parameters During Model Simulation" on page 5-59, you modify the LMS Filter block and change its parameters during simulation.

Modify Adaptive Filter Parameters During Model Simulation

In the previous topic, "LMS Filter Configuration for Adaptive Noise Cancellation" on page 5-54, you created an adaptive filter and used it to remove the noise generated by the Acoustic Environment subsystem. In this topic, you modify the adaptive filter and adjust its parameters during simulation. This topic assumes that you are working on a Windows operating system:

1 If the model you created in "Create an Acoustic Environment in Simulink" on page 5-52 is not open on your desktop, you can open an equivalent model by typing

```
ex_adapt2_audio
```

at the MATLAB command prompt.

2 Double-click the LMS filter block. Set the block parameters as follows, and then click **OK**:

- **Specify step size via** = Input port
- **Initial value of filter weights** = 0
- Select the **Adapt port** check box.
- **Reset port** = Non-zero sample

The **Block Parameters: LMS Filter** dialog box should now look similar to the following figure.
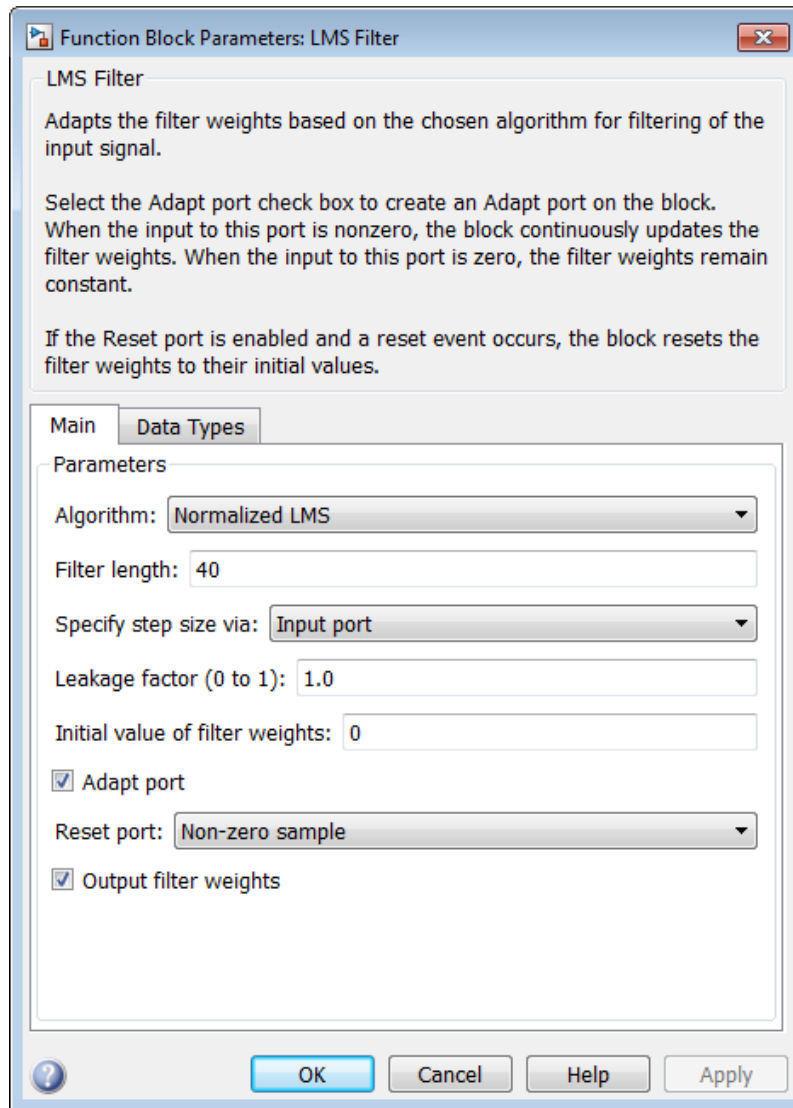
Step-size, Adapt, and Reset ports appear on the LMS Filter block.

3 Click-and-drag the following blocks into your model.

Block	Library	Quantity
Constant	Simulink/Sources	6
Manual Switch	Simulink/Signal Routing	3

4   Connect the blocks as shown in the following figure.
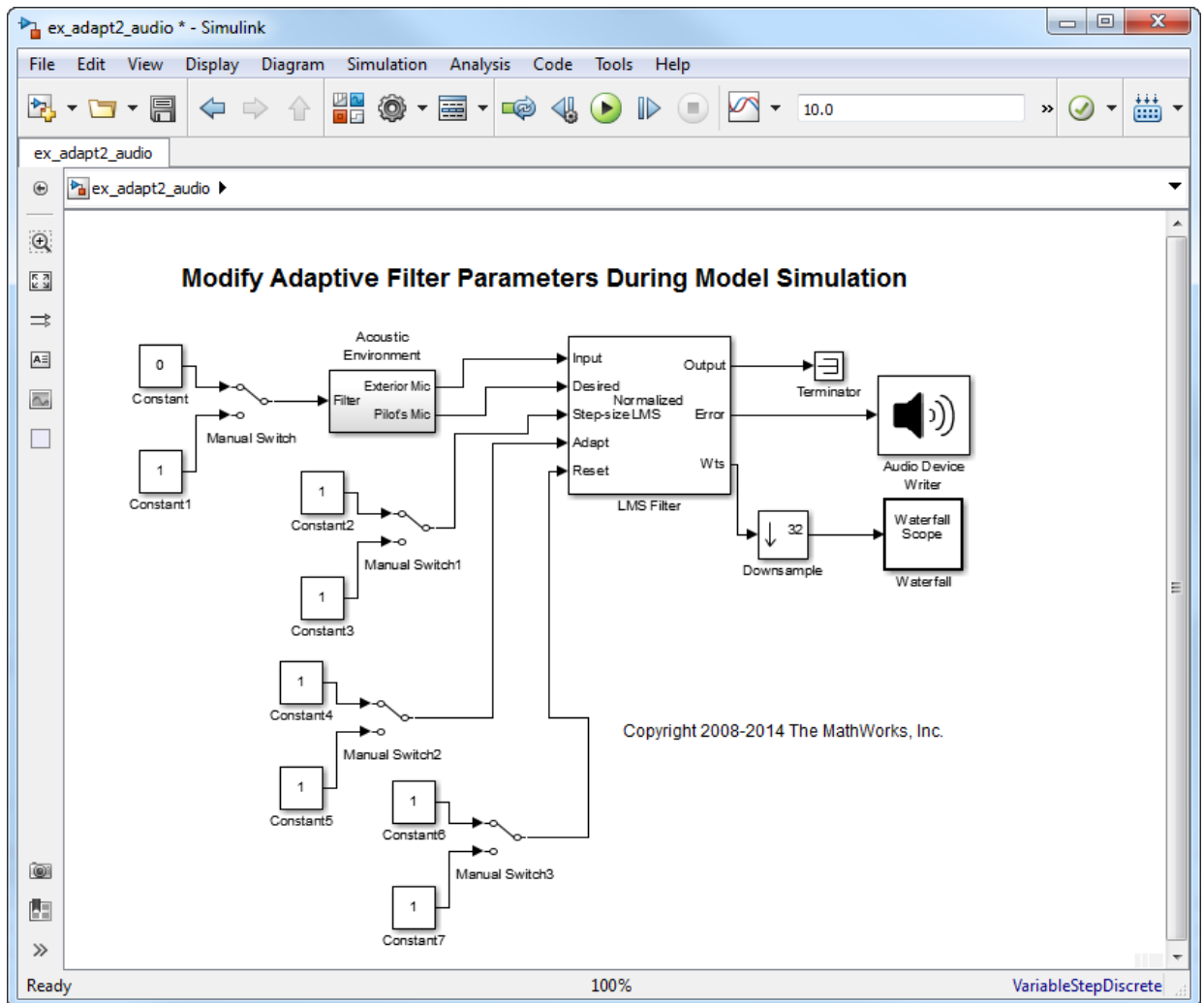
5   Double-click the Constant2 block. Set the block parameters as follows, and then click **OK**:

- **Constant value** = `0.002`
- Select the **Interpret vector parameters as 1-D** check box.
- **Sample time (-1 for inherited)** = `inf`
- **Output data type mode** = `Inherit via back propagation`

6   Double-click the Constant3 block. Set the block parameters as follows, and then click **OK**:

- **Constant value** = `0.04`
- Select the **Interpret vector parameters as 1-D** check box.
- **Sample time (-1 for inherited)** = `inf`
- **Output data type mode** = `Inherit via back propagation`

7   Double-click the Constant4 block. Set the **Constant value** parameter to `0` and then click **OK**.

8   Double-click the Constant6 block. Set the **Constant value** parameter to `0` and then click **OK**.

9   In the model window, from the **Display** menu, point to **Signals & Ports**, and select **Wide Nonscalar Lines** and **Signal Dimensions**.

10   Double-click Manual Switch2 so that the input to the Adapt port is 1.

11   Run the simulation and view the results in the **Waterfall** scope window. You can also listen to the simulation using the speakers attached to your computer.

12   Double-click the Manual Switch block so that the input to the Acoustic Environment subsystem is 1. Then, double-click Manual Switch2 so that the input to the Adapt port to 0.

The filter weights displayed in the **Waterfall** scope window remain constant. When the input to the Adapt port is 0, the filter weights are not updated.

13   Double-click Manual Switch2 so that the input to the Adapt port is 1.

The LMS Filter block updates the coefficients.

14   Connect the Manual Switch1 block to the Constant block that represents 0.002. Then, change the input to the Acoustic Environment subsystem. Repeat this procedure with the Constant block that represents 0.04.

You can see that the system reaches steady state faster when the step size is larger.

15 Double-click the Manual Switch3 block so that the input to the Reset port is 1.

The block resets the filter weights to their initial values. In the **Block Parameters: LMS Filter** dialog box, from the **Reset port** list, you chose `Non-zero sample`. This means that any nonzero input to the Reset port triggers a reset operation.

You have now experimented with adaptive noise cancellation using the LMS Filter block. You adjusted the parameters of your adaptive filter and viewed the effects of your changes while the model was running.

For more information about adaptive filters, see the following block reference pages:

- `LMS Filter`
- `RLS Filter`
- `Block LMS Filter`
- `Fast Block LMS Filter`

Adaptive Filtering Examples

DSP System Toolbox software provides a collection of adaptive filtering examples that illustrate typical applications of the adaptive filtering blocks, listed in the following table.

Adaptive Filtering Examples	Commands for Opening Examples in MATLAB
LMS Adaptive Equalization	lmsadeq
LMS Adaptive Time-Delay Estimation	lmsadtde
Nonstationary Channel Estimation	dspchanest
RLS Adaptive Noise Cancellation	rlsdemo

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

6

Multirate and Multistage Filters

Learn how to analyze, design, and implement multirate and multistage filters in MATLAB and Simulink.

Multirate Filters

In this section...
"Why Are Multirate Filters Needed?" on page 6-2
"Overview of Multirate Filters" on page 6-2

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, see 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.
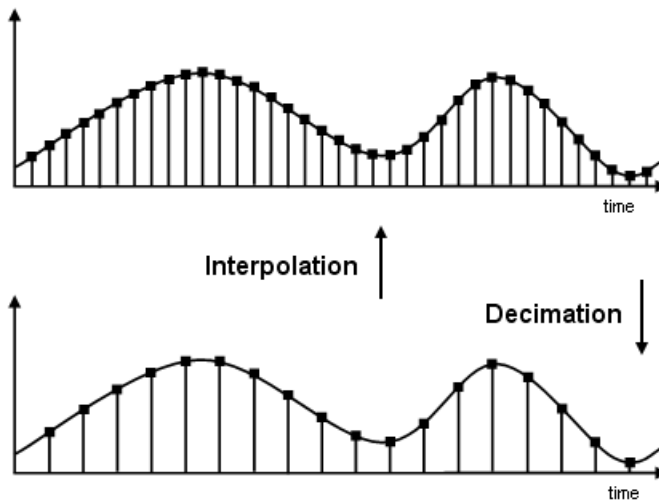
If you start with the top signal, sampled at a frequency Fs, then the bottom signal is sampled at Fs/2 frequency. In this case, the decimation factor, or M, is 2.

The following figure illustrates effect of decimation in the frequency domain.
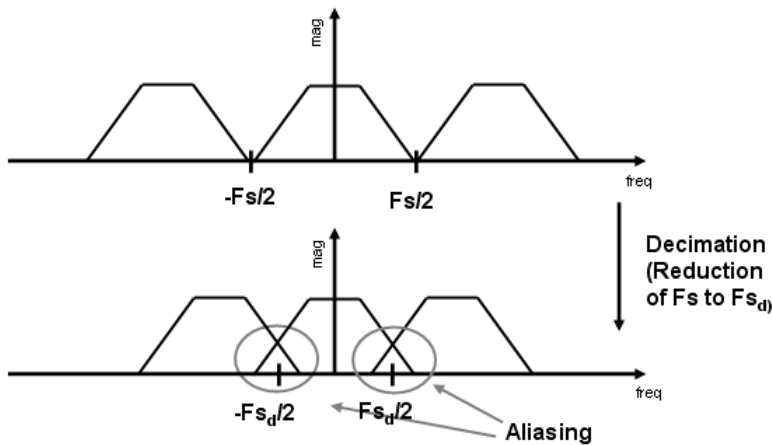
In the first graphic in the figure you can see a signal that is critically sampled, i.e. the sample rate is equal to two times the highest frequency component of the sampled

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.
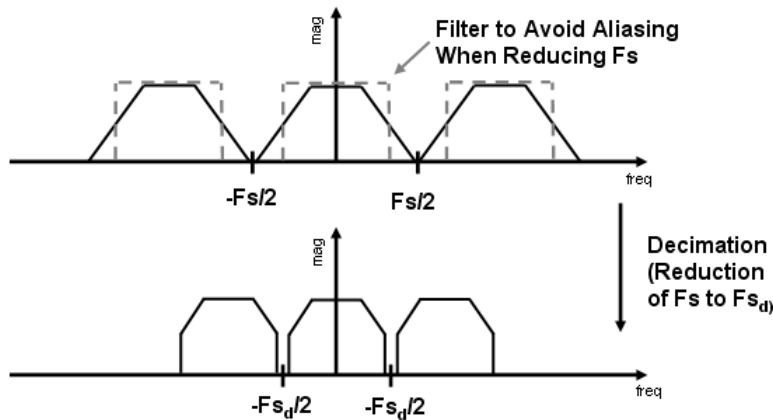
An analogous approach must be taken to avoid *imaging* when performing interpolation on a sampled signal. For more information about the effects of decimation and interpolation on a sampled signal, see "References — Multirate Filters" on page 17-4.

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:
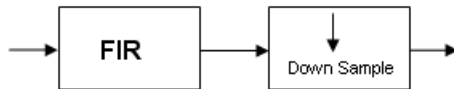
- Conversely, a filter is typically applied after interpolation to avoid imaging:
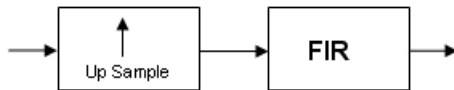
- M must be an integer. Although, if you wish to obtain an M of 4/5, you could interpolate by 4, and then decimate by 5, provided that frequency restrictions are met. This type of multirate filter will be referred to as a *sample rate converter* in the documentation that follows.

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 6-6
"Optimal Multistage Filters in DSP System Toolbox" on page 6-6

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 DSP System Toolbox

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. DSP System 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 DSP System 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 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 example 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 695 multipliers are required.

```
cost(Hd)
```

Reduce 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)
```

Compare the Responses

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')
```
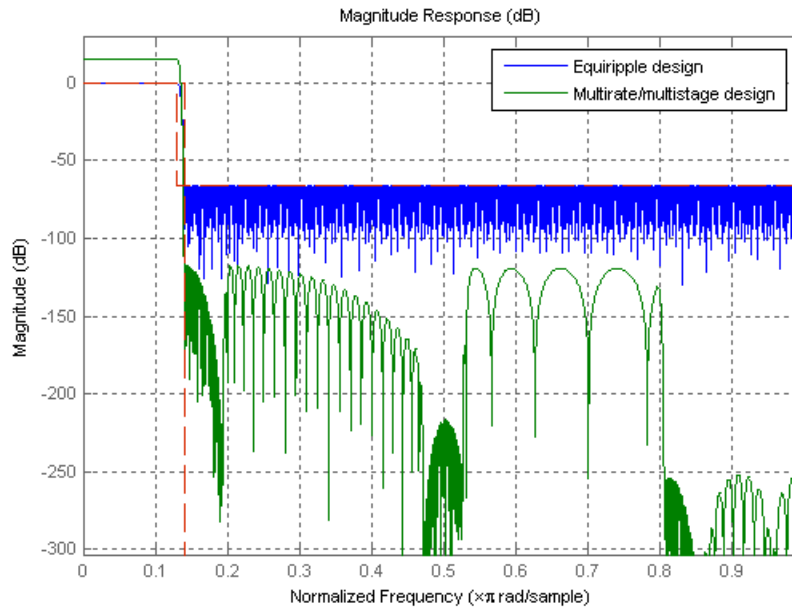
Notice that the stopband attenuation for the multistage design is about twice that of the other designs. This is because the decimators must attenuate out-of-band components by 66 dB in order to avoid aliasing that would violate the specifications. Similarly, the interpolators must attenuate images by 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([0 1 -50 30])
grid on
```
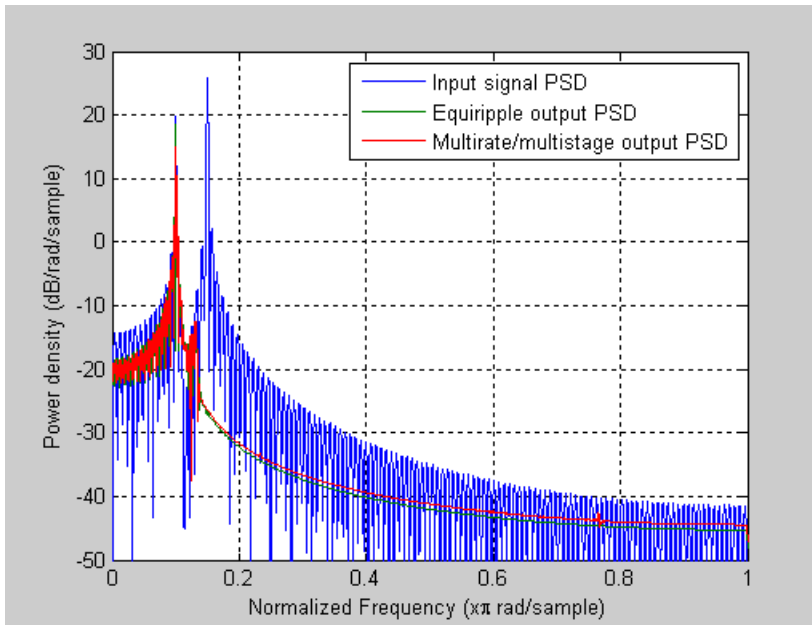
Design of Decimators/Interpolators

This example shows how to design filters for decimation and interpolation. Typically lowpass filters are used for decimation and for interpolation. When decimating, lowpass filters are used to reduce the bandwidth of a signal prior to reducing the sampling rate. This is done to minimize aliasing due to the reduction in the sampling rate. When interpolating, lowpass filters are used to remove spectral images from the low-rate signal. For general notes on lowpass filter design see the example on Designing Low Pass FIR Filters.

Input signal

Before we begin, let us define the signal that we will be throughout the example. The samples of the signal we will be using are drawn from standard normal distribution to have a flat spectrum.

```
HSource = dsp.SignalSource('SamplesPerFrame', 500);
HSource.Signal = randn(1e6,1);        % Gaussian white noise signal
```

Design of Decimators

When decimating, the bandwidth of a signal is reduced to an appropriate value so that minimal aliasing occurs when reducing the sampling rate. Suppose a signal that occupies the full Nyquist interval (i.e. has been critically sampled) has a sampling rate of 800 Hz. The signal's energy extends up to 400 Hz. If we'd like to reduce the sampling rate by a factor of 4 to 200 Hz, significant aliasing will occur unless the bandwidth of the signal is also reduced by a factor of 4. Ideally, a perfect lowpass filter with a cutoff at 100 Hz would be used. In practice, several things will occur: The signal's components between 0 and 100 Hz will be slightly distorted by the passband ripple of a non-ideal lowpass filter; there will be some aliasing due to the finite stopband attenuation of the filter; the filter will have a transition band which will distort the signal in such band. The amount of distortion introduced by each of these effects can be controlled by designing an appropriate filter. In general, to obtain a better filter, a higher filter order will be required.

Let's start by designing a simple lowpass decimator with a decimation factor of 4.

```
M   = 4;   % Decimation factor
Fp  = 80;  % Passband-edge frequency
Fst = 100;  % Stopband-edge frequency
Ap  = 0.1; % Passband peak-to-peak ripple
Ast = 80;  % Minimum stopband attenuation
```

```
Fs  = 800; % Sampling frequency
HfdDecim = fdesign.decimator(M,'lowpass',Fp,Fst,Ap,Ast,Fs)


HfdDecim =

  decimator with properties:

          MultirateType: 'Decimator'
               Response: 'Lowpass'
        DecimationFactor: 4
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: 0
                     Fs: 800
                  Fs_in: 800
                 Fs_out: 200
                  Fpass: 80
                  Fstop: 100
                  Apass: 0.1000
                  Astop: 80
```

The specifications for the filter determine that a transition band of 20 Hz is acceptable between 80 and 100 Hz and that the minimum attenuation for out of band components is 80 dB. Also that the maximum distortion for the components of interest is 0.05 dB (half the peak-to-peak passband ripple). An equiripple filter that meets these specs can be easily obtained as follows:

```
HDecim = design(HfdDecim,'equiripple', 'SystemObject', true);
measure(HDecim)

HSpec = dsp.SpectrumAnalyzer(...                          % Spectrum scope
                    'PlotAsTwoSidedSpectrum', false, ...
                    'SpectralAverages', 50, 'OverlapPercent', 50, ...
                    'Title', 'Decimator with equiripple lowpass filter',...
                    'YLimits', [-50, 0], 'SampleRate', Fs/M*2);

while ~isDone(HSource)
    inputSig = step(HSource);   % Input
    decimatedSig = step(HDecim, inputSig); % Decimator
    step(HSpec, upsample(decimatedSig,2)); % Spectrum
    % The upsampling is done to increase X-limits of SpectrumAnalyzer
    % beyond (1/M)*Fs/2 for better visualization
end
```

```
release(HSpec);
reset(HSource);


ans =

Sample Rate       : 800 Hz
Passband Edge     : 80 Hz
3-dB Point        : 85.621 Hz
6-dB Point        : 87.8492 Hz
Stopband Edge     : 100 Hz
Passband Ripple   : 0.092414 dB
Stopband Atten.   : 80.3135 dB
Transition Width  : 20 Hz
```
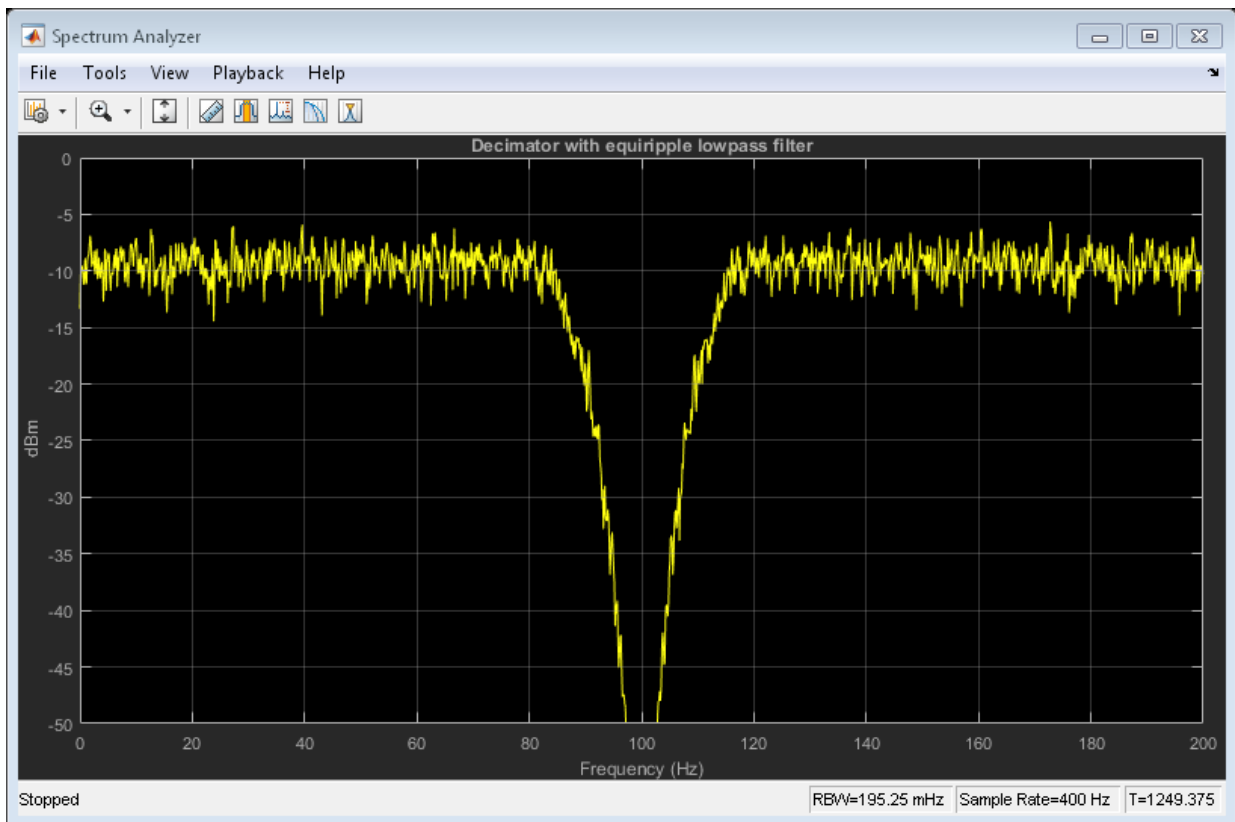
It is clear from the measurements that the design meets the specs.

Using Nyquist Filters

Nyquist filters are attractive for decimation and interpolation due to the fact that a 1/M fraction of the number of coefficients is zero. The band of the Nyquist filter is typically set to be equal to the decimation factor, this centers the cutoff frequency at (1/M)*Fs/2. In this example, the transition band is centered around (1/4)*400 = 100 Hz.

```
TW = 20; % Transition width of 20 Hz
HfdNyqDecim = fdesign.decimator(M,'nyquist',M,TW,Ast,Fs)


HfdNyqDecim =

  decimator with properties:

           MultirateType: 'Decimator'
                Response: 'Nyquist'
        DecimationFactor: 4
           Specification: 'TW,Ast'
             Description: {2x1 cell}
                    Band: 4
      NormalizedFrequency: 0
                      Fs: 800
                   Fs_in: 800
                  Fs_out: 200
         TransitionWidth: 20
                   Astop: 80
```

A Kaiser window design can be obtained in a straightforward manner.

```
HNyqDecim = design(HfdNyqDecim,'kaiserwin','SystemObject', true);

HSpec2 = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
                              'SpectralAverages', 50, 'OverlapPercent', 50, ...
                              'Title', 'Decimator with Nyquist filter',...
                              'YLimits', [-50, 0],...
                              'SampleRate', Fs/M*2);        % Spectrum scope
while ~isDone(HSource)
    inputSig = step(HSource);   % Input
    decimatedSig = step(HNyqDecim, inputSig);   % Decimator
    step(HSpec2, upsample(decimatedSig,2));  % Spectrum
    % The upsampling is done to increase X-limits of SpectrumAnalyzer
```

```
      % beyond (1/M)*Fs/2 for better visualization
end
release(HSpec2);
reset(HSource);
```
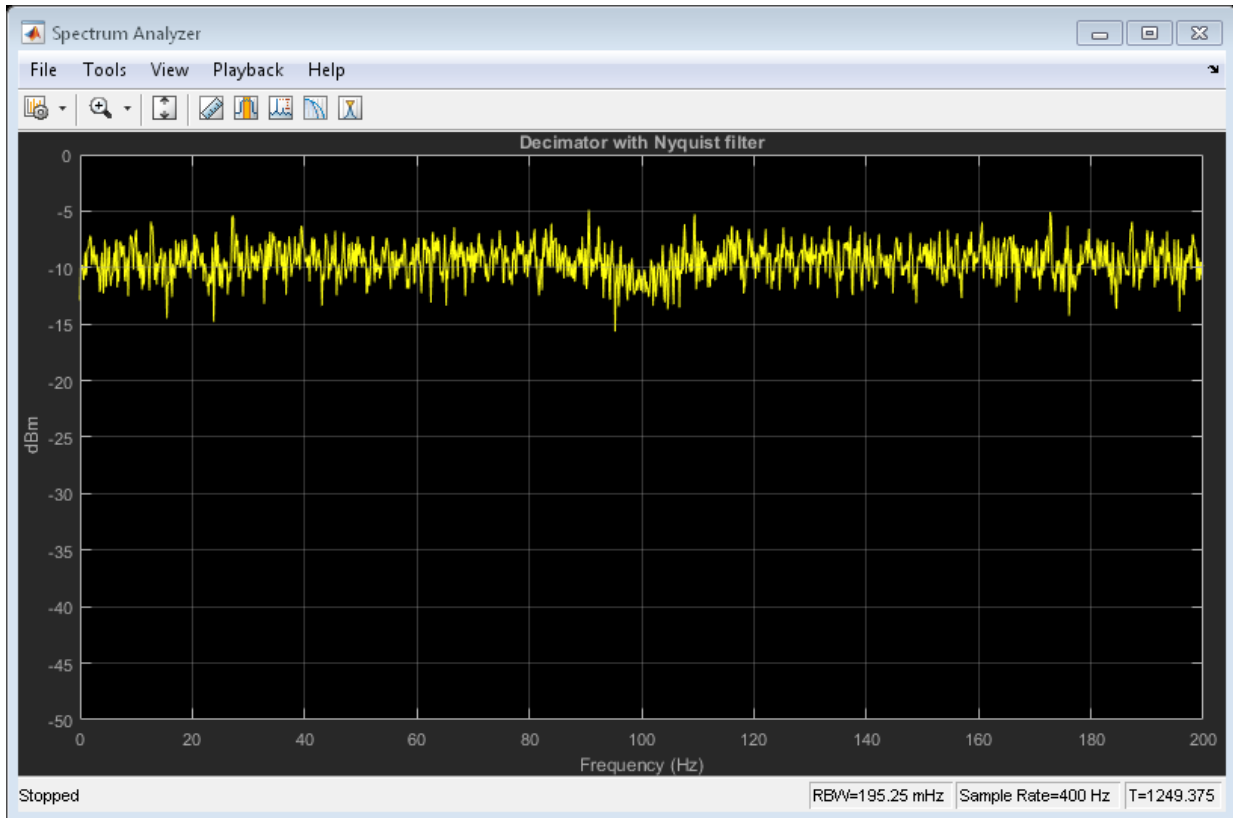

Aliasing with Nyquist Filters

Suppose the signal to be filtered has a flat spectrum. Once filtered, it acquires the spectral shape of the filter. After reducing the sampling rate, this spectrum is repeated with replicas centered around multiples of the new lower sampling frequency. An illustration of the spectrum of the decimated signal can be found from:

```
NFFT = 4096;
[H,f] = freqz(HNyqDecim,NFFT,'whole',Fs);
figure;
```

```matlab
plot(f-Fs/2,20*log10(abs(fftshift(H))))
grid on
hold on
plot(f-Fs/M,20*log10(abs(fftshift(H))),'r-')
plot(f-Fs/2-Fs/M,20*log10(abs(fftshift(H))),'k-')
legend('Baseband spectrum', ...
    'First positive replica', 'First negative replica')
title('Alisasing with Nyquist filter');
fig = gcf;
fig.Color = 'White';
hold off
```
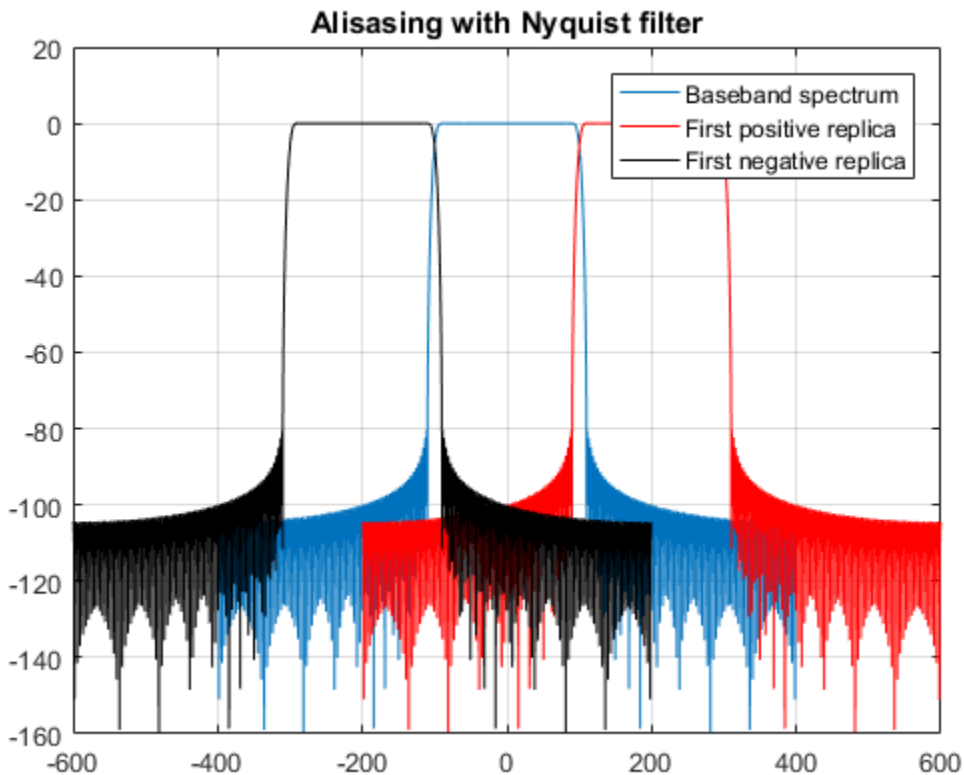

Note that the replicas overlap somewhat, so aliasing is introduced. However, the aliasing only occurs in the transition band. That is, significant energy (above the prescribed 80

dB) from the first replica only aliases into the baseband between 90 and 100 Hz. Since the filter was transitioning in this region anyway, the signal has been distorted in that band and aliasing there is not important.

On the other hand, notice that although we have used the same transition width as with the lowpass design from above, we actually retain a wider usable band (90 Hz rather than 80) when comparing this Nyquist design with the original lowpass design. To illustrate this, let's follow the same procedure to plot the spectrum of the decimated signal when the lowpass design from above is used

```matlab
[H,f] = freqz(HDecim,NFFT,'whole',Fs);
figure;
plot(f-Fs/2,20*log10(abs(fftshift(H))))
grid on
hold on
plot(f-Fs/M,20*log10(abs(fftshift(H))),'r-')
plot(f-Fs/2-Fs/M,20*log10(abs(fftshift(H))),'k-')
legend('Baseband spectrum', ...
    'First positive replica', 'First negative replica')
title('Alisasing with lowpass filter');
fig = gcf;
fig.Color = 'White';
hold off
```
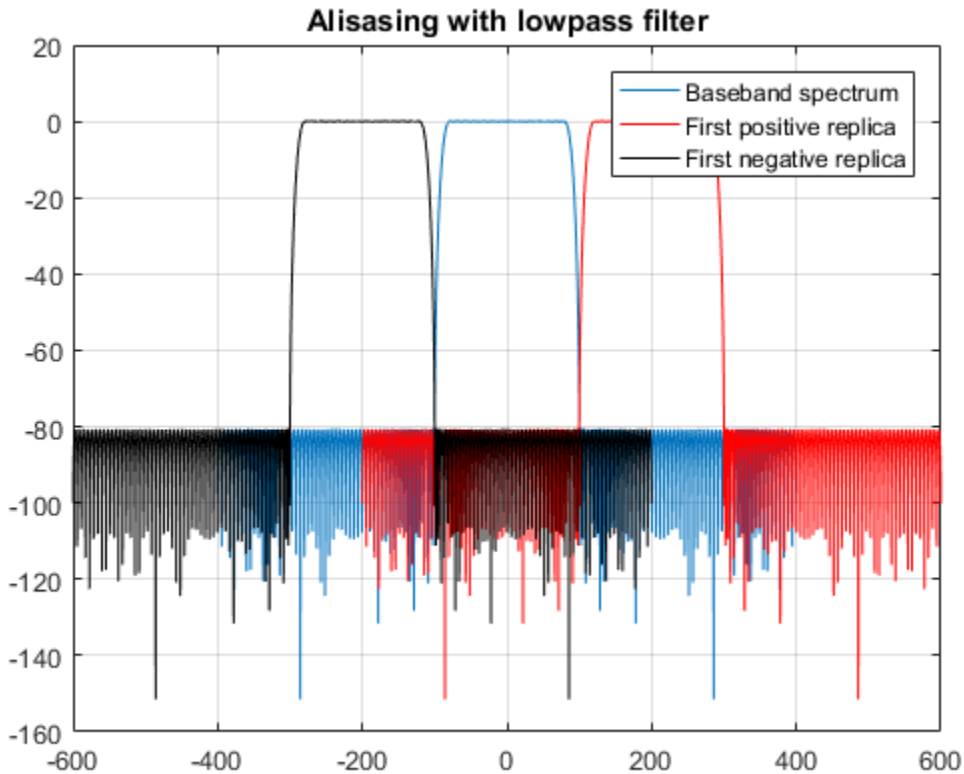
In this case, there is no significant overlap (above 80 dB) between replicas, however because the transition region started at 80 Hz, the resulting decimated signal has a smaller usable bandwidth.

Decimating by 2: Halfband Filters

When the decimation factor is 2, the Nyquist filter becomes a halfband filter. These filters are very attractive because just about half of their coefficients are equal to zero. Often, to design Nyquist filters when the band is an even number, it is desirable to perform a multistage design that uses halfband filters in some/all of the stages.

```
HfdHBDecim = fdesign.decimator(2,'halfband');
HHBDecim = design(HfdHBDecim,'equiripple','SystemObject', true);
HSpec3 = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
```

```
                                'SpectralAverages', 50, 'OverlapPercent', 50, ...
                                'Title', 'Decimator with halfband filter',...
                                'YLimits', [-50, 0],...
                                'SampleRate', Fs);          % Spectrum scope
while ~isDone(HSource)
    inputSig = step(HSource);    % Input
    decimatedSig = step(HHBDecim, inputSig);    % Decimator
    step(HSpec3, upsample(decimatedSig,2));   % Spectrum
end
release(HSpec3);
reset(HSource);
```
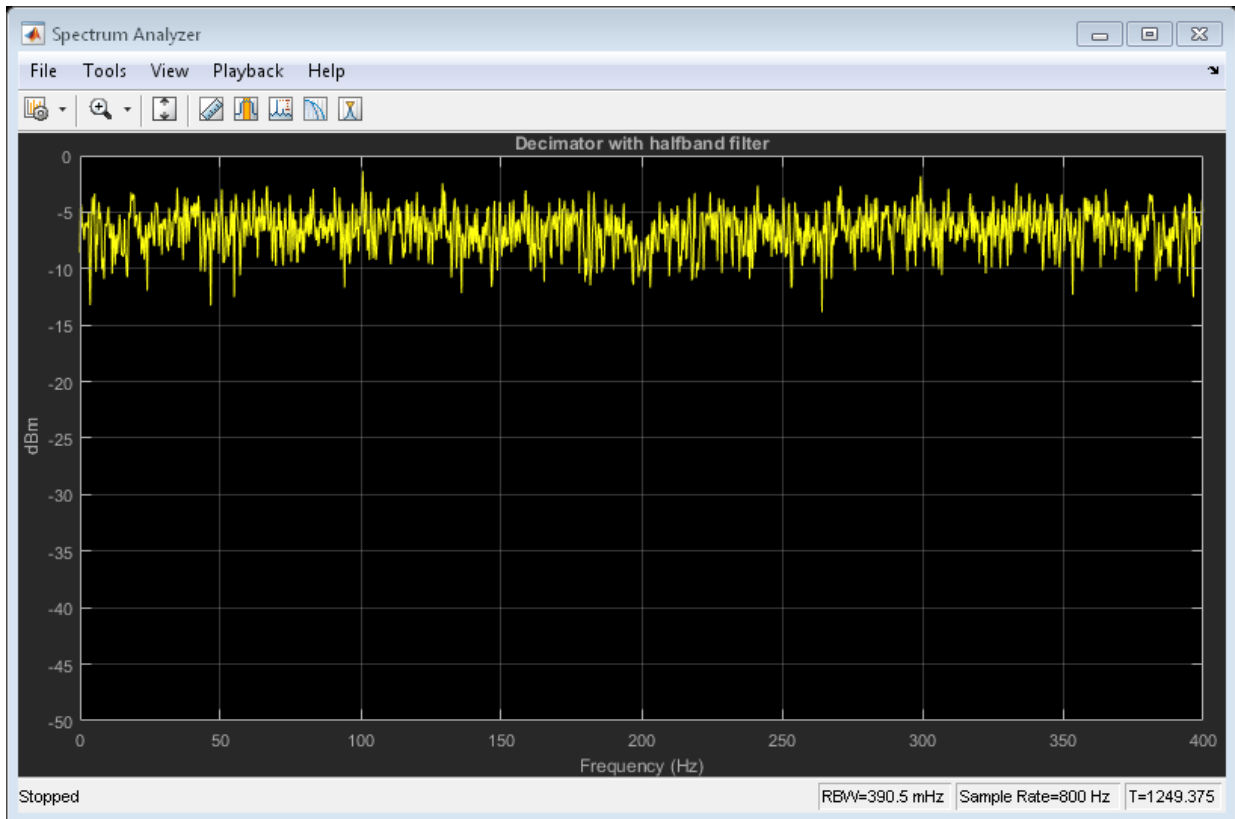

As with other Nyquist filters, when halfbands are used for decimation, aliasing will occur only in the transition region.

Interpolation

When interpolating a signal, the baseband response of the signal should be left as unaltered as possible. Interpolation is obtained by removing spectral replicas when the sampling rate is increased.

Suppose we have a signal sampled at 48 Hz. If it is critically sampled, there is significant energy in the signal up to 24 Hz. If we want to interpolate by a factor of 4, we would ideally design a lowpass filter running at 192 Hz with a cutoff at 24 Hz. As with decimation, in practice an acceptable transition width needs to be incorporated into the design of the lowpass filter used for interpolation along with passband ripple and a finite stopband attenuation. For example, consider the following specs:

```
L   = 4;   % Interpolation factor
Fp  = 22;  % Passband-edge frequency
Fst = 24;  % Stopband-edge frequency
Ap  = 0.1; % Passband peak-to-peak ripple
Ast = 80;  % Minimum stopband attenuation
Fs  = 48;  % Sampling frequency
HfdInterp = fdesign.interpolator(L,'lowpass',Fp,Fst,Ap,Ast,Fs*L)


HfdInterp =

  interpolator with properties:

          MultirateType: 'Interpolator'
               Response: 'Lowpass'
    InterpolationFactor: 4
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
      NormalizedFrequency: 0
                     Fs: 192
                  Fs_in: 48
                 Fs_out: 192
                  Fpass: 22
                  Fstop: 24
                  Apass: 0.1000
                  Astop: 80
```

An equiripple design that meets the specs can be found in the same manner as with decimators

```
HInterp = design(HfdInterp,'equiripple','SystemObject', true);

HSpec4 = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
                  'SpectralAverages', 50, 'OverlapPercent', 50, ...
                  'Title', 'Interpolator with equiripple lowpass filter',...
                  'SampleRate', Fs*L);          % Spectrum scope
while ~isDone(HSource)
    inputSig = step(HSource);   % Input
    interpSig = step(HInterp, inputSig);   % Interpolator
    step(HSpec4, interpSig);  % Spectrum
end
release(HSpec4);
reset(HSource);
```
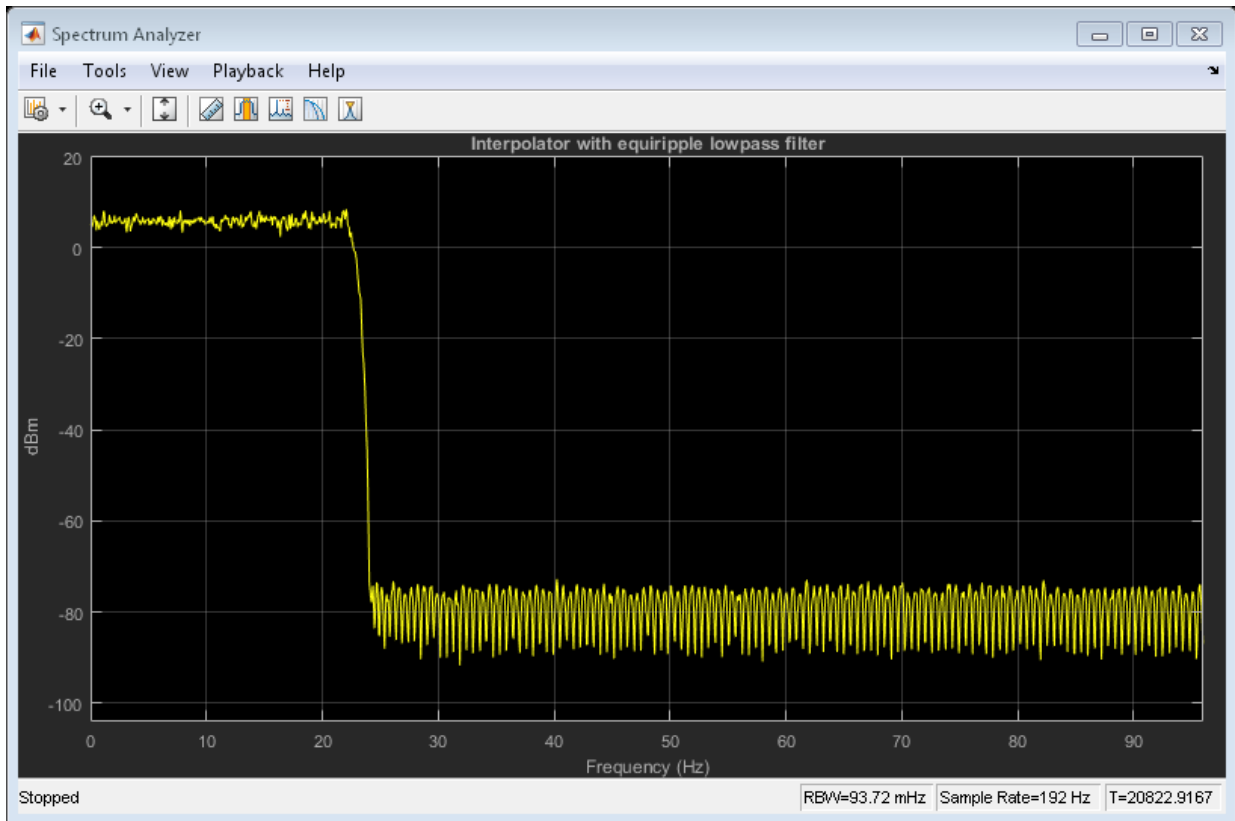
Notice that the filter has a gain of 6 dBm. In general interpolators will have a gain equal to the interpolation factor. This is needed for the signal being interpolated to maintain the same range after interpolation. For example,

```
release(HInterp);
HSin = dsp.SineWave('Frequency', 18, 'SampleRate', Fs, ...
                    'SamplesPerFrame', 100);
interpSig = step(HInterp,step(HSin));
HPlot = dsp.ArrayPlot('YLimits', [-2, 2], ...
                      'Title', 'Sine wave interpolated');
step(HPlot, interpSig(200:300)) % Plot the output
```
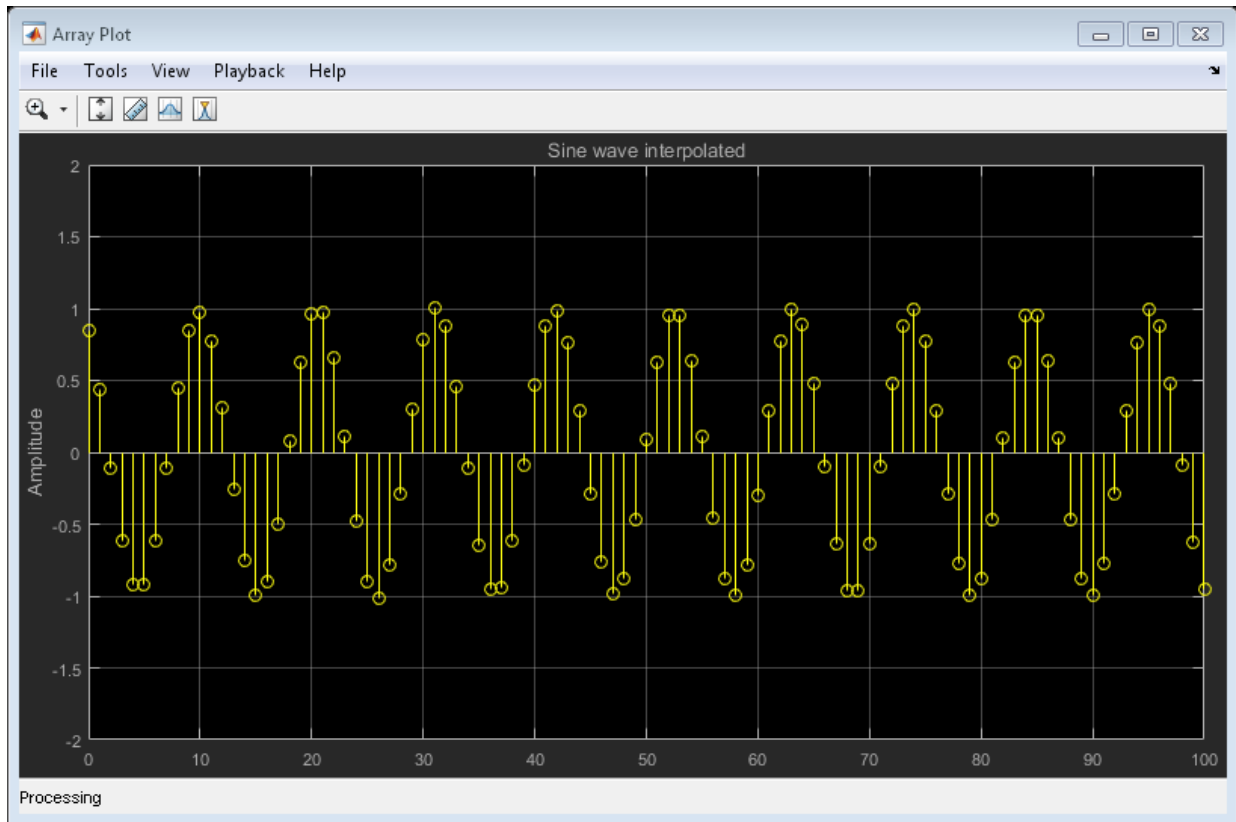

Note that although the filter has a gain of 4, the interpolated signal has the same amplitude as the original signal.

Use of Nyquist Filters for Interpolation

Similar to the decimation case, Nyquist filters are attractive for interpolation purposes. Moreover, given that there is a coefficient equal to zero every L samples, the use of Nyquist filters ensures that the samples from the input signal are retained unaltered at the output. This is not the case for other lowpass filters when used for interpolation (on the other hand, distortion may be minimal in other filters, so this is not necessarily a huge deal).

```
TW = 2;
HfdNyqInterp = fdesign.interpolator(L,'nyquist',L,TW,Ast,Fs*L)
HNyqInterp = design(HfdNyqInterp,'kaiserwin', 'SystemObject', true);

HSpec5 = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
                              'SpectralAverages', 30, 'OverlapPercent', 50, ...
                              'Title', 'Interpolator with Nyquist filter',...
                              'SampleRate', Fs*L);          % Spectrum scope
while ~isDone(HSource)
    inputSig = step(HSource);   % Input
    interpSig = step(HNyqInterp, inputSig);   % Decimator
    step(HSpec5, interpSig);  % Spectrum
end
release(HSpec5);
reset(HSource);


HfdNyqInterp =

  interpolator with properties:

          MultirateType: 'Interpolator'
               Response: 'Nyquist'
    InterpolationFactor: 4
          Specification: 'TW,Ast'
            Description: {2x1 cell}
                   Band: 4
      NormalizedFrequency: 0
                     Fs: 192
                  Fs_in: 48
                 Fs_out: 192
         TransitionWidth: 2
                  Astop: 80
```
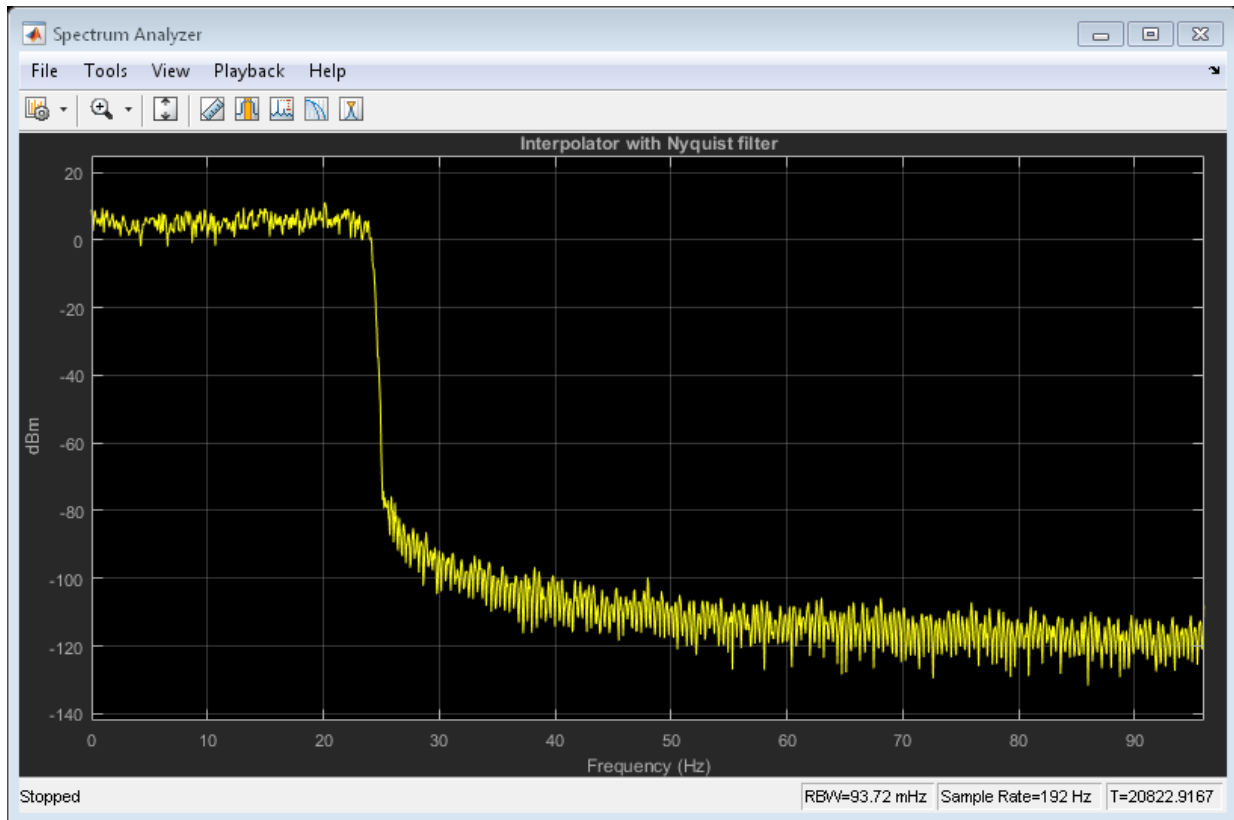
In an analogous manner to decimation, when used for interpolation, Nyquist filters allow some degree of imaging. That is, some frequencies above the cutoff frequency are not attenuated by the value of Ast. However, this occurs only in the transition band of the filter. On the other hand, once again a wider portion of the baseband of the original signal is maintained intact when compared to a lowpass filter with stopband-edge at the ideal cutoff frequency when both filters have the same transition width.

Filter Banks

Multirate filters alter the sample rate of the input signal during the filtering process. Such filters are useful in both rate conversion and filter bank applications.

The `Dyadic Analysis Filter Bank` block decomposes a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The `Dyadic Synthesis Filter Bank` block reconstructs a signal decomposed by the Dyadic Analysis Filter Bank block.

To use a dyadic synthesis filter bank to perfectly reconstruct the output of a dyadic analysis filter bank, the number of levels and tree structures of both filter banks *must* be the same. In addition, the filters in the synthesis filter bank *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction will not be perfect.

Dyadic Analysis Filter Banks

Dyadic analysis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic analysis filter banks with either a symmetric or asymmetric tree structure.
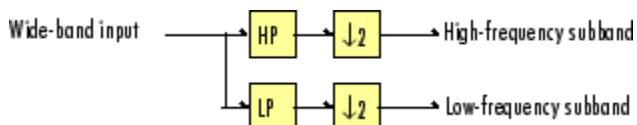
Each unit consists of a lowpass (LP) and highpass (HP) FIR filter pair, followed by a decimation by a factor of 2. The filters are halfband filters with a cutoff frequency of $F_s / 4$, a quarter of the input sampling frequency. Each filter passes the frequency band that the other filter stops.

The unit decomposes its input into adjacent high-frequency and low-frequency subbands. Compared to the input, each subband has half the bandwidth (due to the half-band filters) and half the sample rate (due to the decimation by 2).

Note The following figures illustrate the *concept* of a filter bank, but *not* how the block implements a filter bank; the block uses a more efficient polyphase implementation.
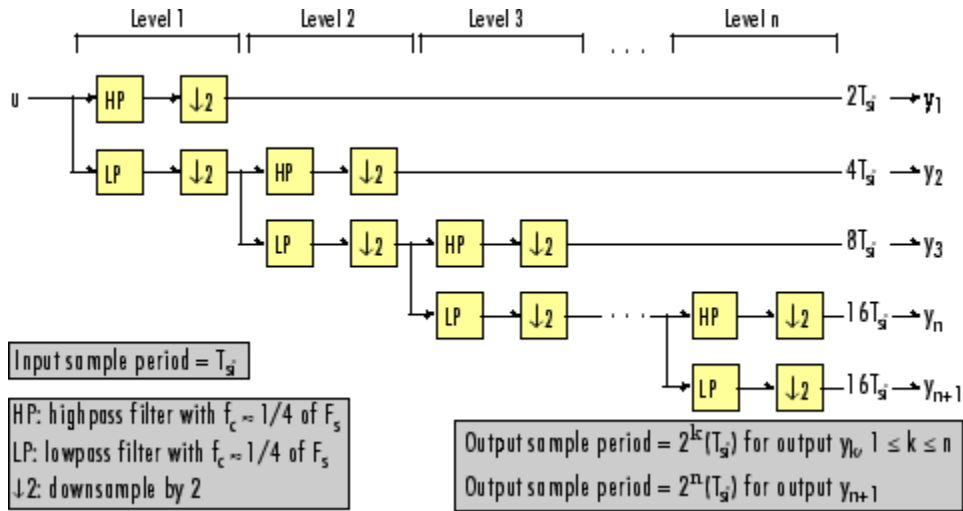
n-Level Asymmetric Dyadic Analysis Filter Bank

Use the above figure and the following figure to compare the two tree structures of the dyadic analysis filter bank. Note that the asymmetric structure decomposes only the low-frequency output from each level, while the symmetric structure decomposes the high- and low-frequency subbands output from each level.
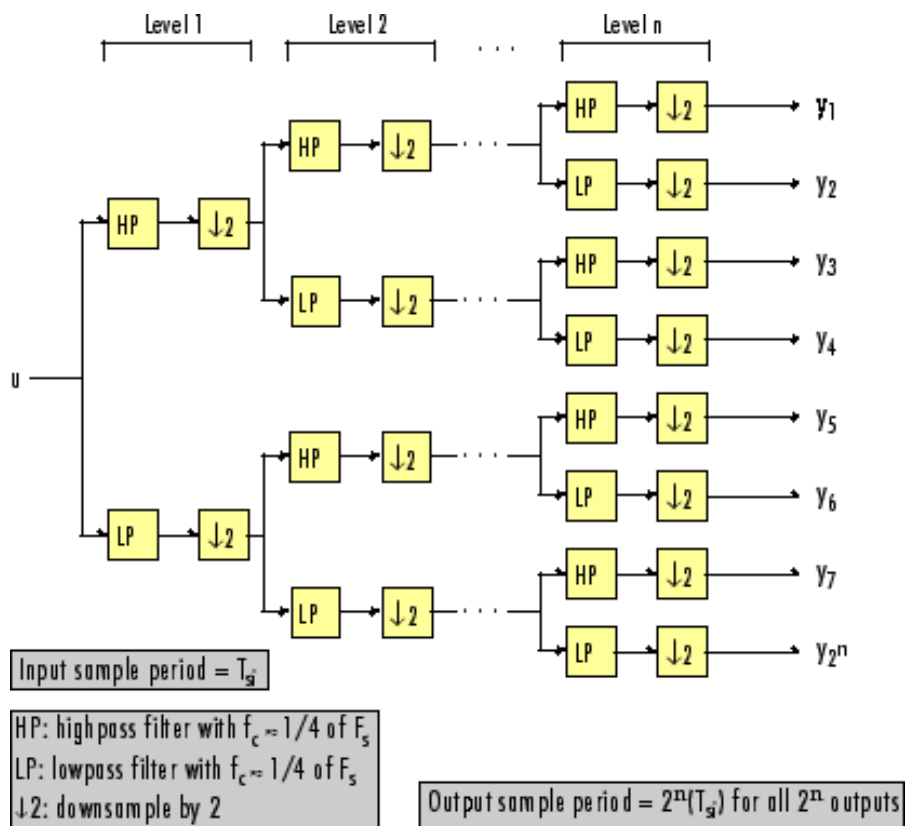
n-Level Symmetric Dyadic Analysis Filter Bank

The following table summarizes the key characteristics of the symmetric and asymmetric dyadic analysis filter bank.

Notable Characteristics of Asymmetric and Symmetric Dyadic Analysis Filter Banks

Characteristic	N-Level Symmetric	N-Level Asymmetric
Low- and High-Frequency Subband Decomposition	All the low-frequency and high-frequency subbands in a level are decomposed in the next level.	Each level's low-frequency subband is decomposed in the next level, and each level's high-frequency band is an output of the filter bank.
Number of Output Subbands	2^n	$n+1$
Bandwidth and Number of Samples in Output Subbands	For an input with bandwidth BW and N samples, all outputs have bandwidth $BW / 2^n$ and $N / 2^n$ samples.	For an input with bandwidth BW and N samples, y_k has the bandwidth BW_k, and N_k samples, where $$BW_k = \begin{cases} BW / 2^k & (1 \le k \le n) \\ BW / 2^n & (k = n + 1) \end{cases}$$ $$N_k = \begin{cases} N / 2^k & (1 \le k \le n) \\ N / 2^n & (k = n + 1) \end{cases}$$ The bandwidth of, and number of samples in each subband (except the last) is half those of the previous subband. The last two subbands have the same bandwidth and number of samples since they originate from the same level in the filter bank.
Output Sample Period	All output subbands have a sample period of $2^n(T_{si})$	Sample period of kth output $$= \begin{cases} 2^k(T_{si}) & (1 \le k \le n) \\ 2^n(T_{si}) & (k = n + 1) \end{cases}$$

Characteristic	N-Level Symmetric	N-Level Asymmetric
		Due to the decimations by 2, the sample period of each subband (except the last) is twice that of the previous subband. The last two subbands have the same sample period since they originate from the same level in the filter bank.
Total Number of Output Samples	The total number of samples in all of the output subbands is equal to the number of samples in the input (due to the of decimations by 2 at each level).	
Wavelet Applications	In wavelet applications, the highpass and lowpass wavelet-based filters are designed so that the aliasing introduced by the decimations are exactly canceled in reconstruction.	

Dyadic Synthesis Filter Banks

Dyadic synthesis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic synthesis filter banks with either a asymmetric or symmetric tree structure as illustrated in the figures entitled n-Level Asymmetric Dyadic Synthesis Filter Bank and n-Level Symmetric Dyadic Synthesis Filter Bank.
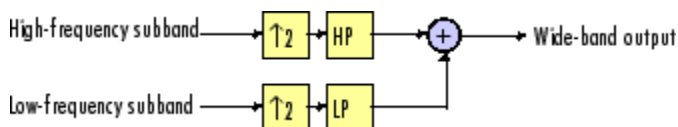
Each unit consists of a lowpass (LP) and highpass (HP) FIR filter pair, preceded by an interpolation by a factor of 2. The filters are halfband filters with a cutoff frequency of $F_s / 4$, a quarter of the input sampling frequency. Each filter passes the frequency band that the other filter stops.

The unit takes in adjacent high-frequency and low-frequency subbands, and reconstructs them into a wide-band signal. Compared to each subband input, the output has twice the bandwidth and twice the sample rate.

Note The following figures illustrate the *concept* of a filter bank, but *not* how the block implements a filter bank; the block uses a more efficient polyphase implementation.
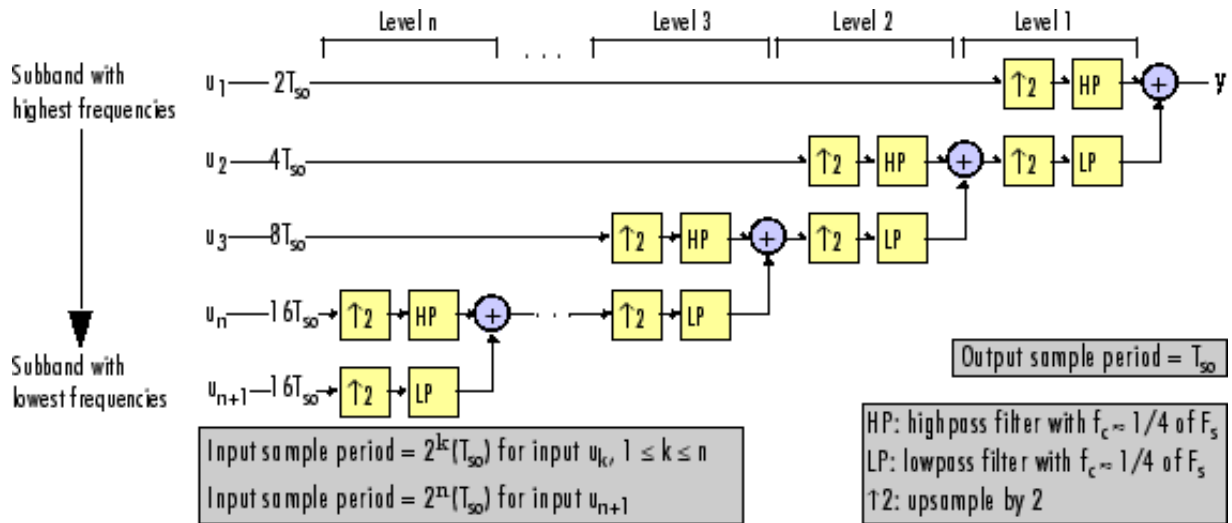
n-Level Asymmetric Dyadic Synthesis Filter Bank

Use the above figure and the following figure to compare the two tree structures of the dyadic synthesis filter bank. Note that in the asymmetric structure, the low-frequency subband input to each level is the output of the previous level, while the high-frequency subband input to each level is an input to the filter bank. In the symmetric structure, both the low- and high-frequency subband inputs to each level are outputs from the previous level.

n-Level Symmetric Dyadic Synthesis Filter Bank

The following table summarizes the key characteristics of symmetric and asymmetric dyadic synthesis filter banks.

Notable Characteristics of Asymmetric and Symmetric Dyadic Synthesis Filter Banks

Characteristic	N-Level Symmetric	N-Level Asymmetric
Input Paths Through the Filter Bank	Both the high-frequency and low-frequency input subbands to each level (except the first) are the outputs of the previous level. The inputs to the first level are the inputs to the filter bank.	The low-frequency subband input to each level (except the first) is the output of the previous level. The low-frequency subband input to the first level, and the high-frequency subband input to each level, are inputs to the filter bank.

Characteristic	N-Level Symmetric	N-Level Asymmetric
Number of Input Subbands	2^n	$n+1$
Bandwidth and Number of Samples in Input Subbands	All inputs subbands have bandwidth BW / 2^n and N / 2^n samples, where the output has bandwidth BW and N samples.	For an output with bandwidth BW and N samples, the kth input subband has the following bandwidth and number of samples. $$BW_k = \begin{cases} BW / 2^k & (1 \le k \le n) \\ BW / 2^n & (k = n + 1) \end{cases}$$ $$N_k = \begin{cases} N / 2^k & (1 \le k \le n) \\ N / 2^n & (k = n + 1) \end{cases}$$
Input Sample Periods	All input subbands have a sample period of $2^n(T_{so})$, where the output sample period is T_{so}.	Sample period of kth input subband $$= \begin{cases} 2^k(T_{so}) & (1 \le k \le n) \\ 2^n(T_{so}) & (k = n + 1) \end{cases}$$ where the output sample period is T_{so}.
Total Number of Input Samples	The number of samples in the output is always equal to the total number of samples in all of the input subbands.	
Wavelet Applications	In wavelet applications, the highpass and lowpass wavelet-based filters are carefully selected so that the aliasing introduced by the decimation in the dyadic *analysis* filter bank is exactly canceled in the reconstruction of the signal in the dyadic *synthesis* filter bank.	

For more information, see Dyadic Synthesis Filter Bank.

Multirate Filtering in Simulink

DSP System Toolbox software provides a collection of multirate filtering examples that illustrate typical applications of the multirate filtering blocks.

Multirate Filtering Examples	Description	Command for Opening Examples in MATLAB
Audio Sample Rate Conversion	Illustrates sample rate conversion of an audio signal from 22.050 kHz to 8 kHz using a multirate FIR rate conversion approach	dspaudiosrc
Sigma-Delta A/D Converter	Illustrates analog-to-digital conversion using a sigma-delta algorithm implementation	dspsdadc
Wavelet Reconstruction and Noise Reduction	Uses the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks to show both the perfect reconstruction property of wavelets and an application for noise reduction	dspwavelet

7

Transforms, Estimation, and Spectral Analysis

Learn about transforms, estimation and spectral analysis.

Transform Time-Domain Data into Frequency Domain

When you want to transform time-domain data into the frequency domain, use the FFT block.

In this example, you use the Sine Wave block to generate two sinusoids, one at 15 Hz and the other at 40 Hz. You sum the sinusoids point-by-point to generate the compound sinusoid

$$u = \sin(30\pi t) + \sin(80\pi t)$$

Then, you transform this sinusoid into the frequency domain using an FFT block:

1 At the MATLAB command prompt, type ex_fft_tut.

The FFT Example opens.

2. Double-click the Sine Wave block. The **Block Parameters: Sine Wave** dialog box opens.

3. Set the block parameters as follows:

   - **Amplitude** = 1
   - **Frequency** = [15 40]
   - **Phase offset** = 0
   - **Sample time** = 0.001

- **Samples per frame** = 128

Based on these parameters, the Sine Wave block outputs two sinusoidal signals with identical amplitudes, phases, and sample times. One sinusoid oscillates at 15 Hz and the other at 40 Hz.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the `Matrix Sum` block. The **Block Parameters: Matrix Sum** dialog box opens.

6 Set the **Sum over** parameter to `Specified dimension` and the **Dimension** parameter to `2`. Click **OK** to save your changes.

Because each column represents a different signal, you need to sum along the individual rows in order to add the values of the sinusoids at each time step.

7 Double-click the `Complex to Magnitude-Angle` block. The **Block Parameters: Complex to Magnitude-Angle** dialog box opens.

8 Set the **Output** parameter to `Magnitude`, and then click **OK**.

This block takes the complex output of the FFT block and converts this output to magnitude.

9 Double-click the Vector Scope block.

10 Set the block parameters as follows, and then click **OK**:

- Click the **Scope Properties** tab.
- **Input domain** = `Frequency`
- Click the **Axis Properties** tab.
- **Frequency units** = `Hertz` (This corresponds to the units of the input signals.)
- **Frequency range** = `[0...Fs/2]`
- Select the **Inherit sample time from input** check box.
- **Amplitude scaling** = `Magnitude`

11 Run the model.

The scope shows the two peaks at 15 and 40 Hz, as expected.
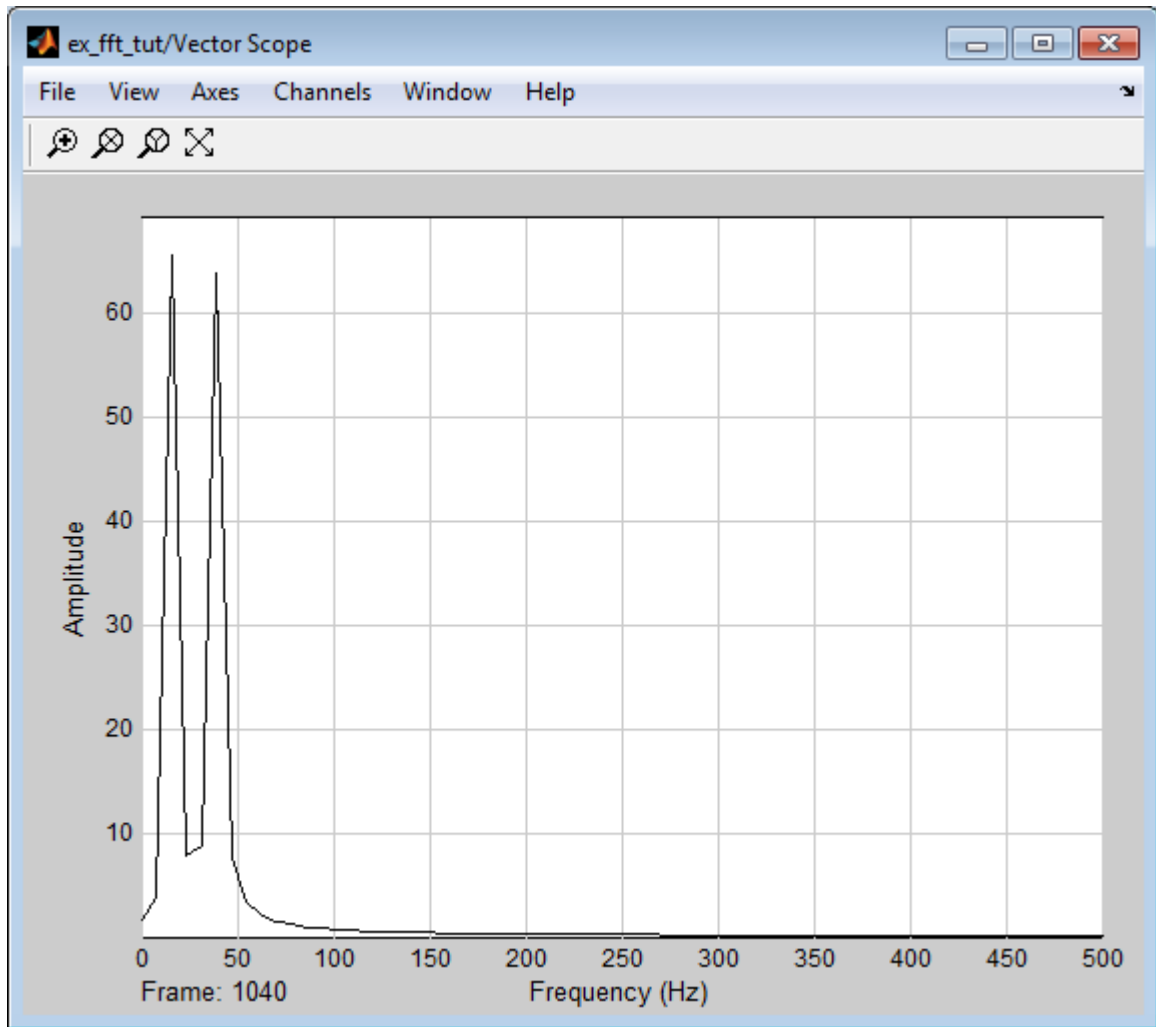
You have now transformed two sinusoidal signals from the time domain to the frequency domain.

Note that the sequence of FFT, Complex to Magnitude-Angle, and Vector Scope blocks could be replaced by a single Spectrum Analyzer block, which computes the magnitude FFT internally. Other blocks that compute the FFT internally are the blocks in the

Power Spectrum Estimation library. See "Estimate the Power Spectral Density in Simulink" on page 7-37for more information about these blocks.

Transform Frequency-Domain Data into Time Domain

When you want to transform frequency-domain data into the time domain, use the IFFT block.

In this example, you use the Sine Wave block to generate two sinusoids, one at 15 Hz and the other at 40 Hz. You sum the sinusoids point-by-point to generate the compound sinusoid, $u = \sin(30\pi t) + \sin(80\pi t)$. You transform this sinusoid into the frequency domain using an FFT block, and then immediately transform the frequency-domain signal back to the time domain using the IFFT block. Lastly, you plot the difference between the original time-domain signal and transformed time-domain signal using a scope:

1   At the MATLAB command prompt, type `ex_ifft_tut`.

    The IFFT Example opens.

2 Double-click the Sine Wave block. The **Block Parameters: Sine Wave** dialog box opens.

3 Set the block parameters as follows:

- **Amplitude** = 1
- **Frequency** = [15 40]
- **Phase offset** = 0
- **Sample time** = 0.001
- **Samples per frame** = 128

Based on these parameters, the Sine Wave block outputs two sinusoidal signals with identical amplitudes, phases, and sample times. One sinusoid oscillates at 15 Hz and the other at 40 Hz.

4   Save these parameters and close the dialog box by clicking **OK**.

5   Double-click the Matrix Sum block. The **Block Parameters: Matrix Sum** dialog box opens.

6   Set the **Sum over** parameter to `Specified dimension` and the **Dimension** parameter to `2`. Click **OK** to save your changes.

Because each column represents a different signal, you need to sum along the individual rows in order to add the values of the sinusoids at each time step.

7   Double-click the FFT block. The **Block Parameters: FFT** dialog box opens.

8   Select the **Output in bit-reversed order** check box., and then click **OK**.

9   Double-click the IFFT block. The **Block Parameters: IFFT** dialog box opens.

10 Set the block parameters as follows, and then click **OK**:

   ·   Select the **Input is in bit-reversed order** check box.
   ·   Select the **Input is conjugate symmetric** check box.

Because the original sinusoidal signal is real valued, the output of the FFT block is conjugate symmetric. By conveying this information to the IFFT block, you optimize its operation.

Note that the Sum block subtracts the original signal from the output of the IFFT block, which is the estimation of the original signal.

11 Double-click the Vector Scope block.

12 Set the block parameters as follows, and then click **OK**:

   ·   Click the **Scope Properties** tab.
   ·   **Input domain** = `Time`

13 Run the model.
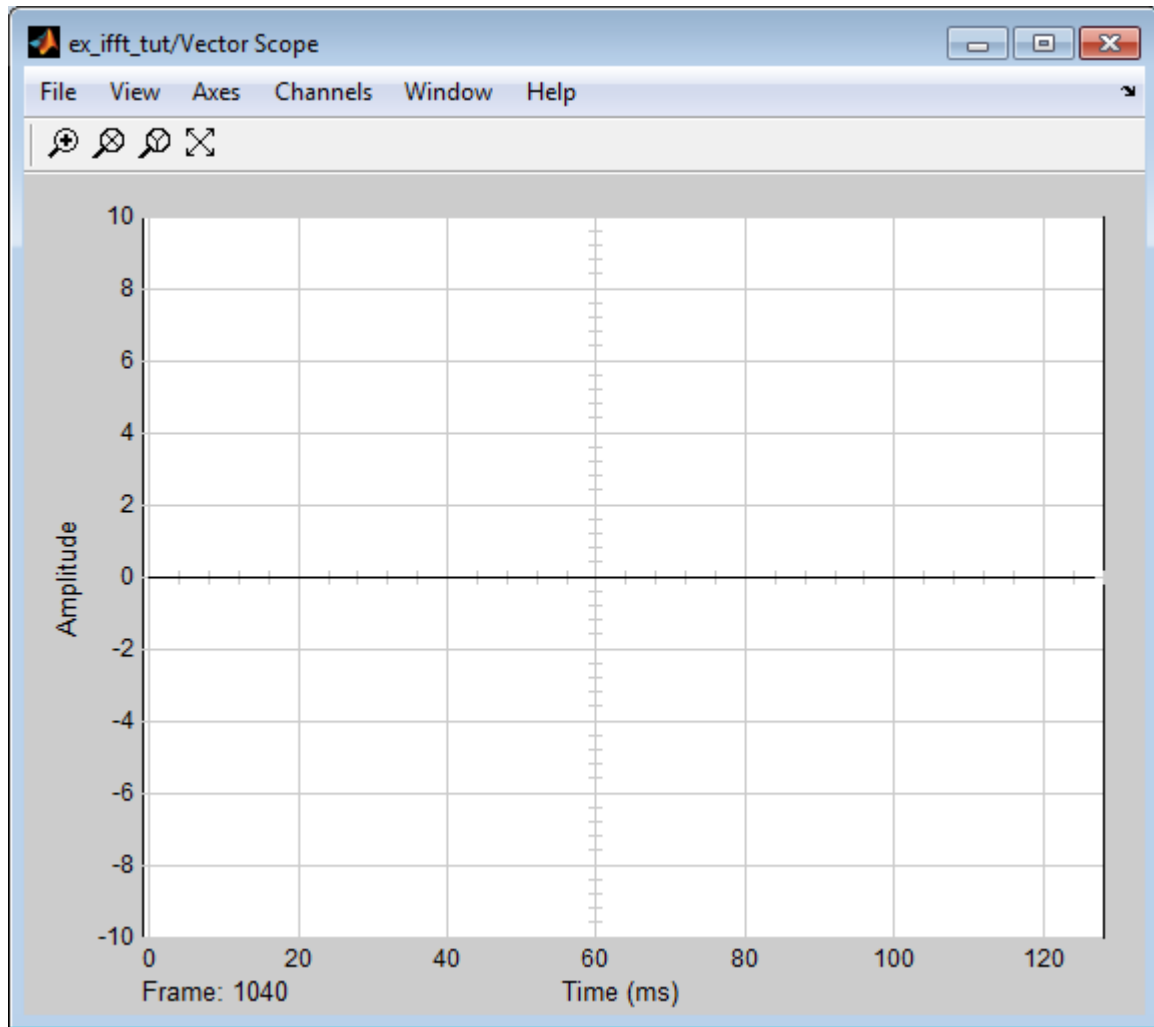
The flat line on the scope suggests that there is no difference between the original signal and the estimate of the original signal. Therefore, the IFFT block has accurately reconstructed the original time-domain signal from the frequency-domain input.

14 Right-click in the Vector Scope window, and select **Autoscale**.
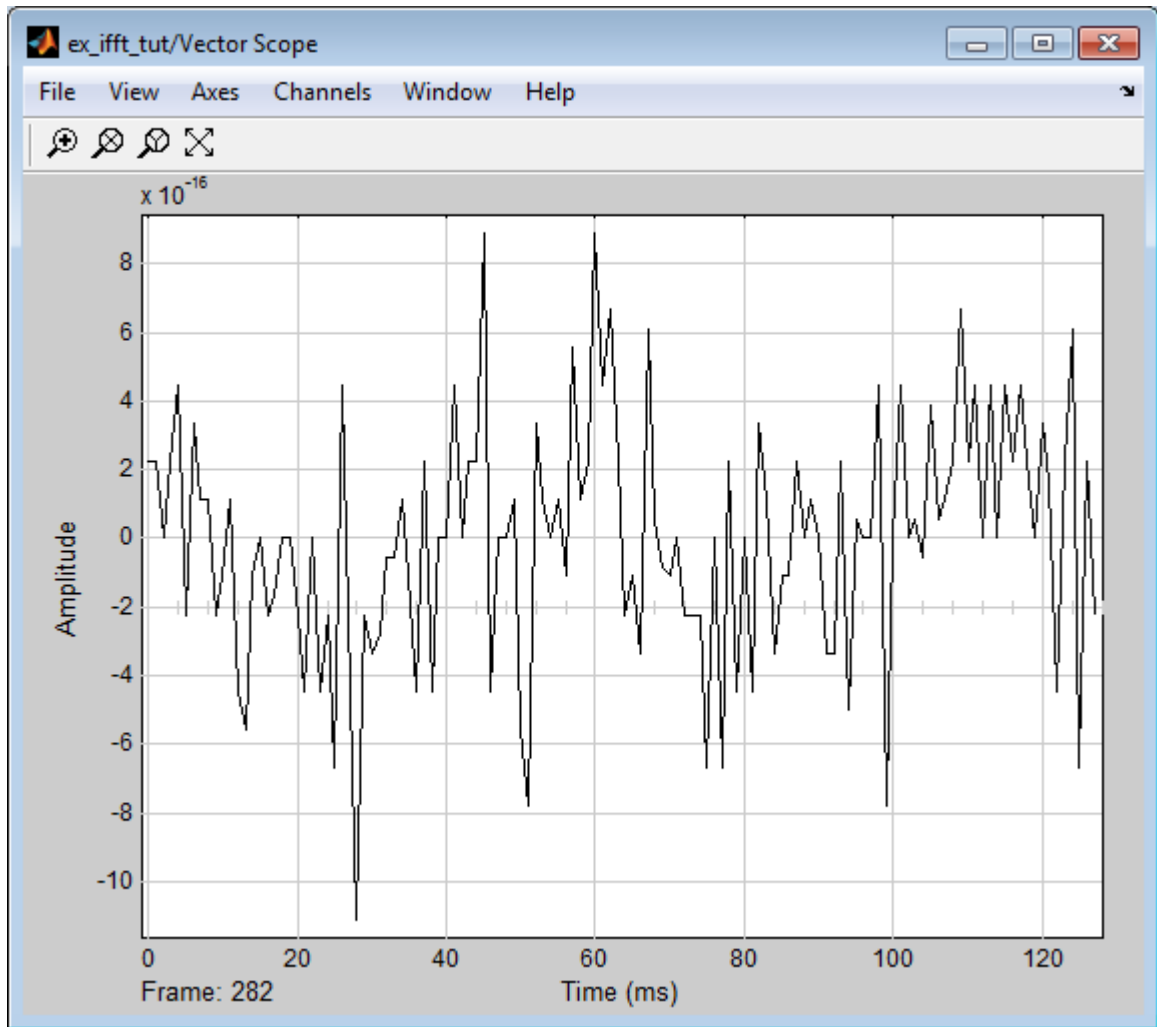
In actuality, the two signals are identical to within round-off error. The previous figure shows the enlarged trace. The differences between the two signals is on the order of 10^{-15}.

Linear and Bit-Reversed Output Order

In this section...
"FFT and IFFT Blocks Data Order" on page 7-12
"Find the Bit-Reversed Order of Your Frequency Indices" on page 7-12

FFT and IFFT Blocks Data Order

The FFT block enables you to output the frequency indices in linear or bit-reversed order. Because linear ordering of the frequency indices requires a bit-reversal operation, the FFT block may run more quickly when the output frequencies are in bit-reversed order.

The input to the IFFT block can be in linear or bit-reversed order. Therefore, you do not have to alter the ordering of your data before transforming it back into the time domain. However, the IFFT block may run more quickly when the input is provided in bit-reversed order.

Find the Bit-Reversed Order of Your Frequency Indices

Two numbers are bit-reversed values of each other when the binary representation of one is the mirror image of the binary representation of the other. For example, in a three-bit system, one and four are bit-reversed values of each other, since the three-bit binary representation of one, 001, is the mirror image of the three-bit binary representation of four, 100. In the diagram below, the frequency indices are in linear order. To put them in bit-reversed order

1   Translate the indices into their binary representation with the minimum number of bits. In this example, the minimum number of bits is three because the binary representation of 7 is 111.

2   Find the mirror image of each binary entry, and write it beside the original binary representation.

3   Translate the indices back to their decimal representation.

The frequency indices are now in bit-reversed order.
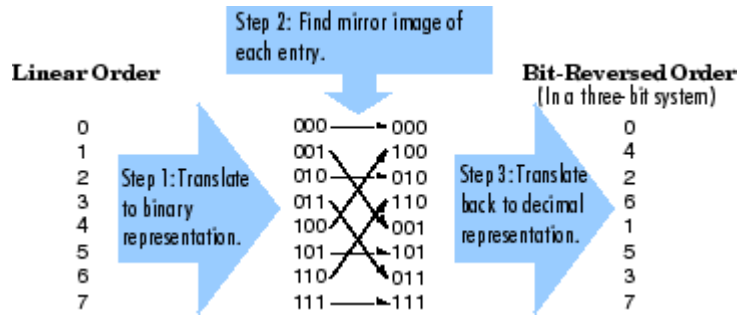
The next diagram illustrates the linear and bit-reversed outputs of the FFT block. The output values are the same, but they appear in different order.
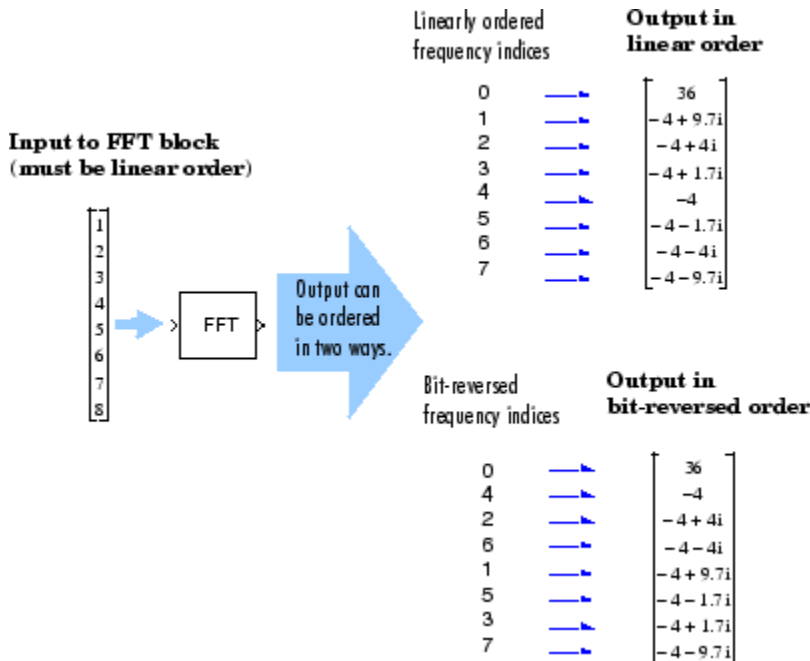
Calculate Channel Latencies Required for Wavelet Reconstruction

In this section...

Analyze Your Model

The following sections guide you through the process of calculating the channel latencies required for perfect wavelet reconstruction. This example uses the `ex_wavelets` model, but you can apply the process to perform perfect wavelet reconstruction in any model. To open the example model, type ex_wavelets at the MATLAB command line.

Note: You must have a Wavelet Toolbox™ product license to run the `ex_wavelets` model.
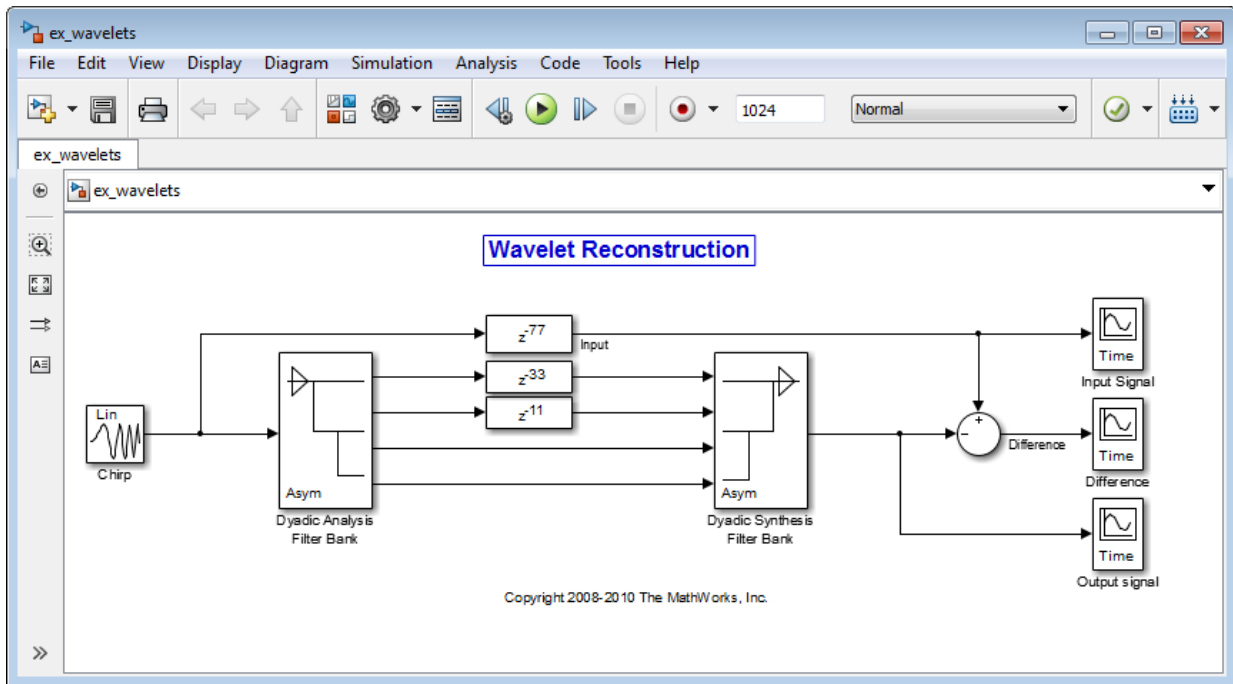
Before you can begin calculating the latencies required for perfect wavelet reconstruction, you must know the types of filters being used in your model.

The Dyadic Analysis Filter Bank and the Dyadic Synthesis Filter Bank blocks in the `ex_wavelets` model have the following settings:

- **Filter** = `Biorthogonal`
- **Filter order [synthesis/analysis]** = `[3/5]`
- **Number of levels** = `3`
- **Tree structure** = `Asymmetric`
- **Input** = `Multiple ports`

Based on these settings, the Dyadic Analysis Filter Bank and the Dyadic Synthesis Filter Bank blocks construct biorthogonal filters using the Wavelet Toolbox `wfilters` function.

Calculate the Group Delay of Your Filters

Once you know the types of filters being used by the Dyadic Analysis and Dyadic Synthesis Filter Bank blocks, you need to calculate the group delay of those filters. To do so, you can use the Signal Processing Toolbox `fvtool`.

Before you can use `fvtool`, you must first reconstruct the filters in the MATLAB workspace. To do so, type the following code at the MATLAB command line:

```
[Lo_D, Hi_D, Lo_R, Hi_R] = wfilters('bior3.5')
```

Where `Lo_D` and `Hi_D` represent the low- and high-pass filters used by the Dyadic Analysis Filter Bank block, and `Lo_R` and `Hi_R` represent the low- and high-pass filters used by the Dyadic Synthesis Filter Bank block.

After you construct the filters in the MATLAB workspace, you can use `fvtool` to determine the group delay of the filters. To analyze the low-pass biorthogonal filter used by the Dyadic Analysis Filter Bank block, you must do the following:

- Type `fvtool(Lo_D)` at the MATLAB command line to launch the Filter Visualization Tool.

- When the Filter Visualization Tool opens, click the Group delay response button ( ) on the toolbar, or select **Group Delay Response** from the **Analysis** menu.

Based on the Filter Visualization Tool's analysis, you can see that the group delay of the Dyadic Analysis Filter Bank block's low-pass biorthogonal filter is 5.5.
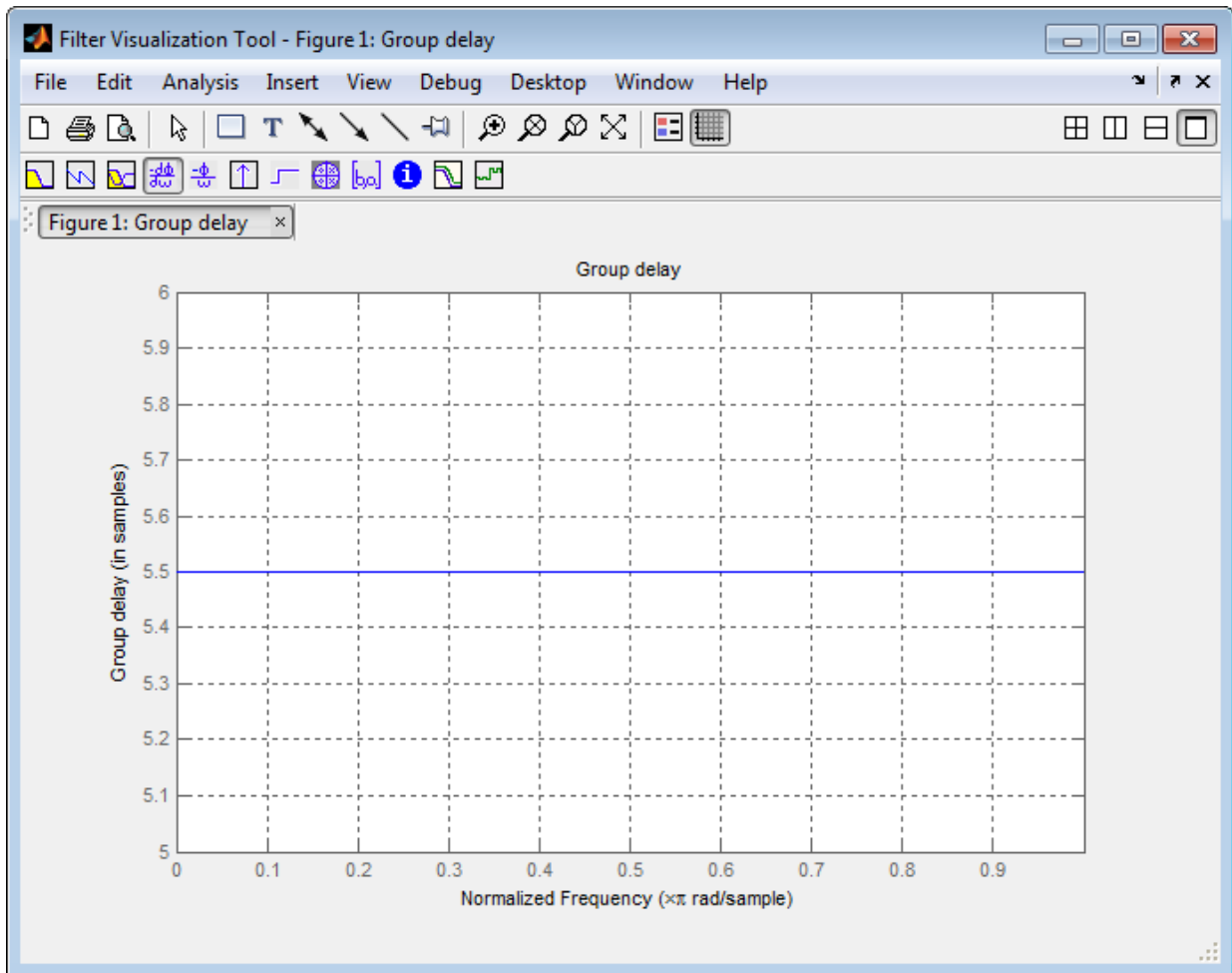
Note: Repeat this procedure to analyze the group delay of each of the filters in your model. This section does not show the results for each filter in the `ex_wavelets` model because all wavelet filters in this particular example have the same group delay.

Reconstruct the Filter Bank System

To determine the delay introduced by the analysis and synthesis filter bank system, you must reconstruct the tree structures of the Dyadic Analysis Filter Bank and the Dyadic Synthesis Filter Bank blocks. To learn more about constructing tree structures for the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks, see the following sections of the DSP System Toolbox User's Guide:

- "Dyadic Analysis Filter Banks" on page 6-25
- "Dyadic Synthesis Filter Banks" on page 6-29

Because the filter blocks in the `ex_wavelets` model use biorthogonal filters with three levels and an asymmetric tree structure, the filter bank system appears as shown in the following figure.
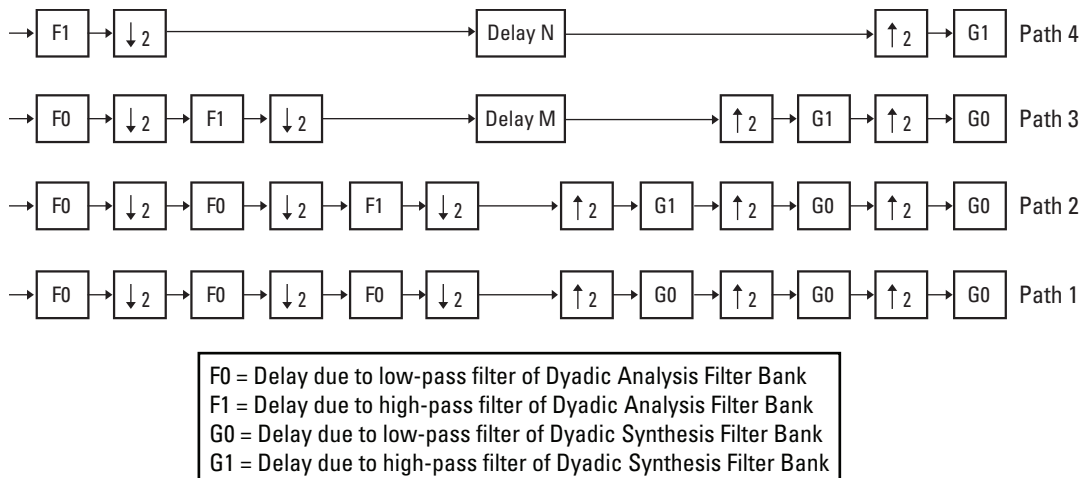
The extra delay values of M and N on paths 3 and 4 in the previous figure ensure that the total delay on each of the four filter paths is identical.

Equalize the Delay on Each Filter Path

Now that you have reconstructed the filter bank system, you can calculate the delay on each filter path. To do so, use the following Noble identities:

First Noble Identity
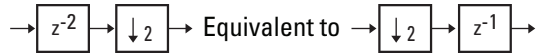
Second Noble Identity
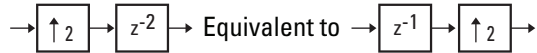
You can apply the Noble identities by summing the delay on each signal path from right to left. The first Noble identity indicates that moving a delay of 1 before a downsample of 2 is equivalent to multiplying that delay value by 2. Similarly, the second Noble identity indicates that moving a delay of 2 before an upsample of 2 is equivalent to dividing that delay value by 2.

The `fvtool` analysis in step 1 found that both the low- and high-pass filters of the analysis filter bank have the same group delay ($F_0 = F_1 = 5.5$). Thus, you can use F to represent the group delay of the analysis filter bank. Similarly, the group delay of the low- and high-pass filters of the synthesis filter bank is the same ($G_0 = G_1 = 5.5$), so you can use G to represent the group delay of the synthesis filter bank.

The following figure shows the filter bank system with the intermediate delay sums displayed below each path.

F	↓2				Delay N				↑2	G	Path 4
2N+(F+G)	2N+G			N+0.5G			0.5G			G	

F = Delay due to Dyadic Analysis Filter Bank
G = Delay due to Dyadic Synthesis Filter Bank
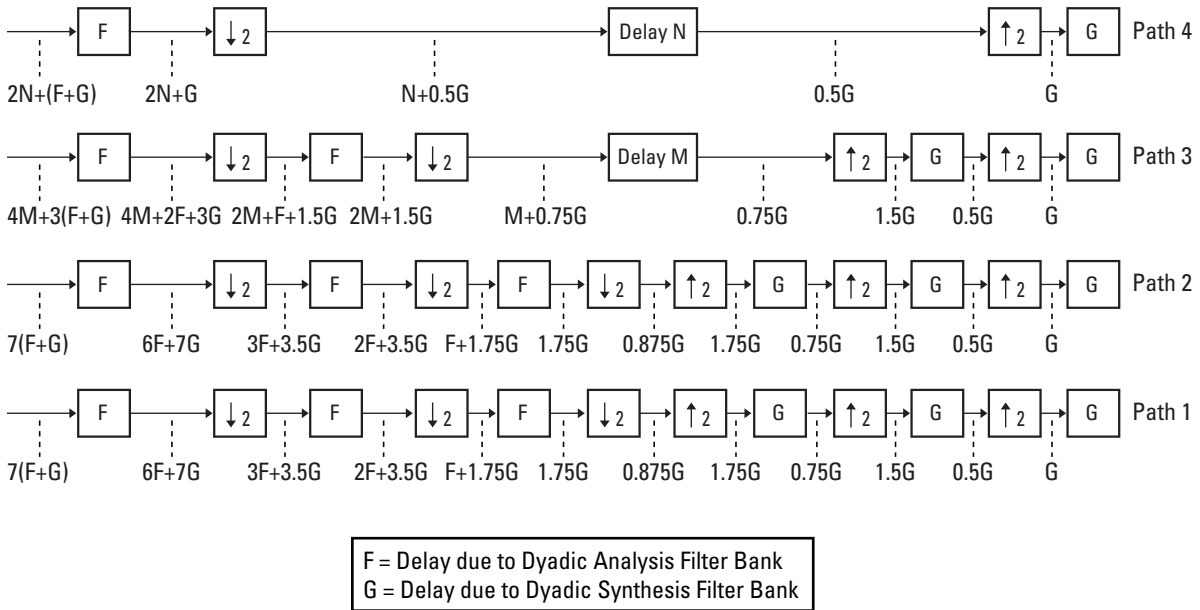
You can see from the previous figure that the signal delays on paths 1 and 2 are identical: $7(F+G)$. Because each path of the filter bank system has identical delay, you can equate the delay equations for paths 3 and 4 with the delay equation for paths 1 and 2. After constructing these equations, you can solve for M and N, respectively:

$$\text{Path 3} = \text{Path 1} \Rightarrow 4M + 3(F + G) = 7(F + G)$$
$$\Rightarrow M = F + G$$

$$\text{Path 4} = \text{Path 1} \Rightarrow 2N + (F + G) = 7(F + G)$$
$$\Rightarrow N = 3(F + G)$$

The `fvtool` analysis in step 1 found the group delay of each biorthogonal wavelet filter in this model to be 5.5 samples. Therefore, $F = 5.5$ and $G = 5.5$. By inserting these values into the two previous equations, you get $M = 11$ and $N = 33$. Because the total delay on each filter path must be the same, you can find the overall delay of the filter bank system by inserting $F = 5.5$ and $G = 5.5$ into the delay equation for any of the four filter paths. Inserting the values of F and G into $7(F+G)$ yields an overall delay of 77 samples for the filter bank system of the `ex_wavelets` model.

Update and Run the Model

Now that you know the latencies required for perfect wavelet reconstruction, you can incorporate those delay values into the model. The `ex_wavelets` model has already been updated with the correct delay values ($M = 11$, $N = 33$, *Overall* = 77), so it is ready to run.
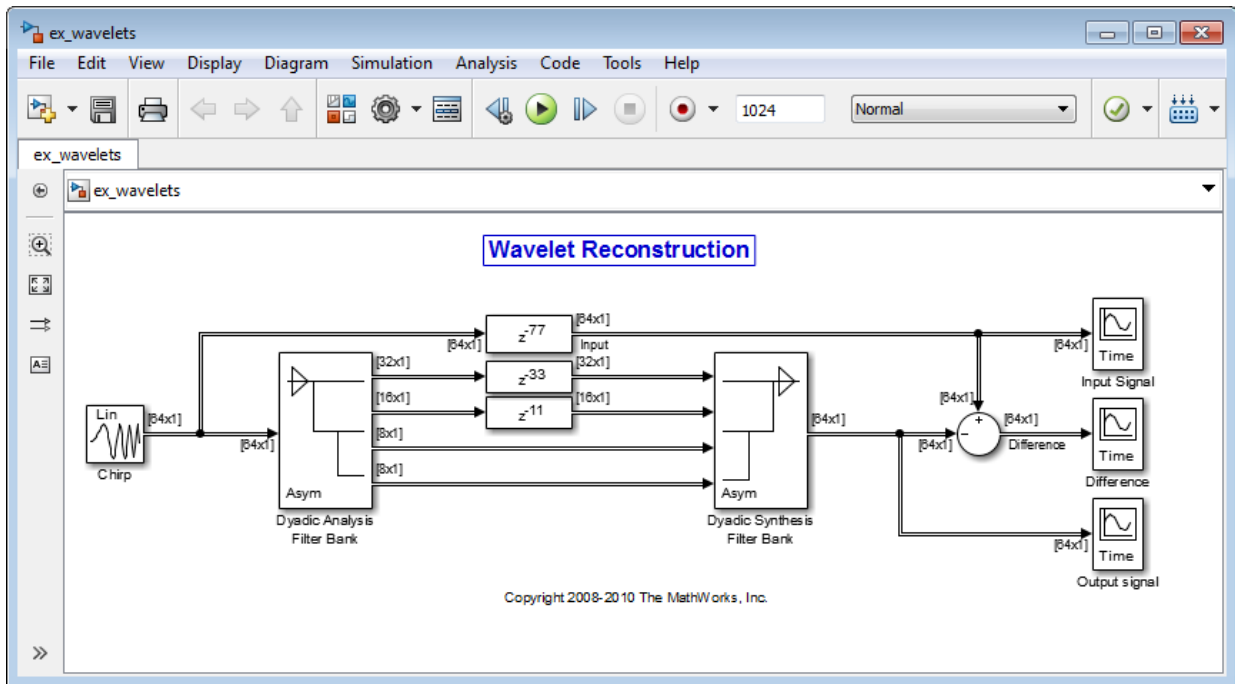
After you run the model, examine the reconstruction error in the Difference scope. To further examine any particular areas of interest, use the zoom tools available on the toolbar of the scope window or from the **View** menu.
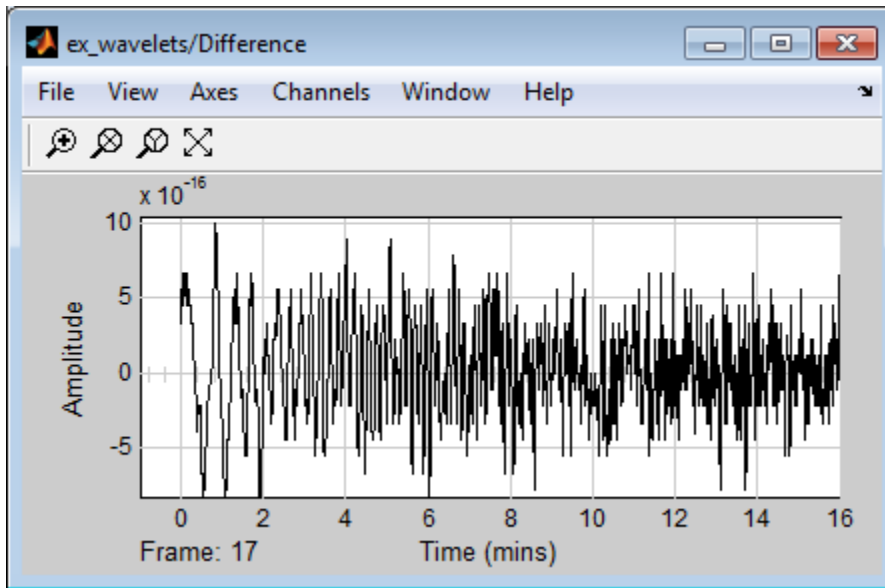
References

[1] Strang, G. and Nguyen, T. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Estimate the Power Spectral Density in MATLAB

In this section...
"Estimate the PSD Using dsp.SpectrumAnalyzer" on page 7-23
"Convert the Power in Watts to dBW and dBm" on page 7-32
"Estimate the PSD Using dsp.SpectrumEstimator" on page 7-33

The power spectral density (PSD) of a time-domain signal is the distribution of power contained within the signal over frequency, based on a finite set of data. The frequency-domain representation of the signal is often easier to analyze than the time-domain representation. Many signal processing applications, such as noise cancellation and system identification, are based on the frequency-specific modifications of signals. The goal of the spectral density estimation is to estimate the spectral density of a signal from a sequence of time samples. Depending on what is known about the signal, estimation techniques can involve parametric or nonparametric approaches, and can be based on time-domain or frequency-domain analysis. For example, a common parametric technique involves fitting the observations to an autoregressive model. A common nonparametric technique is the periodogram. The spectral density is estimated using Fourier transform methods such as the Welch Method. You can also use other techniques such as the maximum entropy method.

In MATLAB, you can perform real-time spectral analysis of a dynamic signal using the dsp.SpectrumAnalyzer System object. Alternately, you can use the dsp.SpectrumEstimator System object to acquire and process the data. In this approach, the power spectrum data is available for further processing.

Estimate the PSD Using dsp.SpectrumAnalyzer

To view the PSD of a signal, you can use the `dsp.SpectrumAnalyzer` System object™. You can change the dynamics of the input signal and see the effect those changes have on the spectral density of the signal in real time. The PSD data can only be viewed and is not available for processing. To acquire and process the data, use the `dsp.SpectrumEstimator` System object. For more information, see the section 'Estimate PSD Using dsp.SpectrumEstimator'.

Initialization

Initialize the sine wave source to generate the sine wave and the spectrum analyer to show the PSD of the signal. The input sine wave has two frequencies: one at 1000 Hz and

the other at 5000 Hz. Create two `dsp.SineWave` objects, one to generate the 1000 Hz sine wave and the other to generate the 5000 Hz sine wave.

```
Fs = 44100;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,'PhaseOffset',10,...
    'SampleRate',Fs,'Frequency',1000);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',5000);
SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'SpectrumType','Power density',...
    'PlotAsTwoSidedSpectrum',false,'Window','Hann',...
    'ChannelNames',{'PSD of the input'},'ShowLegend',true);
```

The spectrum analyzer uses the `Hann` window to compute the power spectral density of the signal.

Estimation

Stream in and estimate the PSD of the signal. Construct a `for`-loop to run for 5000 iterations. In each iteration, stream in 1024 samples (one frame) of each sine wave and compute the PSD of each frame. To generate the input signal, add the two sine waves. The resultant signal is a sine wave with two frequencies: one at 1000 Hz and the other at 5000 Hz. Add Gaussian noise with mean at 0 and a standard deviation of 0.001.

```
for Iter = 1:7000
    Sinewave1 = step(Sineobject1);
    Sinewave2 = step(Sineobject2);
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    step(SA,NoisyInput);
end
```
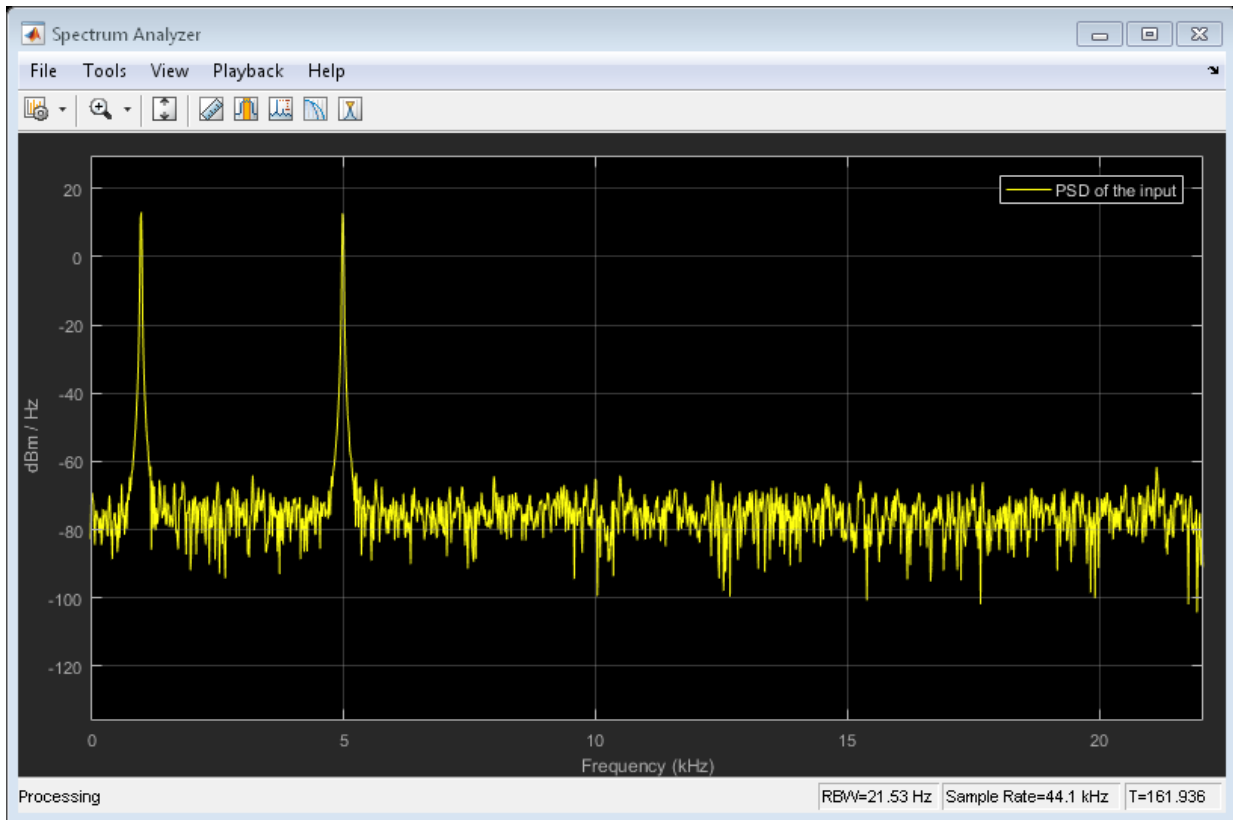
In the spectrum analyzer output, you can see two distinct peaks: one at 1000 Hz and the other at 5000 Hz.

Resoultion Bandwidth (RBW) is the minimum frequency bandwidth that can be resolved by the spectrum analyzer. By default, the RBWSource property of the dsp.SpectrumAnalyzer object is set to Auto. In this mode, RBW is the ratio of the frequency span to 1024. In a two-sided spectrum, this value is $\frac{F_s}{1024}$, while in a one-sided spectrum, it is $\frac{\frac{F_s}{2}}{1024}$.

Using this value of RBW, the window length($N_{samples}$) is computed iteratively using this relationship: $N_{samples} = \frac{(1-\frac{Overlap}{100})*(NENBW)*F_s}{RBW}$

$Overlap$ is the amount of overlap between the previous and current buffered data segments. $NENBW$ is the equivalent noise bandwidth of the window. For more information on the details of the spectral estimation algorithm, see Spectral Analysis.

RBW calculated in this mode gives a good frequency resolution.

To distinguish between two frequencies in the display, the distance between the two frequencies must be at least RBW. In this example, the distance between the two peaks is 4000 Hz, which is greater than RBW. Hence, you can see the peaks distinctly. Change the frequency of the second sine wave to 1015 Hz. The difference between the two frequencies is less than RBW.

```
release(Sineobject2);
Sineobject2.Frequency = 1015;
for Iter = 1:5000
    Sinewave1 = step(Sineobject1);
    Sinewave2 = step(Sineobject2);
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    step(SA,NoisyInput);
end
```
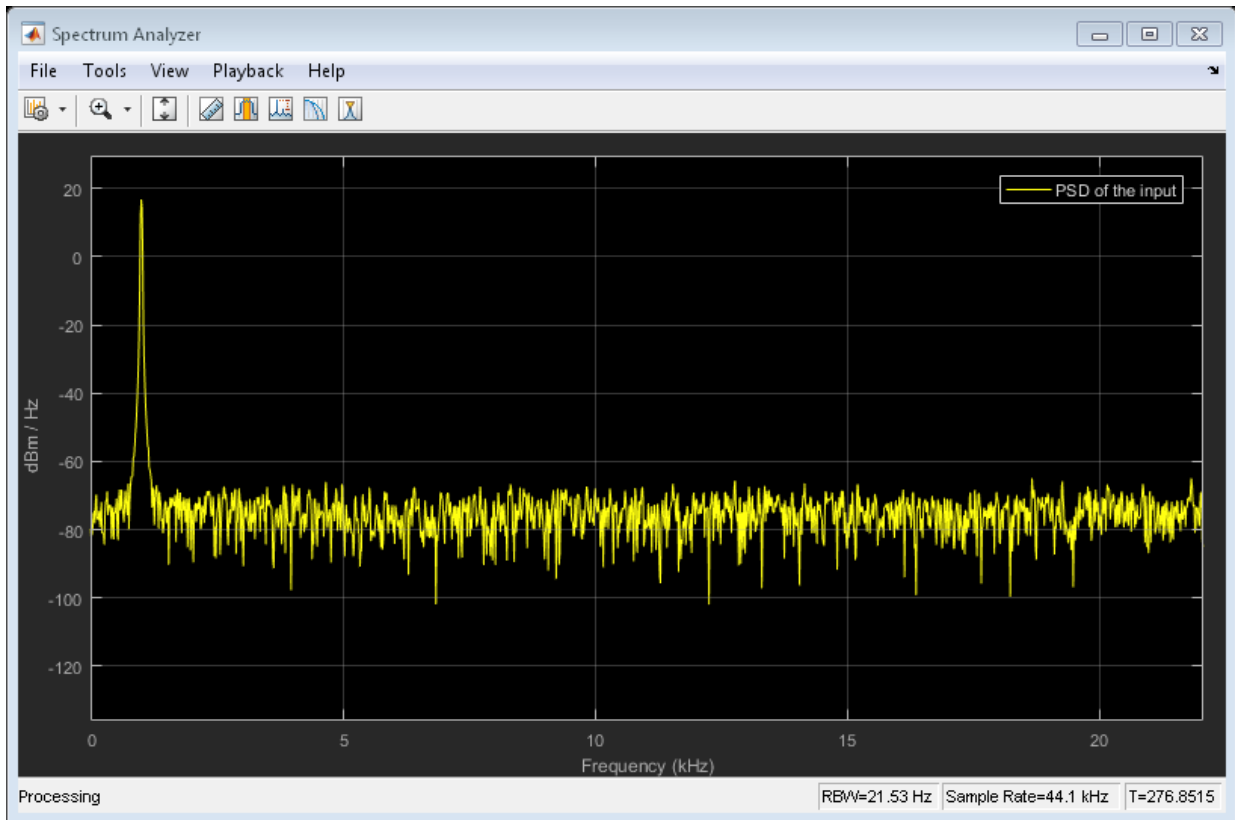
The peaks are not distinguishable.
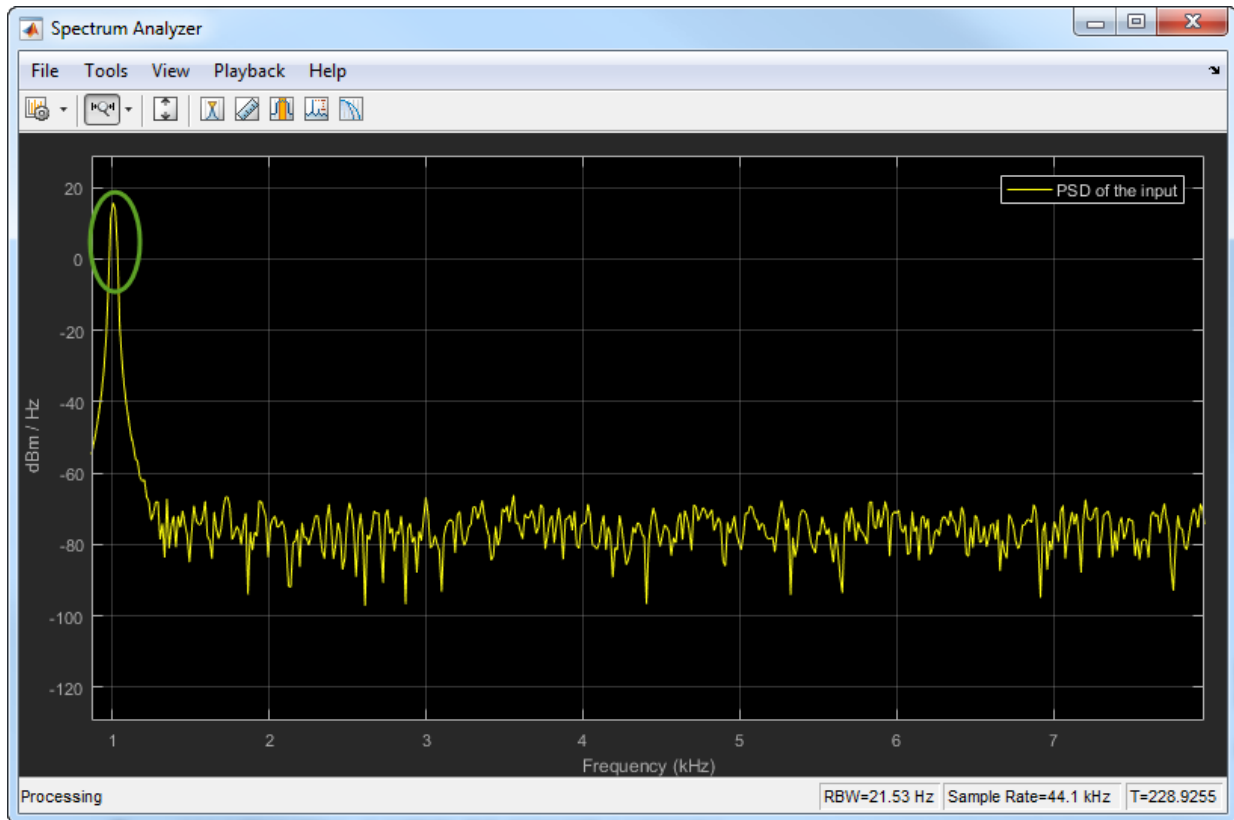
To increase the frequency resolution, decrease RBW to 1 Hz.

```
SA.RBWSource = 'property';
SA.RBW = 1;
for Iter = 1:5000
    Sinewave1 = step(Sineobject1);
    Sinewave2 = step(Sineobject2);
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    step(SA,NoisyInput);
end
```
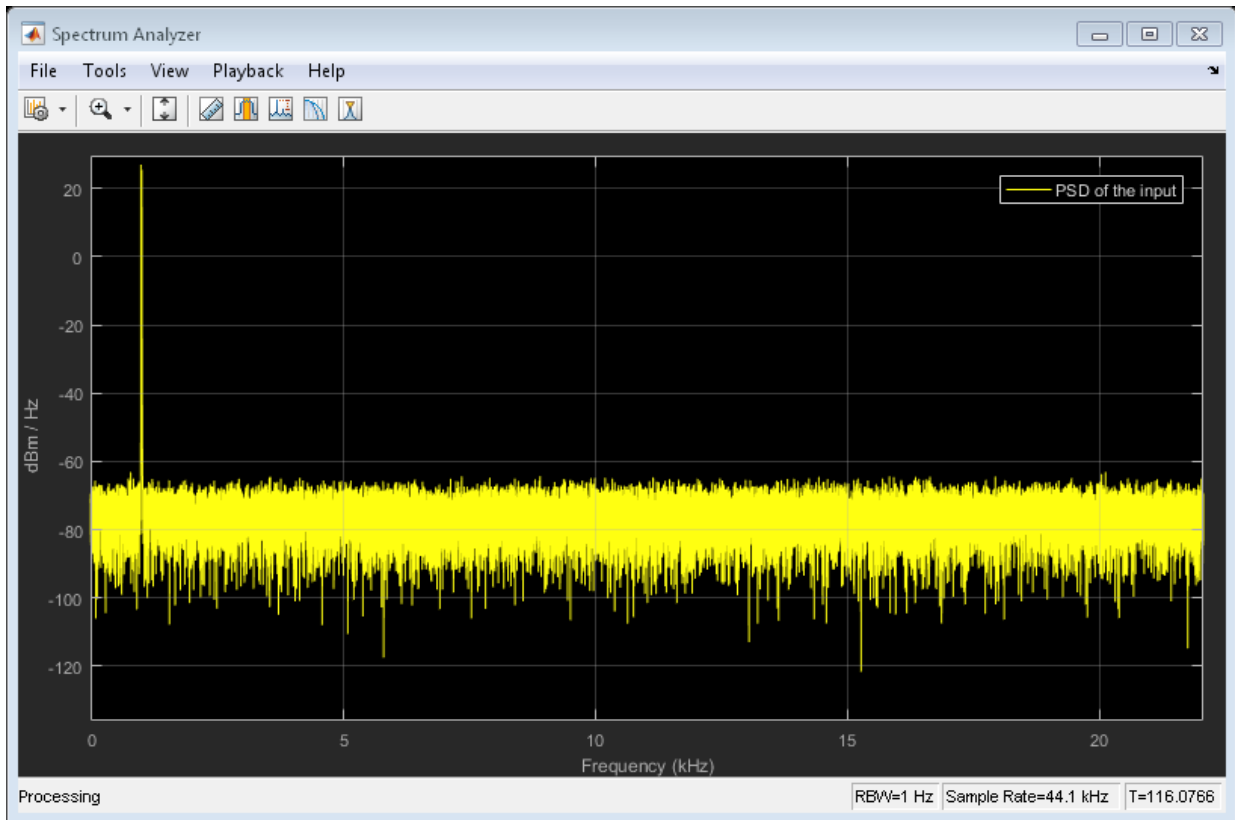
On zooming, the two peaks, which are 15 Hz apart, are now distinguishable.
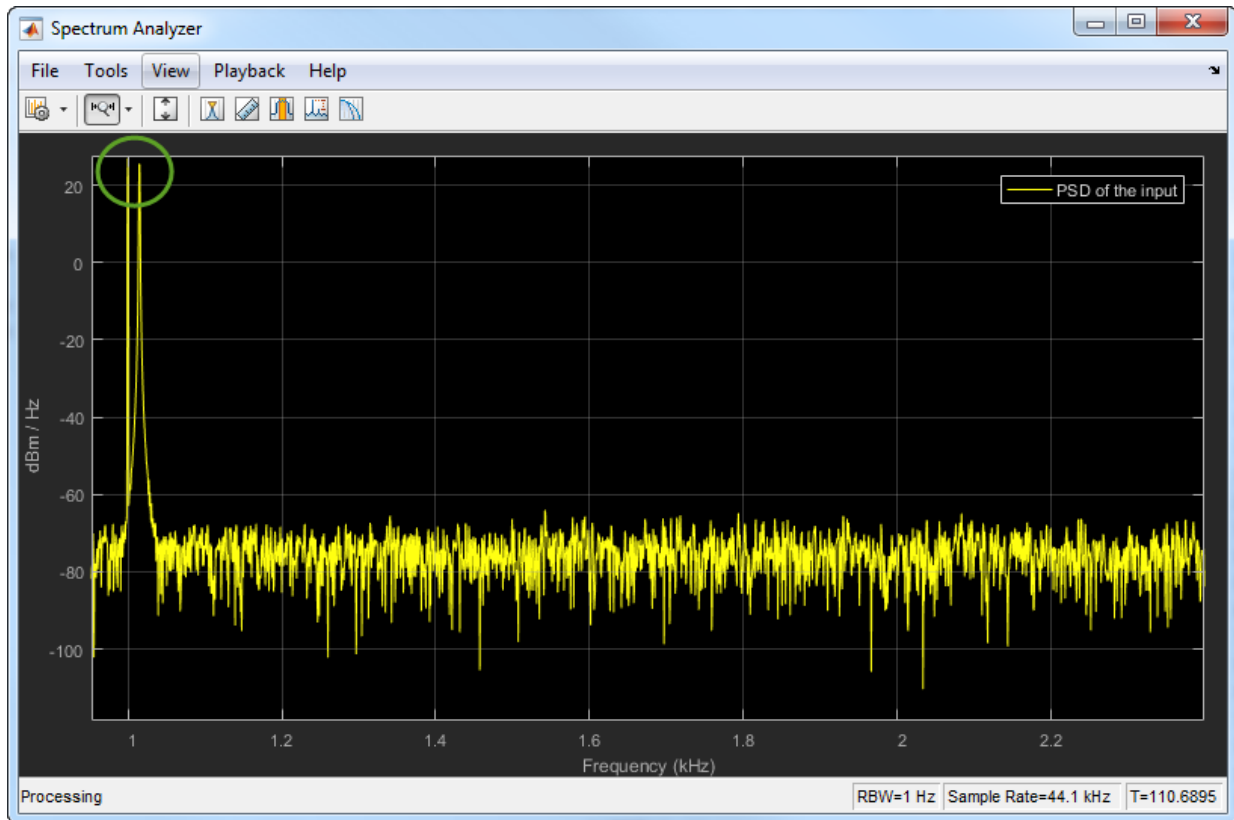
When you increase the frequency resolution, the window length increases, but the tradeoff is that the time resolution decreases.

During streaming, you can change the input properties or the spectrum analyzer properties and see the effect on the spectrum analyzer output immediately. For example, change the frequency of the second sine wave when the index of the loop is a multiple of 1000.

To maintain a good balance between the frequency resolution and time resolution, change the RBWSource property to Auto.

```
release(Sineobject2);
SA.RBWSource = 'Auto';
for Iter = 1:5000
```

```
Sinewave1 = step(Sineobject1);
if (mod(Iter,1000) == 0)
    release(Sineobject2);
    Sineobject2.Frequency = Iter;
    Sinewave2 = step(Sineobject2);
else
    Sinewave2 = step(Sineobject2);
end
Input = Sinewave1 + Sinewave2;
NoisyInput = Input + 0.001*randn(1024,1);
step(SA,NoisyInput);
end
```
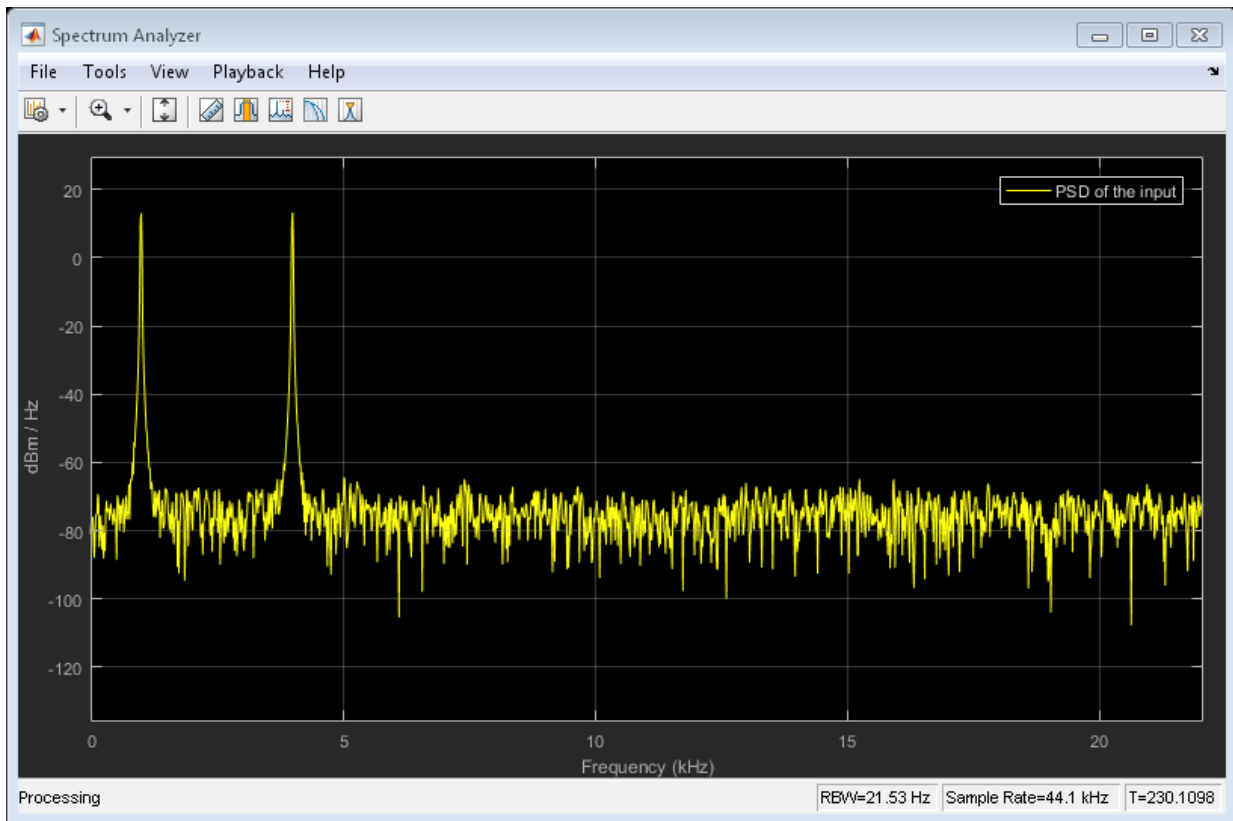

While running the streaming loop, you can see that the peak of the second sine wave changes according to the iteration value. Similarly, you can change any of the spectrum

analyzer properties while the simulation is running and see a corresponding change in the output.

Convert the Power in Watts to dBW and dBm

The spectrum analyzer provides three units to specify the power spectral density: `Watts/Hz`, `dBm/Hz`, and `dBW/Hz`. Corresponding units of power are `Watts`, `dBm`, and `dBW`. The default unit of **Power** is `dBm`.

Power in `dBW` is given by:

$$P_{dBW} = 10\log 10(power\ in\ watt\ /\ 1\ watt)$$

Power in `dBm` is given by:

$$P_{dBm} = 10\log 10(power\ in\ watt\ /\ 1\ milliwatt)$$

For a sine wave signal with an amplitude (A) of 1 V, the power of a one-sided spectrum in `Watts` is given by:

$$A^2\ /\ 2$$

In this example, this power equals 0.5 W. Corresponding power in dBm is given by:

$$P_{dBm} = 10\log 10(power\ in\ watt\ /\ 1\ milliwatt)$$
$$P_{dBm} = 10\log 10(0.5\ /\ 10\ \wedge -3)$$

Here, the power equals 26.9897 dBm. To confirm this value with a peak finder, click **Tools** > **Measurements** > **Peak Finder**.

For a white noise signal, the spectrum is flat for all frequencies. The spectrum analyzer in this example shows a one-sided spectrum in the range [0 Fs/2]. For a white noise signal with a variance of 1e-4, the power per unit bandwidth ($P_{unitbandwidth}$) is 1e-4. The total power in watts over the entire frequency range is:

$$P_{whitenoise} = P_{unitbandwidth} * number\ of\ frequency\ bins,$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{Fs\ /\ 2}{RBW} \right),$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{22050}{21.53} \right)$$

The number of frequency bins is the ratio of total bandwidth to RBW. For a one-sided spectrum, the total bandwidth is half the sampling rate. RBW in this example is 21.53 Hz. With all these values, the total power of the white noise is -39.87 dBm.

Estimate the PSD Using dsp.SpectrumEstimator

Using the dsp.SpectrumAnalyzer System object™, you can view and analyze, but cannot process the PSD data. To process the data, you must compute the spectral density using one of the Estimator objects in the Estimation library. To view the objects in the Estimation library, type help dsp in the MATLAB® command prompt, and click Estimation.

Initialization

Use the same source as in the previous section on using the dsp.SpectrumAnalyzer to estimate the PSD. The input sine wave has two frequencies: one at 1000 Hz and the other at 5000 Hz. Initialize dsp.SpectrumEstimator to compute the PSD of the signal, and dsp.ArrayPlot object to view the PSD of the signal.

```
Fs = 44100;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,'PhaseOffset',10,...
    'SampleRate',Fs,'Frequency',1000);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',5000);

SpecEst = dsp.SpectrumEstimator('SpectrumType','Power density',...
    'PowerUnits','dBm','SampleRate',Fs,...
    'FrequencyRange','onesided');
ArrPlot = dsp.ArrayPlot('PlotType','Line','ChannelNames',{'PSD of the input'},...
    'YLimits',[-90 20],'XLabel','Number of samples per frame','YLabel',...
    'Power (dBm)','Title','One-sided power spectrum with respect to samples');
```

Estimation

Stream in and estimate the PSD of the signal. Construct a for-loop to run for 5000 iterations. In each iteration, stream in 1024 samples (one frame) of each sine wave and compute the PSD of each frame. Add Gaussian noise with mean at 0 and a standard deviation of 0.001 to the input signal.

```
for Iter = 1:5000
    Sinewave1 = step(Sineobject1);
    Sinewave2 = step(Sineobject2);
```

```
        Input = Sinewave1 + Sinewave2;
        NoisyInput = Input + 0.001*randn(1024,1);
        PSDoutput = step(SpecEst,NoisyInput);
        step(ArrPlot,PSDoutput);
end
```
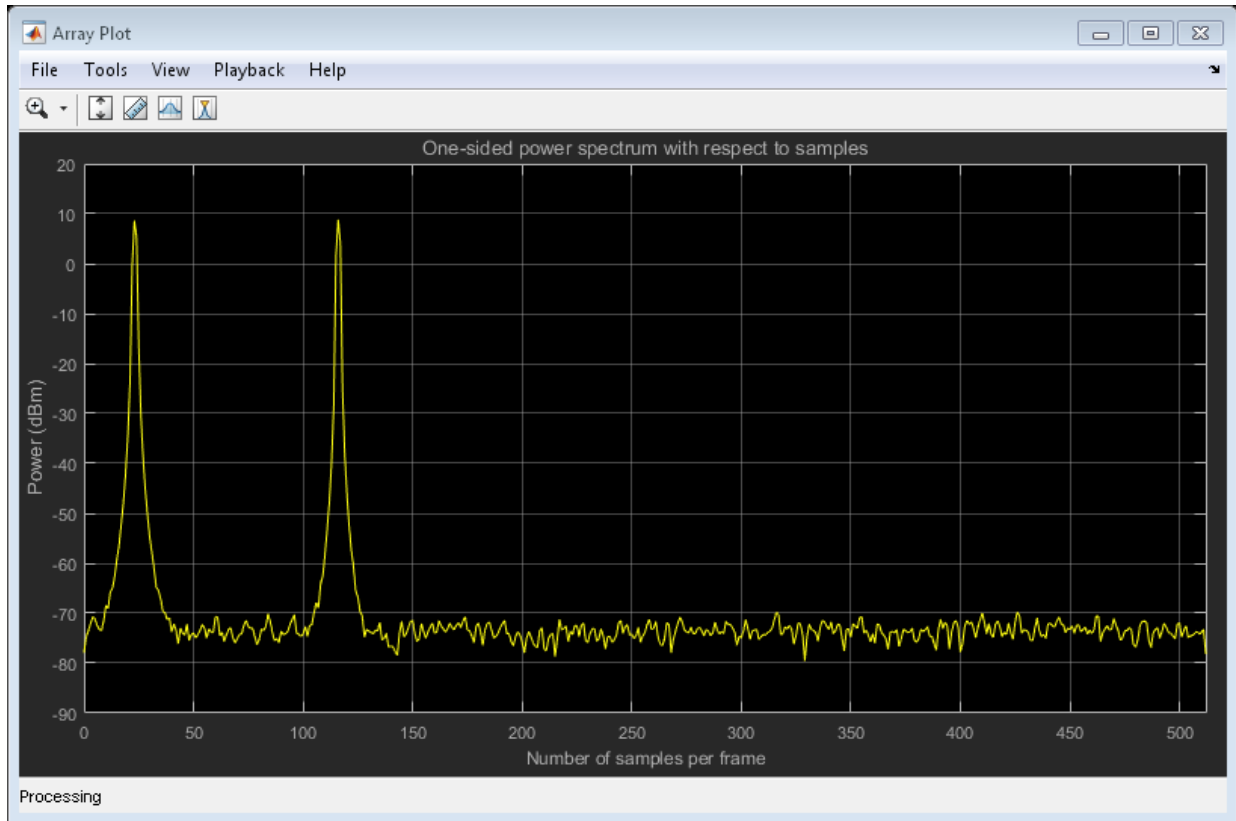

Convert *x*-axis to Represent Frequency

By default, dsp.ArrayPlot plots the PSD data with respect to the number of samples per frame. The number of points on the *x*-axis equals the length of the input frame. For a one-sided spectrum, the span of *x*-axis equals half the input frame. The spectrum analyzer plots the PSD data with respect to frequency. For a one-sided spectrum, the frequency varies in the range [0 Fs/2]. For a two-sided spectrum, frequency varies in the

range [-Fs/2 Fs/2]. To convert the *x*-axis of the array plot from sample based to frequency based, you must set the $SampleIncrement$ property of the `dsp.ArrayPlot` object to $Fs/framelength$. For a one-sided spectrum, $XOffset$ property of `dsp.ArrayPlot` must be 0. For a two-sided spectrum, $XOffset$ must be -Fs/2. In this example, for a one-sided spectrum, $SampleIncrement$ must be 44100/1024.

```
ArrPlot.SampleIncrement = Fs/1024;
ArrPlot.XLabel = 'Frequency (kHz)';
ArrPlot.Title = 'One-sided power spectrum with respect to frequency';

for Iter = 1:5000
    Sinewave1 = step(Sineobject1);
    Sinewave2 = step(Sineobject2);
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    PSDoutput = step(SpecEst,NoisyInput);
    step(ArrPlot,PSDoutput);
end
```

Live Processing

To process the spectral data while streaming, pass the output of the estimator block into a processing logic.

More About

Estimate the Power Spectral Density in Simulink

In this section...
"Estimate PSD Using the Spectrum Analyzer" on page 7-37
"Convert the Power in Watts to dBW and dBm" on page 7-32
"Estimate PSD Using the Spectrum Estimator Block" on page 7-48

The power spectral density (PSD) of a time-domain signal is the distribution of power contained within the signal over frequency, based on a finite set of data. The frequency-domain representation of the signal is often easier to analyze than the time-domain representation. Many signal processing applications, such as noise cancellation and system identification, are based on the frequency-specific modifications of signals. The goal of the spectral density estimation is to estimate the spectral density of a signal from a sequence of time samples. Depending on what is known about the signal, estimation techniques can involve parametric or nonparametric approaches, and can be based on time-domain or frequency-domain analysis. For example, a common parametric technique involves fitting the observations to an autoregressive model. A common nonparametric technique is the periodogram. The spectral density is estimated using Fourier transform methods such as the Welch Method. You can also use other techniques such as the maximum entropy method.

In DSP System Toolbox, you can perform real-time spectral analysis of a dynamic signal using the Spectrum Analyzer block. Alternatively, you can use the Spectrum Estimator block from the `dspspect3` library to compute the power spectrum, and `Array Plot` block to view the spectrum.

Estimate PSD Using the Spectrum Analyzer

You can view the Power Spectral Density (PSD) of a signal using the Spectrum Analyzer block. The PSD is computed in real time and varies with the input signal, and with changes in the properties of the `Spectrum Analyzer` block. You can change the dynamics of the input signal and see what effect those changes have on the spectral density of the signal in real time. The PSD data can only be viewed and is not available for processing. To acquire and process the data, use the `Spectrum Estimator` block in the `dspspect3` library.

The model `ex_psd_sa` feeds a noisy sine wave signal to the Spectrum analyzer block. The sine wave signal is a sum of two sinusoids: one at a frequency of 5000 Hz and the

other at a frequency of 10,000 Hz. The noise at the input is Gaussian, with zero mean and a standard deviation of 0.01.

Open and Inspect the Model

To open the model, enter `ex_psd_sa` in the MATLAB command prompt.

Here are the settings of the blocks in the model.

Block	Parameter Changes	Purpose of the block
Sine Wave 1	• **Frequency** to 5000 • **Sample time** to 1/44100 • **Sample per frame** to 1024	Sinusoid signal with frequency at 5000 Hz
Sine Wave 2	• **Frequency** to 10000 • **Phase offset (rad)** to 10 • **Sample time** to 1/44100 • **Sample per frame** to 1024	Sinusoid signal with frequency at 10000 Hz
Random Source	• **Source type** to Gaussian • **Variance** to 1e-4 • **Sample time** to 1/44100 • **Sample per frame** to 1024	Random Source block generates a random noise signal with properties specified through the block dialog box
Add	**List of signs** to +++.	Add block adds random noise to the input signal
Spectrum Analyzer	Click the **Spectrum Settings** icon . A pane appears on the right. • In the **Main options** pane, under **Type**, select Power density. • In the **Trace options** pane, clear the **Two-sided spectrum** check box. This shows only the real-half of the spectrum.	Spectrum Analyzer block shows the Power Spectrum Density of the signal

Block	Parameter Changes	Purpose of the block
	• Clear the **Max-hold trace** and **Min-hold trace** check boxes if needed.	

Play the model. Open the Spectrum Analyzer block to view the PSD of the sine wave signal. There are two tones at frequencies 5000 Hz and 10,000 Hz, which correspond to the two frequencies at the input.
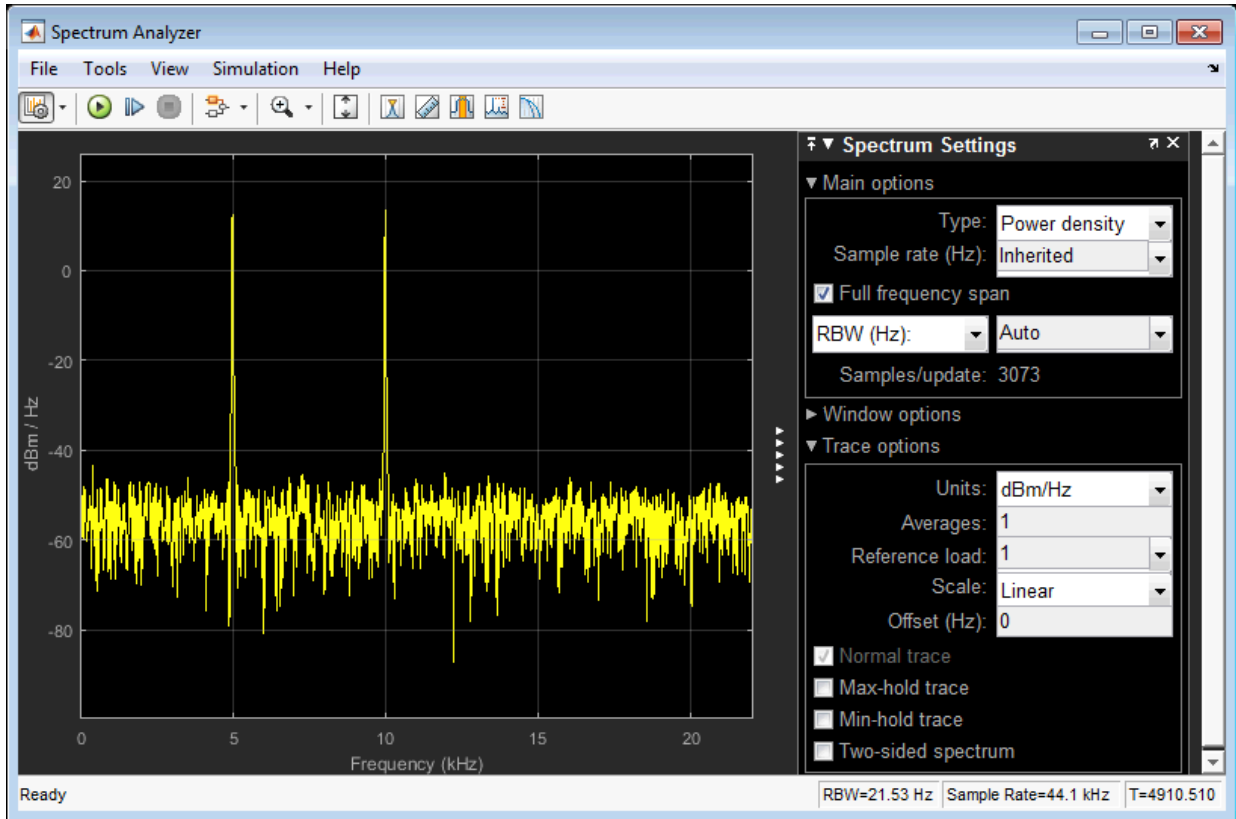
RBW is the resolution bandwidth. It is the minimum frequency bandwidth that can be resolved by the spectrum analyzer. By default, **RBW (Hz)** is set to Auto. In this mode,

RBW is the ratio of the frequency span to 1024. In a two-sided spectrum, this value is $F_s/1024$, while in a one-sided spectrum, it is $(F_s/2)/1024$.

Using this value of *RBW*, the window length ($N_{samples}$) is computed iteratively using this relationship:

$$N_{samples} = \frac{\left(1 - \dfrac{Overlap}{100}\right) * NENBW * F_s}{RBW}$$

Overlap is the amount of overlap between the previous and current buffered data segments. *NENBW* is the equivalent noise bandwidth of the window. For more information on the details of the spectral estimation algorithm, see "Spectral Analysis" on page 7-68.

RBW calculated in this mode gives a good frequency resolution.

To distinguish between two frequencies in the display, the distance between the two frequencies must be at least RBW. In this example, the distance between the two peaks is 5000 Hz, which is greater than *RBW*. Hence, you can see the peaks distinctly. Change the frequency of the second sine wave from 10000 Hz to 5015 Hz. The difference between the two frequencies is less than *RBW*.
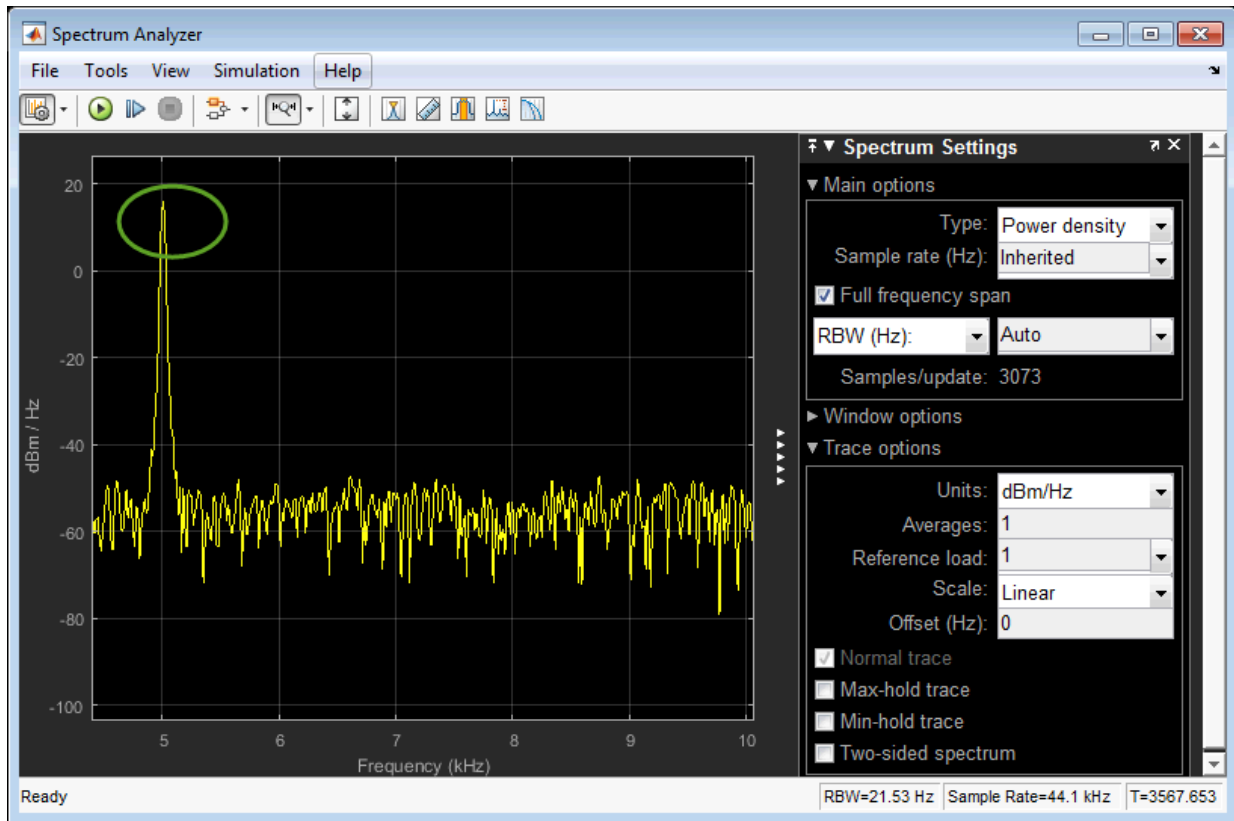
The peaks are not distinguishable. To increase the frequency resolution, decrease *RBW* to 1 Hz.
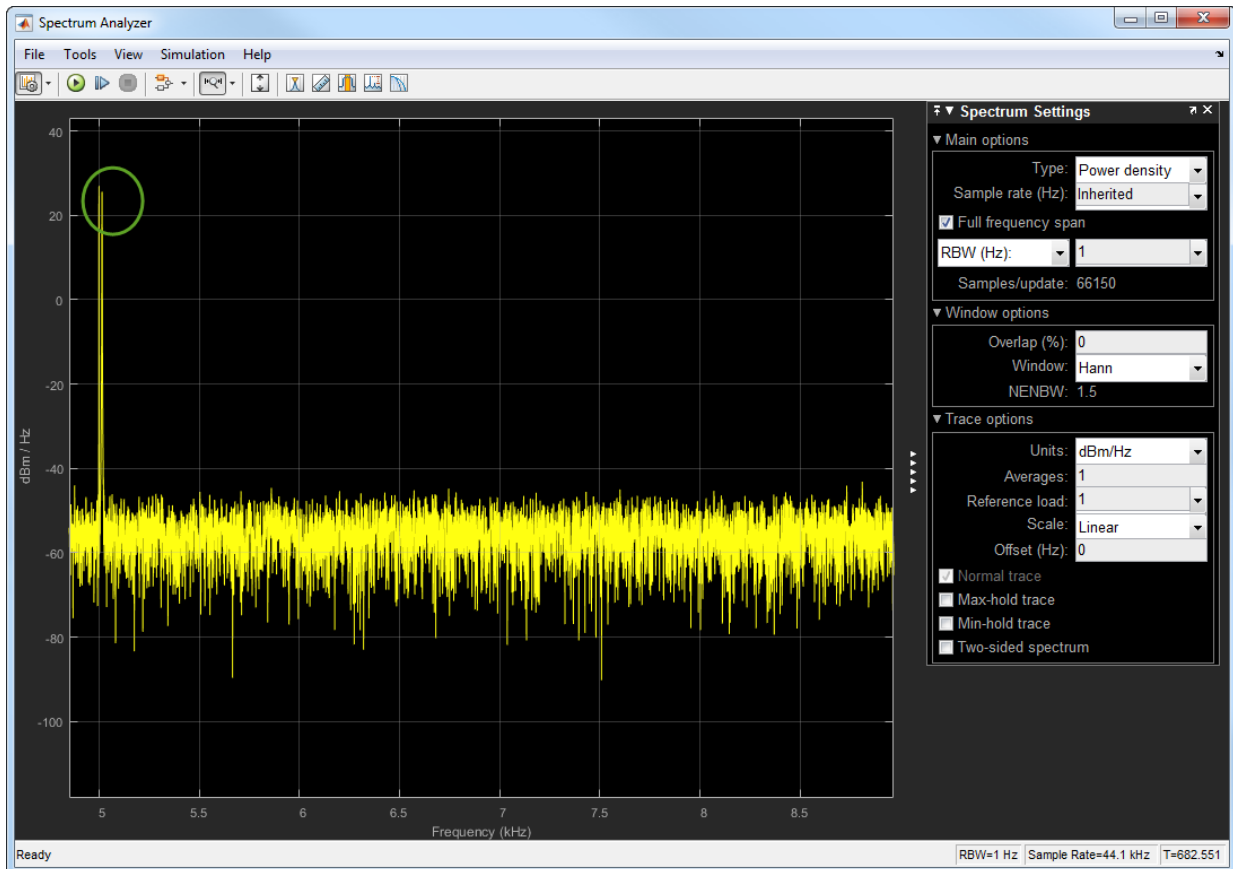
The two peaks, which are 15 Hz apart, are now distinguishable.

When you increase the frequency resolution, the window length increases, but the tradeoff is that the time resolution decreases.

Change the Input Signal

When you change the dynamics of the input signal during simulation, the PSD of the signal also changes in real time. While the simulation is running, change the **Frequency** of the `Sine Wave 1` block to `8000` and click **Apply**. The second tone in the spectral analyzer output shifts to 8000 Hz and you can see the change in real time.
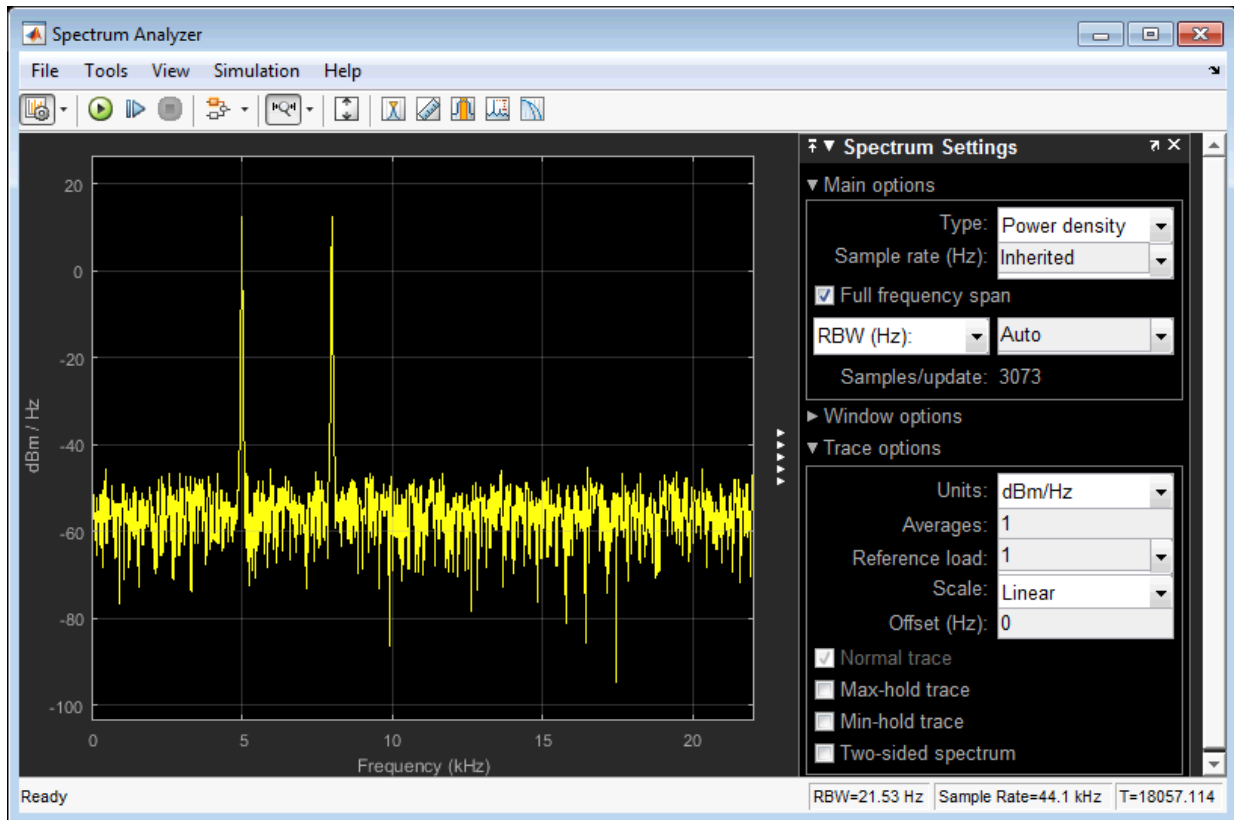
Change the Spectrum Analyzer Settings

When you change the settings in the Spectrum Analyzer block, the effect can be seen on the spectral data in real time.

As an example, in the Spectrum Analyzer block, on the **Main** options pane, select Window length. The window length is 1024. Run the model.
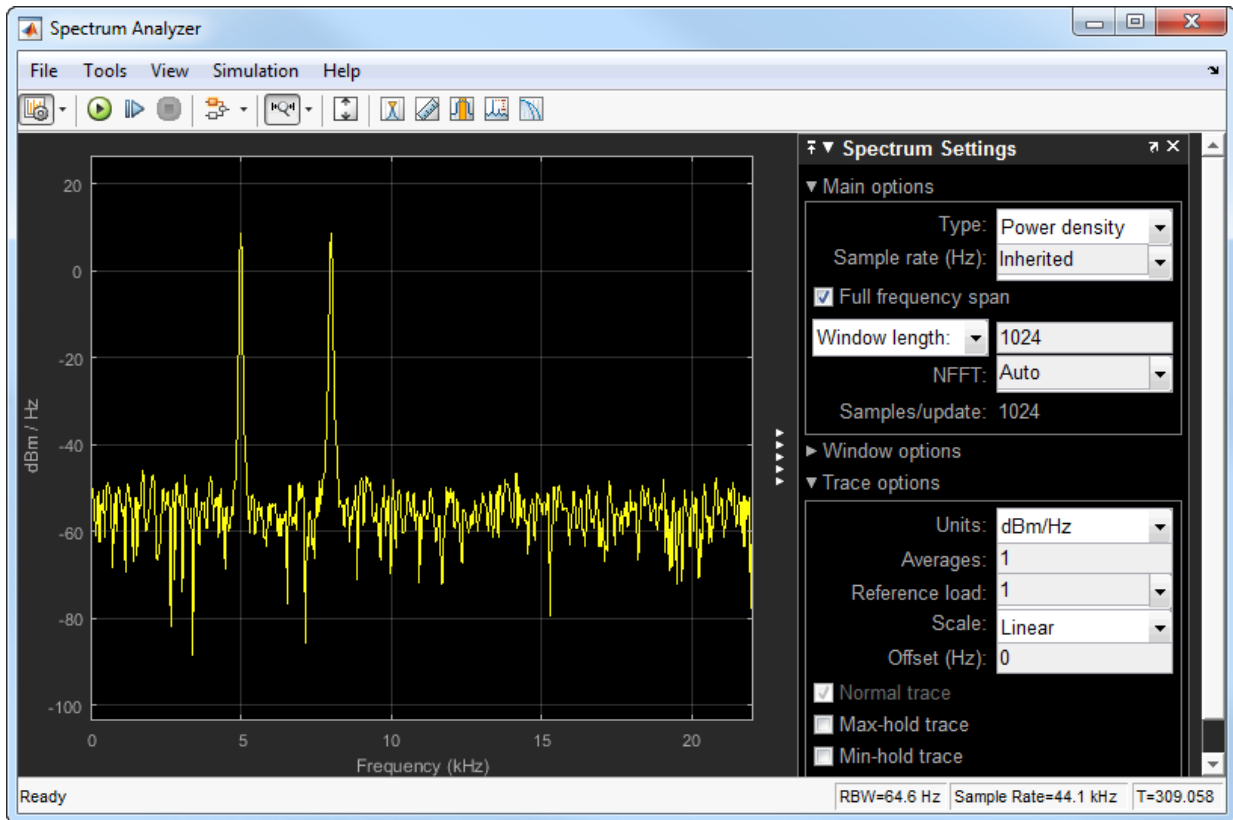
During the simulation, change the **Window length** from 1024 to 500 and click anywhere on the screen. The **Samples/update** ($N_{samples}$) parameter changes to 500.
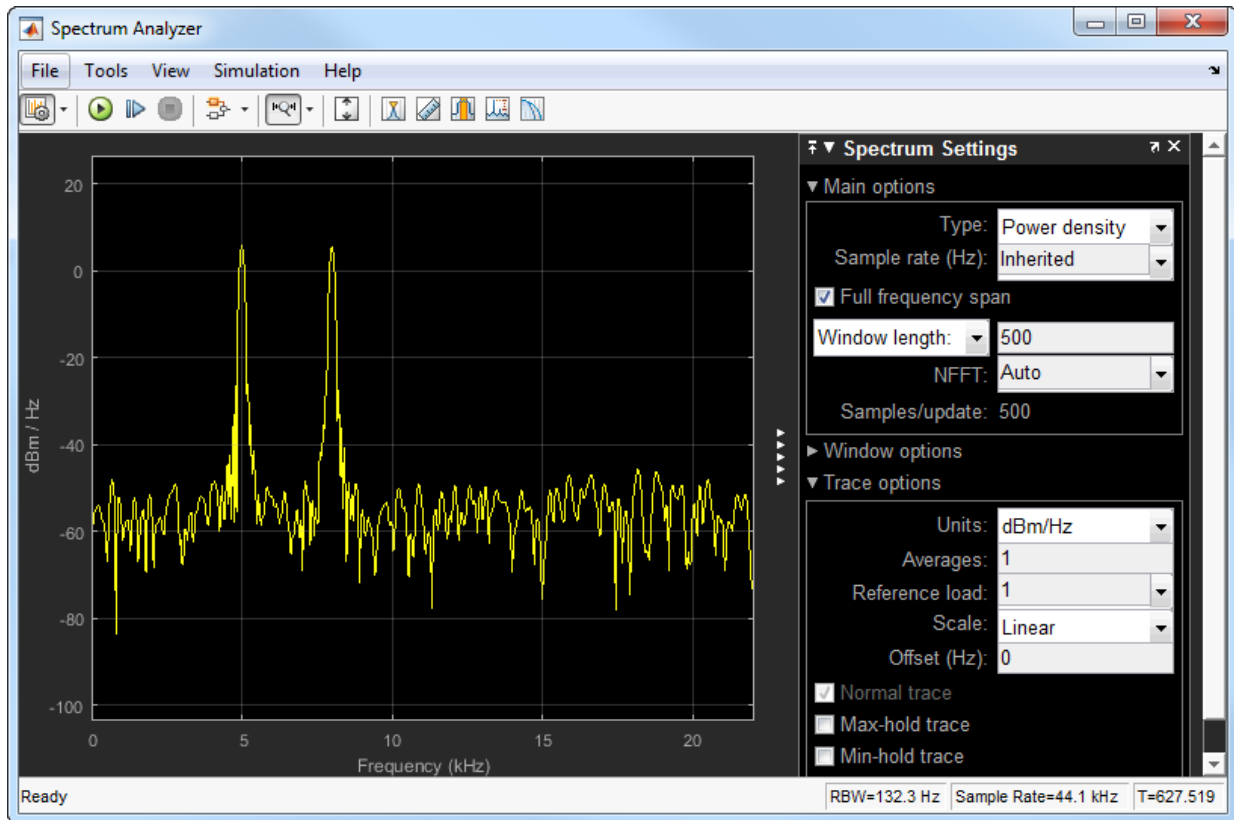
When N$_{samples}$ decreases, *RBW* increases using this relationship:

$$RBW = \frac{\left(1 - \dfrac{Overlap}{100}\right) * NENBW * F_s}{N_{samples}}$$

When you change any parameter in the spectrum analyzer settings, the effect is immediately seen on the PSD data. For more information on how the `Spectrum Analyzer` settings affect the spectral density data, see the 'Algorithms' section of the `Spectrum Analyzer` block reference page.

Convert the Power in Watts to dBW and dBm

The spectrum analyzer provides three units to specify the power spectral density: `Watts/Hz`, `dBm/Hz`, and `dBW/Hz`. Corresponding units of power are `Watts`, `dBm`, and `dBW`. The default unit of **Power** is `dBm`.

Power in `dBW` is given by:

$$P_{dBW} = 10 \log 10 (\text{power in watt} / 1 \, \text{watt})$$

Power in `dBm` is given by:

$$P_{dBm} = 10 \log 10 (\text{power in watt} / 1 \, \text{milliwatt})$$

For a sine wave signal with an amplitude (A) of 1 V, the power of a one-sided spectrum in `Watts` is given by:

$$A^2 / 2$$

In this example, this power equals 0.5 W. Corresponding power in dBm is given by:

$$P_{dBm} = 10 \log 10 (\text{power in watt} / 1 \, \text{milliwatt})$$
$$P_{dBm} = 10 \log 10 (0.5 / 10 \wedge -3)$$

Here, the power equals 26.9897 dBm. To confirm this value with a peak finder, click **Tools** > **Measurements** > **Peak Finder**.

For a white noise signal, the spectrum is flat for all frequencies. The spectrum analyzer in this example shows a one-sided spectrum in the range [0 Fs/2]. For a white noise signal with a variance of 1e-4, the power per unit bandwidth ($P_{\text{unitbandwidth}}$) is 1e-4. The total power in watts over the entire frequency range is:

$$P_{whitenoise} = P_{unitbandwidth} * \text{number of frequency bins},$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{Fs / 2}{RBW} \right),$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{22050}{21.53} \right)$$

The number of frequency bins is the ratio of total bandwidth to RBW. For a one-sided spectrum, the total bandwidth is half the sampling rate. RBW in this example is 21.53 Hz. With all these values, the total power of the white noise is -39.87 dBm.

Estimate PSD Using the `Spectrum Estimator` Block

Using the Spectrum Analyzer block, you can view and analyze, but cannot process the PSD data. To process the data, you must compute the spectral density using the `Spectrum Estimator` block in the `dspspect3` library.
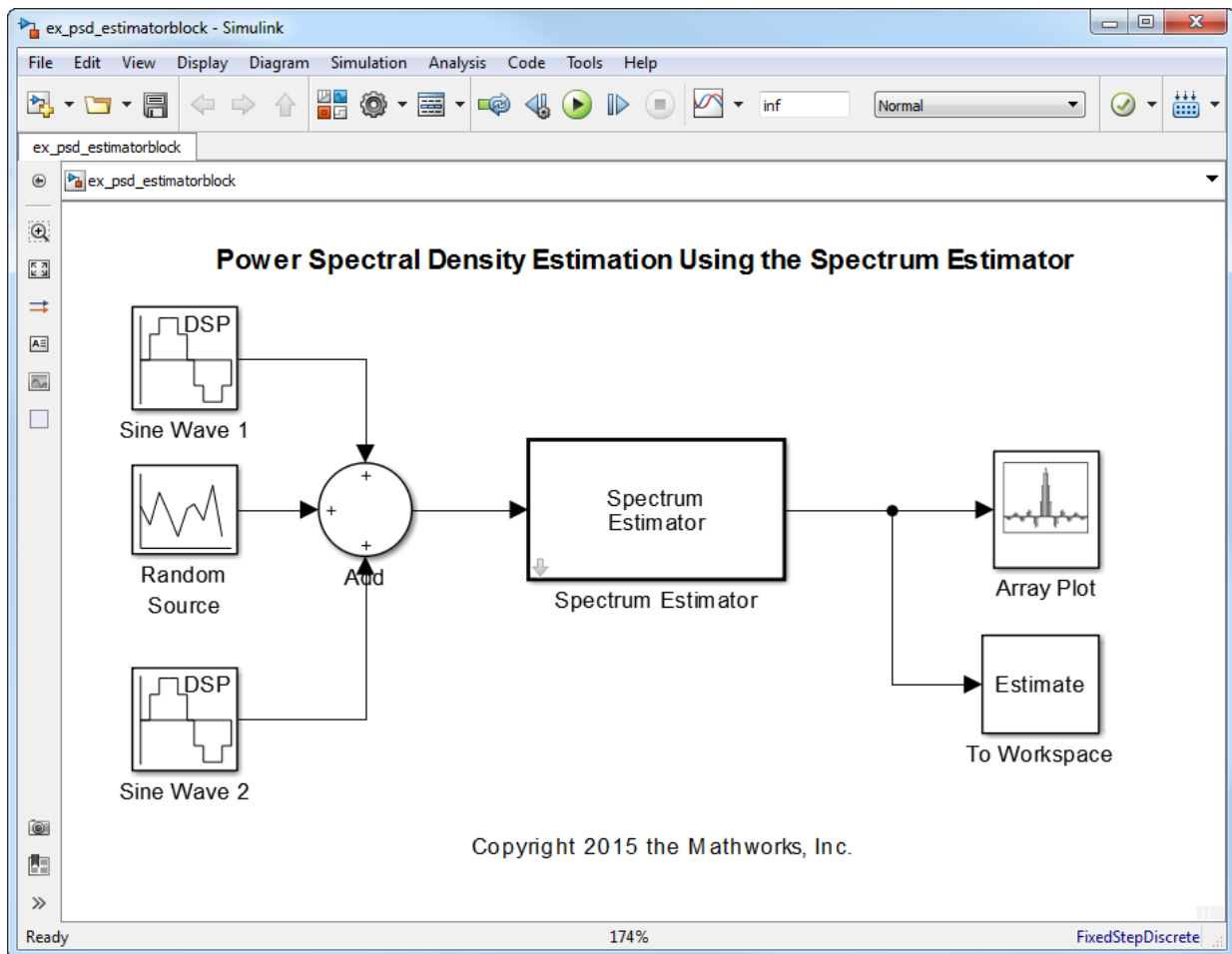
Replace the `Spectrum Analyzer` block in `ex_psd_sa` with the `Spectrum Estimator` block followed by an `Array Plot` block. To view the model, enter `ex_psd_estimatorblock` in the MATLAB command prompt. In addition, to access the spectral estimation data in MATLAB, connect the `To Workspace` block to the output of the `Spectrum Estimator` block. Here are the changes to the settings of the `Spectrum Estimator` block and the `Array Plot` block.

Block	Parameter Changes	Purpose of the block
Spectrum Estimator	The following settings align with the Spectrum Analyzer settings in `ex_psd_sa`. Set: • **Spectrum type** to `Power density`. • **Frequency resolution method** to `Window length`. • **Window length source** to `Specify on dialog`. • **Window length** to `1024`.	Computes the power spectral density of the input signal using Welch's method of averaging periodograms
Array Plot	Click **View** and • select `Style`. In the Style window, select the **Plot type** as `Stairs`. • select `Configuration Properties`. In the Configuration Properties window, set the **Sample increment** as `44100/1024`. For details, see the section 'Convert the X-axis to Represent Frequency'.	Displays the power spectral density data

The spectrum displayed in the `Array Plot` block is similar to the spectrum seen in the `Spectrum Analyzer` block in `ex_psd_sa`.
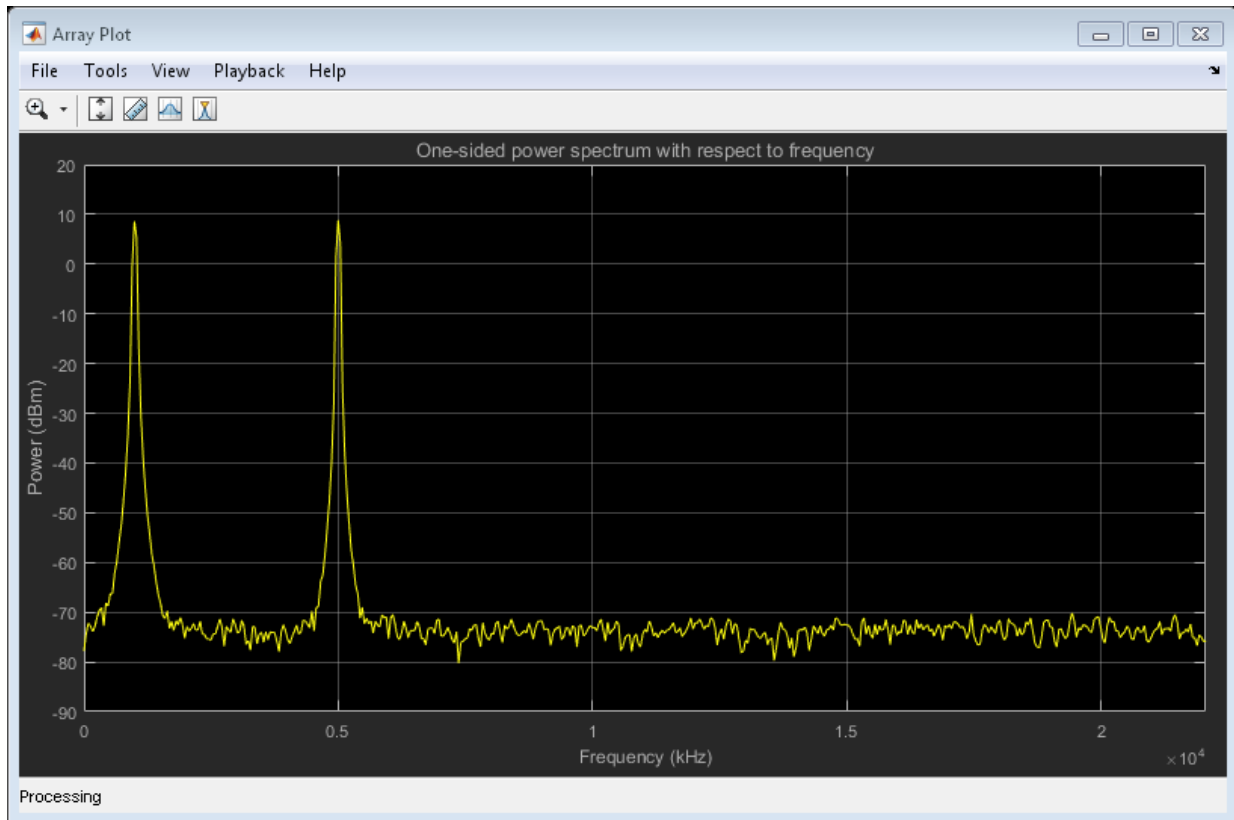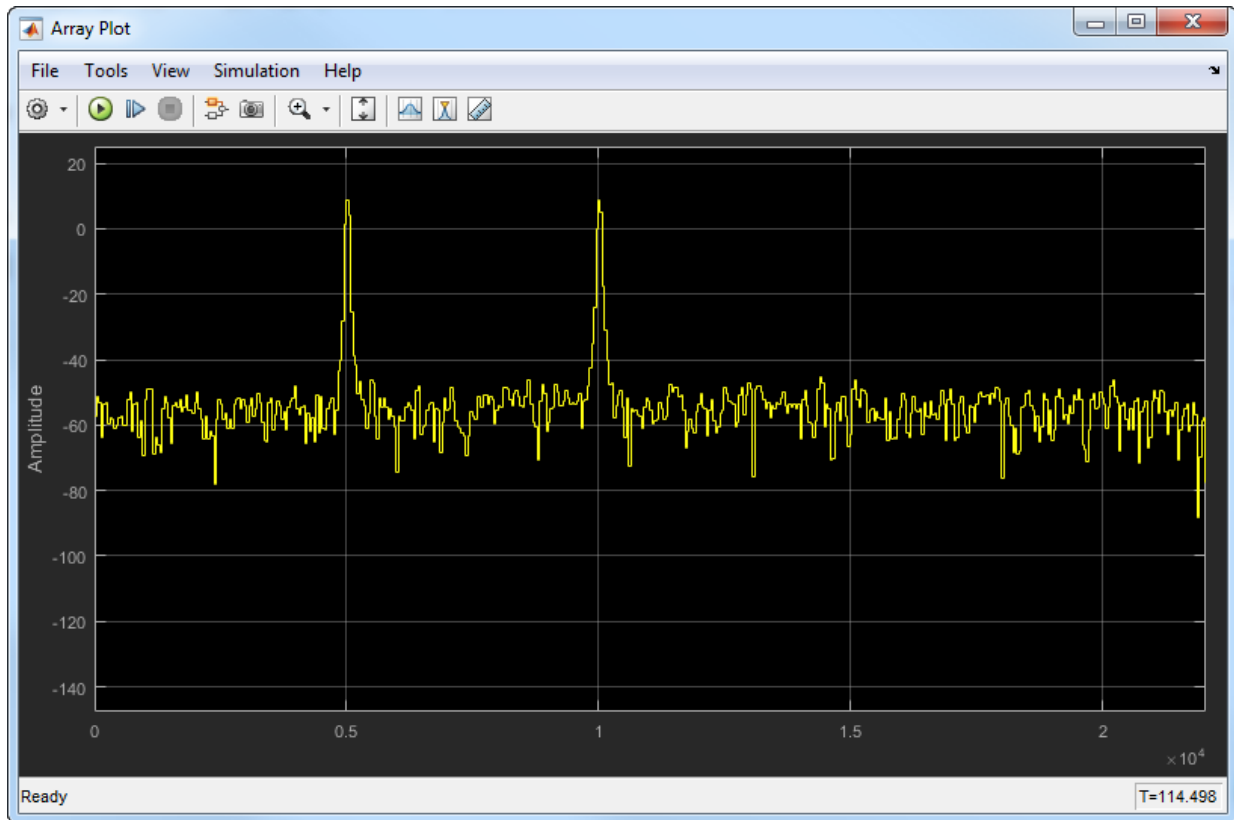
Convert x-axis to Represent Frequency

By default, the `Array Plot` block plots the PSD data with respect to the number of samples per frame. The number of points on the x-axis equals the length of the input frame. The Spectrum analyzer plots the PSD data with respect to frequency. For a one-sided spectrum, the frequency varies in the range [0 Fs/2]. For a two-sided spectrum, the frequency varies in the range [-Fs/2 Fs/2]. To convert the x-axis of the array plot from sample-based to frequency-based, set the **Sample Increment** parameter in the configuration properties window to *Fs/framelength*. For a one-sided spectrum, *XOffset* parameter must be 0. For a two-sided spectrum, *XOffset* must be -Fs/2. In this example, for a one-sided spectrum, **Sample Increment** must be 44100/1024.

Live Processing

To process the spectral data while streaming, pass the output of the estimator block into a processing logic.

More About

Estimate the Transfer Function of an Unknown System

In this section...
"Estimate the Transfer Function in MATLAB" on page 7-53
"Estimate the Transfer Function in Simulink" on page 7-58

You can estimate the transfer function of an unknown system based on the system's measured input and output data.

In DSP System Toolbox, you can estimate the transfer function of a system using the dsp.TransferFunctionEstimator System object in MATLAB and the `Discrete Transfer Function Estimator` block in Simulink. The relationship between the input x and output y is modeled by the linear, time-invariant transfer function T_{xy}. The transfer function is the ratio of the cross power spectral density of x and y, P_{yx}, to the power spectral density of x, P_{xx}:

$$T_{xy}(f) = \frac{P_{yx}(f)}{P_{xx}(f)}$$

The `dsp.TransferFunctionEstimator` object and `Discrete Transfer Function Estimator` block use the Welch's averaged periodogram method to compute the P_{xx} and P_{xy}. For more details on this method, see "Spectral Analysis" on page 7-68.

Coherence

The coherence, or magnitude-squared coherence, between x and y is defined as:

$$C_{xy}(f) = \frac{\left|P_{xy}\right|^2}{P_{xx} * P_{yy}}$$

The coherence function estimates the extent to which you can predict y from x. The value of the coherence is in the range $0 \leq C_{xy}(f) \leq 1$. If $C_{xy} = 0$, the input x and output y are unrelated. A C_{xy} value greater than 0 and less than 1 indicates one of the following:

- Measurements are noisy.

- The system is nonlinear.
- Output *y* is a function of *x* and other inputs.

The coherence of a linear system represents the fractional part of the output signal power that is produced by the input at that frequency. For a particular frequency, $1 - C_{xy}$ is an estimate of the fractional power of the output that the input does not contribute to.

When you set the `OutputCoherence` property of `dsp.TransferFunctionEstimator` to `true`, the object computes the output coherence. In the `Discrete Transfer Function Estimator` block, to compute the coherence spectrum, select the **Output magnitude squared coherence estimate** check box.

Estimate the Transfer Function in MATLAB

To estimate the transfer function of a system in MATLAB, use the `dsp.TransferFunctionEstimator` System object. The object implements the Welch's average modified periodogram method and uses the measured input and output data for estimation.

Initialize the System

The system is a cascade of two filter stages: `dsp.LowpassFilter` and a parallel connection of `dsp.AllpassFilter` and `dsp.AllpoleFilter`.

```
allpole = dsp.AllpoleFilter;
allpass = dsp.AllpassFilter;
lpfilter = dsp.LowpassFilter;
```

Specify Signal Source

The input to the system is a sine wave with a frequency of 100 Hz. The sampling frequency is 44.1 kHz.

```
sine = dsp.SineWave('Frequency',100,'SampleRate',44100,...
    'SamplesPerFrame',1024);
```

Create Transfer Function Estimator

To estimate the transfer function of the system, create the `dsp.TransferFunctionEstimator` System object.

```
tfe  = dsp.TransferFunctionEstimator('FrequencyRange','onesided',...
```

```
    'OutputCoherence', true);
```

Create Array Plot

Initialize two `dsp.ArrayPlot` objects: one to display the magnitude response of the system and the other to display the coherence estimate between the input and the output.

```
tfeplotter = dsp.ArrayPlot('PlotType','Line',...
    'XLabel','Frequency (Hz)',...
    'YLabel','Magnitude Response (dB)',...
    'YLimits',[-120 20],...
    'XOffset',0,...
    'XLabel','Frequency (Hz)',...
    'Title','System Transfer Function',...
    'SampleIncrement',44100/1024);
coherenceplotter = dsp.ArrayPlot('PlotType','Line',...
    'YLimits',[0 1.2],...
    'YLabel','Coherence',...
    'XOffset',0,...
    'XLabel','Frequency (Hz)',...
    'Title','Coherence Estimate',...
    'SampleIncrement',44100/1024);
```

By default, the *x*-axis of the array plot is in samples. To convert this axis into frequency, set the 'SampleIncrement' property of the `dsp.ArrayPlot` object to $\frac{F_s}{1024}$. In this example, this value is 44100/1024, or 43.0664. For a two-sided spectrum, the `XOffset` property of the `dsp.ArrayPlot` object must be -_Fs_/2. The frequency varies in the range [-_Fs_/2 *Fs*/2]. In this example, the array plot shows a one-sided spectrum. Hence, set the `XOffset` to 0. The frequency varies in the range [0 *Fs*/2].

Estimate the Transfer Function

The transfer function estimator accepts two signals: input to the two-stage filter and output of the two-stage filter. The input to the filter is a sine wave containing additive white Gaussian noise. The noise has a mean of zero and a standard deviation of 0.1. The estimator estimates the transfer function of the two-stage filter. The output of the estimator is the frequency response of the filter, which is complex. To extract the magnitude portion of this complex estimate, use the `abs` function. To convert the result into dB, apply a conversion factor of 20*log10(magnitude).

```
for Iter = 1:1000
```

```
input = step(sine) + .1*randn(1024,1);
lpfout = step(lpfilter,input);
allpoleout = step(allpole,lpfout);
allpassout = step(allpass,lpfout);
output = allpoleout + allpassout;
[tfeoutput,outputcoh] = step(tfe,input,output);
step(tfeplotter,20*log10(abs(tfeoutput)));
step(coherenceplotter,outputcoh);
end
```
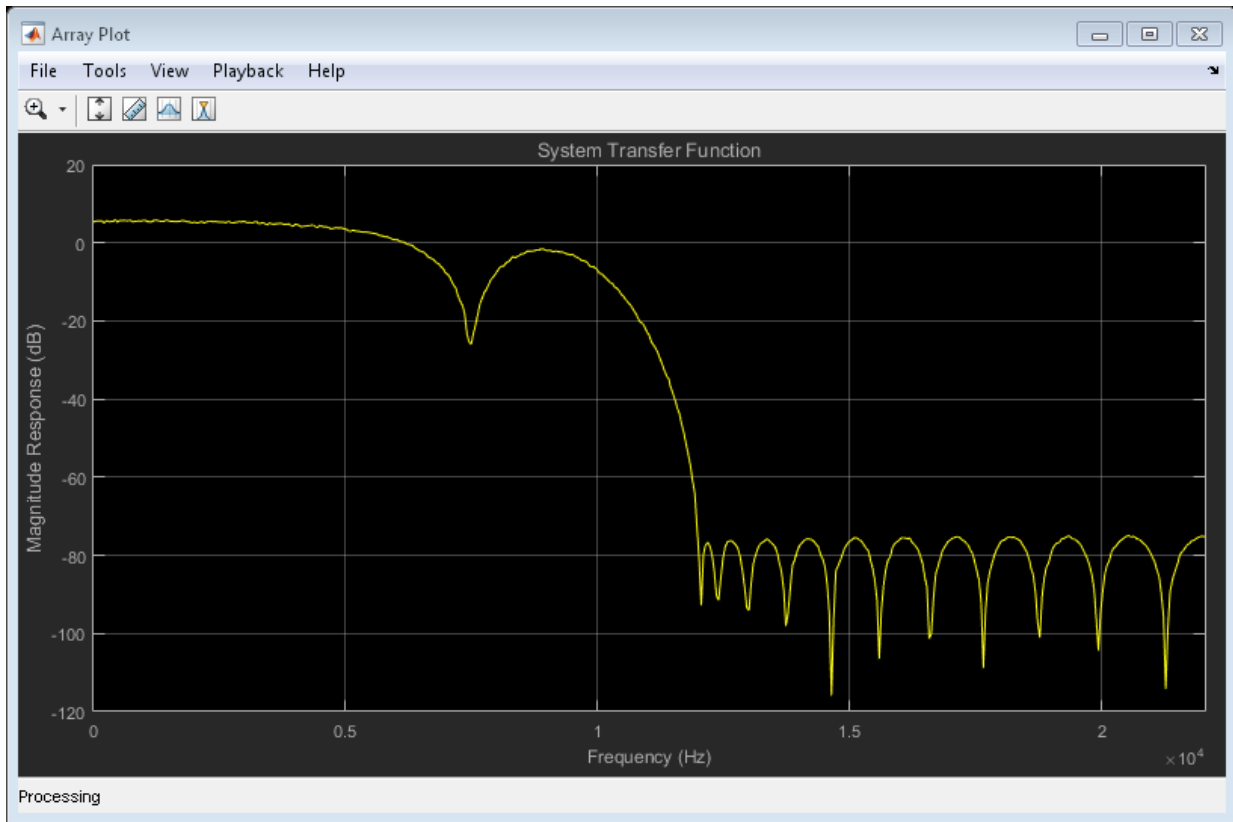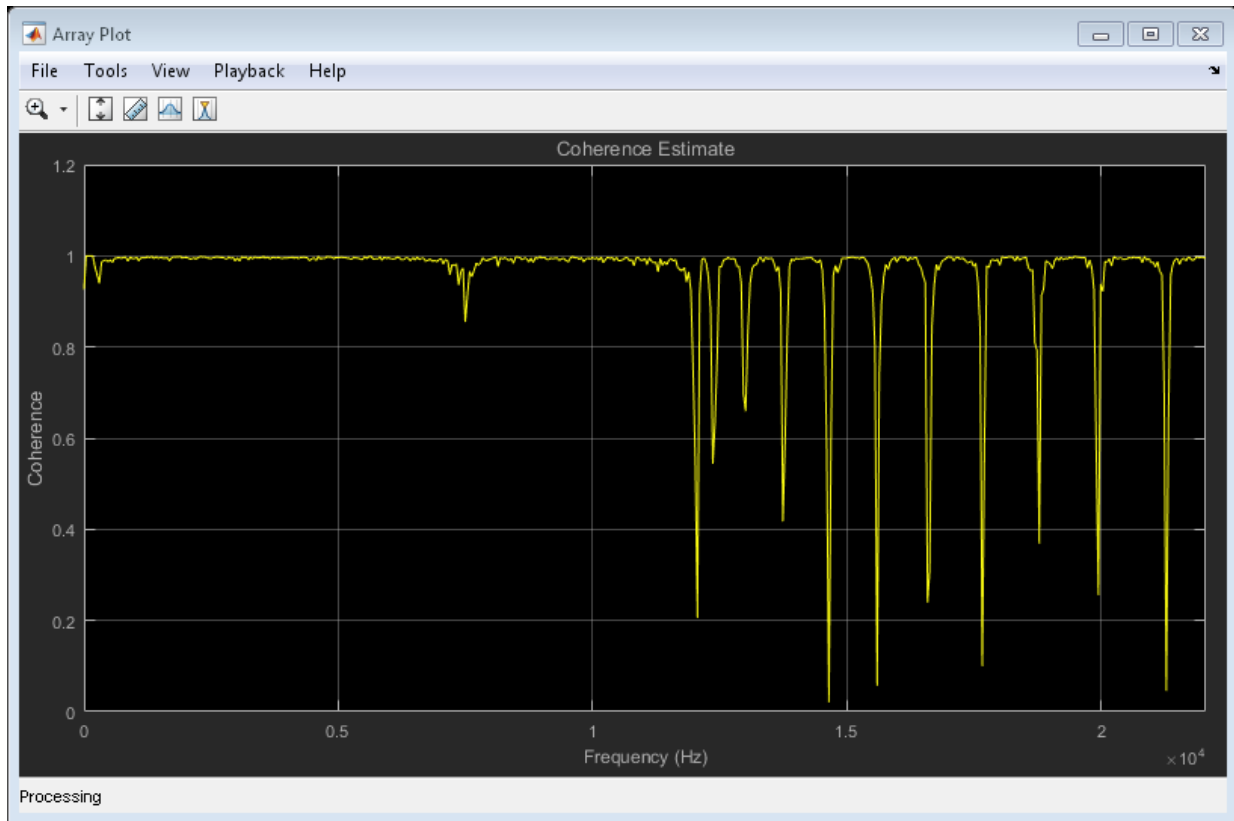
The first plot shows the magnitude response of the system. The second plot shows the coherence estimate between the input and output of the system. Coherence in the plot varies in the range [0 1] as expected.

Magnitude Response of the Filter Using fvtool

The filter is a cascade of two filter stages - `dsp.LowpassFilter` and a parallel connection of `dsp.AllpassFilter` and `dsp.AllpoleFilter`. All the filter objects are used in their default state. Using the filter coefficients, derive the system transfer function and plot the frequency response using `freqz`. Below are the coefficients in the [Num] [Den] format:

- All pole filter - [1 0] [1 0.1]
- All pass filter - [0.5 -1/sqrt(2) 1] [1 -1/sqrt(2) 0.5]

- Lowpass filter - Determine the coefficients using the following commands:

```
lpf = dsp.LowpassFilter;
Coefficients = coeffs(lpf);
```

`Coefficients.Numerator` gives the coefficients in an array format. The mathemtical derivation of the overall system transfer function is not shown here. Once you derive the transfer function, run fvtool and you can see the frequency response below:
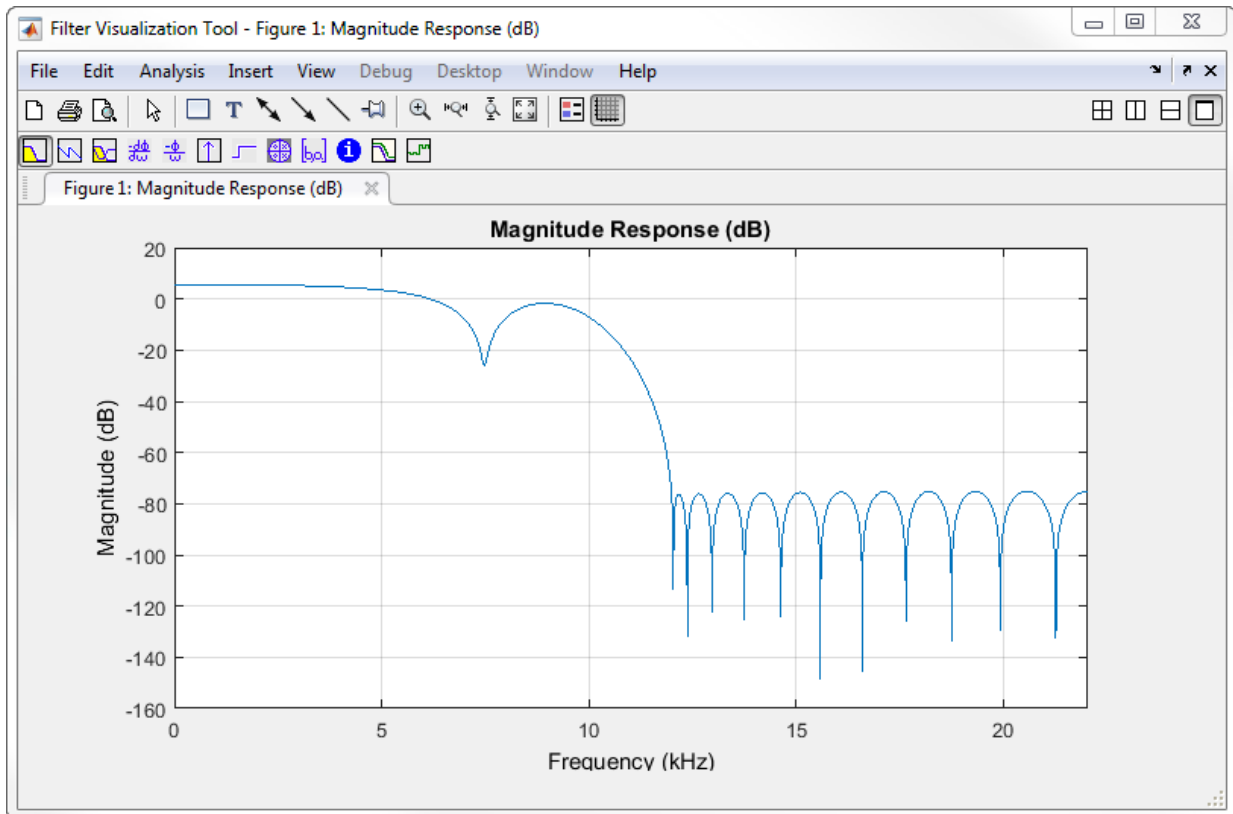
The magnitude response that fvtool shows matches the magnitude response that the `dsp.TransferFunctionEstimator` object estimates.

Estimate the Transfer Function in Simulink

To estimate the transfer function of a system in Simulink, use the `Discrete Transfer Function Estimator` block. The block implements the Welch's average modified periodogram method and uses the measured input and output data for estimation.

The system is a cascade of two filter stages: a lowpass filter and a parallel connection of an allpole filter and allpass filter. The input to the system is a sine wave containing additive white Gaussian noise. The noise has a mean of zero and a standard deviation of 0.1. The input to the estimator is the system input and the system output. The output of the estimator is the frequency response of the system, which is complex. To extract the magnitude portion of this complex estimate, use the `Abs` block. To convert the result into dB, the system uses a `dB (1 ohm)` block.
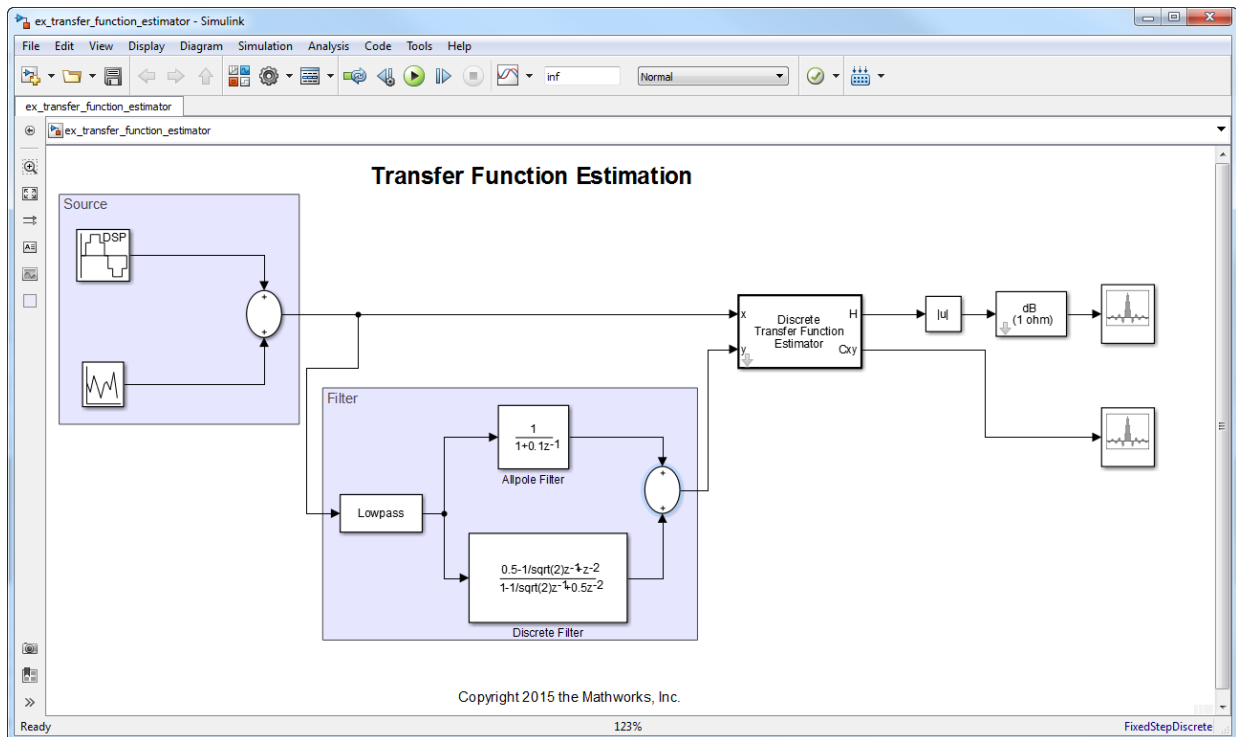
Open and Inspect the Model

To open the model, enter `ex_transfer_function_estimator` in the MATLAB command prompt.

Here are the settings of the blocks in the model.

Block	Parameter Changes	Purpose of the block
Sine Wave	• **Sample time** to `1/44100` • **Sample per frame** to `1024`	Sinusoid signal with frequency at 100 Hz
Random Source	• **Source type** to `Gaussian` • **Variance** to 0.01 • **Sample time** to 1/44100 • **Sample per frame** to `1024`	`Random Source` block generates a random noise signal with properties specified through the block dialog box
Lowpass Filter	No change	Lowpass filter
Allpole Filter	No change	Allpole filter with coefficients `[1 0.1]`
Discrete Filter	• **Numerator** to `[0.5 -1/sqrt(2) 1]` • **Denominator** to `[1 -1/sqrt(2) 0.5]`	Allpass filter with coefficients `[-1/sqrt(2) 0.5]`
Discrete Transfer Function Estimator	• **Frequency range** to `One-sided` • **Number of spectral averages** to 8	Transfer function estimator
Abs	No change	Extracts the magnitude information from the output of the transfer function estimator
First `Array Plot` block	Click **View**:	Shows the magnitude response of the system

Block	Parameter Changes	Purpose of the block
	• Select **Style** and set **Plot type** to `Line`. • Select **Configuration Properties**: From the **Main** tab, set **Sample increment** to `44100/1024` and **X-offset** to `0`. In the **Display** tab, specify the **Title** as `Magnitude Response of the System in dB`, **X-label** as `Frequency (Hz)`, and **Y-label** as `Amplitude (dB)`	
Second `Array Plot` block	Click **View**: • Select **Style** and set **Plot type** to `Line`. • Select **Configuration Properties**: From the **Main** tab, set **Sample increment** to `44100/1024` and **X-offset** to `0`. In the **Display** tab, specify the **Title** as `Coherence Estimate`, **X-label** as `Frequency (Hz)`, and **Y-label** as `Amplitude`	Shows the coherence estimate

By default, the *x*-axis of the array plot is in samples. To convert this axis into frequency, the **Sample increment** parameter is set to `Fs/1024`. In this example, this value is `44100/1024`, or `43.0664`. For a two-sided spectrum, the **X-offset** parameter must be − `Fs/2`. The frequency varies in the range `[-Fs/2 Fs/2]`. In this example, the array plot shows a one-sided spectrum. Hence, the **X-offset** is set to 0. The frequency varies in the range `[0 Fs/2]`.

Run the Model
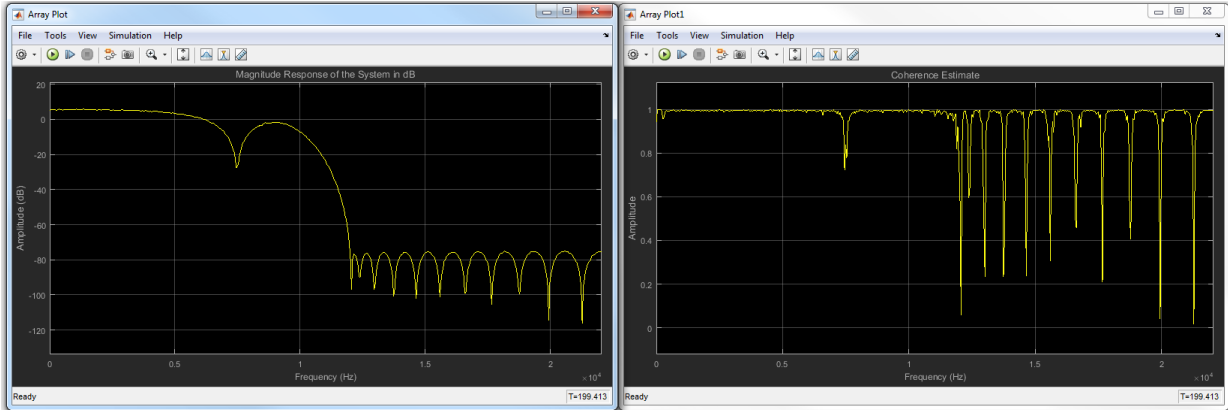
The first plot shows the magnitude response of the system. The second plot shows the coherence estimate between the input and output of the system. Coherence in the plot varies in the range [0 1] as expected.

More About

- "Spectral Analysis" on page 7-68
- "Estimate the Power Spectral Density in MATLAB" on page 7-23
- "Estimate the Power Spectral Density in Simulink" on page 7-37

View the Spectrogram Using Spectrum Analyzer

In this section...

Spectrograms are a two-dimensional representation of the power spectrum of a signal as this signal sweeps through time. They give a visual understanding of the frequency content of your signal. Each line of the spectrogram is one periodogram computed using the Welch's algorithm of averaging modified periodogram.

To show the concepts of the spectrogram, this example uses the model `ex_psd_sa` as the starting point. Open the model and double-click the `Spectrum Analyzer` block. In the **Spectrum Settings** pane, change **Type** to `Spectrogram`. Run the model. You can see the spectrogram output in the Spectrum Analyzer window.
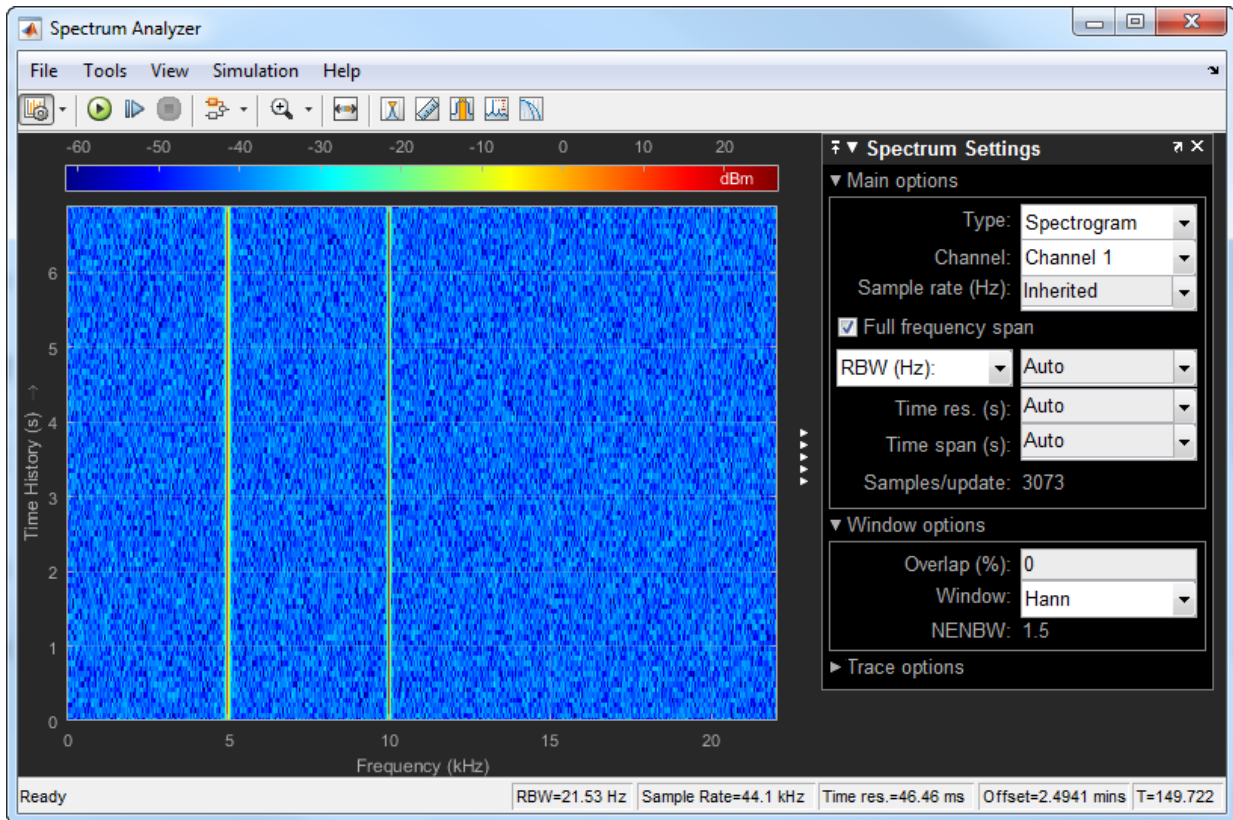
Colormap

Power spectrum density is computed as a function of frequency f and is plotted as a horizontal line. Each point on this line is given a specific color based on the value of the power density at that particular frequency. The color is chosen based on the colormap seen at the top of the display. To change the colormap, click **View** > **Configuration Properties**, and choose one of the options in **color map**. Make sure **Type** is set to Spectrogram. By default, **color map** is set to jet(256).

The two frequencies of the sine wave are distinctly visible at 5 kHz and 10 kHz. The sine wave is embedded in Gaussian noise, which has a variance of 0.0001. This value corresponds to a power of -40 dBm. The color that maps to -40 dBm is assigned to the noise spectrum. The power of the sine wave is 26.9 dBm at 5 kHz and 10 kHz. The color

used in the display at these two frequencies corresponds to 26.9 dBm on the colormap. For more information on how the power is computed in dBm, see 'Conversion of power in watts to dBW and dBm'.

To confirm the dBm values, change **Type** to `Power`. This view shows the power of the signal at various frequencies.
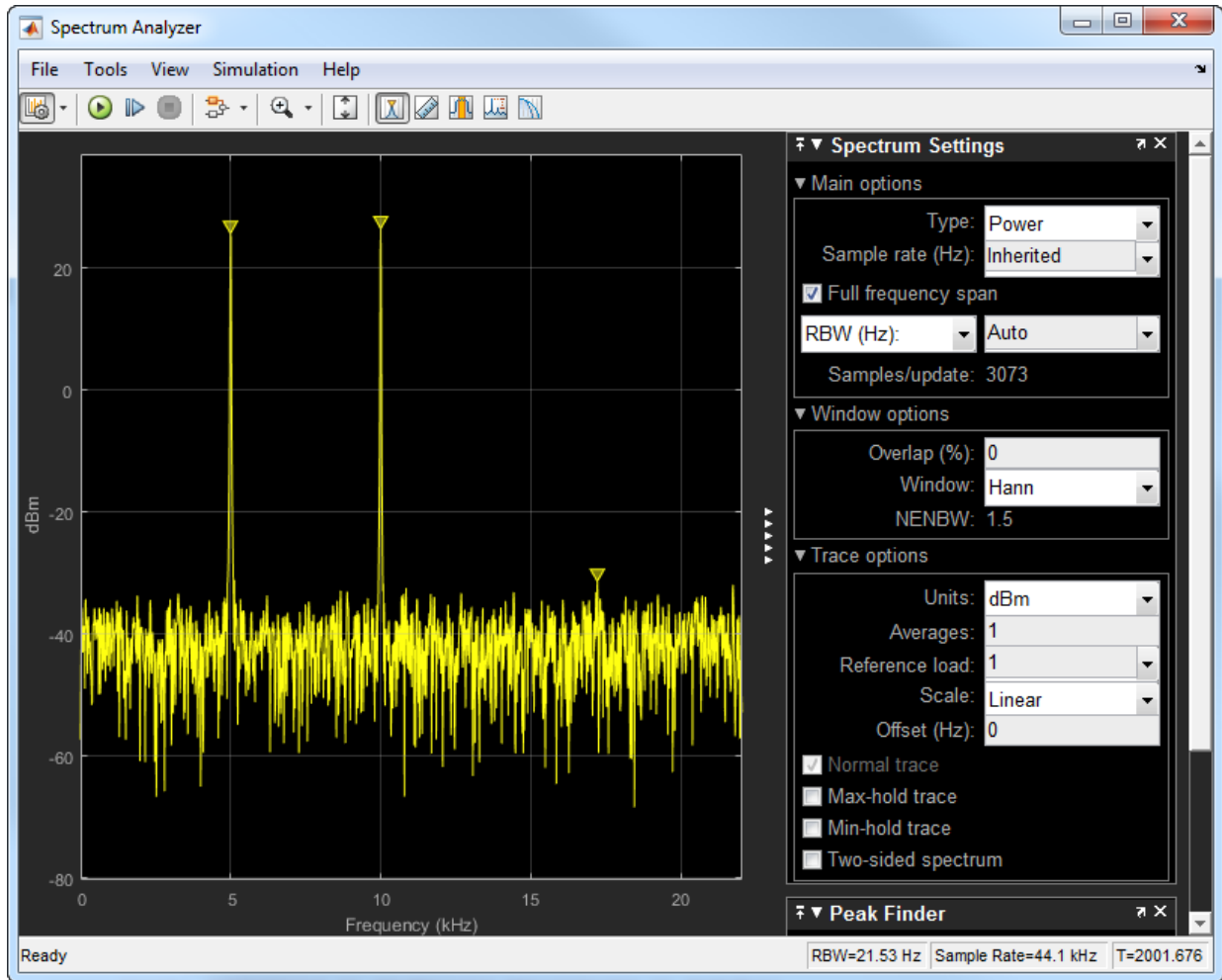
You can see that the two peaks in the power display have an amplitude of 26.9 dBm and the white noise is averaging around -40 dBm.

Display

In the spectrogram display, time scrolls from bottom to top, so the most recent data is shown at the bottom of the display. As the simulation time increases, the offset time also increases to keep the vertical axis limits constant while accounting for the incoming data. The Offset value, along with the simulation time, is displayed at the bottom-right corner of the spectrogram scope.

Resolution Bandwidth (RBW)

Resolution Bandwidth (RBW) is the minimum frequency bandwidth that can be resolved by the spectrum analyzer. By default, **RBW (Hz)** is set to Auto. In this mode, *RBW* is the ratio of the frequency span to 1024. In a two-sided spectrum, this value is $F_s/1024$, while in a one-sided spectrum, it is $(F_s/2)/1024$.

Using this value of *RBW*, the window length ($N_{samples}$) is computed iteratively using this relationship:

$$N_{samples} = \frac{\left(1 - \frac{Overlap}{100}\right) * NENBW * F_s}{RBW}$$

Overlap is the amount of overlap between the previous and current buffered data segments. *NENBW* is the equivalent noise bandwidth of the window. For more information on the details of the spectral estimation algorithm, see "Spectral Analysis" on page 7-68.

To distinguish between two frequencies in the display, the distance between the two frequencies must be at least RBW. In this example, the distance between the two peaks is 5000 Hz, which is greater than *RBW*. Hence, you can see the peaks distinctly.

Time Resolution

Time resolution is the distance between two spectral lines in the vertical axis. By default, **Time res (s)** is set to Auto. In this mode, the value of time resolution is 1/RBW s, which is the minimum attainable resolution. You can also specify the **Time res (s)** as a numeric value.

Conversion of Power in Watts to dBW and dBm

The spectrum analyzer provides three units to specify the power: `Watts`, `dBm`, and `dBW`. By default, **Units** is set to `dBm`.

Power in dBW is:

$$P_{dBW} = 10\log 10(\textit{power in watt} / 1\,\textit{watt})$$

Power in dBm is:

$$P_{dBm} = 10\log 10(\textit{power in watt} / 1\,\textit{milliwatt})$$

For a sine wave signal with an amplitude (A) of 1 V, the power of a one-sided spectrum in Watts is:

$$A^2 / 2$$

In this example, this power equals 0.5 W. Corresponding power in dBm is:

$$P_{dBm} = 10\log 10(\textit{power in watt} / 1\,\textit{milliwatt})$$
$$P_{dBm} = 10\log 10(0.5 / 10 \wedge -3)$$

Here, the power equals 26.9897 dBm. For this reason, the two tones at 5 kHz and 10 kHz in the spectrogram have a color corresponding to 26.9897 dBm.

For a white noise signal, the spectrum is flat for all frequencies. The spectrogram in this example shows a one sided spectrum in the range [0 Fs/2]. For a white noise signal with a variance of 1e-4, the power per unit bandwidth ($P_{unitbandwidth}$) is 1e-4. The total power in watts over the entire frequency range is:

$$P_{whitenoise} = P_{unitbandwidth} * \textit{number of frequency bins},$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{Fs / 2}{RBW} \right),$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{22050}{21.53} \right)$$

The number of frequency bins is the ratio of total bandwidth to RBW. For a one-sided spectrum, the total bandwidth is half the sampling rate. RBW in this example is 21.53 Hz. With all these values, total power of white noise is -39.87 dBm.

Scale Color Limits

When you run the model and do not see the spectrogram colors, click the **Scale Color Limits** button. This option autoscales the colors.

The spectrogram updates in real time. During simulation, if you change any of the tunable parameters in the model, the changes are effective immediately in the spectrogram.

More About

- "Estimate the Power Spectral Density in MATLAB" on page 7-23
- "Estimate the Power Spectral Density in Simulink" on page 7-37

Spectral Analysis

Spectral analysis is the process of estimating the power spectral density (PSD) of a signal from its time-domain representation. Spectral density characterizes the frequency content of a signal or a stochastic process. Intuitively, the spectrum decomposes the signal or the stochastic process into the different frequencies, and identifies periodicities. The most commonly used instrument for performing spectral analysis is the spectrum analyzer.

Spectral analysis is done based on the nonparametric methods and the parametric methods. Nonparametric methods are based on dividing the time-domain data into segments, applying Fourier transform on each segment, computing the squared-magnitude of the transform, and summing and averaging the transform. Nonparametric methods such as modified periodogram, Bartlett, Welch, and the Blackman-Tukey methods, are a variation of this approach. These methods are based on measured data and do not require prior knowledge about the data or the model. Parametric methods are model-based approaches. The model for generating the signal can be constructed with a number of parameters that can estimated from the observed data. From the model and estimated parameters, the algorithm computes the power density spectrum implied by the model.

The spectrum analyzer in DSP System Toolbox uses Welch's nonparametric method of averaging modified periodogram to estimate the power density spectrum of a streaming signal in real time.

Welch's Algorithm of Averaging Modified Periodogram

Welch's technique to reduce the variance of the periodogram breaks the time series into overlapping segments. This method computes a modified periodogram for each segment and then averages these estimates to produce the estimate of the power spectral density. Because the process is wide-sense stationary and Welch's method uses PSD estimates of different segments of the time series, the modified periodograms represent approximately uncorrelated estimates of the true PSD. The averaging reduces the variability.

The segments are multiplied by a window function, such as a Hann window, so that Welch's method amounts to averaging modified periodograms. Because the segments usually overlap, data values at the beginning and end of the segment tapered by the window in one segment, occur away from the ends of adjacent segments. The overlap guards against the loss of information caused by windowing. In the Spectrum Analyzer block, you can specify the window using the **Window** parameter.

The algorithm in the Spectrum Analyzer block consists of these steps:

1 The block buffers the input into N point data segments. Each data segment is split up into L overlapping data segments, each of length M, overlapping by D points. The data segments can be represented as:

$$x_i(n) = x(n + iD), \quad n = 0,1,...,M-1$$
$$i = 0,1,...,L-1$$

- If $D = M/2$, the overlap is 50%.
- If $D = 0$, the overlap is 0%.

The block uses the `RBW` or the `Window Length` setting in the **Spectrum Settings** pane to determine the data window length. Then, it partitions the input signal into a number of windowed data segments.

The spectrum analyzer requires a minimum number of samples ($N_{samples}$) to compute a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to the resolution bandwidth, RBW, by the following equation:

$$N_{samples} = \frac{\left(1 - \dfrac{Overlap}{100}\right) * NENBW * F_s}{RBW}.$$

- *Overlap*, the amount of overlap (%) between the previous and current buffered data segments, is specified through the **Overlap (%)** parameter in the **Window options** pane.
- *NENBW*, the normalized effective noise bandwidth of the window depends on the windowing method. This parameter is shown in the **Window options** pane.
- F_s is the sample rate of the input signal.

When in `RBW` mode, the window length required to compute one spectral update, N_{window}, is directly related to the resolution bandwidth and normalized effective noise bandwidth:

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

When in `Window length` mode, the window length is used as specified.

The number of input samples required to compute one spectral update, $N_{samples}$, is directly related to the window length and the amount of overlap:

$$N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$$

When you increase the overlap percentage, fewer new input samples are needed to compute a new spectral update. For example, the table shows the number of input samples required to compute one spectral update when the window length is 100.

Overlap	$N_{samples}$
0%	100
50%	50
80%	20

The normalized effective noise bandwidth, *NENBW*, is a window parameter determined by the window length, N_{window}, and the type of window used. If $w(n)$ denotes the vector of N_{window} window coefficients, then *NENBW* is:

$$NENBW = N_{window} \frac{\displaystyle\sum_{n=1}^{N_{window}} w^2(n)}{\left[\displaystyle\sum_{n=1}^{N_{window}} w(n)\right]^2}$$

When in RBW mode, you can set the resolution bandwidth using the value of the **RBW** parameter on the **Main options** pane. You must specify a value so that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:

$$\frac{span}{RBW} > 2$$

By default, the **RBW** parameter on the **Main options** pane is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value so that there are 1024 RBW intervals over the specified frequency span. Thus, when you set **RBW** to `Auto`,

RBW is calculated by: $RBW_{auto} = \dfrac{span}{1024}$

When in window length mode, you specify N_{window} and the resulting RBW is

$\dfrac{NENBW * Fs}{N_{window}}$.

2 Apply a window to each of the L overlapping data segments in the time domain. Most window functions afford more influence to the data at the center of the set than to the data at the edges, which represents a loss of information. To mitigate that loss, the individual data sets are commonly overlapped in time. For each windowed segment, compute the periodogram by computing the discrete Fourier transform. Then compute the squared magnitude of the result, and divide the result by M.

$$P_{xx}^i(f) = \frac{1}{MU} \left| \sum_{n=0}^{M-1} x_i(n)w(n)e^{-j2\pi fn} \right|^2, \qquad i = 0,1,...,L-1$$

where U is a normalization factor for the power in the window function and is given by

$$U = \frac{1}{M} \sum_{n=0}^{M-1} w^2(n)$$

.

You can specify the window using the **Window** parameter.

3 To determine the Welch power spectrum estimate, the Spectrum Analyzer block averages the result of the periodograms for the last L data segments. The averaging reduces the variance, compared to the original N point data segment.

$$P_{xx}^{W}(f) = \frac{1}{L} \sum_{i=0}^{L-1} P_{xx}^{i}(f)$$

L is specified through the **Averages** parameter in the **Trace options** pane.

4 The Spectrum Analyzer block computes the power spectral density using:

$$P_{xx}^{W}(f) = \frac{1}{L * F_s} \sum_{i=0}^{L-1} P_{xx}^{i}(f)$$

.

References

[1] Proakis, John G., and Dimitris G. Manolakis. *Digital Signal Processing*. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.

[2] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996.

More About

- "Estimate the Power Spectral Density in MATLAB" on page 7-23
- "Estimate the Power Spectral Density in Simulink" on page 7-37
- "Estimate the Transfer Function of an Unknown System" on page 7-52
- "View the Spectrogram Using Spectrum Analyzer" on page 7-62

Mathematics

Learn about statistics and linear algebra.

Statistics

Statistics Blocks

The Statistics library provides fundamental statistical operations such as minimum, maximum, mean, variance, and standard deviation. Most blocks in the Statistics library support two types of operations; basic and running.

The blocks listed below toggle between basic and running modes using the **Running** check box in the parameter dialog box:

- `Histogram`
- `Mean`
- `RMS`
- `Standard Deviation`
- `Variance`

An unselected **Running** check box means that the block is operating in basic mode, while a selected **Running** box means that the block is operating in running mode.

The Maximum and Minimum blocks are slightly different from the blocks above, and provide a **Mode** parameter in the block dialog box to select the type of operation. The `Value and Index`, `Value`, and `Index` options in the **Mode** menu all specify basic operation, in each case enabling a different set of output ports on the block. The `Running` option in the **Mode** menu selects running operation.

Basic Operations

A *basic operation* is one that processes each input independently of previous and subsequent inputs. For example, in basic mode (with `Value and Index` selected, for example) the Maximum block finds the maximum value in each column of the current input, and returns this result at the top output (`Val`). Each consecutive `Val` output

therefore has the same number of columns as the input, but only one row. Furthermore, the values in a given output only depend on the values in the corresponding input. The block repeats this operation for each successive input.

This type of operation is exactly equivalent to the MATLAB command

```
val = max(u)      % Equivalent MATLAB code
```

which computes the maximum of each column in input u.

The next section is an example of a basic statistical operation.

Create a Sliding Window

You can use the basic statistics operations in conjunction with the Buffer block to implement basic sliding window statistics operations. A *sliding window* is like a stencil that you move along a data stream, exposing only a set number of data points at one time.

For example, you may want to process data in 128-sample frames, moving the window along by one sample point for each operation. One way to implement such a sliding window is shown in the following ex_mean_tut model.
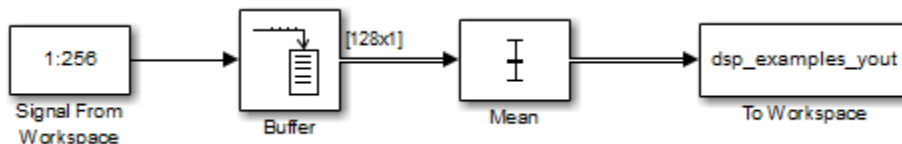
The Buffer block's **Buffer size** (M_o) parameter determines the size of the window. The **Buffer overlap** (L) parameter defines the "slide factor" for the window. At each sample instant, the window slides by M_o-L points. The **Buffer overlap** is often M_o-1, so that a new statistic is computed for every new signal sample.

Running Operations

A *running operation* is one that processes successive inputs, and computes a result that reflects both current and past inputs. In this mode, you must use the **Input processing** parameter to specify whether the block performs sample- or frame-based processing on the inputs. A reset port enables you to restart this tracking at any time. The running

statistic is computed for each input channel independently, so the block's output is the same size as the input.

For example, in running mode (`Running` selected from the **Mode** parameter) the Maximum block outputs a record of the input's maximum value over time.

The following figure illustrates how a Maximum block in running mode operates on a 3-by-2 matrix input, u, when the **Input processing** parameter is set to Columns as channels (frame based). The running maximum is reset at *t*=2 by an impulse to the block's optional Rst port.
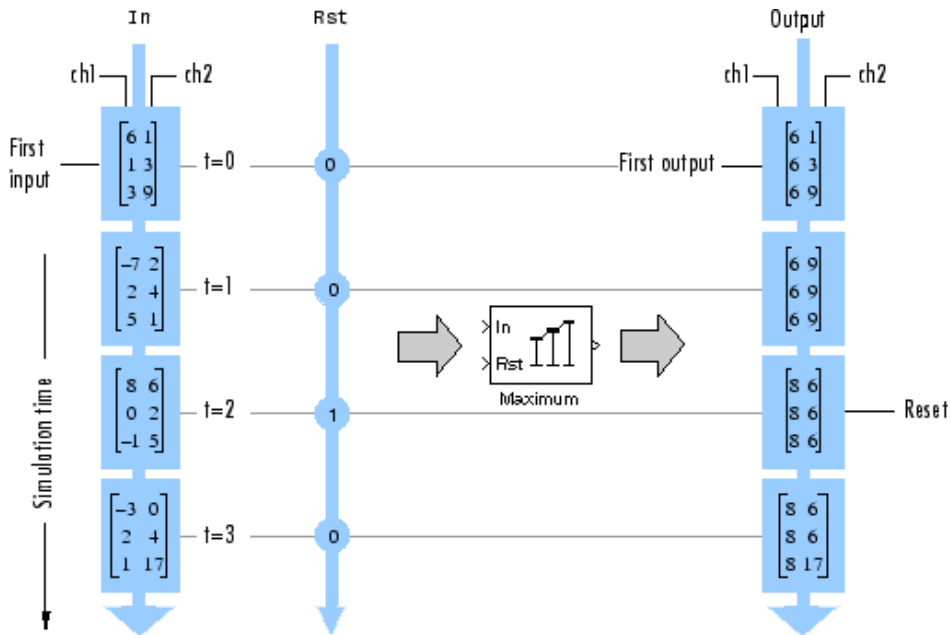
Linear Algebra and Least Squares

In this section...
"Linear Algebra Blocks" on page 8-6
"Linear System Solvers" on page 8-6
"Matrix Factorizations" on page 8-8
"Matrix Inverses" on page 8-9

Linear Algebra Blocks

The Matrices and Linear Algebra library provides three large sublibraries containing blocks for linear algebra; Linear System Solvers, Matrix Factorizations, and Matrix Inverses. A fourth library, Matrix Operations, provides other essential blocks for working with matrices.

Linear System Solvers

The Linear System Solvers library provides the following blocks for solving the system of linear equations $AX = B$:

- Autocorrelation LPC
- Cholesky Solver
- Forward Substitution
- LDL Solver
- Levinson-Durbin
- LU Solver
- QR Solver
- SVD Solver

Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Solver block is particularly adapted for a square Hermitian positive definite matrix A, whereas the Backward Substitution block is particularly suited for an upper triangular matrix A.

Solve AX=B Using the LU Solver Block

In the following ex_lusolver_tut model, the LU Solver block solves the equation $Ax = b$, where

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix}$$

and finds x to be the vector `[-2 0 1]'`.
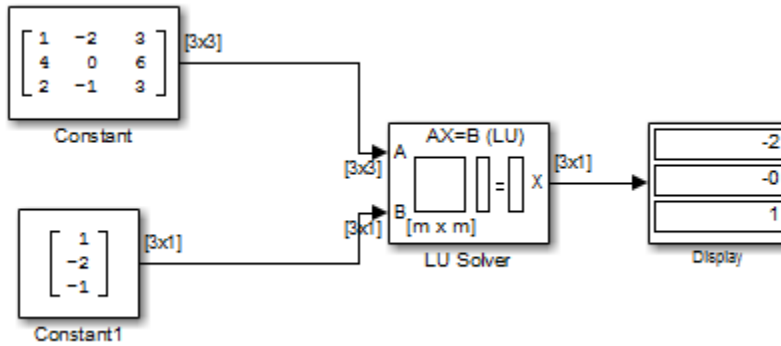
You can verify the solution by using the Matrix Multiply block to perform the multiplication Ax, as shown in the following ex_matrixmultiply_tut1 model.
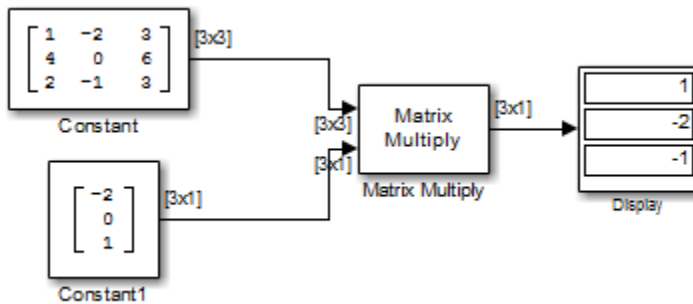
Matrix Factorizations

The Matrix Factorizations library provides the following blocks for factoring various kinds of matrices:

- `Cholesky Factorization`
- `LDL Factorization`
- `LU Factorization`
- `QR Factorization`
- `Singular Value Decomposition`

Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Factorization block is particularly suited to factoring a Hermitian positive definite matrix into triangular components, whereas the QR Factorization is particularly suited to factoring a rectangular matrix into unitary and upper triangular components.

Factor a Matrix into Upper and Lower Submatrices Using the LU Factorization Block

In the following ex_lufactorization_tut model, the LU Factorization block factors a matrix A_p into upper and lower triangular submatrices U and L, where A_p is row equivalent to input matrix A, where

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$
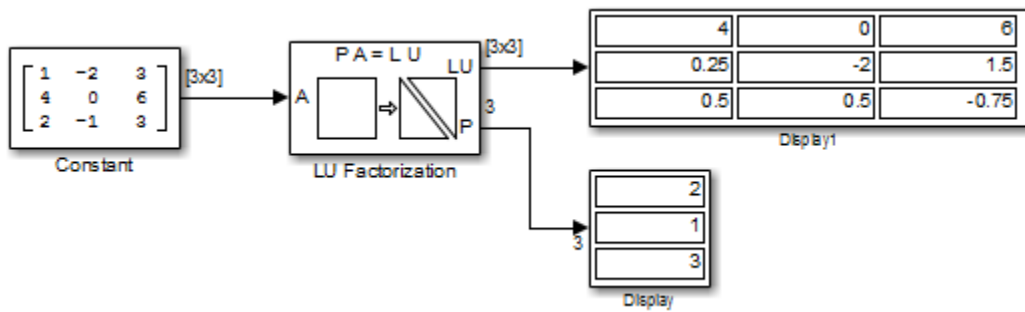
The lower output of the LU Factorization, P, is the permutation index vector, which indicates that the factored matrix A_p is generated from A by interchanging the first and second rows.

$$\mathbf{A}_p = \begin{bmatrix} 4 & 0 & 6 \\ 1 & -2 & 3 \\ 2 & -1 & 3 \end{bmatrix}$$

The upper output of the LU Factorization, LU, is a composite matrix containing the two submatrix factors, U and L, whose product LU is equal to A_p.

$$\mathbf{U} = \begin{bmatrix} 4 & 0 & 6 \\ 0 & -2 & 1.5 \\ 0 & 0 & -0.75 \end{bmatrix} \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0.25 & 1 & 0 \\ 0.5 & 0.5 & 1 \end{bmatrix}$$

You can check that $LU = A_p$ with the Matrix Multiply block, as shown in the following ex_matrixmultiply_tut2 model.
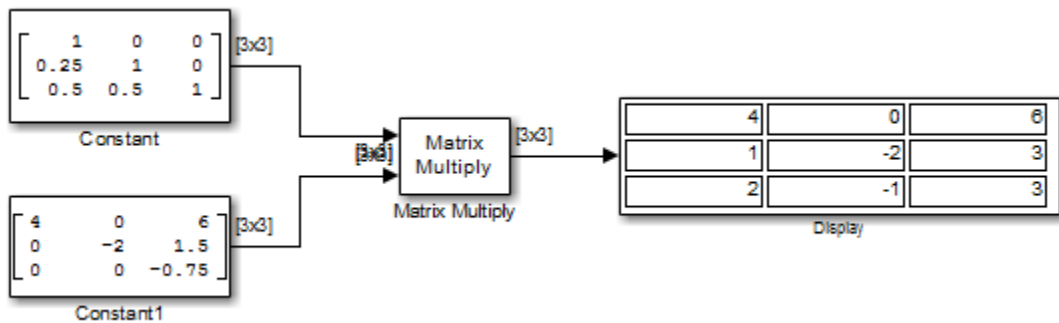
Matrix Inverses

The Matrix Inverses library provides the following blocks for inverting various kinds of matrices:

*   `Cholesky Inverse`
*   `LDL Inverse`

- LU Inverse
- Pseudoinverse

Find the Inverse of a Matrix Using the LU Inverse Block

In the following ex_luinverse_tut model, the LU Inverse block computes the inverse of input matrix A, where

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$

and then forms the product $A^{-1}A$, which yields the identity matrix of order 3, as expected.
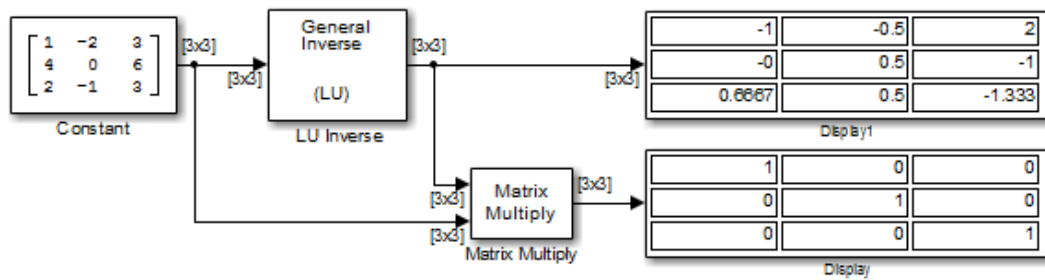
As shown above, the computed inverse is

$$\mathbf{A}^{-1} = \begin{bmatrix} -1 & -0.5 & 2 \\ 0 & 0.5 & -1 \\ 0.6667 & 0.5 & -1.333 \end{bmatrix}$$

9

Fixed-Point Design

Learn about fixed-point data types and how to convert floating-point models to fixed point.

Fixed-Point Signal Processing

In this section...
"Fixed-Point Features" on page 9-2
"Benefits of Fixed-Point Hardware" on page 9-2
"Benefits of Fixed-Point Design with System Toolboxes Software" on page 9-3

Note: To take full advantage of fixed-point support in System Toolbox software, you must install Fixed-Point Designer software.

Fixed-Point Features

Many of the blocks in this product have fixed-point support, so you can design signal processing systems that use fixed-point arithmetic. Fixed-point support in DSP System Toolbox software includes

- Signed two's complement and unsigned fixed-point data types
- Word lengths from 2 to 128 bits in simulation
- Word lengths from 2 to the size of a `long` on the Simulink Coder C code-generation target
- Overflow handling and rounding methods
- C code generation for deployment on a fixed-point embedded processor, with Simulink Coder code generation software. The generated code uses all allowed data types supported by the embedded target, and automatically includes all necessary shift and scaling operations

Benefits of Fixed-Point Hardware

There are both benefits and trade-offs to using fixed-point hardware rather than floating-point hardware for signal processing development. Many signal processing applications require low-power and cost-effective circuitry, which makes fixed-point hardware a natural choice. Fixed-point hardware tends to be simpler and smaller. As a result, these units require less power and cost less to produce than floating-point circuitry.

Floating-point hardware is usually larger because it demands functionality and ease of development. Floating-point hardware can accurately represent real-world numbers, and

its large dynamic range reduces the risk of overflow, quantization errors, and the need for scaling. In contrast, the smaller dynamic range of fixed-point hardware that allows for low-power, inexpensive units brings the possibility of these problems. Therefore, fixed-point development must minimize the negative effects of these factors, while exploiting the benefits of fixed-point hardware; cost- and size-effective units, less power and memory usage, and fast real-time processing.

Benefits of Fixed-Point Design with System Toolboxes Software

Simulating your fixed-point development choices before implementing them in hardware saves time and money. The built-in fixed-point operations provided by the System Toolboxes software save time in simulation and allow you to generate code automatically.

This software allows you to easily run multiple simulations with different word length, scaling, overflow handling, and rounding method choices to see the consequences of various fixed-point designs before committing to hardware. The traditional risks of fixed-point development, such as quantization errors and overflow, can be simulated and mitigated in software before going to hardware.

Fixed-point C code generation with System Toolbox software and Simulink Coder code generation software produces code ready for execution on a fixed-point processor. All the choices you make in simulation in terms of scaling, overflow handling, and rounding methods are automatically optimized in the generated code, without necessitating time-consuming and costly hand-optimized code.

Fixed-Point Concepts and Terminology

In this section...
"Fixed-Point Data Types" on page 9-4
"Scaling" on page 9-5
"Precision and Range" on page 9-6

Note: The "Glossary" defines much of the vocabulary used in these sections. For more information on these subjects, see the "Fixed-Point Designer" documentation.

Fixed-Point Data Types

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). How hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

Binary numbers are represented as either fixed-point or floating-point data types. In this section, we discuss many terms and concepts relating to fixed-point numbers, data types, and mathematics.

A fixed-point data type is characterized by the word length in bits, the position of the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a generalized fixed-point number (either signed or unsigned) is shown below:
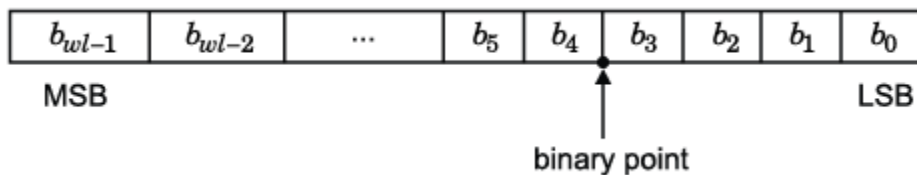
where

- b_i is the ith binary digit.

- wl is the word length in bits.
- b_{wl-1} is the location of the most significant, or highest, bit (MSB).
- b_0 is the location of the least significant, or lowest, bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example, therefore, the number is said to have four fractional bits, or a fraction length of four.

Fixed-point data types can be either signed or unsigned. Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude
- One's complement
- Two's complement

Two's complement is the most common representation of signed fixed-point numbers and is used by System Toolbox software. See "Two's Complement" on page 9-10 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = slope\ adjustment \times 2^{exponent}$$

The integer is sometimes called the *stored integer*. This is the raw binary number, in which the binary point assumed to be at the far right of the word. In System Toolboxes, the negative of the exponent is often referred to as the *fraction length*.

The slope and bias together represent the scaling of the fixed-point number. In a number with zero bias, only the slope affects the scaling. A fixed-point number that is only scaled by binary point position is equivalent to a number in the Fixed-Point Designer [Slope Bias] representation that has a bias equal to zero and a slope adjustment equal to one. This is referred to as binary point-only scaling or power-of-two scaling:

$$real\text{-}world\ value = 2^{exponent} \times integer$$

or

$$real\text{-}world\ value = 2^{-fraction\ length} \times integer$$

In System Toolbox software, you can define a fixed-point data type and scaling for the output or the parameters of many blocks by specifying the word length and fraction length of the quantity. The word length and fraction length define the whole of the data type and scaling information for binary-point only signals.

All System Toolbox blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [Slope Bias] scaling.

Precision and Range

You must pay attention to the precision and range of the fixed-point data types and scalings you choose for the blocks in your simulations, in order to know whether rounding methods will be invoked or if overflows will occur.

Range

The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl, scaling S, and bias B is illustrated below:
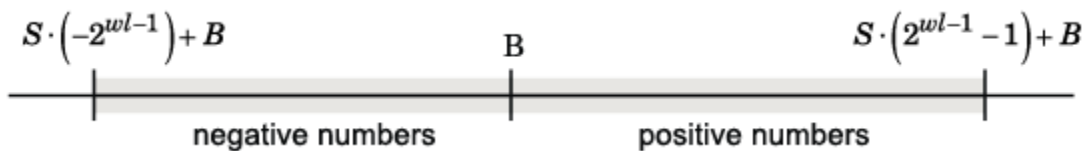
For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl}.

For example, in two's complement, negative numbers must be represented as well as zero, so the maximum value is 2^{wl-1}. Because there is only one representation for zero, there are an unequal number of positive and negative numbers. This means there is a representation for -2^{wl-1} but not for 2^{wl-1}:

For slope = 1 and bias = 0:
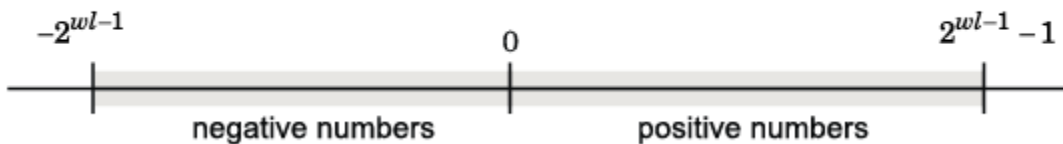
Overflow Handling

Because a fixed-point data type represents numbers within a finite range, overflows can occur if the result of an operation is larger or smaller than the numbers in that range.

System Toolbox software does not allow you to add guard bits to a data type on-the-fly in order to avoid overflows. Any guard bits must be allocated upon model initialization. However, the software does allow you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. See "Modulo Arithmetic" on page 9-9 for more information.

Precision

The precision of a fixed-point number is the difference between successive values representable by its data type and scaling, which is equal to the value of its least significant bit. The value of the least significant bit, and therefore the precision of the number, is determined by the number of fractional bits. A fixed-point value can be represented to within half of the precision of its data type and scaling.

For example, a fixed-point representation with four bits to the right of the binary point has a precision of 2^{-4} or 0.0625, which is the value of its least significant bit. Any number within the range of this data type and scaling can be represented to within $(2^{-4})/2$ or 0.03125, which is half the precision. This is an example of representing a number with finite precision.

Rounding Modes

When you represent numbers with finite precision, not every number in the available range can be represented exactly. If a number cannot be represented exactly by the specified data type and scaling, it is *rounded* to a representable number. Although precision is always lost in the rounding operation, the cost of the operation and the

amount of bias that is introduced depends on the rounding mode itself. To provide you with greater flexibility in the trade-off between cost and bias, DSP System Toolbox software currently supports the following rounding modes:

- `Ceiling` rounds the result of a calculation to the closest representable number in the direction of positive infinity.

- `Convergent` rounds the result of a calculation to the closest representable number. In the case of a tie, `Convergent` rounds to the nearest even number. This is the least biased rounding mode provided by the toolbox.

- `Floor`, which is equivalent to truncation, rounds the result of a calculation to the closest representable number in the direction of negative infinity.

- `Nearest` rounds the result of a calculation to the closest representable number. In the case of a tie, `Nearest` rounds to the closest representable number in the direction of positive infinity.

- `Round` rounds the result of a calculation to the closest representable number. In the case of a tie, `Round` rounds positive numbers to the closest representable number in the direction of positive infinity, and rounds negative numbers to the closest representable number in the direction of negative infinity.

- `Simplest` rounds the result of a calculation using the rounding mode (`Floor` or `Zero`) that adds the least amount of extra rounding code to your generated code. For more information, see "Rounding Mode: Simplest" in the Fixed-Point Designer documentation.

- `Zero` rounds the result of a calculation to the closest representable number in the direction of zero.

To learn more about each of these rounding modes, see "Rounding" in the Fixed-Point Designer documentation.

For a direct comparison of the rounding modes, see "Choosing a Rounding Method" in the Fixed-Point Designer documentation.

Arithmetic Operations

Note: These sections will help you understand what data type and scaling choices result in overflows or a loss of precision.

Modulo Arithmetic

Binary math is based on modulo arithmetic. Modulo arithmetic uses only a finite set of numbers, wrapping the results of any calculations that fall outside the given set back into the set.

For example, the common everyday clock uses modulo 12 arithmetic. Numbers in this system can only be 1 through 12. Therefore, in the "clock" system, 9 plus 9 equals 6. This can be more easily visualized as a number circle:

9 ...

... plus 9 more ...

... equals 6.
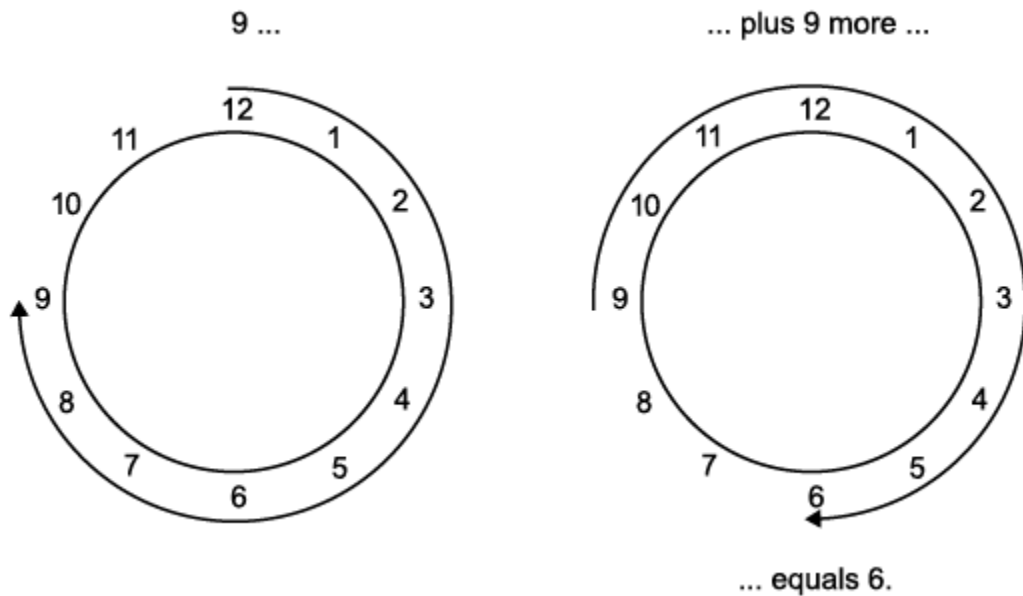
Similarly, binary math can only use the numbers 0 and 1, and any arithmetic results that fall outside this range are wrapped "around the circle" to either 0 or 1.

Two's Complement

Two's complement is a way to interpret a binary number. In two's complement, positive numbers always start with a 0 and negative numbers always start with a 1. If the leading bit of a two's complement number is 0, the value is obtained by calculating the standard binary value of the number. If the leading bit of a two's complement number is 1, the value is obtained by assuming that the leftmost bit is negative, and then calculating the binary value of the number. For example,

$$01 = (0 + 2^0) = 1$$
$$11 = ((-2^1) + (2^0)) = (-2 + 1) = -1$$

To compute the negative of a binary number using two's complement,

1 Take the one's complement, or "flip the bits."

2 Add a 1 using binary math.

3 Discard any bits carried beyond the original word length.

For example, consider taking the negative of 11010 (-6). First, take the one's complement of the number, or flip the bits:

11010 → 00101

Next, add a 1, wrapping all numbers to 0 or 1:

```
  00101
    +1
  00110 (6)
```

Addition and Subtraction

The addition of fixed-point numbers requires that the binary points of the addends be aligned. The addition is then performed using binary arithmetic so that no number other than 0 or 1 is used.

For example, consider the addition of 010010.1 (18.5) with 0110.110 (6.75):

```
   010010.1    (18.5)
  +0110.110    (6.75)
  011001.010 (25.25)
```

Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

Most fixed-point DSP System Toolbox blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further

shifting is necessary during the addition to line up the binary points. See "Casts" on page 9-14 for more information.

Multiplication

The multiplication of two's complement fixed-point numbers is directly analogous to regular decimal multiplication, with the exception that the intermediate results must be sign extended so that their left sides align before you add them together.

For example, consider the multiplication of 10.11 (-1.25) with 011 (3):

The extra 1 is the result of necessary sign extension.

The number of fractional bits of the result is the sum of the number of fractional bits of the factors.

Multiplication Data Types

The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. See individual reference pages to determine whether a particular block accepts complex fixed-point inputs.

In most cases, you can set the data types used during multiplication in the block mask. See Accumulator Parameters, Intermediate Product Parameters, Product Output Parameters, and Output Parameters. These data types are defined in "Casts" on page 9-14.

Note: The following diagrams show the use of fixed-point data types in multiplication in System Toolbox software. They do not represent actual subsystems used by the software to perform multiplication.

Real-Real Multiplication

The following diagram shows the data types used in the multiplication of two real numbers in System Toolbox software. The software returns the output of this operation in the product output data type, as the next figure shows.
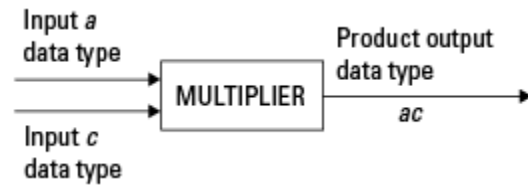
Real-Complex Multiplication

The following diagram shows the data types used in the multiplication of a real and a complex fixed-point number in System Toolbox software. Real-complex and complex-real multiplication are equivalent. The software returns the output of this operation in the product output data type, as the next figure shows.
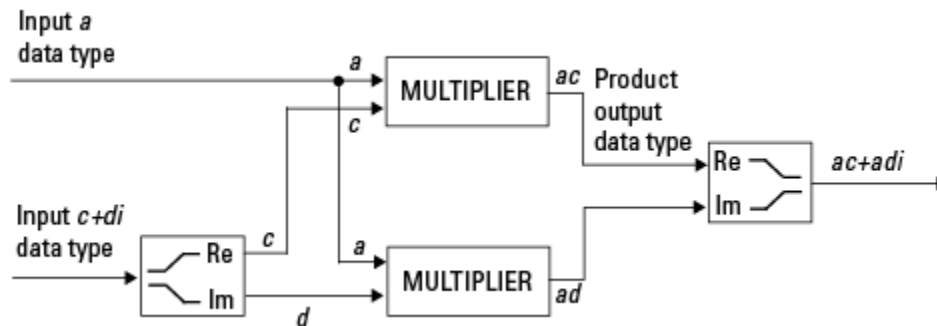
Complex-Complex Multiplication

The following diagram shows the multiplication of two complex fixed-point numbers in System Toolbox software. Note that the software returns the output of this operation in the accumulator output data type, as the next figure shows.
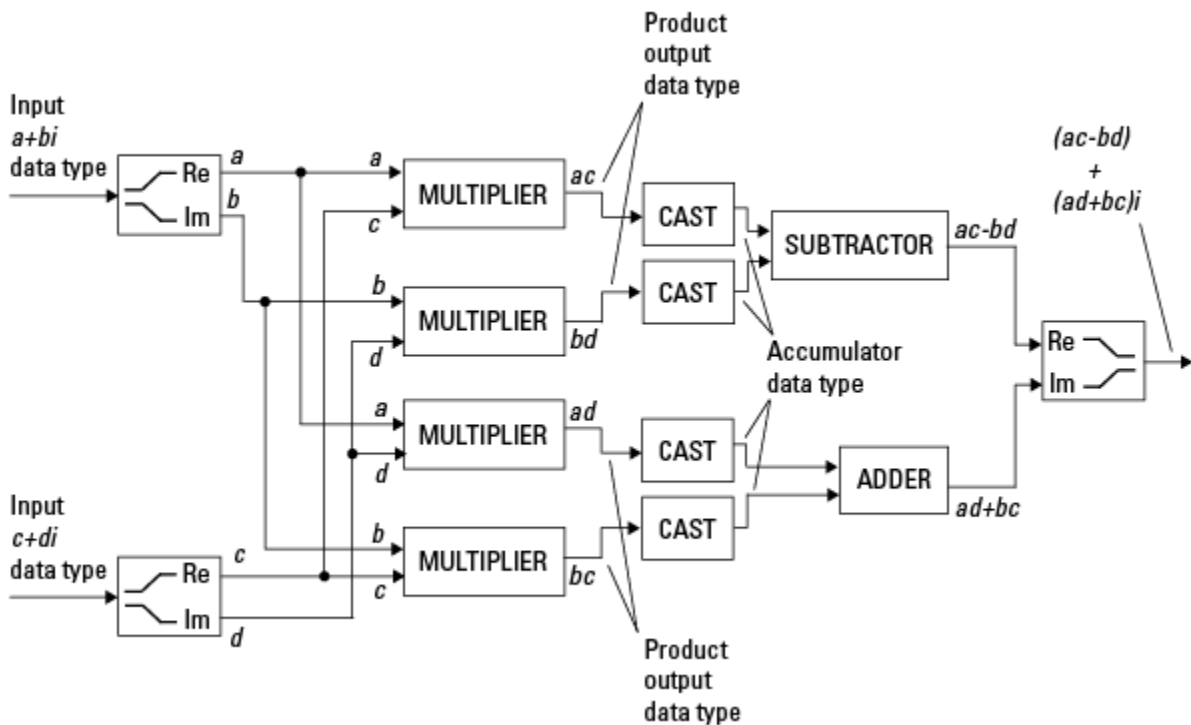
System Toolbox blocks cast to the accumulator data type before performing addition or subtraction operations. In the preceding diagram, this is equivalent to the C code

```
acc=ac;
acc-=bd;
```

for the subtractor, and

```
acc=ad;
acc+=bc;
```

for the adder, where *acc* is the accumulator.

Casts

Many fixed-point System Toolbox blocks that perform arithmetic operations allow you to specify the accumulator, intermediate product, and product output data types, as

applicable, as well as the output data type of the block. This section gives an overview of the casts to these data types, so that you can tell if the data types you select will invoke sign extension, padding with zeros, rounding, and/or overflow.

Casts to the Accumulator Data Type

For most fixed-point System Toolbox blocks that perform addition or subtraction, the operands are first cast to an accumulator data type. Most of the time, you can specify the accumulator data type on the block mask. See Accumulator Parameters. Since the addends are both cast to the same accumulator data type before they are added together, no extra shift is necessary to insure that their binary points align. The result of the addition remains in the accumulator data type, with the possibility of overflow.

Casts to the Intermediate Product or Product Output Data Type

For System Toolbox blocks that perform multiplication, the output of the multiplier is placed into a product output data type. Blocks that then feed the product output back into the multiplier might first cast it to an intermediate product data type. Most of the time, you can specify these data types on the block mask. See Intermediate Product Parameters and Product Output Parameters.

Casts to the Output Data Type

Many fixed-point System Toolbox blocks allow you to specify the data type and scaling of the block output on the mask. Remember that the software does not allow mixed types on the input and output ports of its blocks. Therefore, if you would like to specify a fixed-point output data type and scaling for a System Toolbox block that supports fixed-point data types, you must feed the input port of that block with a fixed-point signal. The final cast made by a fixed-point System Toolbox block is to the output data type of the block.

Note that although you can not mix fixed-point and floating-point signals on the input and output ports of blocks, you can have fixed-point signals with different word and fraction lengths on the ports of blocks that support fixed-point signals.

Casting Examples

It is important to keep in mind the ramifications of each cast when selecting these intermediate data types, as well as any other intermediate fixed-point data types that are allowed by a particular block. Depending upon the data types you select, overflow and/or rounding might occur. The following two examples demonstrate cases where overflow and rounding can occur.

Cast from a Shorter Data Type to a Longer Data Type

Consider the cast of a nonzero number, represented by a four-bit data type with two fractional bits, to an eight-bit data type with seven fractional bits:

source

The source bits must be shifted up to match the binary point position of the destination data type.

destination

This bit from the source data type "falls off" the high end with the shift up. Overflow might occur. The result will saturate or wrap.

These bits of the destination data type are padded with 0's or 1's.
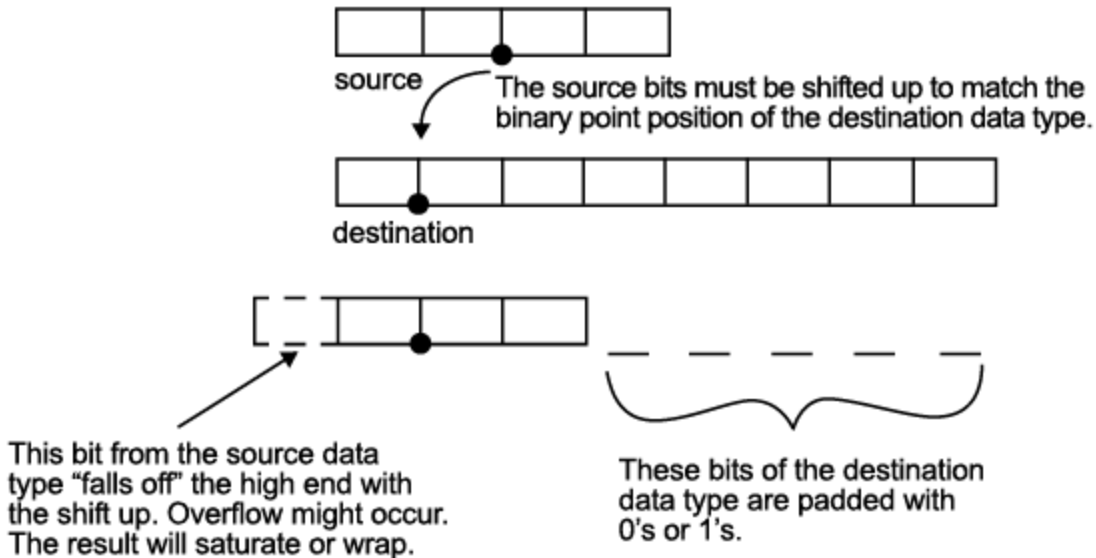
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 0's or 1's:

- If overflow does not occur, the empty bits are padded with 0's.

- If wrapping occurs, the empty bits are padded with 0's.

- If saturation occurs,

  - The empty bits of a positive number are padded with 1's.

  - The empty bits of a negative number are padded with 0's.

You can see that even with a cast from a shorter data type to a longer data type, overflow might still occur. This can happen when the integer length of the source data type (in this case two) is longer than the integer length of the destination data type (in this case

one). Similarly, rounding might be necessary even when casting from a shorter data type to a longer data type, if the destination data type and scaling has fewer fractional bits than the source.

Cast from a Longer Data Type to a Shorter Data Type

Consider the cast of a nonzero number, represented by an eight-bit data type with seven fractional bits, to a four-bit data type with two fractional bits:
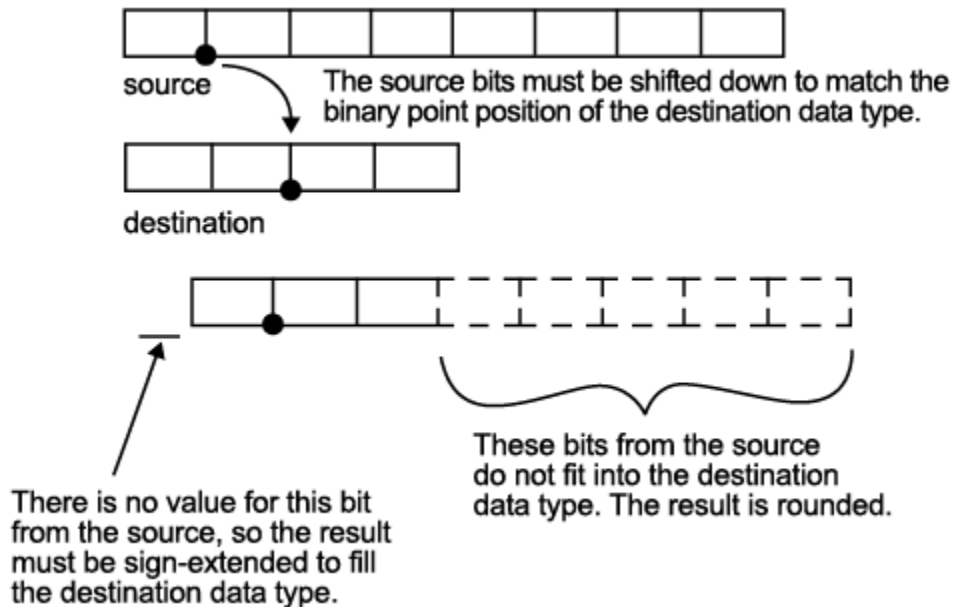
As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so the result is sign extended to fill the integer portion of the destination data type. The bottom five bits of the source do not fit into the fraction length of the destination. Therefore, precision can be lost as the result is rounded.

In this case, even though the cast is from a longer data type to a shorter data type, all the integer bits are maintained. Conversely, full precision can be maintained even if you cast to a shorter data type, as long as the fraction length of the destination data type is the same length or longer than the fraction length of the source data type. In that case, however, bits are lost from the high end of the result and overflow might occur.

The worst case occurs when both the integer length and the fraction length of the destination data type are shorter than those of the source data type and scaling. In that case, both overflow and a loss of precision can occur.

Fixed-Point Support for MATLAB System Objects in DSP System Toolbox

In this section...
"Get Information About Fixed-Point System Objects " on page 9-19
"Set System Object Fixed-Point Properties" on page 9-23
"Full Precision for Fixed-Point System Objects" on page 9-24

Get Information About Fixed-Point System Objects

System objects that support fixed-point data processing have fixed-point properties. When you display the properties of a System object, click `Show all properties` at the end of the property list to display the fixed-point properties for that object. You can also display the fixed-point properties for a particular object by typing `dsp.<ObjectName>.helpFixedPoint` at the command line.

DSP System Toolbox System Objects That Support Fixed Point

Object	Description
Sources	
dsp.SignalSource	Import a variable from the MATLAB workspace
dsp.SineWave	Generate discrete sine wave
Sinks	
dsp.ArrayPlot	Display vectors or arrays
dsp.AudioFileWriter	Write audio samples to audio file
dsp.SignalSink	Log MATLAB simulation data
dsp.SpectrumAnalyzer	Display frequency spectrum of time-domain signals
dsp.TimeScope	Display time-domain signals
Adaptive Filters	
dsp.LMSFilter	Compute output, error, and weights using LMS adaptive algorithm
Filter Designs	
dsp.CICCompensationDecimator	Compensate for CIC filter using a FIR decimator

Object	Description
dsp.CICCompensationInterpolator	Compensate for CIC filter using a FIR interpolator
dsp.Differentiator	Direct form FIR full band differentiator filter
dsp.FIRHalfbandDecimator	Halfband decimator
dsp.FIRHalfbandInterpolator	Halfband interpolator
dsp.HighpassFilter	FIR or IIR highpass filter
dsp.LowpassFilter	FIR or IIR lowpass filter
Filter Implementations	
dsp.AllpoleFilter	IIR Filter with no zeros
dsp.BiquadFilter	Model biquadratic IIR (SOS) filters
dsp.DigitalFilter	Filter each channel of input over time using discrete-time filter implementations
dsp.FIRFilter	Static or time-varying FIR filter
dsp.IIRFilter	Infinite Impulse Response (IIR) filter
Multirate Filters	
dsp.CICDecimator	Decimate inputs using a Cascaded Integrator-Comb (CIC) filter
dsp.CICInterpolator	Interpolate inputs using a Cascaded Integrator-Comb (CIC) filter
dsp.FIRDecimator	Filter and downsample input signals
dsp.FIRInterpolator	Upsample and filter input signals
dsp.FIRRateConverter	Upsample, filter, and downsample input signals
dsp.HDLFIRRateConverter	Upsample, filter, and downsample—optimized for HDL code generation
dsp.SubbandAnalysisFilter	Decompose signal into high-frequency and low-frequency subbands
dsp.SubbandSynthesisFilter	Reconstruct a signal from high-frequency and low-frequency subbands
Linear Prediction	
dsp.LevinsonSolver	Solve linear system of equations using Levinson-Durbin recursion

Object	Description
Transforms	
dsp.DCT	Compute discrete cosine transform (DCT) of input
dsp.FFT	Compute fast Fourier transform (FFT) of input
dsp.HDLFFT	Compute fast Fourier transform (FFT) of input — optimized for HDL Code generation
dsp.HDLIFFT	Compute inverse fast Fourier transform (IFFT) of input — optimized for HDL Code generation
dsp.IDCT	Compute inverse discrete cosine transform (IDCT) of input
dsp.IFFT	Compute inverse fast Fourier transform (IFFT) of input
Statistics	
dsp.Autocorrelator	Compute autocorrelation of vector inputs
dsp.Crosscorrelator	Compute cross-correlation of two inputs
dsp.Histogram	Output histogram of an input or sequence of inputs
dsp.Maximum	Compute maximum value in input
dsp.Mean	Compute average or mean value in input
dsp.Median	Compute median value in input
dsp.Minimum	Compute minimum value in input
dsp.Variance	Compute variance of input or sequence of inputs
Quantizers	
dsp.ScalarQuantizerDecoder	Convert each index value into quantized output value
dsp.ScalarQuantizerEncoder	Perform scalar quantization encoding
dsp.VectorQuantizerDecoder	Find vector quantizer codeword for given index value
dsp.VectorQuantizerEncoder	Perform vector quantization encoding
Signal Operations	
dsp.Convolver	Compute convolution of two inputs

Object	Description
dsp.DCBlocker	Remove DC component
dsp.Delay	Delay input by specified number of samples or frames
dsp.DigitalDownConverter	Translate digital signal from Intermediate Frequency (IF) band to baseband and decimate it
dsp.DigitalUpConverter	Interpolate digital signal and translate it from baseband to Intermediate Frequency (IF) band
dsp.FarrowRateConverter	Polynomial sample rate converter with arbitrary conversion factor
dsp.HDLNCO	Generate real or complex sinusoidal signals — optimized for HDL code generation
dsp.NCO	Generate real or complex sinusoidal signals
dsp.PeakFinder	Determine extrema (maxima or minima) in input signal
dsp.VariableFractionalDelay	Delay input by time-varying fractional number of sample periods
dsp.VariableIntegerDelay	Delay input by time-varying integer number of sample periods
dsp.Window	Window object
dsp.ZeroCrossingDetector	Zero crossing detector
Math Operations	
dsp.CumulativeProduct	Compute cumulative product of channel, column, or row elements
dsp.CumulativeSum	Compute cumulative sum of channel, column, or row elements
dsp.HDLComplexToMagnitudeAngle	Compute magnitude and phase angle of complex signal—optimized for HDL code generation
dsp.Normalizer	Normalize input
Matrix Operations	
dsp.ArrayVectorAdder	Add vector to array along specified dimension
dsp.ArrayVectorDivider	Divide array by vector along specified dimension

Object	Description
dsp.ArrayVectorMultiplier	Multiply array by vector along specified dimension
dsp.ArrayVectorSubtractor	Subtract vector from array along specified dimension
Matrix Factorizations	
dsp.LDLFactor	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
dsp.LUFactor	Factor square matrix into lower and upper triangular matrices
Linear System Solvers	
dsp.LowerTriangularSolver	Solve LX = B for X when L is lower triangular matrix
dsp.UpperTriangularSolver	Solve UX = B for X when U is upper triangular matrix
Switches and Counters	
dsp.Counter	Count up or down through specified range of numbers
Buffers	
dsp.Buffer	Buffer an input signal

Set System Object Fixed-Point Properties

Several properties affect the fixed-point data processing used by a System object. Objects perform fixed-point processing and use the current fixed-point property settings when they receive fixed-point input.

You change the values of fixed-point properties in the same way as you change any System object property value. See "Component Properties". You also use the Fixed-Point Designer `numerictype` object to specify the desired data type as fixed point, the signedness, and the word- and fraction-lengths. System objects support these values of `DataTypeMode`: `Boolean`, `Double`, `Single`, and `Fixed-point: binary point scaling`.

In the same way as for blocks, the data type properties of many System objects can set the appropriate word lengths and scalings automatically by using full precision. System

objects assume that the target specified on the Configuration Parameters Hardware Implementation target is `ASIC/FPGA`.

If you have not set the property that activates a dependent property and you attempt to change that dependent property, a warning message displays. For example, for the`dsp.FFT` object, before you set `CustomOutputDataType` to `numerictype(1,32,30)`,set `OutputDataType` to `'Custom'`.

Note: System objects do not support fixed-point word lengths greater than 128 bits.

For any System object provided in the Toolbox, the fimath settings for any fimath attached to a fi input or a fi property are ignored. Outputs from a System object never have an attached fimath.

Full Precision for Fixed-Point System Objects

`FullPrecisionOverride` is a convenience property that, when you set to `true`, automatically sets the appropriate properties for an object to use full-precision to process fixed-point input. For System objects, full precision, fixed-point operation refers to growing just enough additional bits to compute the ideal full precision result. This operation has no minimum or maximum range overflow nor any precision loss due to rounding or underflow. It is also independent of any hardware-specific settings. The data types chosen are based only on known data type ranges and not on actual numeric values. Unlike full precision for `dfilt` objects, full precision for System objects does not optimize coefficient values.

When you set the `FullPrecisionOverride` property to `true`, the other fixed-point properties it controls no longer apply and any of their non-default values are ignored. These properties are also hidden. To specify individual fixed-point properties, first set `FullPrecisionOverride` to `false`.

More About

Fixed-Point Support for Simulink blocks in DSP System Toolbox

This section lists the Simulink blocks in DSP System Toolbox which support fixed-point operations. This information is also available in the Simulink block data type support table for DSP System Toolbox. To access this table, type the command below in the MATLAB command prompt.

```
showsignalblockdatatypetable
```

Block	Description
Sources	
Constant	Generate constant value
Discrete Impulse	Generate discrete impulse
Identity Matrix	Generate matrix with ones on main diagonal and zeros elsewhere
NCO	Generate real or complex sinusoidal signals
NCO HDL Optimized	Generate real or complex sinusoidal signals —optimized for HDL code generation
Signal From Workspace	Import signal from MATLAB workspace
Sine Wave	Generate continuous or discrete sine wave
Sinks	
Array Plot	Display vectors or arrays
Display	Show value of input
Matrix Viewer	Display matrices as color images
Spectrum Analyzer	Display frequency spectrum of time-domain signals
Time Scope	Display time-domain signals
To Workspace	Write data to MATLAB workspace
Triggered To Workspace	Write input sample to MATLAB workspace when triggered
Vector Scope	Display vector or matrix of time-domain, frequency-domain, or user-defined data
Waterfall	View vectors of data over time

Block	Description
Adaptive Filters	
LMS Filter	Compute output, error, and weights using LMS adaptive algorithm
Filter Designs	
CIC Compensation Decimator	Compensate for CIC filter using FIR decimator
CIC Compensation Interpolator	Compensate for CIC filter using FIR interpolator
Differentiator Filter	Direct form FIR full band differentiator filter
FIR Halfband Decimator	Decimate signal using polyphase FIR halfband filter
FIR Halfband Interpolator	Interpolate signal using polyphase FIR half band filter
Highpass Filter	Design FIR or IIR highpass filter
Lowpass Filter	Design FIR or IIR lowpass filter
Filter Implementations	
Allpole Filter	Model allpole filters
Biquad Filter	Model biquadratic IIR (SOS) filters
Discrete FIR Filter	Model FIR filters
Discrete Filter	Model Infinite Impulse Response (IIR) filters
Filter Realization Wizard	Construct filter realizations using digital filter blocks or Sum, Gain, and Delay blocks
Multirate Filters	
CIC Decimation	Decimate signal using Cascaded Integrator-Comb filter
CIC Interpolation	Interpolate signal using Cascaded Integrator-Comb filter
FIR Decimation	Filter and downsample input signals

Block	Description
`FIR Interpolation`	Upsample and filter input signals
`FIR Rate Conversion`	Upsample, filter, and downsample input signals
`FIR Rate Conversion HDL Optimized`	Upsample, filter, and downsample input signals—optimized for HDL code generation
`Two-Channel Analysis Subband Filter`	Decompose signal into high-frequency and low-frequency subbands
`Two-Channel Synthesis Subband Filter`	Reconstruct signal from high-frequency and low-frequency subbands
Linear Prediction	
`Levinson-Durbin`	Solve linear system of equations using Levinson-Durbin recursion
Transforms	
`DCT`	Discrete cosine transform (DCT) of input
`FFT`	Fast Fourier transform (FFT) of input
`FFT HDL Optimized`	Fast Fourier transform—optimized for HDL code generation
`IDCT`	Inverse discrete cosine transform (IDCT) of input
`IFFT`	Inverse fast Fourier transform (IFFT) of input
`IFFT HDL Optimized`	Inverse fast Fourier transform—optimized for HDL code generation
`Magnitude FFT`	Compute nonparametric estimate of spectrum using periodogram method
`Short-Time FFT`	Nonparametric estimate of spectrum using short-time, fast Fourier transform (FFT) method
Statistics	
`Autocorrelation`	Autocorrelation of vector or matrix input

Block	Description
Correlation	Cross-correlation of two inputs
Histogram	Generate histogram of input or sequence of inputs
Maximum	Find maximum values in input or sequence of inputs
Mean	Find mean value of input or sequence of inputs
Median	Find median value of input
Minimum	Find minimum values in input or sequence of inputs
Sort	Sort input elements by value
Variance	Compute variance of input or sequence of inputs
Quantizers	
Scalar Quantizer Decoder	Convert each index value into quantized output value
Scalar Quantizer Encoder	Encode each input value by associating it with index value of quantization region
Vector Quantizer Decoder	Find vector quantizer codeword that corresponds to given, zero-based index value
Vector Quantizer Encoder	For given input, find index of nearest codeword based on Euclidean or weighted Euclidean distance measure
Signal Operations	
Constant Ramp	Generate ramp signal with length based on input dimensions
Convolution	Convolution of two inputs
DC Blocker	lock DC component
Digital Down-Converter	Translate digital signal from Intermediate Frequency (IF) band to baseband and decimate it

Block	Description
Digital Up-Converter	Interpolate digital signal and translate it from baseband to Intermediate Frequency (IF) band
Downsample	Resample input at lower rate by deleting samples
Farrow Rate Converter	Polynomial sample-rate converter with arbitrary conversion factor
NCO	Generate real or complex sinusoidal signals
NCO HDL Optimized	Generate real or complex sinusoidal signals —optimized for HDL code generation
Offset	Truncate vectors by removing or keeping beginning or ending values
Pad	Pad or truncate specified dimension(s)
Peak Finder	Determine whether each value of input signal is local minimum or maximum
Repeat	Resample input at higher rate by repeating values
Sample and Hold	Sample and hold input signal
Triggered Signal From Workspace	Import signal samples from MATLAB workspace when triggered
Upsample	Resample input at higher rate by inserting zeros
Variable Fractional Delay	Delay input by time-varying fractional number of sample periods
Variable Integer Delay	Delay input by time-varying integer number of sample periods
Window Function	Compute and/or apply window to input signal
Zero Crossing	Count number of times signal crosses zero in single time step
Math Operations	

Block	Description
Complex to Magnitude-Angle HDL Optimized	Compute magnitude and/or phase angle of complex signal—optimized for HDL code generation using the CORDIC algorithm
Cumulative Product	Cumulative product of channel, column, or row elements
Cumulative Sum	Cumulative sum of channel, column, or row elements
Difference	Compute element-to-element difference along specified dimension of input
Normalization	Perform vector normalization along rows, columns, or specified dimension
dB Gain	Apply decibel gain
Matrix Operations	
Array-Vector Add	Add vector to array along specified dimension
Array-Vector Divide	Divide array by vector along specified dimension
Array-Vector Multiply	Multiply array by vector along specified dimension
Array-Vector Subtract	Subtract vector from array along specified dimension
Create Diagonal Matrix	Create square diagonal matrix from diagonal elements
Extract Diagonal	Extract main diagonal of input matrix
Extract Triangular Matrix	Extract lower or upper triangle from input matrices
Identity Matrix	Generate matrix with ones on main diagonal and zeros elsewhere
Matrix Concatenate	Concatenate input signals of same data type to create contiguous output signal
Matrix Product	Multiply matrix elements along rows, columns, or entire input

Block	Description
Matrix Square	Compute square of input matrix
Matrix Sum	Sum matrix elements along rows, columns, or entire input
Matrix 1-Norm	Compute 1-norm of matrix
Matrix Multiply	Multiply or divide inputs
Overwrite Values	Overwrite submatrix or subdiagonal of input
Permute Matrix	Reorder matrix rows or columns
Submatrix	Select subset of elements (submatrix) from matrix input
Toeplitz	Generate matrix with Toeplitz symmetry
Matrix Factorizations	
LDL Factorization	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
LU Factorization	Factor square matrix into lower and upper triangular components
Linear System Solvers	
Backward Substitution	Solve UX=B for X when U is upper triangular matrix
Forward Substitution	Solve LX=B for X when L is lower triangular matrix
Levinson-Durbin	Solve linear system of equations using Levinson-Durbin recursion
Switches and Counters	
Edge Detector	Detect transition from zero to nonzero value
Event-Count Comparator	Detect threshold crossing of accumulated nonzero inputs
N-Sample Switch	Switch between two inputs after specified number of sample periods

Block	Description
Buffers	
Buffer	Buffer input sequence to smaller or larger frame size
Delay Line	Rebuffer sequence of inputs
Queue	Store inputs in FIFO register
Stack	Store inputs into LIFO register
Unbuffer	Unbuffer input frame into sequence of scalar outputs
Indexing	
Flip	Flip input vertically or horizontally
Multiport Selector	Distribute arbitrary subsets of input rows or columns to multiple output ports
Overwrite Values	Overwrite submatrix or subdiagonal of input
Selector	Select input elements from vector, matrix, or multidimensional signal
Submatrix	Select subset of elements (submatrix) from matrix input
Variable Selector	Select subset of rows or columns from input
Signal Attributes	
Check Signal Attributes	Error when input signal does or does not match selected attributes exactly
Convert 1-D to 2-D	Reshape 1-D or 2-D input to 2-D matrix with specified dimensions
Convert 2-D to 1-D	Convert 2-D matrix input to 1-D vector
Data Type Conversion	Convert input signal to specified data type
Frame Conversion	Specify sampling mode of output signal
Inherit Complexity	Change complexity of input to match reference signal

More About

- "Fixed-Point Support for MATLAB System Objects in DSP System Toolbox" on page 9-19

Using the Fixed-Point Converter App with System Objects

You can use the Fixed-Point Converter app to automatically propose and apply data types for commonly used system objects. The proposed data types are based on simulation data from the System object.

Automated conversion is available for these DSP System Toolbox System Objects:

- dsp.ArrayVectorAdder
- dsp.BiquadFilter
- dsp.FIRDecimator
- dsp.FIRInterpolator
- dsp.FIRFilter (Direct Form and Direct Form Transposed only)
- dsp.FIRRateConverter
- dsp.LowerTriangularSolver
- dsp.LUFactor
- dsp.UpperTriangularSolver
- dsp.VariableFractionalDelay
- dsp.Window

The Fixed-Point Converter app can display simulation minimum and maximum values, whole number information, and histogram data.

- You cannot propose data types for these System objects based on static range data.
- You must configure the System object to use `'Custom'` fixed-point settings.
- The app applies the proposed data types only if the input signal is floating point, not fixed-point.

  The app treats scaled doubles as fixed-point. The scaled doubles workflow for System objects is the same as that for regular variables.

- The app ignores the **Default word length** setting in the **Settings** menu. The app also ignores specified rounding and overflow modes. Data-type proposals are based on the settings of the System object.

Related Examples

- "Use the Fixed-Point Converter App with a System object" on page 9-35

Use the Fixed-Point Converter App with a System object

This example converts a `dsp.FIRFilter` System object, which filters a high-frequency sinusoid signal, to fixed-point using the Fixed-Point Converter app. This example requires Fixed-Point Designer and DSP System Toolbox licenses.

Create DSP Filter Function and Test Bench

Create a `myFIRFilter` function from a `dsp.FIRFilter` System object.

By default, System objects are configured to use full-precision fixed-point arithmetic. To gather range data and get data type proposals from the Fixed-Point Converter app, configure the System object to use 'Custom' settings.

Save the function to a local writable folder.

```matlab
function output = myFIRFilter(input, num)

    persistent lowpassFIR;
    if isempty(lowpassFIR)
        lowpassFIR  = dsp.FIRFilter('NumeratorSource', 'Input port', ...
            'FullPrecisionOverride', false, ...
            'ProductDataType', 'Full precision', ... % default
            'AccumulatorDataType', 'Custom', ...
            'CustomAccumulatorDataType', numerictype(1,16,4), ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype(1,8,2));
    end
    output = step(lowpassFIR, input, num);

end
```

Create a test bench, `myFIRFilter_tb`, for the filter. The test bench generates a signal that gathers range information for conversion. Save the test bench.

```matlab
% Test bench for myFIRFilter
% Remove high-frequency sinusoid using an FIR filter.

% Initialize
f1 = 1000;
f2 = 3000;
Fs = 8000;
Fcutoff = 2000;
```

```
% Generate input
SR = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs,...
    'SamplesPerFrame',1024);

% Filter coefficients
num = fir1(130,Fcutoff/(Fs/2));

% Visualize input and output spectra
plot = dsp.SpectrumAnalyzer('SampleRate',Fs,'PlotAsTwoSidedSpectrum',...
    false,'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Input Signal (Channel 1) Output Signal (Channel 2)');

% Stream
for k = 1:100
    input = sum(step(SR),2); % Add the two sinusoids together
    filteredOutput = myFIRFilter(input, num); % Filter
    step(plot,[input,filteredOutput]); % Visualize
end
```

Convert the Function to Fixed-Point

1 Open the Fixed-Point Converter app.

   - MATLAB Toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
   - MATLAB command prompt: Enter

     ```
     fixedPointConverter
     ```

2 To add the entry-point function `myFIRFilter` to the project, browse to the file `myFIRFilter.m`, and then click **Open**.

   By default, the app saves information and settings for this project in the current folder in a file named `myFirFilter.prj`.
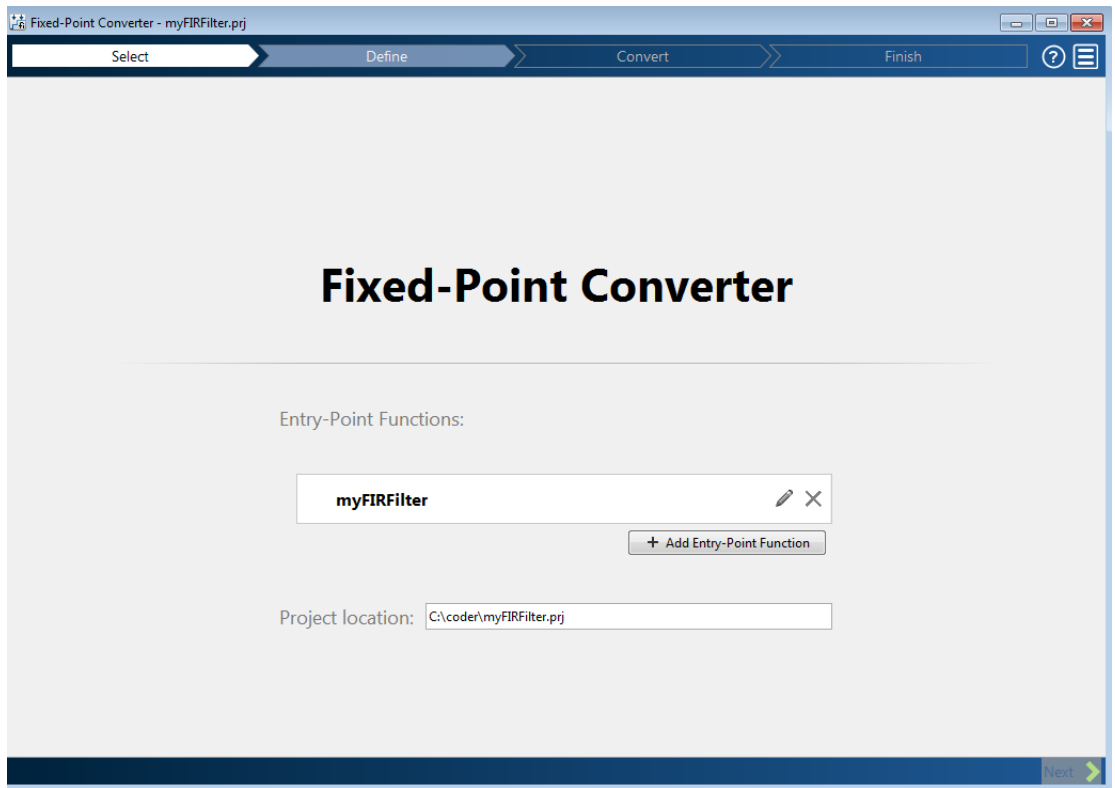
3   Click **Next** to go to the **Define Input Types** step.

The app screens `myFIRFilter.m` for code violations and readiness issues. The app does not find issues in `myFIRFilter.m`.

4   On the **Define Input Types** page, to add `myFIRFilter_tb` as a test file, browse to `myFIRFilter_tb.m`, and then click **Autodefine Input Types**.

The app determines from the test file that the type of `input` is `double(1024 x 1)` and the type of `num` is `double(1 x 131)`.
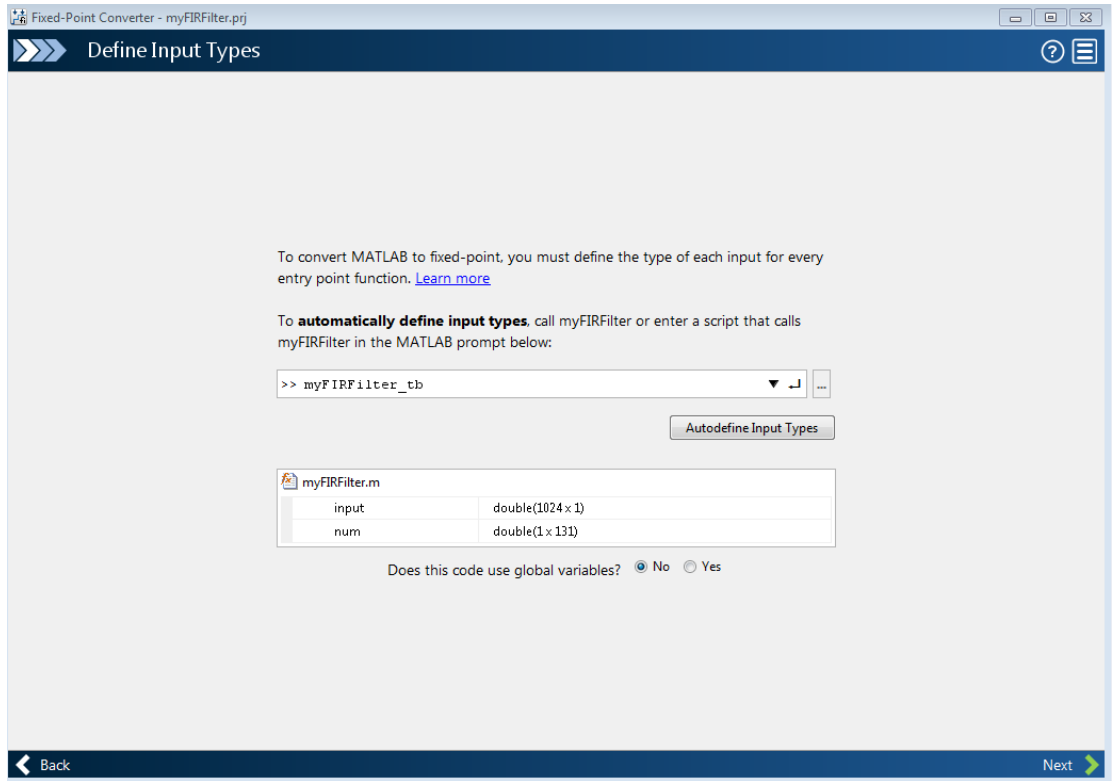
5 Click **Next** to go to the **Convert to Fixed Point** step.

6 On the **Convert to Fixed Point** page, click **Simulate** to collect range information.

The **Variables** tab displays the collected range information and type proposals. Manually edit the data type proposals as needed.

In the **Variables** tab, the **Proposed Type** field for
`lowpassFIR.CustomProductDataType` is `Full Precision`. The Fixed-Point Converter app did not propose a data type for this field because its `'ProductDataType'` setting is not set to `'Custom'`.

7 Click **Convert** to apply the proposed data types to the function.

The Fixed-Point Converter app applies the proposed data types and generates a fixed-point function, `myFIRFilter_fixpt`.
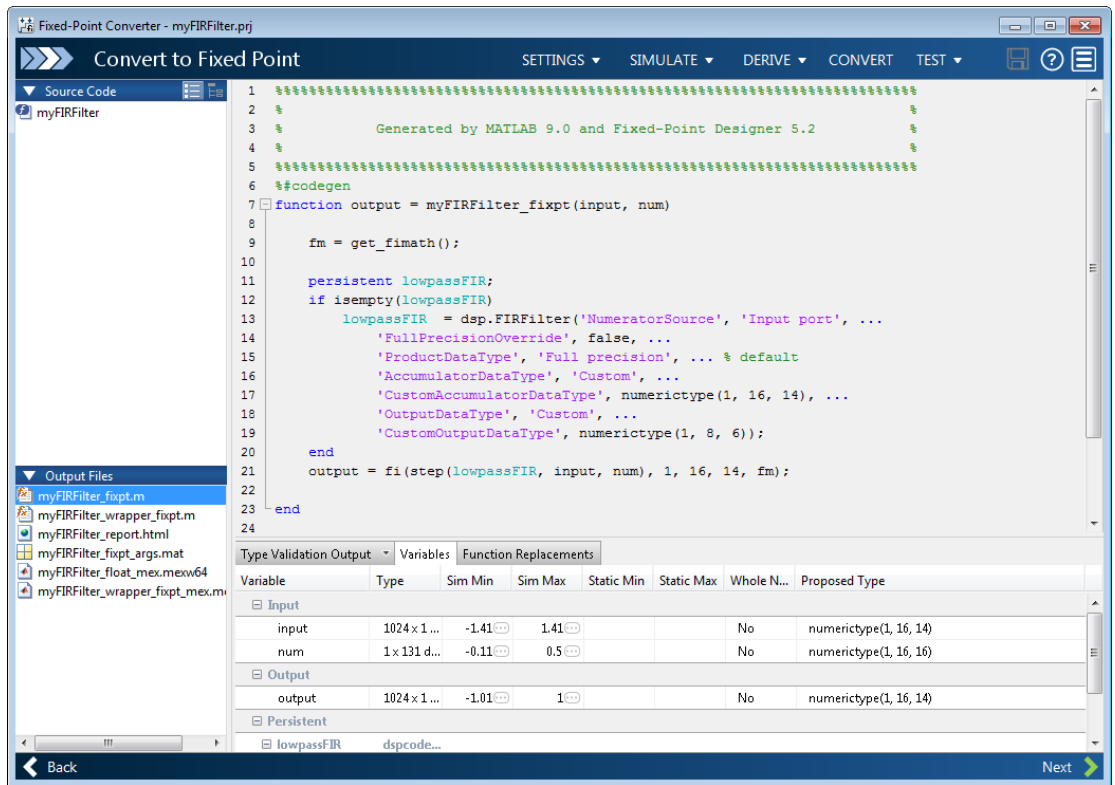
```
function output = myFIRFilter_fixpt(input, num)

    fm = get_fimath();

    persistent lowpassFIR;
    if isempty(lowpassFIR)
        lowpassFIR  = dsp.FIRFilter('NumeratorSource', 'Input port', ...
            'FullPrecisionOverride', false, ...
            'ProductDataType', 'Full precision', ... % default
            'AccumulatorDataType', 'Custom', ...
            'CustomAccumulatorDataType', numerictype(1, 16, 14), ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype(1, 8, 6));
    end
    output = fi(step(lowpassFIR, input, num), 1, 16, 14, fm);
```

```
end

function fm = get_fimath()
 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',..
 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode',..
 'FullPrecision', 'MaxSumWordLength', 128);
end
```

More About

- "Using the Fixed-Point Converter App with System Objects" on page 9-34

Specify Fixed-Point Attributes for Blocks

Fixed-Point Block Parameters

System Toolbox blocks that have fixed-point support usually allow you to specify fixed-point characteristics through block parameters. By specifying data type and scaling information for these fixed-point parameters, you can simulate your target hardware more closely.

Note: Floating-point inheritance takes precedence over the settings discussed in this section. When the block has floating-point input, all block data types match the input.

You can find most fixed-point parameters on the **Data Types** pane of System Toolbox blocks. The following figure shows a typical **Data Types** pane.

All System Toolbox blocks with fixed-point capabilities share a set of common parameters, but each block can have a different subset of these fixed-point parameters. The following table provides an overview of the most common fixed-point block parameters.

Fixed-Point Data Type Parameter	Description
Rounding Mode	Specifies the rounding mode for the block to use when the specified data type and scaling cannot exactly represent the result of a fixed-point calculation. See "Rounding Modes" on page 9-7 for more information on the available options.
Overflow Mode	Specifies the overflow mode to use when the result of a fixed-point calculation does not fit into the representable range of the specified data type. See "Overflow Handling" on page 9-7 for more information on the available options.

Fixed-Point Data Type Parameter	Description
Intermediate Product	Specifies the data type and scaling of the intermediate product for fixed-point blocks. Blocks that feed multiplication results back to the input of the multiplier use the intermediate product data type. See the reference page of a specific block to learn about the intermediate product data type for that block.
Product Output	Specifies the data type and scaling of the product output for fixed-point blocks that must compute multiplication results. See the reference page of a specific block to learn about the product output data type for that block. For or complex-complex multiplication, the multiplication result is in the accumulator data type. See "Multiplication Data Types" on page 9-12 for more information on complex fixed-point multiplication in System toolbox software.
Accumulator	Specifies the data type and scaling of the accumulator (sum) for fixed-point blocks that must hold summation results for further calculation. Most such blocks cast to the accumulator data type before performing the add operations (summation). See the reference page of a specific block for details on the accumulator data type of that block.
Output	Specifies the output data type and scaling for blocks.

Using the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool available on the **Data Types** pane of some fixed-point System Toolbox blocks.

To learn more about using the **Data Type Assistant** to help you specify block data type parameters, see the following section of the Simulink documentation:
"Specify Data Types Using Data Type Assistant"

Checking Signal Ranges

Some fixed-point System Toolbox blocks have **Minimum** and **Maximum** parameters on the **Data Types** pane. When a fixed-point data type has these parameters, you can use them to specify appropriate minimum and maximum values for range checking purposes.

To learn how to specify signal ranges and enable signal range checking, see "Signal Ranges" in the Simulink documentation.

Specify System-Level Settings

You can monitor and control fixed-point settings for System Toolbox blocks at a system or subsystem level with the Fixed-Point Tool. For additional information on these subjects, see

- The `fxptdlg` reference page — A reference page on the Fixed-Point Tool in the Simulink documentation
- "Fixed-Point Tool" — A tutorial that highlights the use of the Fixed-Point Tool in the Fixed-Point Designer software documentation

Logging

The Fixed-Point Tool logs overflows, saturations, and simulation minimums and maximums for fixed-point System Toolbox blocks. The Fixed-Point Tool does not log overflows and saturations when the `Data overflow` line in the **Diagnostics > Data Integrity** pane of the Configuration Parameters dialog box is set to `None`.

Autoscaling

You can use the Fixed-Point Tool autoscaling feature to set the scaling for System Toolbox fixed-point data types.

Data type override

System Toolbox blocks obey the `Use local settings`, `Double`, `Single`, and `Off` modes of the **Data type override** parameter in the Fixed-Point Tool. The `Scaled double` mode is also supported for System Toolboxes source and byte-shuffling blocks, and for some arithmetic blocks such as Difference and Normalization.

Inherit via Internal Rule

Selecting appropriate word lengths and scalings for the fixed-point parameters in your model can be challenging. To aid you, an `Inherit via internal rule` choice is often available for fixed-point block data type parameters, such as the **Accumulator** and **Product output** signals. The following sections describe how the word and fraction lengths are selected for you when you choose `Inherit via internal rule` for a fixed-point block data type parameter in System Toolbox software:

- "Internal Rule for Accumulator Data Types" on page 9-46
- "Internal Rule for Product Data Types" on page 9-47
- "Internal Rule for Output Data Types" on page 9-47
- "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 9-47
- "Internal Rule Examples" on page 9-49

Note: In the equations in the following sections, WL = word length and FL = fraction length.

Internal Rule for Accumulator Data Types

The internal rule for accumulator data types first calculates the ideal, full-precision result. Where N is the number of addends:

$$WL_{ideal\ accumulator} = WL_{input\ to\ accumulator} + \text{floor}(\log_2(N-1)) + 1$$

$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$

For example, consider summing all the elements of a vector of length 6 and data type sfix10_En8. The ideal, full-precision result has a word length of 13 and a fraction length of 8.

The accumulator can be real or complex. The preceding equations are used for both the real and imaginary parts of the accumulator. For any calculation, after the full-precision result is calculated, the final word and fraction lengths set by the internal rule are

affected by your particular hardware. See "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 9-47 for more information.

Internal Rule for Product Data Types

The internal rule for product data types first calculates the ideal, full-precision result:

$$WL_{ideal\ product} = WL_{input\ 1} + WL_{input\ 2}$$

$$FL_{ideal\ product} = FL_{input\ 1} + FL_{input\ 2}$$

For example, multiplying together the elements of a real vector of length 2 and data type sfix10_En8. The ideal, full-precision result has a word length of 20 and a fraction length of 16.

For real-complex multiplication, the ideal word length and fraction length is used for both the complex and real portion of the result. For complex-complex multiplication, the ideal word length and fraction length is used for the partial products, and the internal rule for accumulator data types described above is used for the final sums. For any calculation, after the full-precision result is calculated, the final word and fraction lengths set by the internal rule are affected by your particular hardware. See "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 9-47 for more information.

Internal Rule for Output Data Types

A few System Toolbox blocks have an `Inherit via internal rule` choice available for the block output. The internal rule used in these cases is block-specific, and the equations are listed in the block reference page.

As with accumulator and product data types, the final output word and fraction lengths set by the internal rule are affected by your particular hardware, as described in "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 9-47.

The Effect of the Hardware Implementation Pane on the Internal Rule

The internal rule selects word lengths and fraction lengths that are appropriate for your hardware. To get the best results using the internal rule, you must specify the type of hardware you are using on the **Hardware Implementation** pane of the Configuration

Parameters dialog box. You can open this dialog box from the **Simulation** menu in your model.
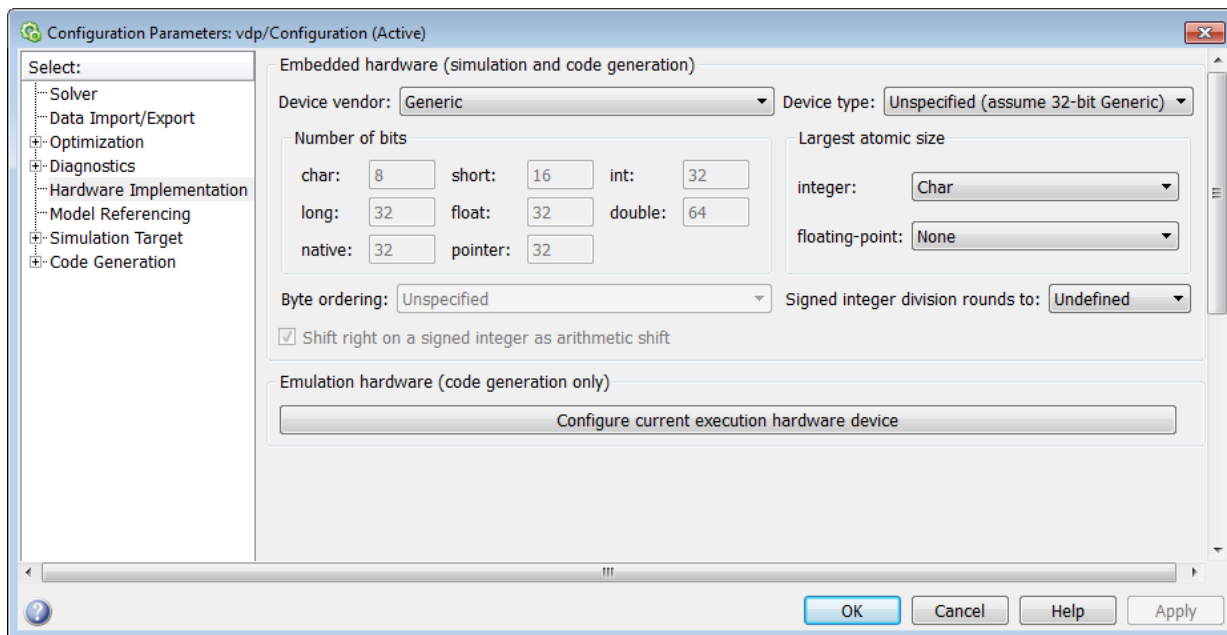
ASIC/FPGA

On an ASIC/FPGA target, the ideal, full-precision word length and fraction length calculated by the internal rule are used. If the calculated ideal word length is larger than the largest allowed word length, you receive an error. The largest word length allowed for Simulink and System Toolbox software is 128 bits.

Other targets

For all targets other than ASIC/FPGA, the ideal, full-precision word length calculated by the internal rule is rounded up to the next available word length of the target. The calculated ideal fraction length is used, keeping the least-significant bits.

If the calculated ideal word length for a product data type is larger than the largest word length on the target, you receive an error. If the calculated ideal word length for an accumulator or output data type is larger than the largest word length on the target, the largest target word length is used.

Internal Rule Examples

The following sections show examples of how the internal rule interacts with the **Hardware Implementation** pane to calculate accumulator data types and product data types.

Accumulator Data Types

Consider the following model ex_internalRule_accumExp.
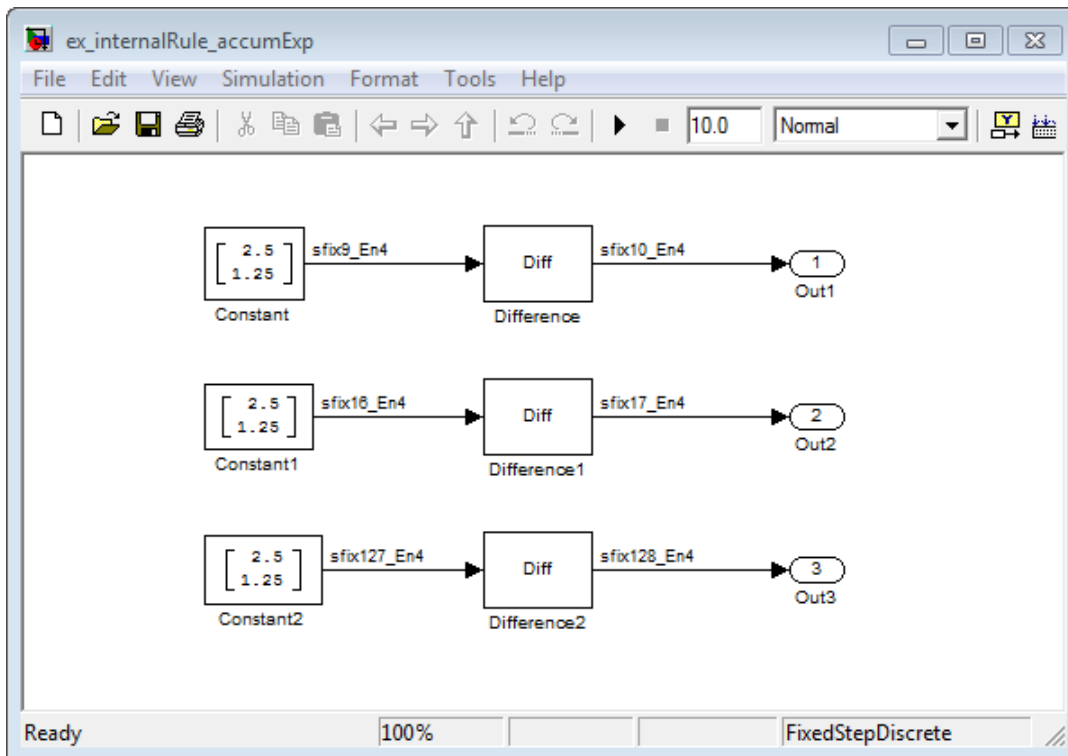
In the `Difference` blocks, the **Accumulator** parameter is set to `Inherit: Inherit via internal rule`, and the **Output** parameter is set to `Inherit: Same as accumulator`. Therefore, you can see the accumulator data type calculated by the internal rule on the output signal in the model.

In the preceding model, the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box is set to ASIC/FPGA. Therefore, the accumulator data type used by the internal rule is the ideal, full-precision result.

Calculate the full-precision word length for each of the Difference blocks in the model:

$$WL_{ideal\ accumulator} = WL_{input\ to\ accumulator} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator} = 9 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator} = 9 + 0 + 1 = 10$$

$$WL_{ideal\ accumulator1} = WL_{input\ to\ accumulator1} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator1} = 16 + 0 + 1 = 17$$

$$WL_{ideal\ accumulator2} = WL_{input\ to\ accumulator2} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator2} = 127 + 0 + 1 = 128$$

Calculate the full-precision fraction length, which is the same for each Matrix Sum block in this example:

$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$
$$FL_{ideal\ accumulator} = 4$$

Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to 32—bit Embedded Processor, by changing the parameters as shown in the following figure.
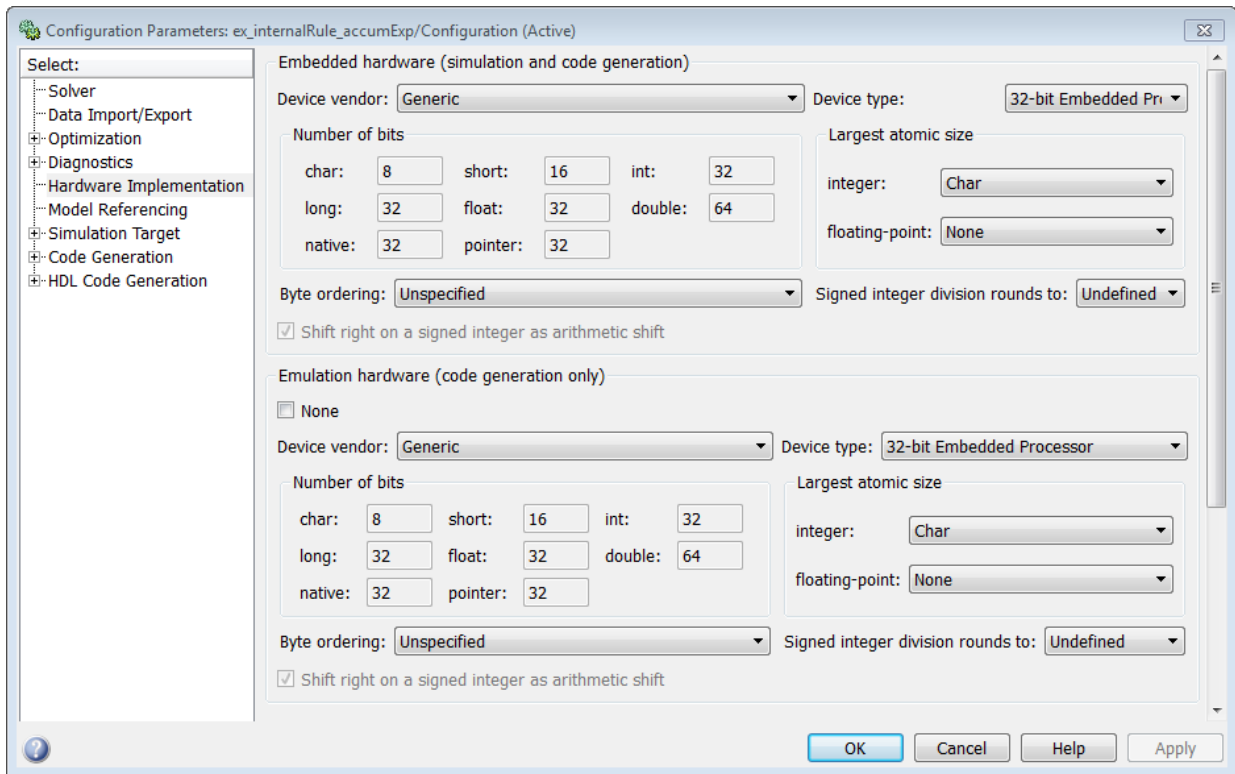
As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 10, 17, and 128 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case You can see this if you rerun the model, as shown in the following figure.
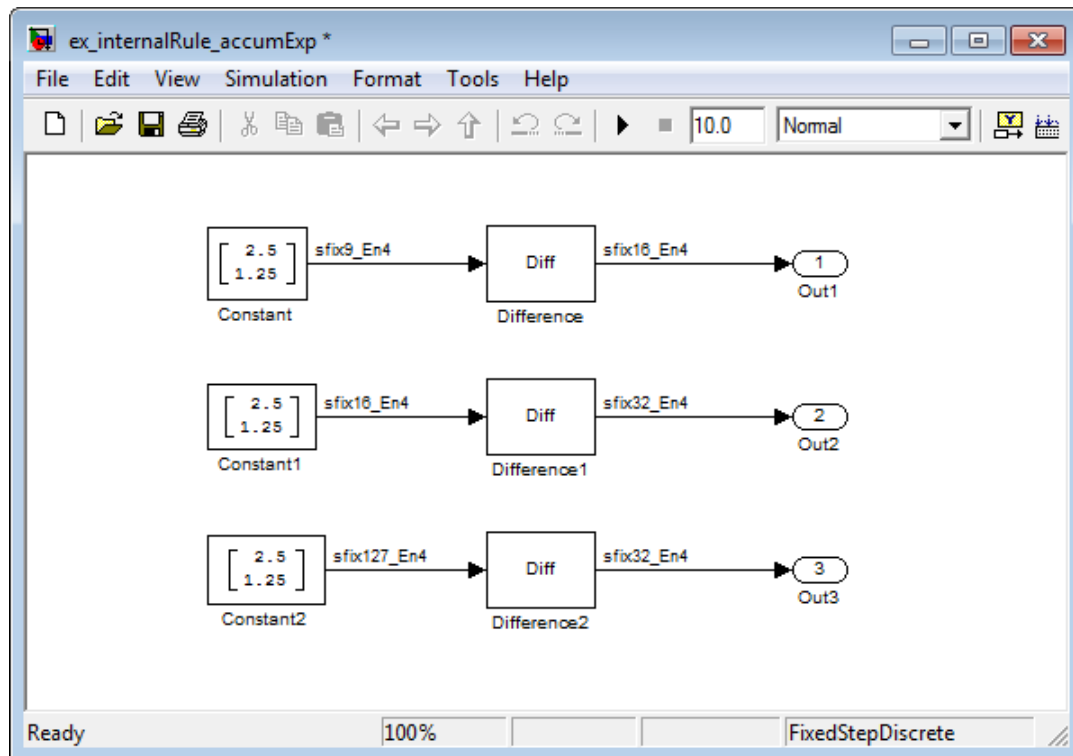
Product Data Types

Consider the following model ex_internalRule_prodExp.
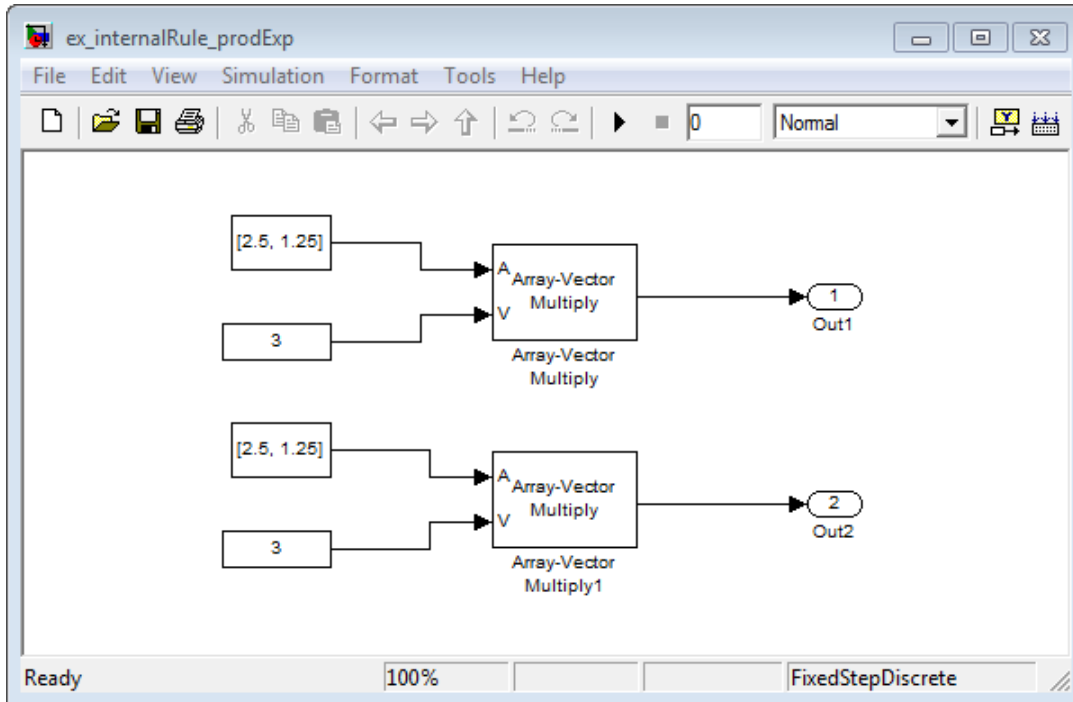
In the `Array-Vector Multiply` blocks, the **Product Output** parameter is set to `Inherit: Inherit via internal rule`, and the **Output** parameter is set to `Inherit: Same as product output`. Therefore, you can see the product output data type calculated by the internal rule on the output signal in the model. The setting of the **Accumulator** parameter does not matter because this example uses real values.

For the preceding model, the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box is set to `ASIC/FPGA`. Therefore, the product data type used by the internal rule is the ideal, full-precision result.

Calculate the full-precision word length for each of the Array-Vector Multiply blocks in the model:

$$WL_{ideal\ product} = WL_{input\ a} + WL_{input\ b}$$
$$WL_{ideal\ product} = 7 + 5 = 12$$

$$WL_{ideal\ product1} = WL_{input\ a} + WL_{input\ b}$$
$$WL_{ideal\ product1} = 16 + 15 = 31$$

Calculate the full-precision fraction length, which is the same for each Array-Vector Multiply block in this example:

$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$
$$FL_{ideal\ accumulator} = 4$$

Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to 32—bit Embedded Processor, as shown in the following figure.
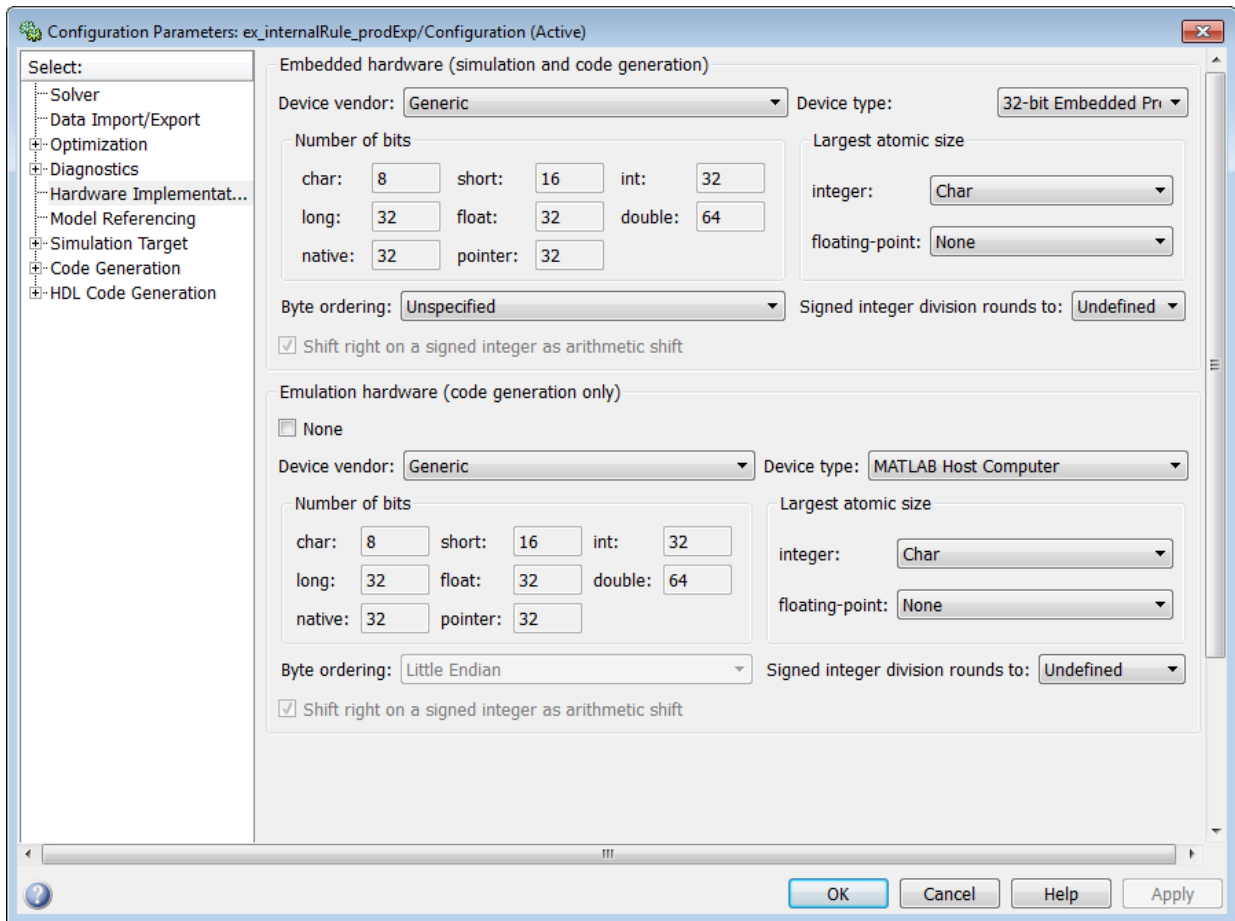
As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 12 and 31 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case. You can see this if you rerun the model, as shown in the following figure.
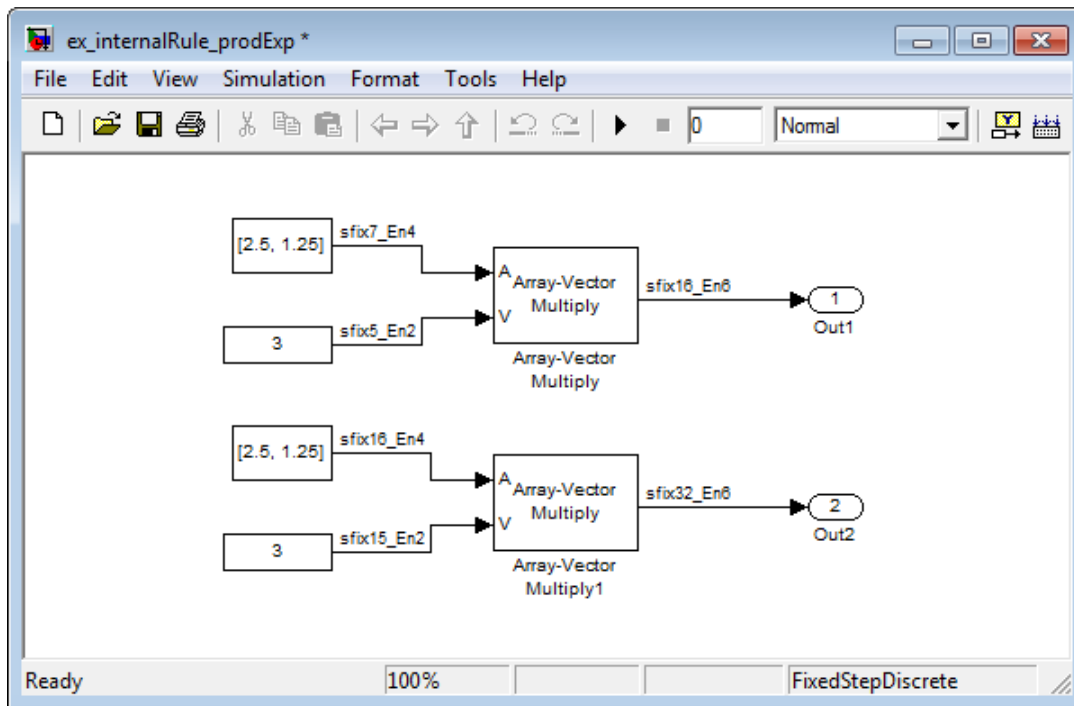
Specify Data Types for Fixed-Point Blocks

The following sections show you how to use the Fixed-Point Tool to select appropriate data types for fixed-point blocks in the `ex_fixedpoint_tut` model:

- "Prepare the Model" on page 9-56
- "Use Data Type Override to Find a Floating-Point Benchmark" on page 9-61
- "Use the Fixed-Point Tool to Propose Fraction Lengths" on page 9-62
- "Examine the Results and Accept the Proposed Scaling" on page 9-62

Prepare the Model

1 Open the model by typing ex_fixedpoint_tut at the MATLAB command line.
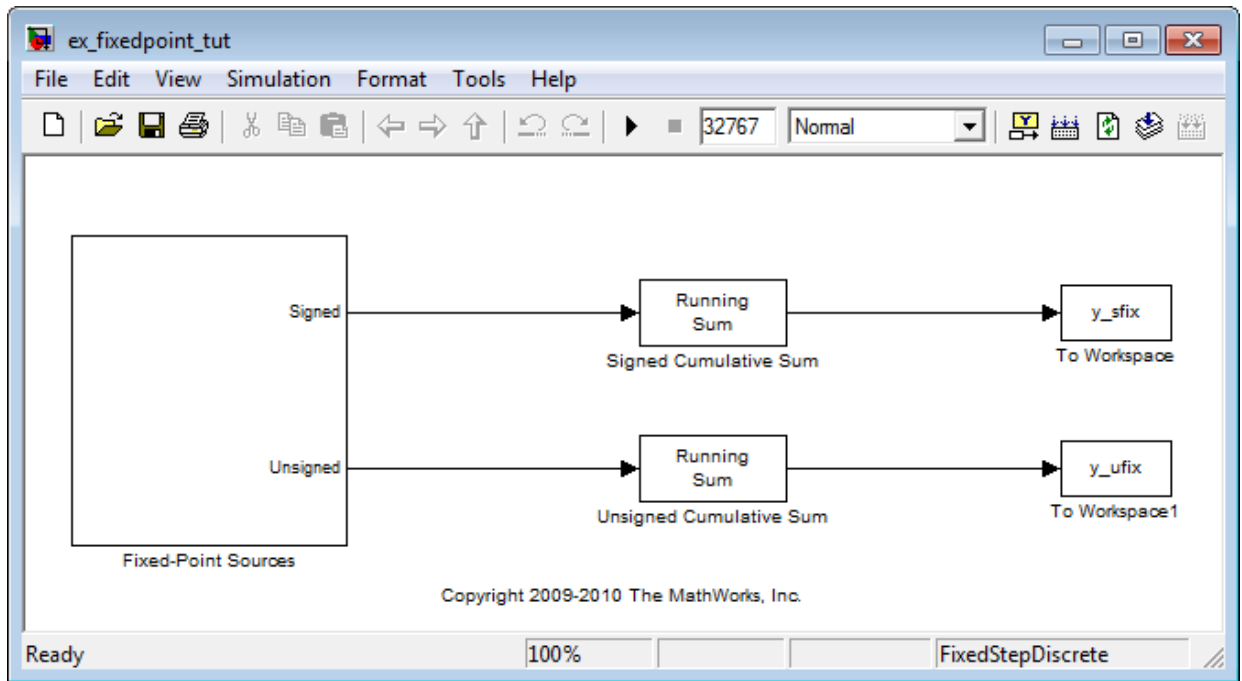
This model uses the Cumulative Sum block to sum the input coming from the Fixed-Point Sources subsystem. The Fixed-Point Sources subsystem outputs two signals with different data types:

- The Signed source has a word length of 16 bits and a fraction length of 15 bits.
- The Unsigned source has a word length of 16 bits and a fraction length of 16 bits.

2 Run the model to check for overflow. MATLAB displays the following warnings at the command line:

```
Warning: Overflow occurred. This originated from
'ex_fixedpoint_tut/Signed Cumulative Sum'.
Warning: Overflow occurred. This originated from
'ex_fixedpoint_tut/Unsigned Cumulative Sum'.
```

According to these warnings, overflow occurs in both Cumulative Sum blocks.

3 To investigate the overflows in this model, use the Fixed-Point Tool. You can open the Fixed-Point Tool by selecting **Tools** > **Fixed-Point** > **Fixed-Point Tool**

from the model menu. Turn on logging for all blocks in your model by setting the
Fixed-point instrumentation mode parameter to `Minimums, maximums and overflows`.

4 Now that you have turned on logging, rerun the model by clicking the Simulation button.
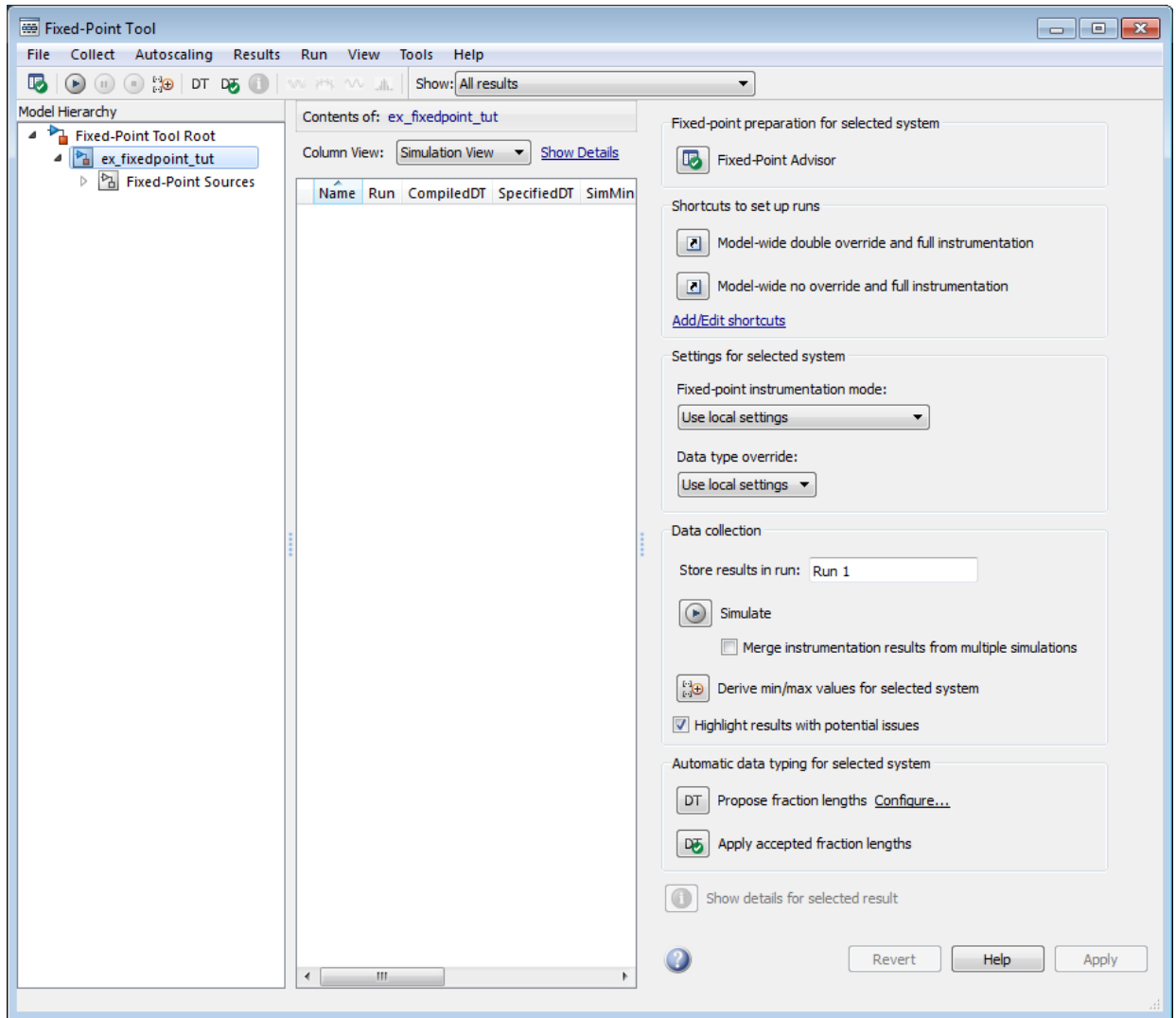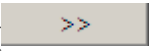
5 The results of the simulation appear in a table in the central **Contents** pane of the Fixed-Point Tool. Review the following columns:

- **Name** — Provides the name of each signal in the following format: `Subsystem Name/Block Name: Signal Name`.

- **SimDT** — The simulation data type of each logged signal.

- **SpecifiedDT** — The data type specified on the block dialog for each signal.

- **SimMin** — The smallest representable value achieved during simulation for each logged signal.

- **SimMax** — The largest representable value achieved during simulation for each logged signal.

- **OverflowWraps** — The number of overflows that wrap during simulation.

For more information on each of the columns in this table, see the "Contents Pane" section of the Simulink `fxptdlg` function reference page.

You can also see that the **SimMin** and **SimMax** values for the Accumulator data types range from `0` to `.9997`. The logged results indicate that 8,192 overflows wrapped during simulation in the Accumulator data type of the Signed Cumulative Sum block. Similarly, the Accumulator data type of the Unsigned Cumulative Sum block had 16,383 overflows wrap during simulation.

To get more information about each of these data types, highlight them in the

Contents pane, and click the **Show details for selected result** button ( )

6 Assume a target hardware that supports 32-bit integers, and set the Accumulator word length in both Cumulative Sum blocks to `32`. To do so, perform the following steps:

1 Right-click the `Signed Cumulative Sum: Accumulator` row in the Fixed-Point Tool pane, and select **Highlight Block In Model**.

2 Double-click the block in the model, and select the **Data Types** pane of the dialog box.

3 Open the **Data Type Assistant for Accumulator** by clicking the Assistant button (`>>`) in the Accumulator data type row.

4 Set the **Mode** to `Fixed Point`. To see the representable range of the current specified data type, click the **Fixed-point details** link. The tool displays the

representable maximum and representable minimum values for the current data type.
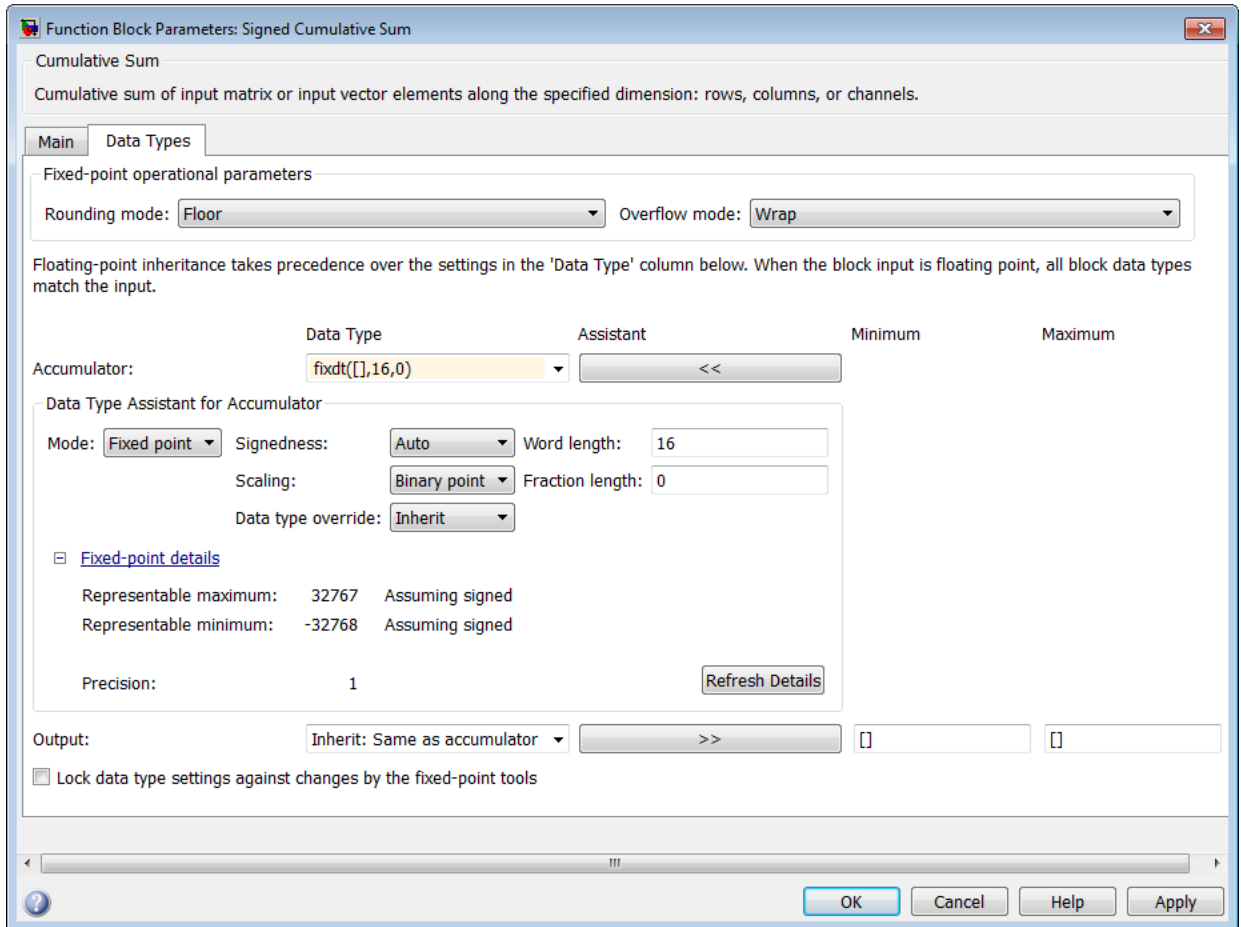
5   Change the **Word length** to 32, and click the **Refresh details** button in the **Fixed-point details** section to see the updated representable range. When you change the value of the **Word length** parameter, the data type string in the **Data Type** edit box automatically updates.

6   Click **OK** on the block dialog box to save your changes and close the window.

7 Set the word length of the Accumulator data type of the Unsigned Cumulative Sum block to 32 bits. You can do so in one of two ways:

- Type the data type string `fixdt([],32,0)` directly into **Data Type** edit box for the Accumulator data type parameter.

- Perform the same steps you used to set the word length of the Accumulator data type of the Signed Cumulative Sum block to 32 bits.

7 To verify your changes in word length and check for overflow, rerun your model. To do so, click the **Simulate** button in the Fixed-Point Tool.

The **Contents** pane of the Fixed-Point Tool updates, and you can see that no overflows occurred in the most recent simulation. However, you can also see that the **SimMin** and **SimMax** values range from 0 to 0. This underflow happens because the fraction length of the Accumulator data type is too small. The **SpecifiedDT** cannot represent the precision of the data values. The following sections discuss how to find a floating-point benchmark and use the Fixed-Point Tool to propose fraction lengths.

Use Data Type Override to Find a Floating-Point Benchmark

The **Data type override** feature of the Fixed-Point tool allows you to override the data types specified in your model with floating-point types. Running your model in `Double` override mode gives you a reference range to help you select appropriate fraction lengths for your fixed-point data types. To do so, perform the following steps:

1 Open the Fixed-Point Tool and set **Data type override** to `Double`.
2 Run your model by clicking the **Run simulation and store active results** button.
3 Examine the results in the **Contents** pane of the Fixed-Point Tool. Because you ran the model in `Double` override mode, you get an accurate, idealized representation of the simulation minimums and maximums. These values appear in the **SimMin** and **SimMax** parameters.
4 Now that you have an accurate reference representation of the simulation minimum and maximum values, you can more easily choose appropriate fraction lengths. Before making these choices, save your active results to reference so you can use them as your floating-point benchmark. To do so, select **Results** > **Move Active Results To Reference** from the Fixed-Point Tool menu. The status displayed in the **Run** column changes from `Active` to `Reference` for all signals in your model.

Use the Fixed-Point Tool to Propose Fraction Lengths

Now that you have your `Double` override results saved as a floating-point reference, you are ready to propose fraction lengths.

1 To propose fraction lengths for your data types, you must have a set of `Active` results available in the Fixed-Point Tool. To produce an active set of results, simply rerun your model. The tool now displays both the `Active` results and the `Reference` results for each signal.

2 Select the **Use simulation min/max if design min/max is not available** check box. You did not specify any design minimums or maximums for the data types in this model. Thus, the tool uses the logged information to compute and propose fraction lengths. For information on specifying design minimums and maximums, see "Signal Ranges" in the Simulink documentation.

3
Click the **Propose fraction lengths** button (button). The tool populates the proposed data types in the **ProposedDT** column of the **Contents** pane. The corresponding proposed minimums and maximums are displayed in the **ProposedMin** and **ProposedMax** columns.

Examine the Results and Accept the Proposed Scaling

Before accepting the fraction lengths proposed by the Fixed-Point Tool, it is important to look at the details of that data type. Doing so allows you to see how much of your data the suggested data type can represent. To examine the suggested data types and accept the proposed scaling, perform the following steps:

1 In the **Contents** pane of the Fixed-Point Tool, you can see the proposed fraction lengths for the data types in your model.

  - The proposed fraction length for the Accumulator data type of both the Signed and Unsigned Cumulative Sum blocks is 17 bits.

  - To get more details about the proposed scaling for a particular data type, highlight the data type in the **Contents** pane of the Fixed-Point Tool.

  - Open the Autoscale Information window for the highlighted data type by clicking

    the **Show autoscale information for the selected result** button (button).

2 When the Autoscale Information window opens, check the **Value** and **Percent Proposed Representable** columns for the **Simulation Minimum** and **Simulation Maximum** parameters. You can see that the proposed data type can represent 100% of the range of simulation data.

3   To accept the proposed data types, select the check box in the **Accept** column for each data type whose proposed scaling you want to keep. Then, click the **Apply accepted fraction lengths** button ( ). The tool updates the specified data types on the block dialog boxes and the **SpecifiedDT** column in the **Contents** pane.

4   To verify the newly accepted scaling, set the **Data type override** parameter back to **Use local settings**, and run the model. Looking at **Contents** pane of the Fixed-Point Tool, you can see the following details:

- The **SimMin** and **SimMax** values of the `Active` run match the **SimMin** and **SimMax** values from the floating-point `Reference` run.

- There are no longer any overflows.

- The **SimDT** does not match the **SpecifiedDT** for the Accumulator data type of either Cumulative Sum block. This difference occurs because the Cumulative Sum block always inherits its **Signedness** from the input signal and only allows you to specify a **Signedness** of `Auto`. Therefore, the **SpecifiedDT** for both Accumulator data types is `fixdt([],32,17)`. However, because the Signed Cumulative Sum block has a signed input signal, the **SimDT** for the Accumulator parameter of that block is also signed (`fixdt(1,32,17)`). Similarly, the **SimDT** for the Accumulator parameter of the Unsigned Cumulative Sum block inherits its **Signedness** from its input signal and thus is unsigned (`fixdt(0,32,17)`).

Quantizers

In this section...

Scalar Quantizers

Analysis and Synthesis of Speech

You can use blocks from the DSP System Toolbox Quantizers library to design scalar quantizer encoders and decoders. A speech signal is usually represented in digital format, which is a sequence of binary bits. For storage and transmission applications, it is desirable to compress a signal by representing it with as few bits as possible, while maintaining its perceptual quality. Quantization is the process of representing a signal with a reduced level of precision. If you decrease the number of bits allocated for the quantization of your speech signal, the signal is distorted and the speech quality degrades.

In narrowband digital speech compression, speech signals are sampled at a rate of 8000 samples per second. Each sample is typically represented by 8 bits. This 8-bit represenation corresponds to a bit rate of 64 kbits per second. Further compression is possible at the cost of quality. Most of the current low bit rate speech coders are based on the principle of linear predictive speech coding. This topic shows you how to use the `Scalar Quantizer Encoder` and `Scalar Quantizer Decoder` blocks to implement a simple speech coder.

1   Type `ex_sq_example1` at the MATLAB command line to open the example model.
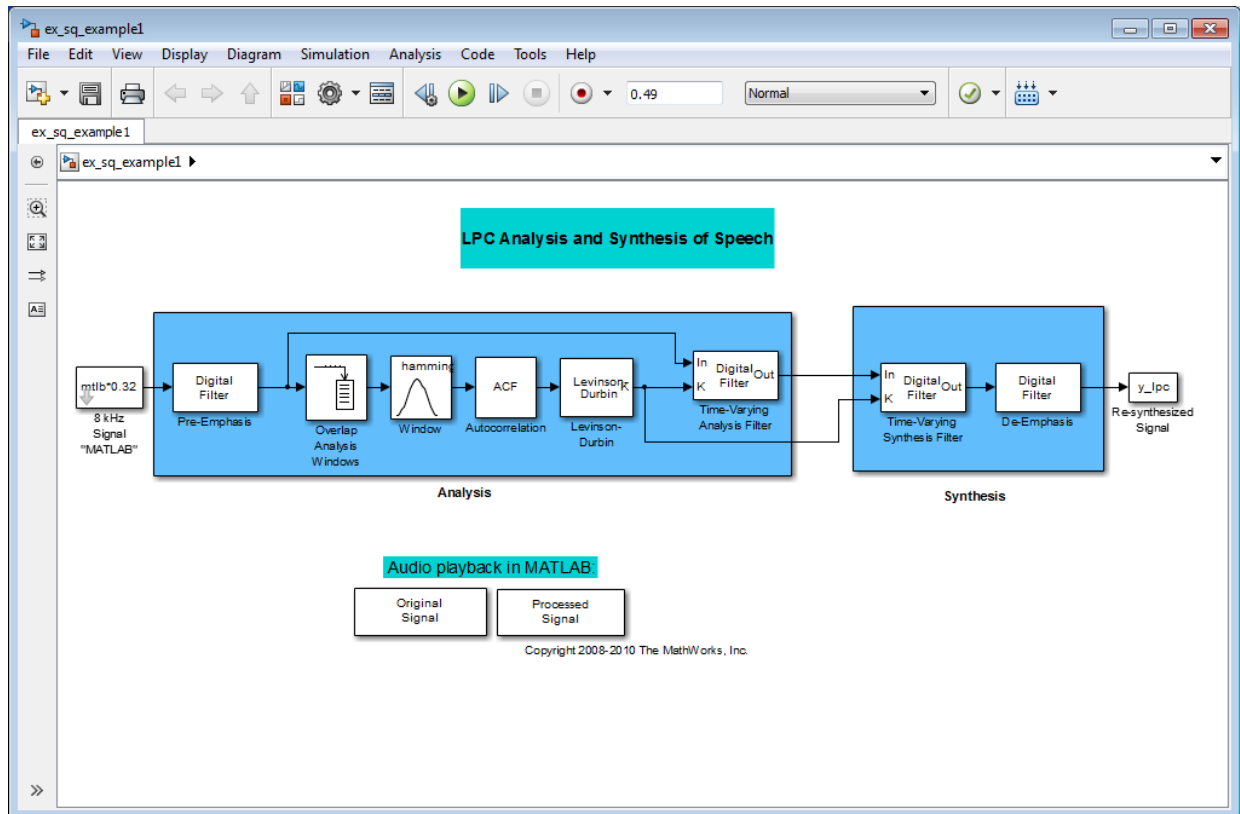
This model pre-emphasizes the input speech signal by applying an FIR filter. Then, it calculates the reflection coefficients of each frame using the Levinson-Durbin algorithm. The model uses these reflection coefficients to create the linear prediction analysis filter (lattice-structure). Next, the model calculates the residual signal by filtering each frame of the pre-emphasized speech samples using the reflection coefficients. The residual signal, which is the output of the analysis stage, usually has a lower energy than the input signal. The blocks in the synthesis stage of the model filter the residual signal using the reflection coefficients and apply an all-pole de-emphasis filter. The de-emphasis filter is the inverse of the pre-emphasis filter. The result is the full recovery of the original signal.

2 Run this model.

3 Double-click the Original Signal and Processed Signal blocks and listen to both the original and the processed signal.

There is no significant difference between the two because no quantization was performed.

To better approximate a real-world speech analysis and synthesis system, quantize the residual signal and reflection coefficients before they are transmitted. The following topics show you how to design scalar quantizers to accomplish this task.

Identify Your Residual Signal and Reflection Coefficients

In the previous topic, "Analysis and Synthesis of Speech" on page 9-64, you learned the theory behind the LPC Analysis and Synthesis of Speech example model. In this topic, you define the residual signal and the reflection coefficients in your MATLAB workspace as the variables *E* and *K*, respectively. Later, you use these values to create your scalar quantizers:

1 Open the example model by typing `ex_sq_example2` at the MATLAB command line.

2 Save the model file as `ex_sq_example2` in your working folder.

3 From the Simulink Sinks library, click-and-drag two To Workspace blocks into your model.

4 Connect the output of the Levinson-Durbin block to one of the To Workspace blocks.

5 Double-click this To Workspace block and set the **Variable name** parameter to K. Click **OK**.

6 Connect the output of the Time-Varying Analysis Filter block to the other To Workspace block.

7 Double-click this To Workspace block and set the **Variable name** parameter to E. Click **OK**.

Your model should now look similar to this figure.
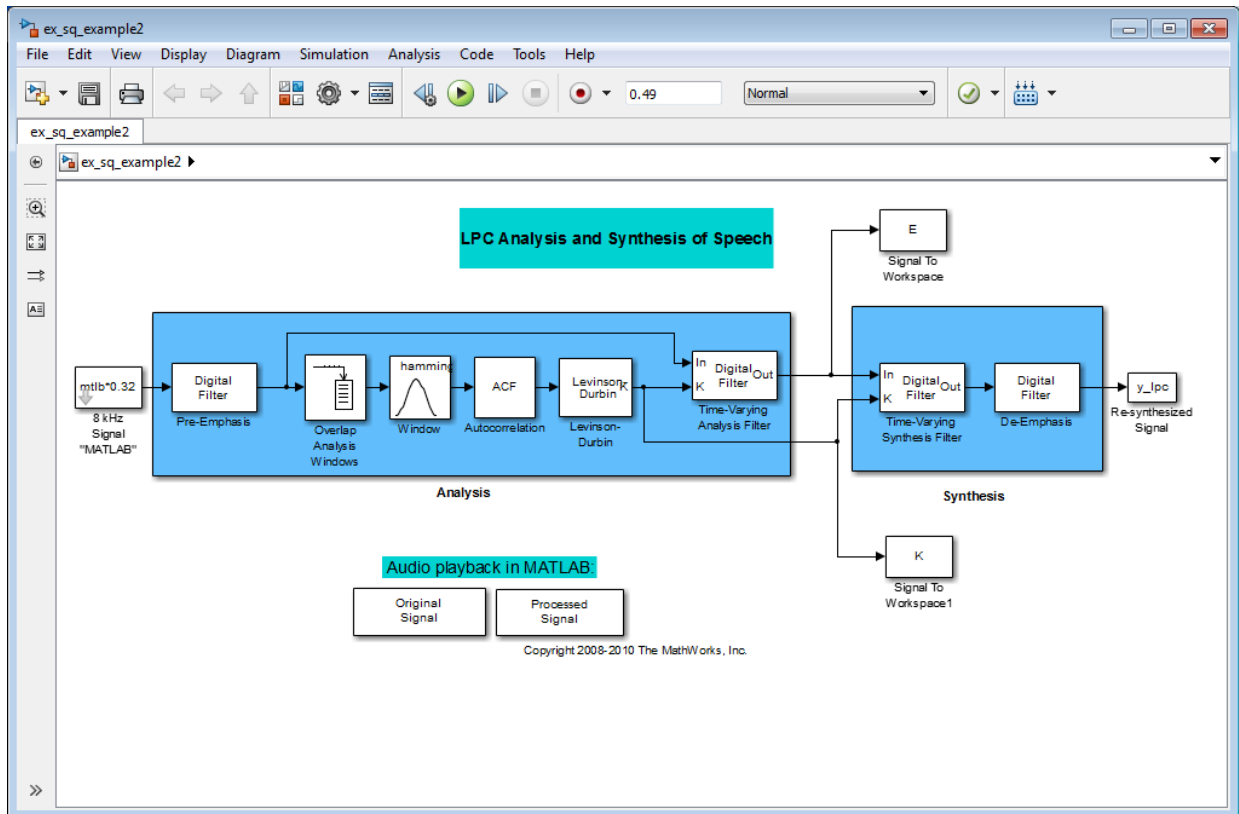
8 Run your model.

The residual signal, *E*, and your reflection coefficients, *K*, are defined in the MATLAB workspace. In the next topic, you use these variables to design your scalar quantizers.

Create a Scalar Quantizer

In this topic, you create scalar quantizer encoders and decoders to quantize the residual signal, *E*, and the reflection coefficients, *K*:

1 If the model you created in "Identify Your Residual Signal and Reflection Coefficients" on page 9-66 is not open on your desktop, you can open an equivalent model by typing ex_sq_example2 at the MATLAB command prompt.

2   Run this model to define the variables *E* and *K* in the MATLAB workspace.

3   From the Quantizers library, click-and-drag a Scalar Quantizer Design block into your model. Double-click this block to open the SQ Design Tool GUI.

4   For the **Training Set** parameter, enter K.

The variable *K* represents the reflection coefficients you want to quantize. By definition, they range from -1 to 1.

Note: Theoretically, the signal that is used as the **Training Set** parameter should contain a representative set of values for the parameter to be quantized. However, this example provides an approximation to this global training process.

---

5   For the **Number of levels** parameter, enter 128.

Assume that your compression system has 7 bits to represent each reflection coefficient. This means it is capable of representing 2^7 or 128 values. The **Number of levels** parameter is equal to the total number of codewords in the codebook.

6   Set the **Block type** parameter to Both.

7   For the **Encoder block name** parameter, enter SQ Encoder - Reflection Coefficients.

8   For the **Decoder block name** parameter, enter SQ Decoder - Reflection Coefficients.

9   Make sure that your desired destination model, ex_sq_example2, is the current model. You can type gcs in the MATLAB Command Window to display the name of your current model.

10 In the SQ Design Tool GUI, click the **Design and Plot** button to apply the changes you made to the parameters.

The GUI should look similar to the following figure.
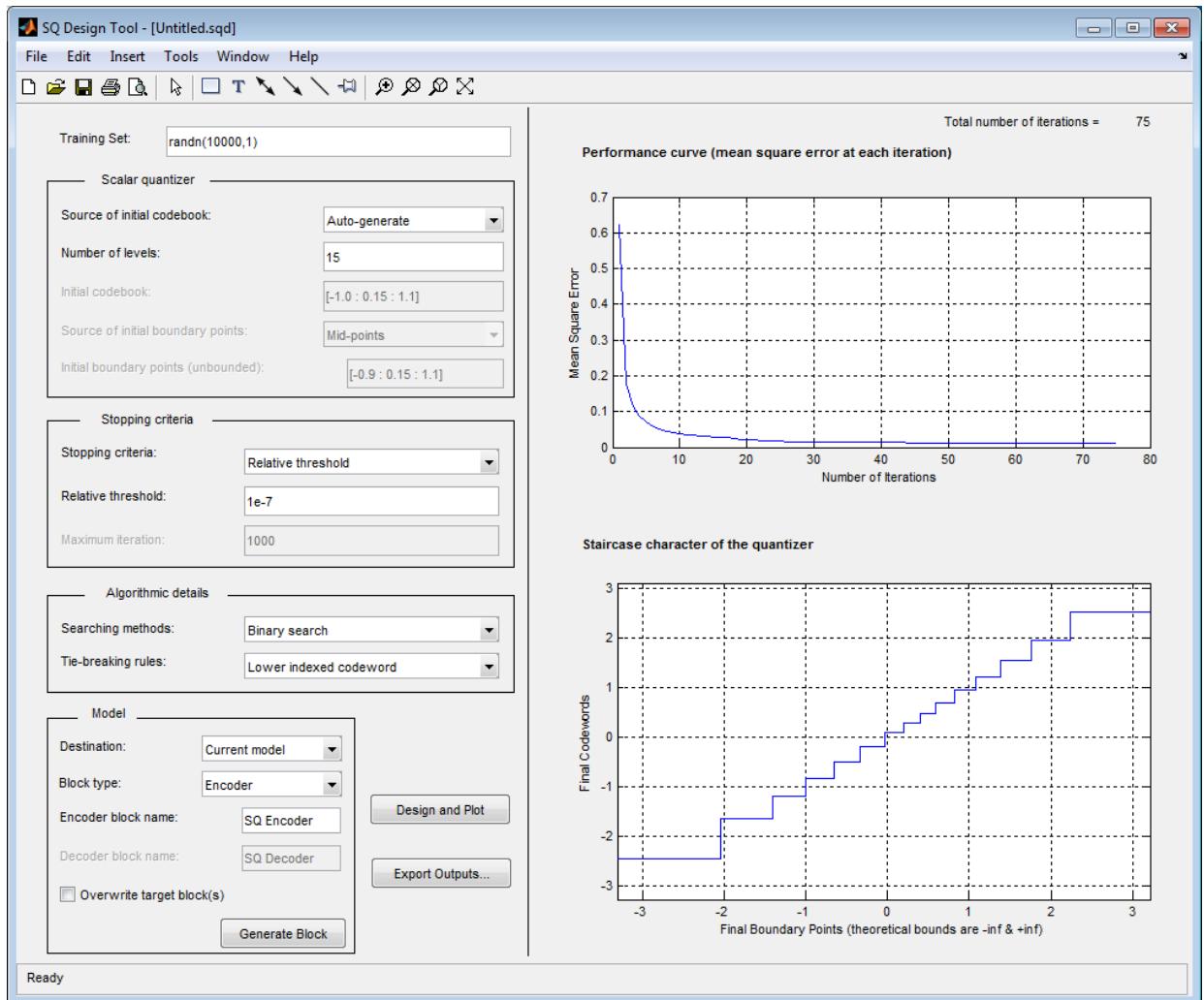
11 Click the **Generate Model** button.

Two new blocks, SQ Encoder - Reflection Coefficients and SQ Decoder - Reflection Coefficients, appear in your model file.

12 Click the SQ Design Tool GUI and, for the **Training Set** parameter, enter E.

13 Repeat steps 5 to 11 for the variable *E*, which represents the residual signal you want to quantize. In steps 6 and 7, name your blocks SQ Encoder - Residual and SQ Decoder - Residual.

Once you have completed these steps, two new blocks, SQ Encoder - Residual and SQ Decoder - Residual, appear in your model file.

14 Close the SQ Design Tool GUI. You do not need to save the SQ Design Tool session.

You have now created a scalar quantizer encoder and a scalar quantizer decoder for each signal you want to quantize. You are ready to quantize the residual signal, *E*, and the reflection coefficients, *K*.

15 Save the model as `ex_sq_example3`. Your model should look similar to the following figure.
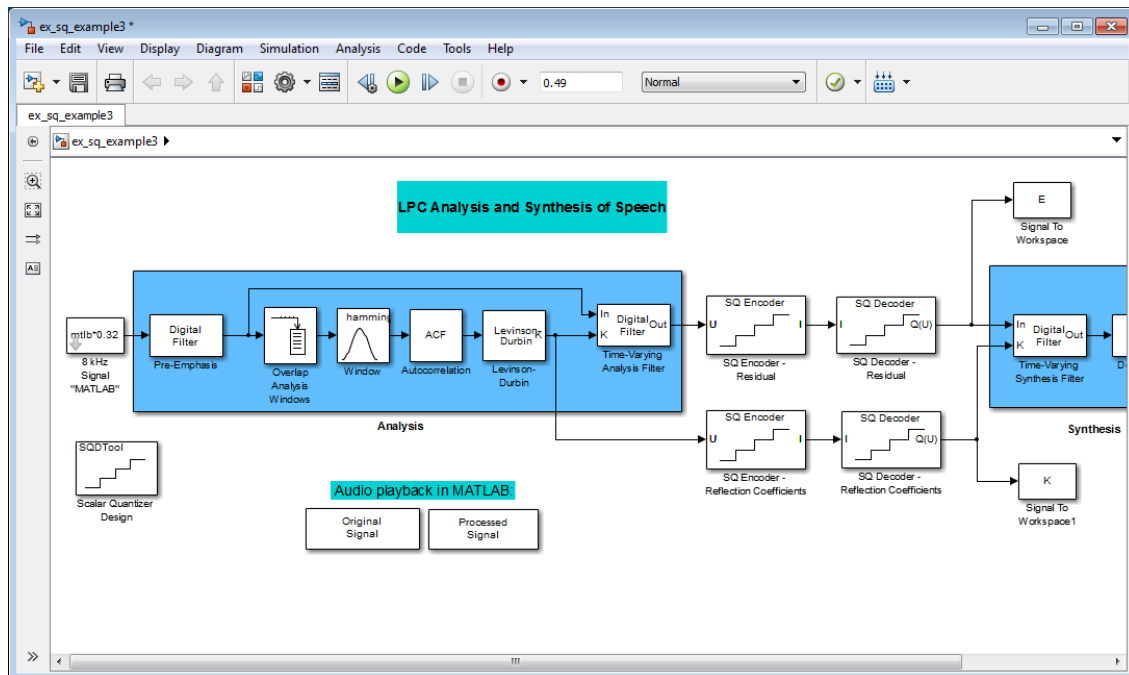
16 Run your model.

17 Double-click the Original Signal and Processed Signal blocks, and listen to both signals.

Again, there is no perceptible difference between the two. You can therefore conclude that quantizing your residual and reflection coefficients did not affect the ability of your system to accurately reproduce the input signal.

You have now quantized the residual and reflection coefficients. The bit rate of a quantization system is calculated as (bits per frame)*(frame rate).

In this example, the bit rate is [(80 residual samples/frame)*(7 bits/sample) + (12 reflection coefficient samples/frame)*(7 bits/sample)]*(100 frames/second), or 64.4 kbits per second. This is higher than most modern speech coders, which typically have a bit rate of 8 to 24 kbits per second. If you decrease the number of bits allocated for the quantization of the reflection coefficients or the residual signal, the overall bit rate would decrease. However, the speech quality would also degrade.

For information about decreasing the bit rate without affecting speech quality, see "Vector Quantizers" on page 9-71.

Vector Quantizers

- "Build Your Vector Quantizer Model" on page 9-71
- "Configure and Run Your Model" on page 9-73

Build Your Vector Quantizer Model

In the previous section, you created scalar quantizer encoders and decoders and used them to quantize your residual signal and reflection coefficients. The bit rate of your scalar quantization system was 64.4 kbits per second. This bit rate is higher than most modern speech coders. To accommodate a greater number of users in each channel, you need to lower this bit rate while maintaining the quality of your speech signal. You can use vector quantizers, which exploit the correlations between each sample of a signal, to accomplish this task.

In this topic, you modify your scalar quantization model so that you are using a split vector quantizer to quantize your reflection coefficients:

1 Open a model similar to the one you created in "Create a Scalar Quantizer" on page 9-67 by typing ex_vq_example1 at the MATLAB command prompt. The example model **ex_vq_example1** adds a new LSF Vector Quantization subsystem to the **ex_sq_example3** model. This subsystem is preconfigured to work as a vector quantizer. You can use this subsystem to encode and decode your reflection coefficients using the split vector quantization method.

2   Delete the SQ Encoder – Reflection Coefficients and SQ Decoder – Reflection Coefficients blocks.

3   From the Simulink Sinks library, click-and-drag a Terminator block into your model.

4   From the DSP System Toolbox Estimation > Linear Prediction library, click-and-drag a LSF/LSP to LPC Conversion block and two LPC to/from RC blocks into your model.

5   Connect the blocks as shown in the following figure. You do not need to connect Terminator blocks to the P ports of the LPC to/from RC blocks. These ports disappear once you set block parameters.
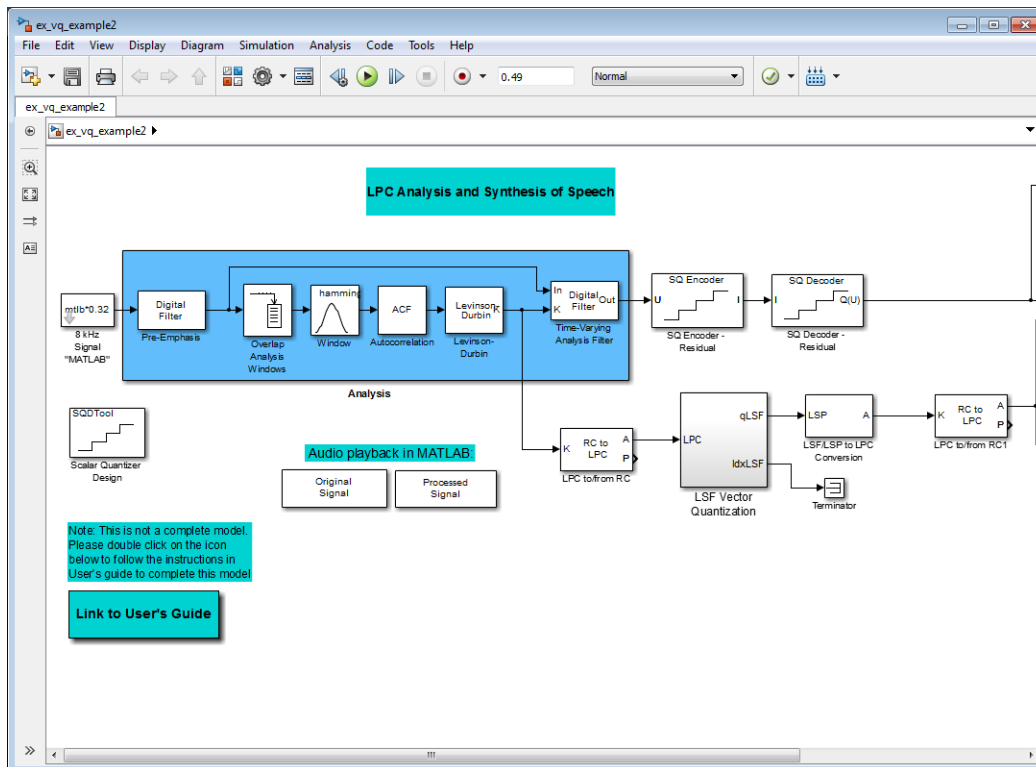
You have modified your model to include a subsystem capable of vector quantization. In the next topic, you reset your model parameters to quantize your reflection coefficients using the split vector quantization method.

Configure and Run Your Model

In the previous topic, you configured your scalar quantization model for vector quantization by adding the LSF Vector Quantization subsystem. In this topic, you set your block parameters and quantize your reflection coefficients using the split vector quantization method.

1 If the model you created in "Build Your Vector Quantizer Model" on page 9-71 is not open on your desktop, you can open an equivalent model by typing ex_vq_example2 at the MATLAB command prompt.

2 Double-click the LSF Vector Quantization subsystem, and then double-click the LSF Split VQ subsystem.

   The subsystem opens, and you see the three Vector Quantizer Encoder blocks used to implement the split vector quantization method.
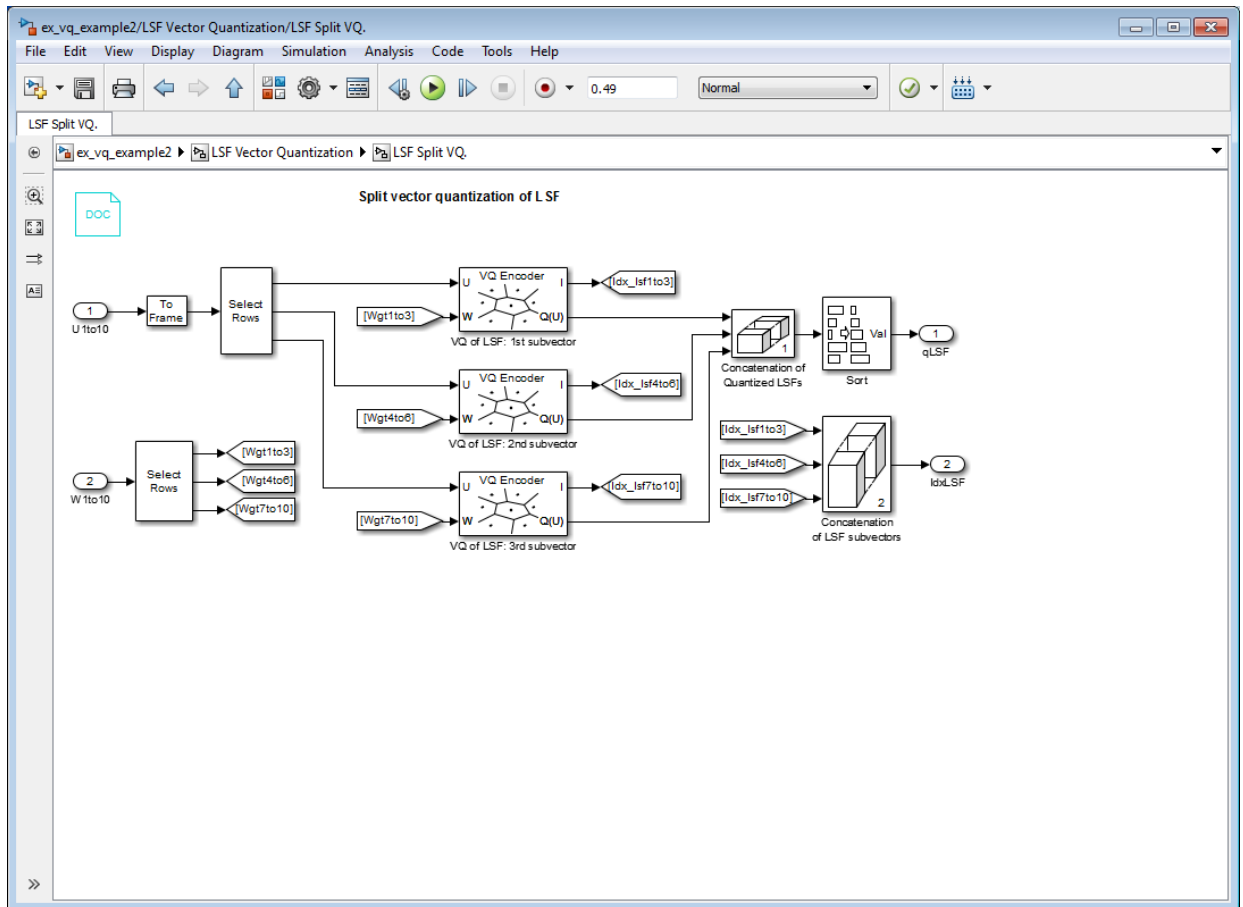
This subsystem divides each vector of 10 line spectral frequencies (LSFs), which represent your reflection coefficients, into three LSF subvectors. Each of these subvectors is sent to a separate vector quantizer. This method is called split vector quantization.

3 Double-click the VQ of LSF: 1st subvector block.

The Block Parameters: VQ of LSF: 1st subvector dialog box opens.
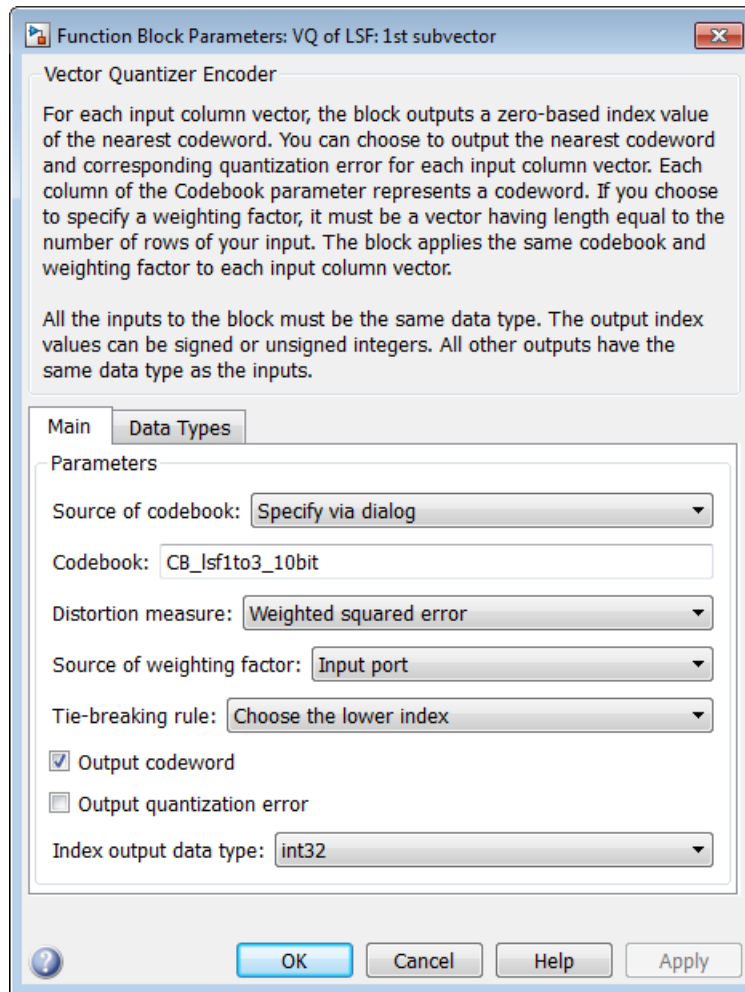
The variable `CB_lsf1to3_10bit` is the codebook for the subvector that contains the first three elements of the LSF vector. It is a 3-by-1024 matrix, where 3 is the number of elements in each codeword and 1024 is the number of codewords in the codebook. Because $2^{10} = 1024$, it takes 10 bits to quantize this first subvector. Similarly, a 10-bit vector quantizer is applied to the second and third subvectors,

which contain elements 4 to 6 and 7 to 10 of the LSF vector, respectively. Therefore, it takes 30 bits to quantize all three subvectors.

> **Note:** If you used the vector quantization method to quantize your reflection coefficients, you would need 2_{30} or 1.0737e9 codebook values to achieve the same degree of accuracy as the split vector quantization method.

4 In your model file, double-click the Autocorrelation block and set the **Maximum non-negative lag (less than input length)** parameter to 10. Click **OK**.

This parameter controls the number of linear polynomial coefficients (LPCs) that are input to the split vector quantization method.

5 Double-click the LPC to/from RC block that is connected to the input of the LSF Vector Quantization subsystem. Clear the **Output normalized prediction error power** check box. Click **OK**.

6 Double-click the LSF/LSP to LPC Conversion block and set the **Input** parameter to LSF in range (0 to pi). Click **OK**.

7 Double-click the LPC to/from RC block that is connected to the output of the LSF/ LSP to LPC Conversion block. Set the **Type of conversion** parameter to LPC to RC, and clear the **Output normalized prediction error power** check box. Click **OK**.

8 Run your model.
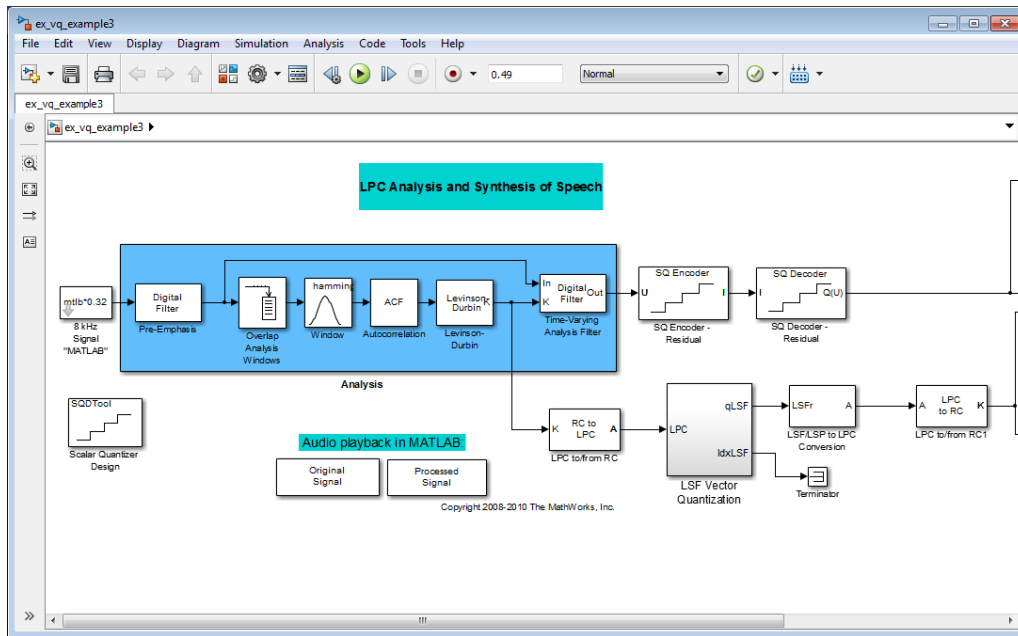
9 Double-click the Original Signal and Processed Signal blocks to listen to both the original and the processed signal.

There is no perceptible difference between the two. Quantizing your reflection coefficients using a split vector quantization method produced good quality speech without much distortion.

You have now used the split vector quantization method to quantize your reflection coefficients. The vector quantizers in the LSF Vector Quantization subsystem use 30 bits to quantize a frame containing 80 reflection coefficients. The bit rate of a quantization system is calculated as (bits per frame)*(frame rate).

In this example, the bit rate is [(80 residual samples/frame)*(7 bits/sample) + (30 bits/frame)]*(100 frames/second), or 59 kbits per second. This is less than 64.4 kbits per second, the bit rate of the scalar quantization system. However, the quality of the speech signal did not degrade. If you want to further reduce the bit rate of your system, you can use the vector quantization method to quantize the residual signal.

Fixed-Point Filter Design

In this section...
"Overview of Fixed-Point Filters" on page 9-78
"Data Types for Filter Functions" on page 9-78
"Floating-Point to Fixed-Point Filter Conversion" on page 9-79
"Create an FIR Filter Using Integer Coefficients" on page 9-88
"Fixed-Point Filtering in Simulink" on page 9-104

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, DSP System 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 arithmetic, see "Arithmetic Operations".

Data Types for Filter Functions

- "Data Type Support" on page 9-78
- "Fixed Data Type Support" on page 9-79
- "Single Data Type Support" on page 9-79

Data Type Support

There are three different data types supported in DSP System Toolbox software:

- Fixed — Requires Fixed-Point Designer and is supported by packages listed in "Fixed Data Type Support" on page 9-79.
- Double — Double precision, floating point and is the default data type for DSP System 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 9-79.

Fixed Data Type Support

To use fixed data type, you must have Fixed-Point Designer. 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';
```

Floating-Point to Fixed-Point Filter Conversion

- "Process Overview" on page 9-80
- "Design the Filter" on page 9-80
- "Quantize the Coefficients" on page 9-81
- "Dynamic Range Analysis" on page 9-84
- "Compare Magnitude Response and Magnitude Response Estimate" on page 9-85

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.

Design the Filter

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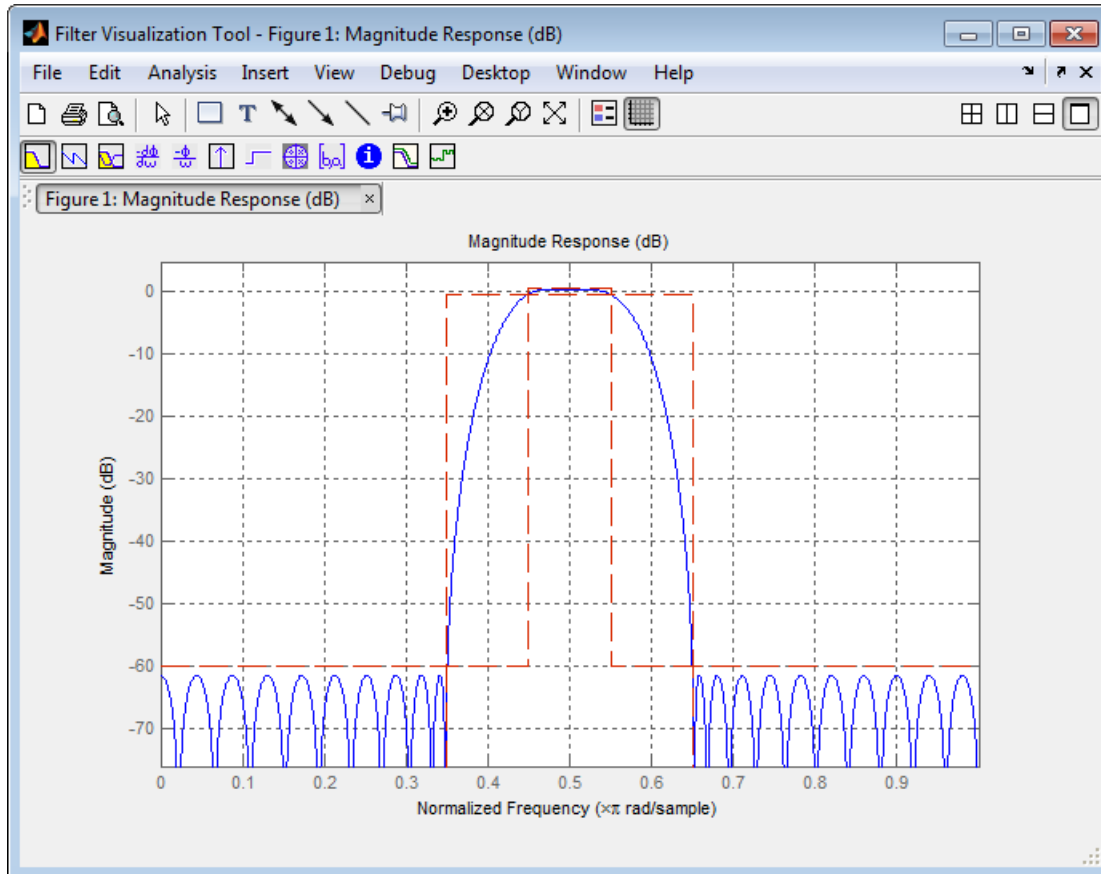
where the passband is from .45 to .55 of normalized frequency, the amount of ripple acceptable in the passband is 1 dB, the first stopband is from 0 to .35 (normalized), the second stopband is from .65 to 1 (normalized), and both stopbands provide 60 dB of attenuation.

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.

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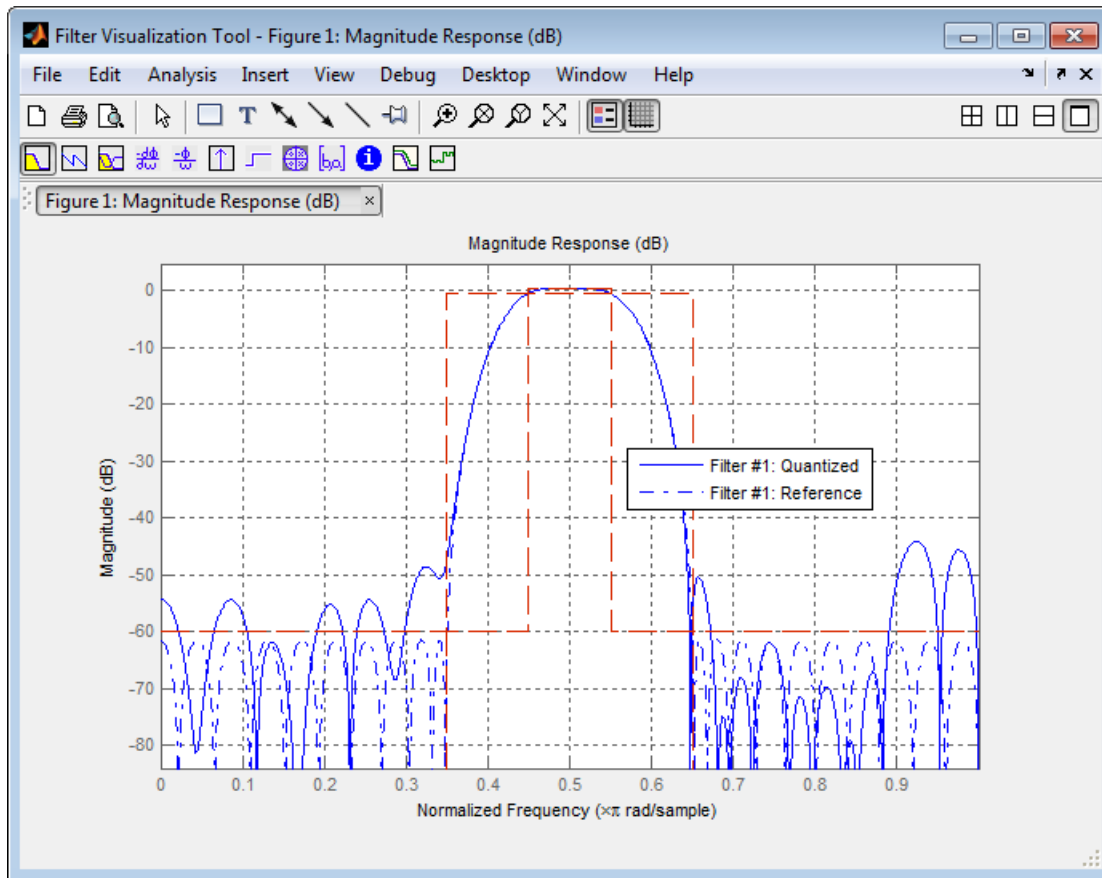
As the figure shows, the filter design constraints are not met. The attenuation is not complete, and there is noise at the edges of the stopbands. You can experiment with different coefficient word lengths if you like. For this example, however, the word length of 12 is sufficient.

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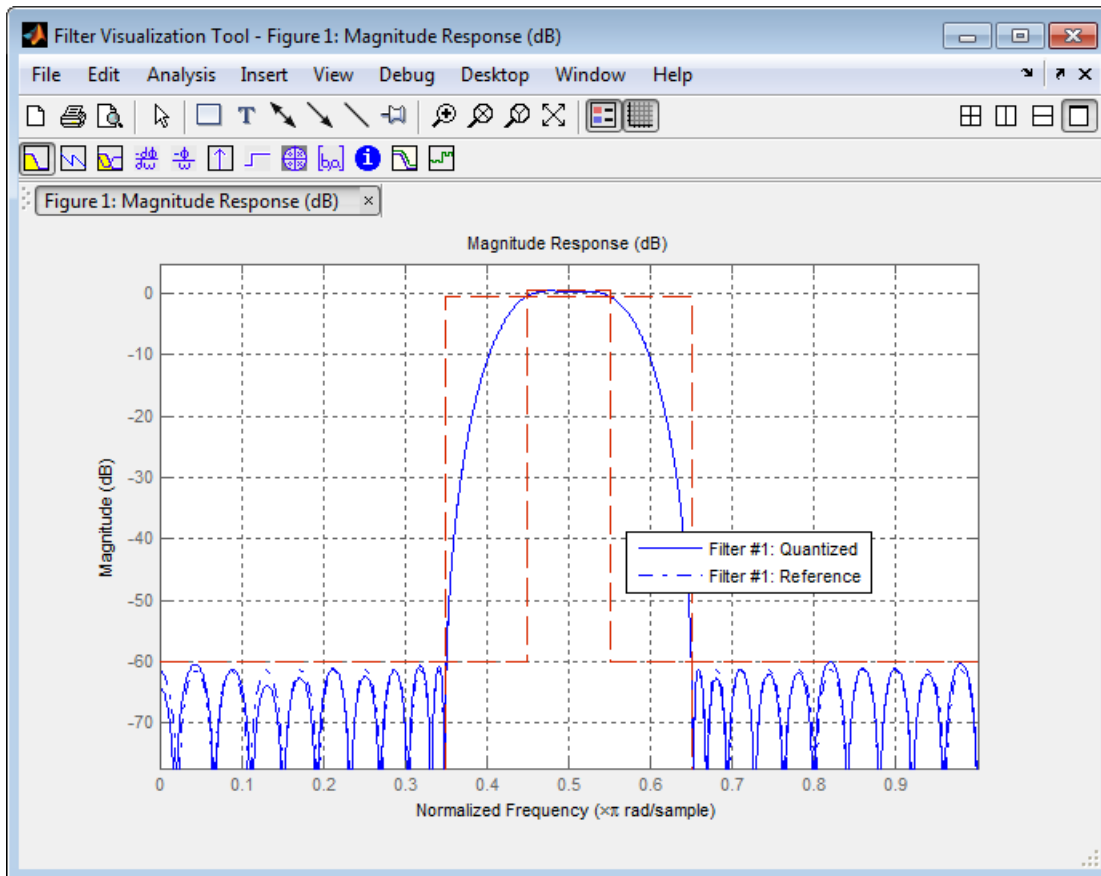
Now that the coefficient word length is set, there are other data width constraints that might require attention. Type the following at the MATLAB command prompt:

```
>> 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.

Compare Magnitude Response and Magnitude Response Estimate

You can use the `fvtool` GUI to 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.

View the Magnitude Response Estimate

Begin by designing a simple lowpass filter using the command.

```
h = design(fdesign.lowpass, 'butter','SOSScaleNorm','Linf');
```

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.
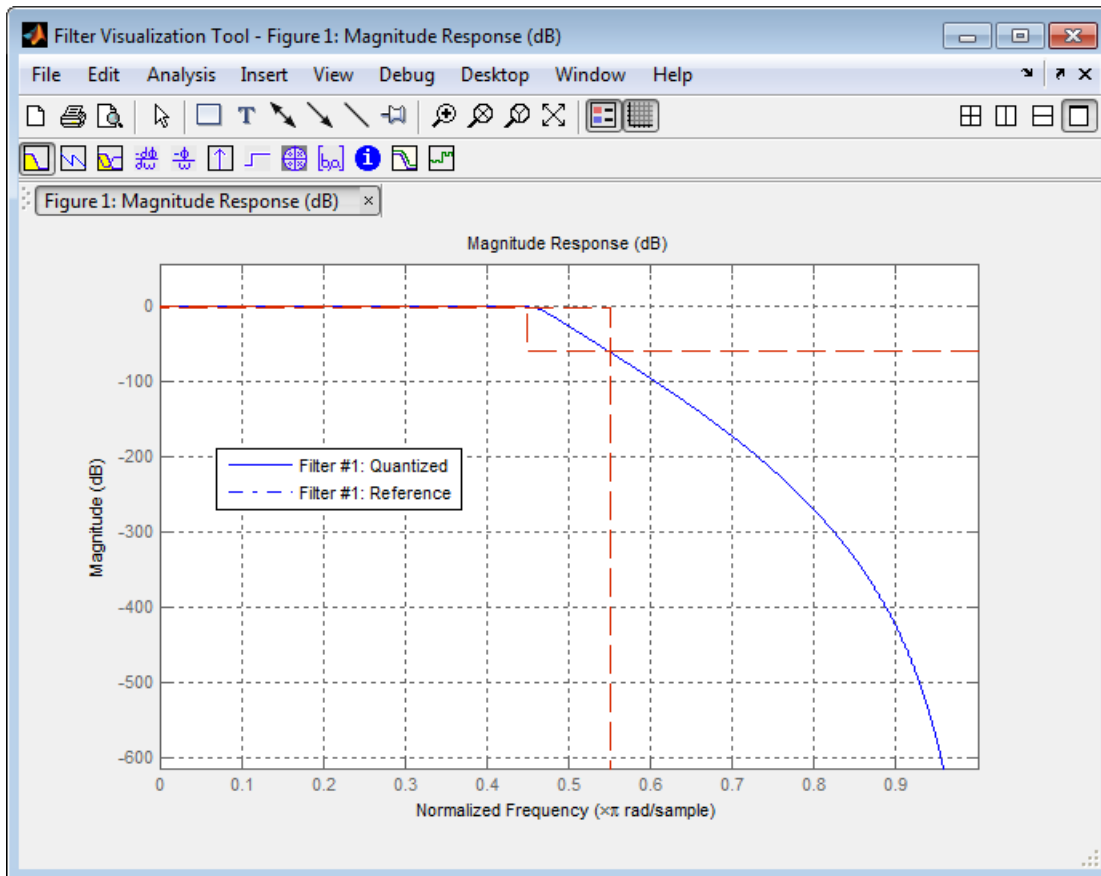
However if you look at the **Magnitude Response Estimate** plot from the **Analysis** menu, you will see that the actual filter created may not perform nearly as well as indicated by the **Magnitude Response** plot.
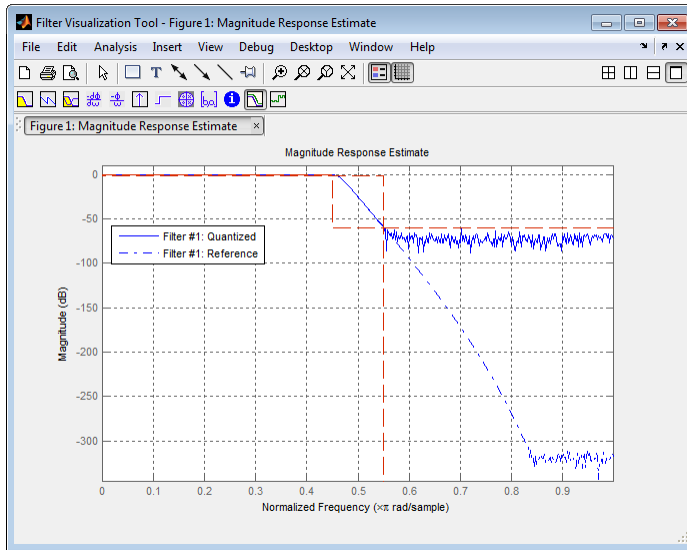
This is because by using the noise-based method of the **Magnitude Response Estimate**, you estimate the complex frequency response for your filter as determined by applying a noise- like signal to the filter input. **Magnitude Response Estimate** uses the Monte Carlo trials to generate a noise signal 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 "Analyze Filters with a Noise-Based Method" on page 4-28 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.

Create an FIR Filter Using Integer Coefficients

Review of Fixed-Point Numbers

Terminology of Fixed-Point Numbers

DSP System Toolbox functions assume 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:
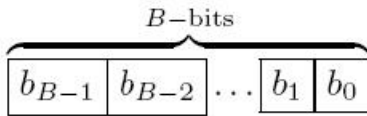
The register has a WordLength of B, or in other words it has B bits. The bits are numbered from left to right from 0 to B-1. The most significant bit (MSB) is the leftmost bit, b_{B-1}. The least significant bit is the right-most bit, b_0. You can think of the FracLength as a quantity specifying how 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:

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:

$$
\boxed{b_3} \ \boxed{b_2} \ \boxed{b_1} \ \boxed{b_0}
$$
$$
{-2^{3-L}} \ {2^{2-L}} \ {2^{1-L}} \ {2^{-L}}
$$

$$
\boxed{b_3} \ \boxed{b_2}
$$
$$
{-2^{3-L}} \ {2^{2-L}}
$$
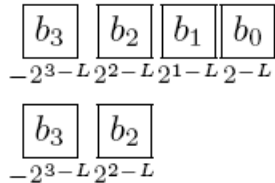
This means that the FracLength has changed from L to $L - 2$.

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

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.

Define the Filter Coefficients

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

```
b = rcosdesign(.25, 12.5, 8, '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.

Build 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')
```
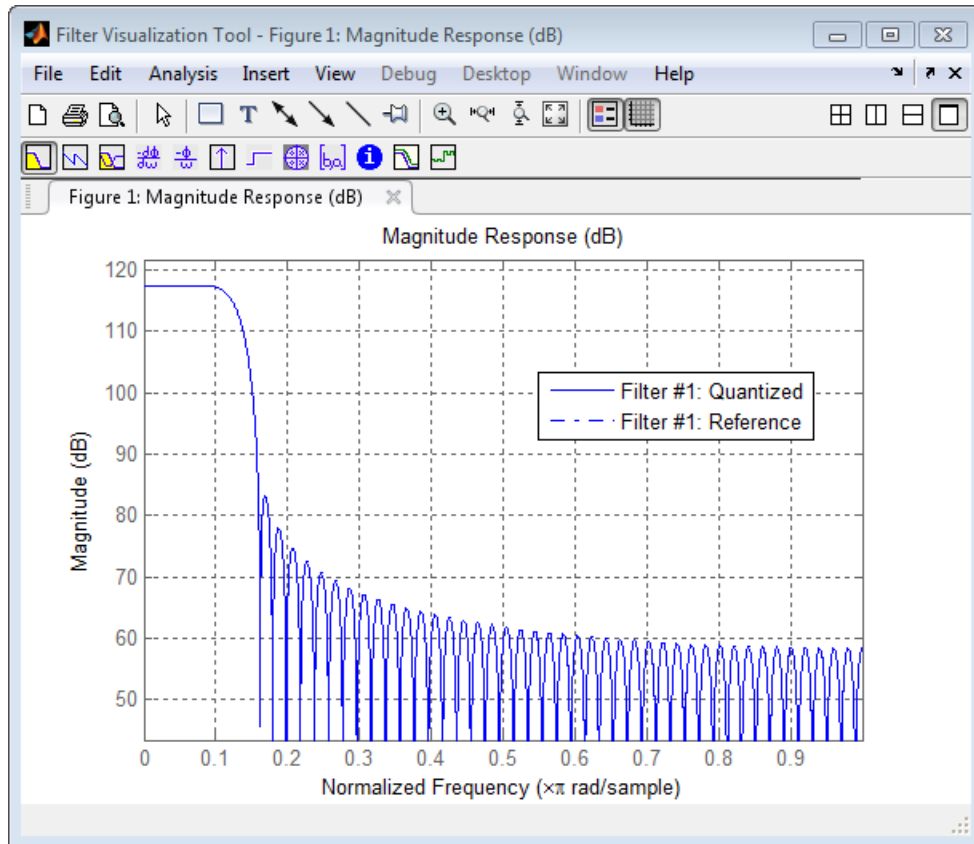

This shows a large gain of 117 dB in the passband, which is due to the large values of the coefficients— this will cause the output of the filter to be much larger than the input. A method of addressing this will be discussed in the following sections.

Set the 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.

Create 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);
```

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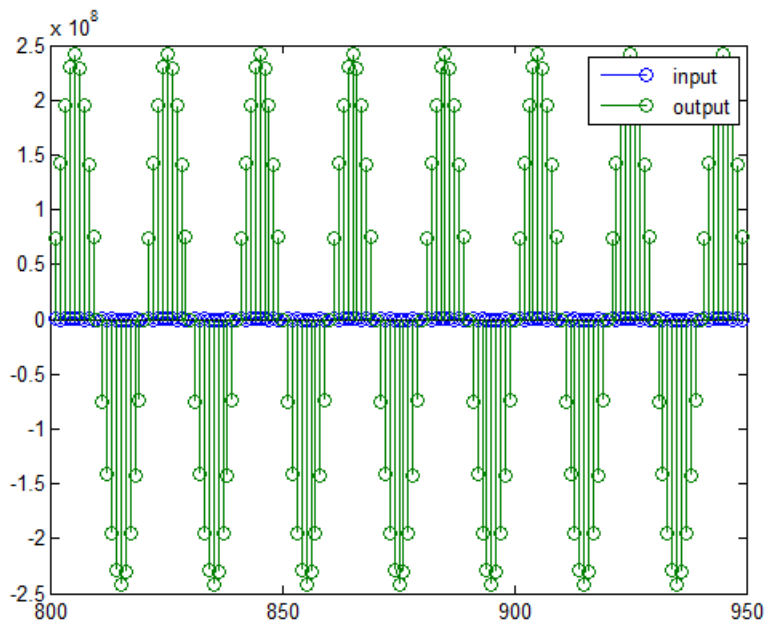
It is difficult to compare the two signals in this figure because of the large difference in scales. This is due to the large gain of the filter, so you will need to compensate for the filter gain:

```
stem(n(idx)', [2^18*xscext, yscext]);
axis([800 950 -5e8 5e8]);
legend('scaled input', 'output');
```
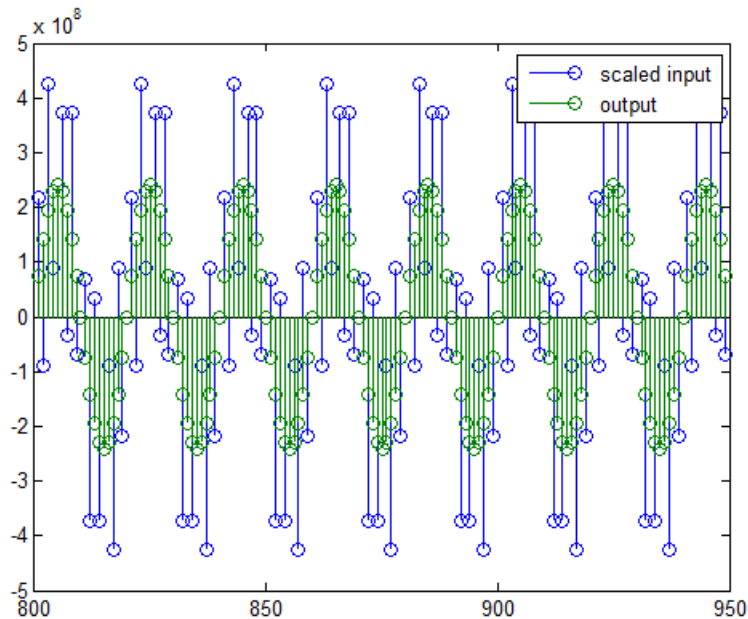

You can see how the signals compare much more easily once the scaling has been done, as seen in the above figure.

Truncate 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 9-88the 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');
```
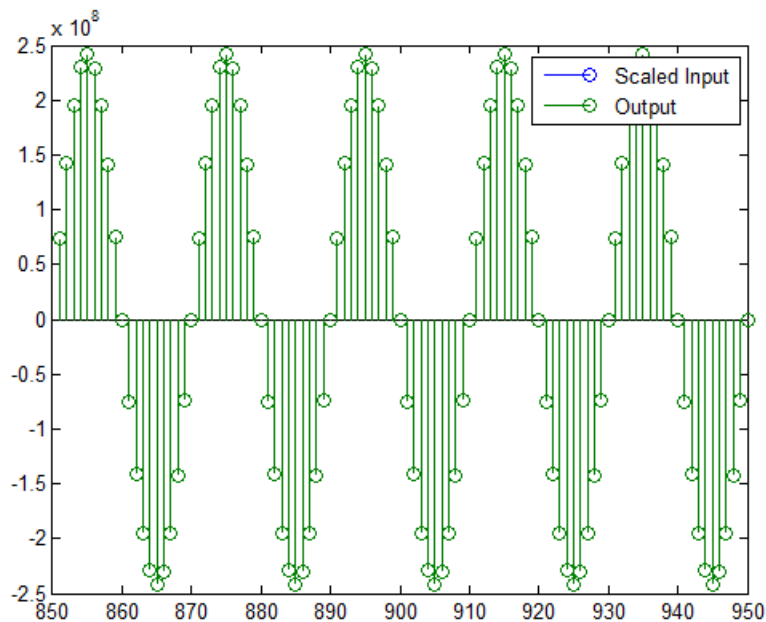

Scale the Output

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');
```
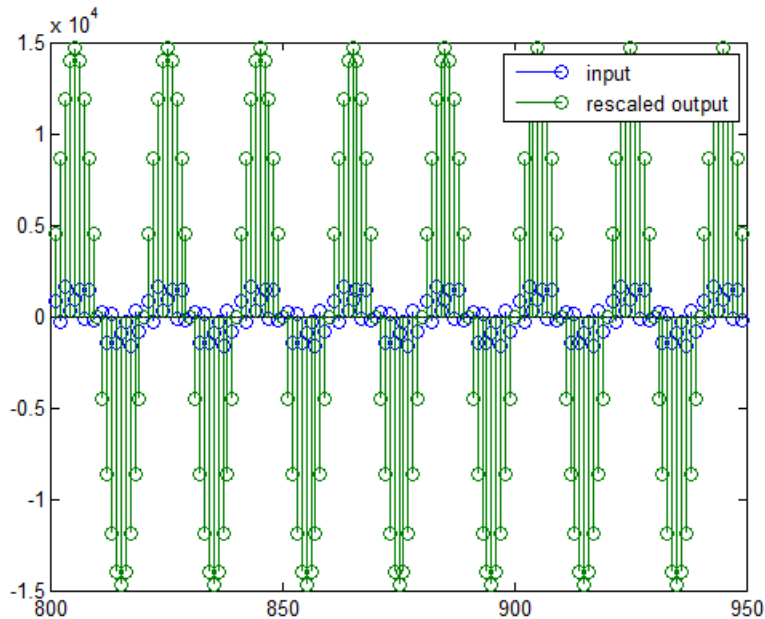

This plot shows that the output is still larger than the input. If you had done the filtering in double-precision floating-point, this would not be the case— because here more bits are being used for the output than for the input, so the MSBs are weighted differently. You can see this another way by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

```
[H,w] = freqz(h);
plot(w/pi, 20*log10(2^(-14)*abs(H)));
```
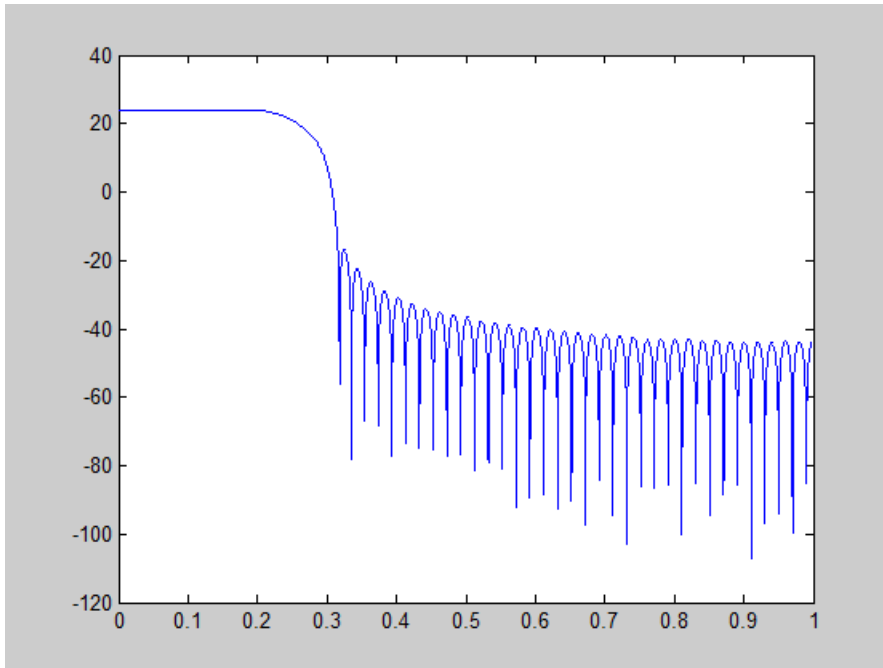
This plot shows that the passband gain is still above 0 dB.

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');
```
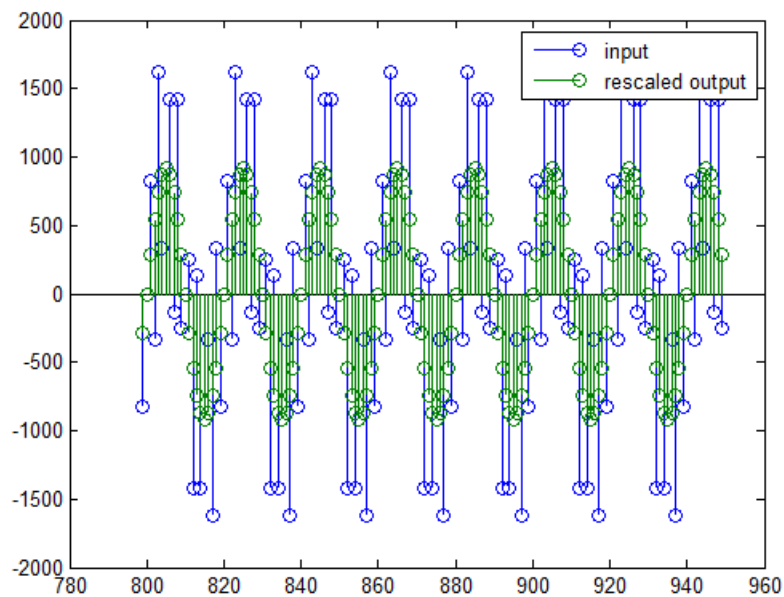
This operation is equivalent to scaling the filter gain down by 2^(-18).

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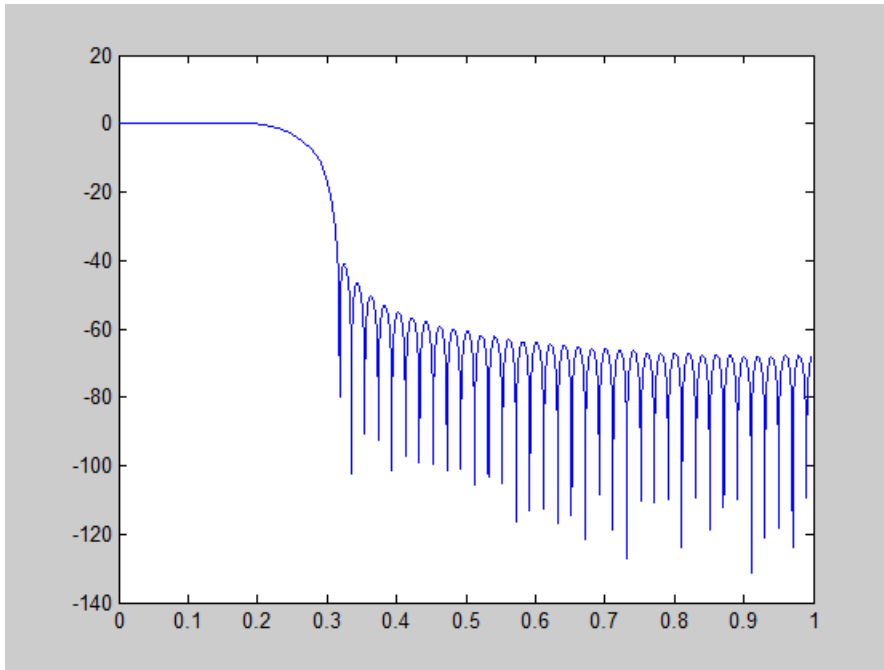
The above plot shows a 0 dB gain in the passband, as desired.

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
```

Configure Filter Parameters to Work with Integers Using the set2int Method

Set the 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.

Reinterpret 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');
```
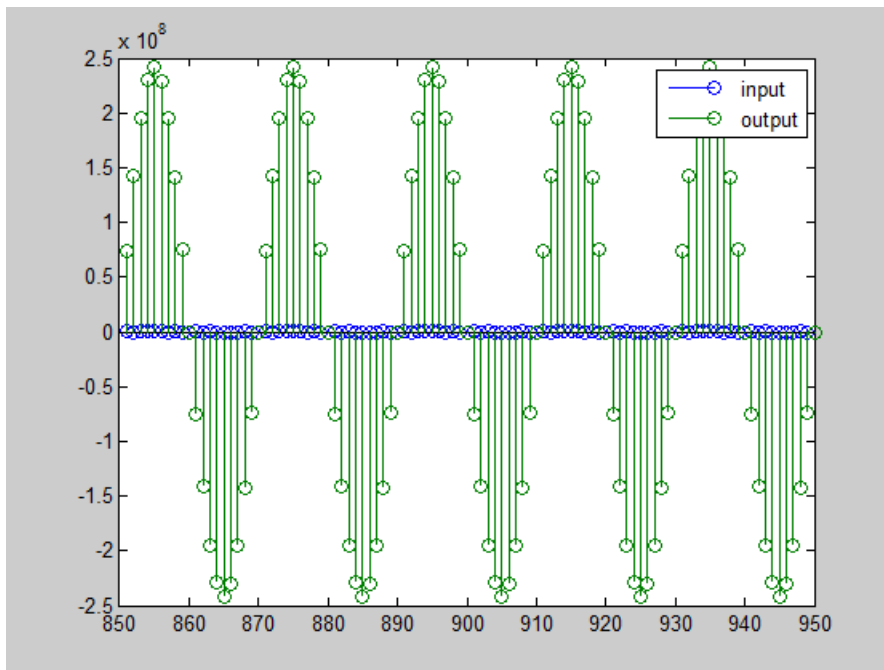

Once again, the plot shows that the input and output are at different scales. In order to scale the output so that the signals can be compared more easily in a plot, you will need

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');
```
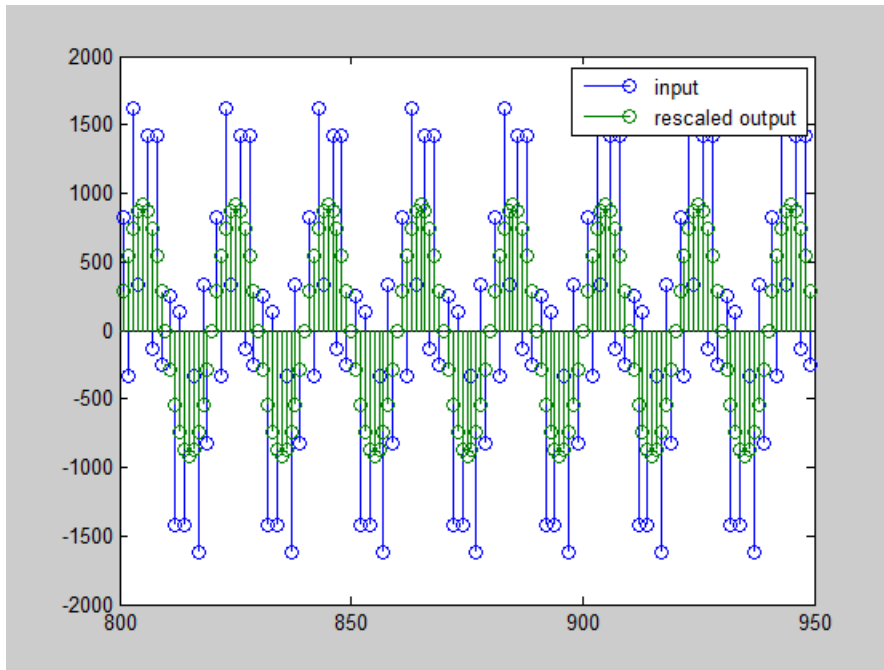

This final plot shows the filtered data re-scaled to match the input scale.

Fixed-Point Filtering in Simulink

- "Fixed-Point Filtering Blocks" on page 9-105
- "Filter Implementation Blocks" on page 9-105

• "Filter Design and Implementation Blocks" on page 9-105

Fixed-Point Filtering Blocks

The following DSP System Toolbox blocks enable you to design and/or realize a variety of fixed-point filters:

• `CIC Decimation`
• `CIC Interpolation`
• `Filter Realization Wizard`
• `FIR Decimation`
• `FIR Interpolation`
• `Two-Channel Analysis Subband Filter`
• `Two-Channel Synthesis Subband Filter`

Filter Implementation Blocks

The FIR Decimation, FIR Interpolation, Two-Channel Analysis Subband Filter, and Two-Channel Synthesis Subband Filter blocks are all implementation blocks. They allow you to implement filters for which you already know the filter coefficients.

Filter Design and Implementation Blocks

The Filter Realization Wizard block invokes part of the Filter Design and Analysis Tool from Signal Processing Toolbox software. This block allows you both to design new filters and to implement filters for which you already know the coefficients. In its implementation stage, the Filter Realization Wizard creates a filter realization using Sum, Gain, and Delay blocks. You can use this block to design and/or implement numerous types of fixed-point and floating-point single-channel filters. See the `Filter Realization Wizard` reference page for more information about this block.

The CIC Decimation and CIC Interpolation blocks allow you to design and implement Cascaded Integrator-Comb filters. See their block reference pages for more information.

Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters

Fixed-point FIR filters are commonly implemented on digital signal processors, FPGAs, and ASICs. A fixed-point filter uses fixed-point arithmetic and is represented by an equation with fixed-point coefficients. If the accumulator and output of the FIR filter do not have sufficient bits to represent their data, overflow occurs and distorts the signal. Use these two rules to determine FIR filter precision settings automatically. The aim is to minimize resource utilization (memory/storage and processing elements) while avoiding overflow. Because the rules are optimized based on the input precision, coefficient precision, and the coefficient values, the FIR filter must have nontunable coefficients.

The precision rules define the minimum and the maximum values of the FIR filter output. To determine these values, perform min/max analysis on the FIR filter coefficients.

Output Limits for FIR Filters

FIR filter is defined by:

$$y[n] = \sum_{k=0}^{N-1} h_k x[n-k]$$

- $x[n]$ is the input signal.
- $y[n]$ is the output signal.
- h_k is the kth filter coefficient.
- N is the length of the filter.

Output Limits for FIR Filters with Real Input and Real Coefficients

Let the minimum value of the input signal be X_{min}, where $X_{min} \leq 0$, and the maximum value be X_{max}, where $X_{max} \geq 0$. The minimum output occurs when you multiply the

positive coefficients by X_{min} and the negative coefficients by X_{max}. Similarly, the maximum output occurs when you multiply the positive coefficients by X_{max} and the negative coefficients by X_{min}.

If the sum of all the positive coefficients is

$$G^+ = \sum_{k=0, h_k > 0}^{N-1} h_k$$

and the sum of all the negative coefficients is denoted as

$$G^- = \sum_{k=0, h_k < 0}^{N-1} h_k$$

then you can express the minimum output of the filter as

$$Y_{\min} = G^+ X_{\min} + G^- X_{\max}$$

and the maximum output of the filter as

$$Y_{\max} = G^+ X_{\max} + G^- X_{\min}$$

Therefore, the output of the filter lies in the interval $[Y_{min}, Y_{max}]$.

Complex Filter Convolution Equations

You can define a complex filter (complex inputs and complex coefficients) in terms of the real and imaginary parts of its signals and coefficients:

$$\mathrm{Re}(y[n]) = \sum_{k=0}^{N-1} \mathrm{Re}(h_k)\mathrm{Re}(x[n-k]) - \sum_{k=0}^{N-1} \mathrm{Im}(h_k)\mathrm{Im}(x[n-k])$$

$$\mathrm{Im}(y[n]) = \sum_{k=0}^{N-1} \mathrm{Re}(h_k)\mathrm{Im}(x[n-k]) + \sum_{k=0}^{N-1} \mathrm{Im}(h_k)\mathrm{Re}(x[n-k])$$

The complex filter is decomposed into four real filters as depicted in the signal flow diagram. Each signal is annotated with an interval denoting its range.
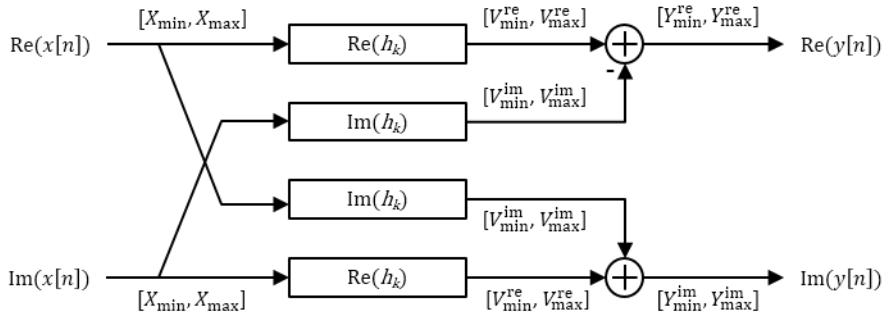
Output Limits for FIR Filters with Complex Input and Complex Coefficients

You can extend the real filter min/max analysis to complex filters. Assume that both the real and imaginary parts of the input signal lie in the interval $[X_{min}, X_{max}]$.

The complex filter contains two instances of the filter $\text{Re}(h_k)$. Both filters have the same input range and therefore the same output range in the interval $[V^{re}_{min}, V^{re}_{max}]$. Similarly, the complex filter contains two instances of the filter $\text{Im}(h_k)$. Both filters have the same output range in the interval $[V^{im}_{min}, V^{im}_{max}]$.

Based on the min/max analysis of real filters, you can express V^{re}_{min}, V^{re}_{max}, V^{im}_{min}, and V^{im}_{max} as:

$$V^{re}_{\min} = G^+_{re}X_{\min} + G^-_{re}X_{\max}$$

$$V^{re}_{\max} = G^+_{re}X_{\max} + G^-_{re}X_{\min}$$

$$V^{im}_{\min} = G^+_{im}X_{\min} + G^-_{im}X_{\max}$$

$$V^{im}_{\max} = G^+_{im}X_{\max} + G^-_{im}X_{\min}$$

- G^+_{re} is the sum of the positive real parts of h_k, given by

$$G^+_{re} = \sum_{k=0, \text{Re}(h_k)>0}^{N-1} \text{Re}(h_k)$$

- G_{re}^{-} is the sum of the negative real parts of h_k, given by

$$G_{re}^{-} = \sum_{k=0, \text{Re}(h_k)<0}^{N-1} \text{Re}(h_k)$$

- G_{im}^{+} is the sum of the positive imaginary parts of h_k, given by

$$G_{im}^{+} = \sum_{k=0, \text{Im}(h_k)>0}^{N-1} \text{Im}(h_k)$$

- G_{im}^{-} is the sum of the negative imaginary parts of h_k, given by

$$G_{im}^{-} = \sum_{k=0, \text{Im}(h_k)<0}^{N-1} \text{Im}(h_k)$$

The minimum and maximum values of the real and imaginary parts of the output are:

$$Y_{\min}^{re} = V_{\min}^{re} - V_{\max}^{im}$$
$$Y_{\max}^{re} = V_{\max}^{re} - V_{\min}^{im}$$
$$Y_{\min}^{im} = V_{\min}^{re} + V_{\min}^{im}$$
$$Y_{\max}^{im} = V_{\max}^{re} + V_{\max}^{im}$$

The worst-case minimum and maximum on either the real or imaginary part of the output is given by

$$Y_{\min} = \min(Y_{\min}^{re}, Y_{\min}^{im})$$
$$Y_{\max} = \max(Y_{\max}^{re}, Y_{\max}^{im})$$

Fixed-Point Precision Rules

The fixed-point precision rules define the output word length and fraction length of the filter in terms of the accumulator word length and fraction length.
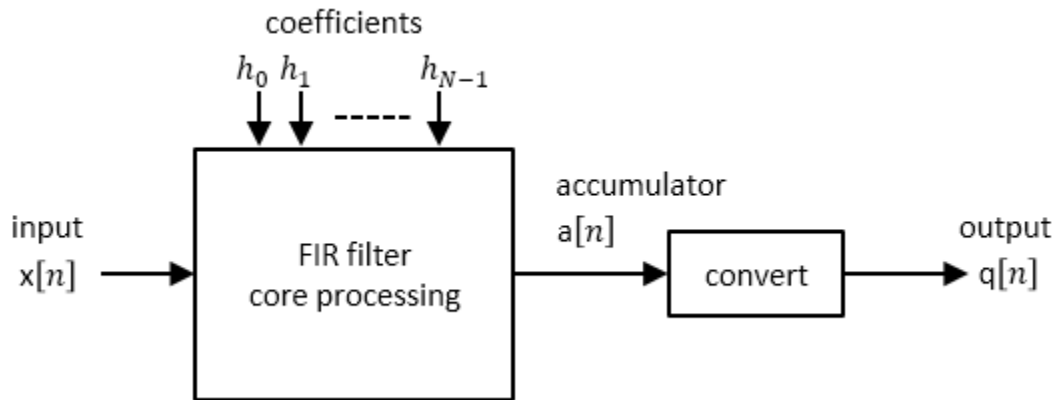
Full-Precision Accumulator Rule

Assume that the input is a signed or unsigned fixed-point signal with word length W_x and fraction length F_x. Also assume that the coefficients are signed or unsigned fixed-point values with fraction length F_h. You can now define full precision as the fixed-point settings that minimize the word length of the accumulator while avoiding overflow or any loss of precision.

- The accumulator fraction length is equal to the product fraction length, which is the sum of the input and coefficient fraction lengths.

$$F_a = F_x + F_h$$

- If $Y_{min} = 0$, then the accumulator is unsigned with word length

$$W_a = \left\lceil \log_2 (Y_{\max} 2^{F_a} + 1) \right\rceil$$

If $Y_{min} > 0$, then the accumulator is signed with word length

$$W_a = \left\lceil \log_2 (\max(-Y_{\min} 2^{F_a}, Y_{\max} 2^{F_a} + 1)) \right\rceil + 1$$

The ceil operator rounds to the nearest integer towards $+\infty$.

Output Same Word Length as Input Rule

This rule sets the output word length to be the same as the input word length. Then, it adjusts the fraction length to avoid overflow. W_q is the output word length and F_q is the output fraction length.

Truncate the accumulator to make the output word length same as the input word length.

$$W_q = W_x$$

.

Set the output fraction length F_q to

$$F_q = F_a - (W_a - W_x)$$

.

Polyphase Interpolators and Decimators

You can extend these rules to polyphase FIR interpolators and decimators.

FIR Interpolators

Treat each polyphase branch of the FIR interpolator as a separate FIR filter. The output data type of the FIR interpolator is the worst-case data type of all the polyphase branches.

FIR Decimators

For decimators, the polyphase branches add up at the output. Hence, the output data type is computed as if it were a single FIR filter with all the coefficients of all the polyphase branches.

More About

·   "Fixed-Point Concepts and Terminology" on page 9-4
·   "Fixed-Point Filter Design" on page 9-78

10

C Code Generation

Learn how to generate code for signal processing applications.

Understanding C Code Generation

In this section...
"C Code Generation with the Simulink Coder Product" on page 10-2
"Highly Optimized Generated ANSI C Code" on page 10-3

C Code Generation with the Simulink Coder Product

You can use the DSP System Toolbox, Simulink Coder, and Embedded Coder® products together to generate code that you can use to implement your model for a practical application. For instance, you can create an executable from your Simulink model to run on a target chip.

This chapter introduces you to the basic concepts of code generation using these tools.

Shared Library Dependencies

In general, the code you generate from DSP System Toolbox blocks is portable ANSI C code. After you generate the code, you can deploy it on another machine. For more information on how to do so, see "Relocate Code to Another Development Environment" in the Simulink Coder documentation.

There are a few DSP System Toolbox blocks that generate code with limited portability. These blocks use precompiled shared libraries, such as DLLs, to support I/O for specific types of devices and file formats. To find out which blocks use precompiled shared libraries, open the DSP System Toolbox Block Support Table. You can identify blocks that use precompiled shared libraries by checking the footnotes listed in the **Code Generation Support** column of the table. All blocks that use shared libraries have the following footnote:

```
Host computer only. Excludes Simulink Desktop Real-Time code
generation.
```

Simulink Coder provides functions to help you set up and manage the build information for your models. For example, one of the functions that Simulink Coder provides is `getNonBuildFiles`. This function allows you to identify the shared libraries required by blocks in your model. If your model contains any blocks that use precompiled shared libraries, you can install those libraries on the target system. The folder that you install

the shared libraries in must be on the system path. The target system does not need to have MATLAB installed, but it does need to be supported by MATLAB.

Highly Optimized Generated ANSI C Code

All DSP System Toolbox blocks generate highly optimized ANSI C code. This C code is often suitable for embedded applications, and includes the following optimizations:

- **Function reuse (run-time libraries)** — The generated code reuses common algorithmic functions via calls to shared utility functions. Shared utility functions are highly optimized ANSI/ISO C functions that implement core algorithms such as FFT and convolution.

- **Parameter reuse (Simulink Coder run-time parameters)** — In many cases, if there are multiple instances of a block that all have the same value for a specific parameter, each block instance points to the same variable in the generated code. This process reduces memory requirements.

- **Blocks have parameters that affect code optimization** — Some blocks, such as the Sine Wave block, have parameters that enable you to optimize the simulation for memory or for speed. These optimizations also apply to code generation.

- **Other optimizations** — Use of contiguous input and output arrays, reusable inputs, overwritable arrays, and inlined algorithms provide smaller generated C code that is more efficient at run time.

Functions and System Objects Supported for C Code Generation

If you have a MATLAB Coder license, you can generate C and C++ code from MATLAB code that contains DSP System Toolbox functions and System objects. For more information about C and C++ code generation from MATLAB code, see the MATLAB Coder documentation. For more information about generating code from System objects, see "System Objects in MATLAB Code Generation".

The following DSP System Toolbox functions and System objects are supported for C and C++ code generation from MATLAB code.

Name	Remarks and Limitations
Estimation	
dsp.BurgAREstimator	"System Objects in MATLAB Code Generation"
dsp.BurgSpectrumEstimator	"System Objects in MATLAB Code Generation"
dsp.CepstralToLPC	"System Objects in MATLAB Code Generation"
dsp.CrossSpectrumEstimator	"System Objects in MATLAB Code Generation"
dsp.LevinsonSolver	"System Objects in MATLAB Code Generation"
dsp.LPCToAutocorrelation	"System Objects in MATLAB Code Generation"
dsp.LPCToCepstral	"System Objects in MATLAB Code Generation"
dsp.LPCToLSF	"System Objects in MATLAB Code Generation"
dsp.LPCToLSP	"System Objects in MATLAB Code Generation"
dsp.LPCToRC	"System Objects in MATLAB Code Generation"
dsp.LSFToLPC	"System Objects in MATLAB Code Generation"
dsp.LSPToLPC	"System Objects in MATLAB Code Generation"
dsp.RCToAutocorrelation	"System Objects in MATLAB Code Generation"
dsp.RCToLPC	"System Objects in MATLAB Code Generation"
dsp.SpectrumEstimator	"System Objects in MATLAB Code Generation"
dsp.TransferFunctionEstimator	"System Objects in MATLAB Code Generation"
Filters	
ca2tf	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Name	Remarks and Limitations
cl2tf	All inputs must be constant. Expressions or variables are allowed if their values do not change.
dsp.AdaptiveLatticeFilter	"System Objects in MATLAB Code Generation"
dsp.AffineProjectionFilter	"System Objects in MATLAB Code Generation"
dsp.AllpassFilter	"System Objects in MATLAB Code Generation" The System object supports code generation only when the `Structure` property is set to `Minimum multiplier` or `Lattice`.
dsp.AllpoleFilter	• "System Objects in MATLAB Code Generation" • Only the `Denominator` property is tunable for code generation.
dsp.BiquadFilter	"System Objects in MATLAB Code Generation"
dsp.CICCompensationDecimator	"System Objects in MATLAB Code Generation"
dsp.CICCompensationInterpolator	"System Objects in MATLAB Code Generation"
dsp.CICDecimator	"System Objects in MATLAB Code Generation"
dsp.CICInterpolator	"System Objects in MATLAB Code Generation"
dsp.Differentiator	"System Objects in MATLAB Code Generation"
dsp.FarrowRateConverter	"System Objects in MATLAB Code Generation"
dsp.FastTransversalFilter	"System Objects in MATLAB Code Generation"
dsp.FilterCascade	• You cannot generate code directly from `dsp.FilterCascade`. You can use the `generateFilteringCode` method to generate a MATLAB function. You can generate C/C++ code from this MATLAB function. • "System Objects in MATLAB Code Generation"
dsp.FilteredXLMSFilter	"System Objects in MATLAB Code Generation"
dsp.FIRDecimator	"System Objects in MATLAB Code Generation"

Name	Remarks and Limitations
dsp.FIRFilter	• "System Objects in MATLAB Code Generation" • Only the `Numerator` property is tunable for code generation.
dsp.FIRHalfbandDecimator	"System Objects in MATLAB Code Generation"
dsp.FIRHalfbandInterpolator	"System Objects in MATLAB Code Generation"
dsp.FIRInterpolator	"System Objects in MATLAB Code Generation"
dsp.FIRRateConverter	"System Objects in MATLAB Code Generation"
dsp.FrequencyDomainAdaptiveFilter	"System Objects in MATLAB Code Generation"
dsp.HighpassFilter	"System Objects in MATLAB Code Generation"
dsp.IIRFilter	• Only the `Numerator` and `Denominator` properties are tunable for code generation. • "System Objects in MATLAB Code Generation"
dsp.IIRHalfbandDecimator	"System Objects in MATLAB Code Generation"
dsp.IIRHalfbandInterpolator	"System Objects in MATLAB Code Generation"
dsp.KalmanFilter	"System Objects in MATLAB Code Generation"
dsp.LMSFilter	"System Objects in MATLAB Code Generation"
dsp.LowpassFilter	"System Objects in MATLAB Code Generation"
dsp.RLSFilter	"System Objects in MATLAB Code Generation"
dsp.SampleRateConverter	"System Objects in MATLAB Code Generation"
dsp.SubbandAnalysisFilter	"System Objects in MATLAB Code Generation"
dsp.SubbandSynthesisFilter	"System Objects in MATLAB Code Generation"
dsp.VariableBandwidthFIRFilter	"System Objects in MATLAB Code Generation"
dsp.VariableBandwidthIIRFilter	"System Objects in MATLAB Code Generation"
firceqrip	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Name	Remarks and Limitations
fireqint	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firgr	• All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.
firhalfband	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firlpnorm	• All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.
firminphase	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firnyquist	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firpr2chfb	All inputs must be constant. Expressions or variables are allowed if their values do not change.
ifir	All inputs must be constant. Expressions or variables are allowed if their values do not change.
iircomb	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Name	Remarks and Limitations
iirgrpdelay	• All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.
iirlpnorm	• All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.
iirlpnormc	• All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.
iirnotch	All inputs must be constant. Expressions or variables are allowed if their values do not change.
iirpeak	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tf2ca	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tf2cl	All inputs must be constant. Expressions or variables are allowed if their values do not change.
Filter Design	
designMultirateFIR	The inputs to the function must be constants
Math Operations	
dsp.ArrayVectorAdder	"System Objects in MATLAB Code Generation"
dsp.ArrayVectorDivider	"System Objects in MATLAB Code Generation"

Name	Remarks and Limitations
dsp.ArrayVectorMultiplier	"System Objects in MATLAB Code Generation"
dsp.ArrayVectorSubtractor	"System Objects in MATLAB Code Generation"
dsp.CumulativeProduct	"System Objects in MATLAB Code Generation"
dsp.CumulativeSum	"System Objects in MATLAB Code Generation"
dsp.LDLFactor	"System Objects in MATLAB Code Generation"
dsp.LevinsonSolver	"System Objects in MATLAB Code Generation"
dsp.LowerTriangularSolver	"System Objects in MATLAB Code Generation"
dsp.LUFactor	"System Objects in MATLAB Code Generation"
dsp.Normalizer	"System Objects in MATLAB Code Generation"
dsp.UpperTriangularSolver	"System Objects in MATLAB Code Generation"
Quantizers	
dsp.ScalarQuantizerDecoder	"System Objects in MATLAB Code Generation"
dsp.ScalarQuantizerEncoder	"System Objects in MATLAB Code Generation"
dsp.VectorQuantizerDecoder	"System Objects in MATLAB Code Generation"
dsp.VectorQuantizerEncoder	"System Objects in MATLAB Code Generation"
Scopes	
dsp.SpectrumAnalyzer	This System object does not generate code. It is automatically declared as an *extrinsic* variable using the `coder.extrinsic` function.
dsp.TimeScope	This System object does not generate code. It is automatically declared as an *extrinsic* variable using the `coder.extrinsic` function.
Signal Management	
dsp.Counter	"System Objects in MATLAB Code Generation"
dsp.DelayLine	"System Objects in MATLAB Code Generation"
Signal Operations	
dsp.Convolver	"System Objects in MATLAB Code Generation"
dsp.DCBlocker	"System Objects in MATLAB Code Generation"
dsp.Delay	"System Objects in MATLAB Code Generation"

Name	Remarks and Limitations
dsp.DigitalDownConverter	"System Objects in MATLAB Code Generation"
dsp.DigitalUpConverter	"System Objects in MATLAB Code Generation"
dsp.Interpolator	"System Objects in MATLAB Code Generation"
dsp.NCO	"System Objects in MATLAB Code Generation"
dsp.PeakFinder	"System Objects in MATLAB Code Generation"
dsp.PhaseExtractor	"System Objects in MATLAB Code Generation"
dsp.PhaseUnwrapper	"System Objects in MATLAB Code Generation"
dsp.VariableFractionalDelay	"System Objects in MATLAB Code Generation"
dsp.VariableIntegerDelay	"System Objects in MATLAB Code Generation"
dsp.Window	• This object has no tunable properties for code generation. • "System Objects in MATLAB Code Generation"
dsp.ZeroCrossingDetector	"System Objects in MATLAB Code Generation"
Sinks	
audioDeviceWriter	"System Objects in MATLAB Code Generation"
dsp.AudioFileWriter	"System Objects in MATLAB Code Generation"
dsp.UDPSender	"System Objects in MATLAB Code Generation"
Sources	
dsp.AudioFileReader	"System Objects in MATLAB Code Generation"
dsp.SignalSource	"System Objects in MATLAB Code Generation"
dsp.SineWave	• This object has no tunable properties for code generation. • "System Objects in MATLAB Code Generation"
dsp.UDPReceiver	"System Objects in MATLAB Code Generation"
Statistics	
dsp.Autocorrelator	"System Objects in MATLAB Code Generation"
dsp.Crosscorrelator	"System Objects in MATLAB Code Generation"

Name	Remarks and Limitations
dsp.Histogram	• This object has no tunable properties for code generation. • "System Objects in MATLAB Code Generation"
dsp.Maximum	"System Objects in MATLAB Code Generation"
dsp.Mean	"System Objects in MATLAB Code Generation"
dsp.Median	"System Objects in MATLAB Code Generation"
dsp.Minimum	"System Objects in MATLAB Code Generation"
dsp.PeakToPeak	"System Objects in MATLAB Code Generation"
dsp.PeakToRMS	"System Objects in MATLAB Code Generation"
dsp.RMS	"System Objects in MATLAB Code Generation"
dsp.StandardDeviation	"System Objects in MATLAB Code Generation"
dsp.StateLevels	"System Objects in MATLAB Code Generation"
dsp.Variance	"System Objects in MATLAB Code Generation"
Transforms	
dsp.AnalyticSignal	"System Objects in MATLAB Code Generation"
dsp.DCT	"System Objects in MATLAB Code Generation"
dsp.FFT	"System Objects in MATLAB Code Generation"
dsp.IDCT	"System Objects in MATLAB Code Generation"
dsp.IFFT	"System Objects in MATLAB Code Generation"

C Code Generation from MATLAB

When you have a license for the MATLAB Coder product, you can generate standalone C and C++ from MATLAB code. See the MATLAB Coder documentation for details on supported functionality and workflow.

C Code Generation from Simulink

In this section...
"Open and Run the Model" on page 10-13
"Generate Code from the Model" on page 10-15
"Build and Run the Generated Code" on page 10-15

Note: You must have both the DSP System Toolbox and Simulink Coder products installed on your computer to complete the procedures in this section.

Open and Run the Model

The `ex_codegen_dsp` model implements a simple adaptive filter to remove noise from a signal while simultaneously identifying a filter that characterizes the noise frequency content. To open this model, enter

```
open_system('ex_codegen_dsp')
```
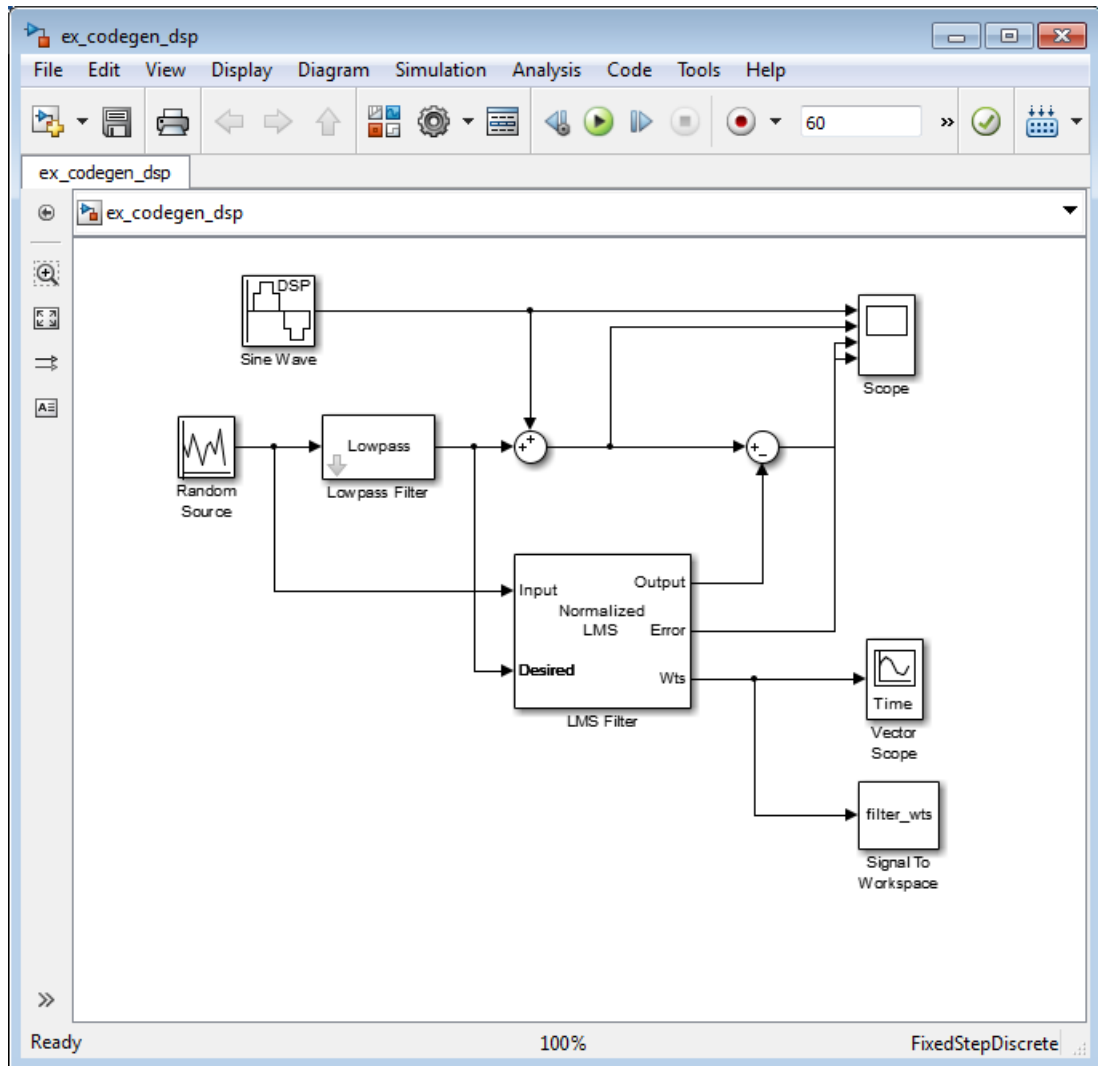
Run the model and observe the output in both scopes. This model saves the filter weights each time they adapt. You can plot the last set of coefficients using the following command:

```
plot(filter_wts(:,:,1201))
```

Generate Code from the Model

To generate code from the model, you must first ensure that you have write permission in your current folder. The code generation process creates a new subfolder inside the current MATLAB working folder. MATLAB saves all of the files created by the code generation process in that subfolder, including those which contain the generated C source code.

To start the code generation process, click the **Build Model** icon ( ) on your model toolbar. After the model finishes generating code, the Code Generation Report appears, allowing you to inspect the generated code. You may also notice that the build process created a new subfolder inside of your current MATLAB working folder. The name of this folder consists of the model name, followed by the suffix `_grt_rtw`. In the case of this example, the subfolder that contains the generated C source code is named `ex_codegen_dsp_grt_rtw`.

Build and Run the Generated Code

Setup the C/C++ Compiler

If you want to build and run the generated code, you need to have access to a C compiler. For more information about which compilers are supported in the current release, see http://www.mathworks.com/support/compilers/current_release/.

To setup your compiler, run the following command:

```
mex -setup
```

Build the Generated Code

After your compiler is setup, you can build and run the generated code. The ex_codegen_dsp model is currently configured to generate code only. To build the generated code, you must first make the following changes:

1 Open the **Model Configuration Parameters** dialog, navigate to the **Code Generation** tab, and clear the **Generate Code Only** checkbox.

2 Click **OK** to apply your changes and close the dialog box.

3
   From the model toolbar, click the **Build Model** icon ( ).

Because you re-configured the model to generate and build code, the code generation process continues until the code is compiled and linked.

Run the Generated Code

To run the generated code, enter the following command at the MATLAB prompt:

```
!ex_codegen_dsp
```

Running the generated code creates a MAT-file which contains the same variables as those generated by simulating the model. The variables in the MAT-file are named with a prefix of rt_. After you run the generated code, you can load the variables from the MAT-file by typing the following command at the MATLAB prompt:

```
load ex_codegen_dsp.mat
```

You can now compare the variables from the generated code with the variables from the model simulation. To plot the last set of coefficients from the generated code, enter the following command at the MATLAB prompt:

```
plot(rt_filter_wts(:,:,1201))
```

The last set of coefficients from the generated code are shown in the following figure.
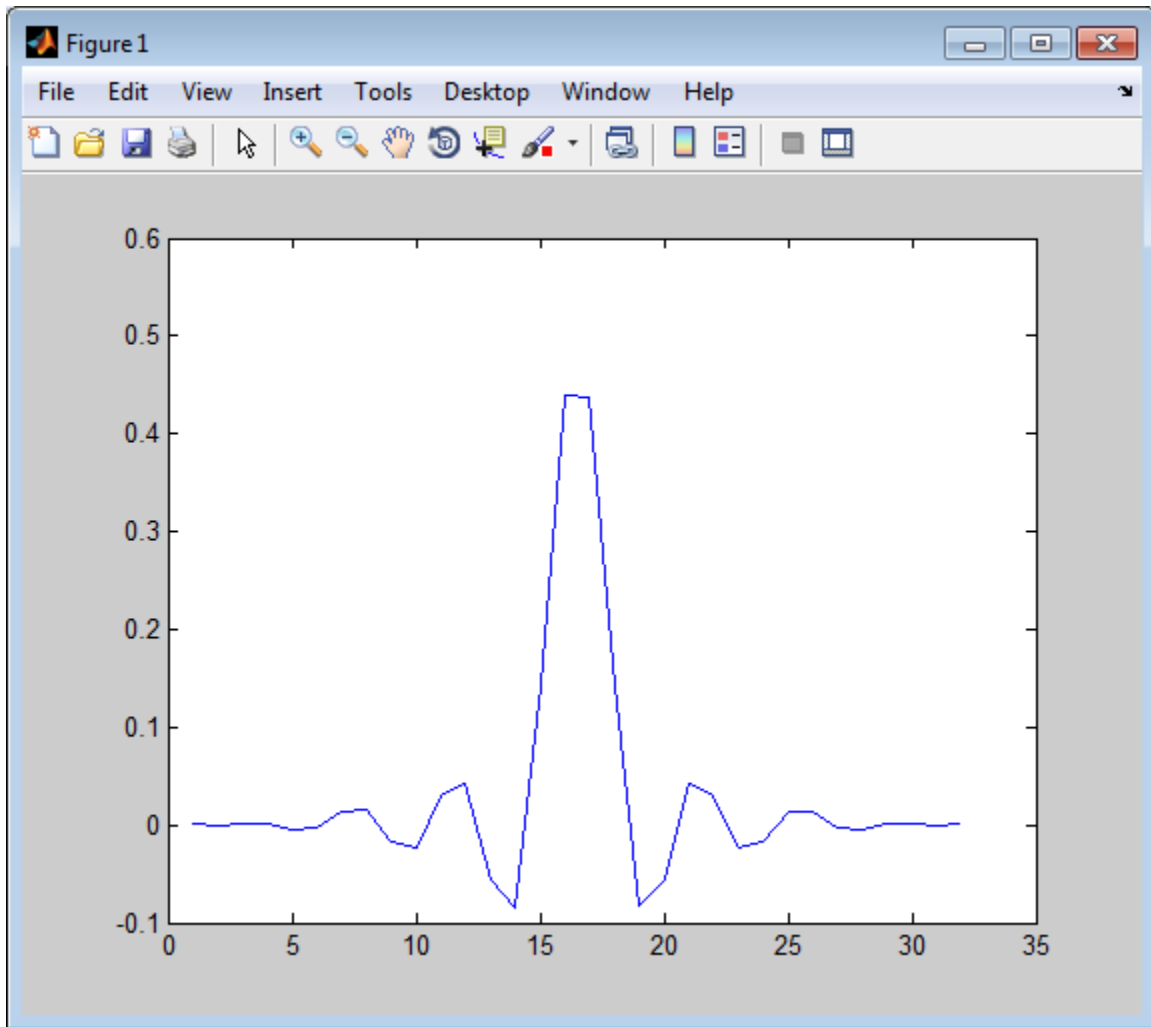
For further information on generating code from Simulink, see the "Simulink Coder" documentation.

How to Run a Generated Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set the environment variables as follows:

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH $DYLD_LIBRARY_PATH: $MATLABROOT/ bin/maci64 (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH $DYLD_LIBRARY_PATH: $MATLABROOT/ bin/maci64 (Bash)`

Linux	`setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/ bin/glnxa64 (csh/tcsh)`
	
	`export LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/ bin/glnxa64 (Bash)`

Windows	`set PATH = $MATLABROOT\bin\win64; %PATH%`

DSP System Toolbox Supported Hardware

As of this release, DSP System Toolbox supports the following hardware.

Support Package	Vendor	Earliest Release Available	Last Release Available
ARM Cortex-M Processors	ARM	R2013b	Current
ARM Cortex-A Processors	ARM	R2014b	Current

For a complete list of supported hardware, see Hardware Support.

How Is `dspunfold` Different from `parfor`?

In this section...
"DSP Algorithms Involve States" on page 10-20
"dspunfold Introduces Latency" on page 10-20
"parfor Requires Significant Restructuring in Code" on page 10-20
"parfor Used with dspunfold" on page 10-21

The `dspunfold` and `parfor` functions accelerate MATLAB algorithms through parallelization. Each function has its own advantages and disadvantages.

When you use `parfor` inside the entry-point MATLAB function, and call `codegen` on this function, the generated MEX file is multi-threaded. For more information, see "Algorithm Acceleration Using Parallel for-Loops (parfor)". However, `parfor` is not ideal for DSP algorithms. The reason being that DSP algorithms involve states.

DSP Algorithms Involve States

Most algorithms in DSP System Toolbox contain states and stream data. States in MATLAB are modeled using persistent variables. Because `parfor` does not support persistent variables, you cannot model states using `parfor` loops. See "Global or Persistent Declarations in parfor-Loop". In addition, you cannot have any data dependency across `parfor` loops. Hence, you cannot maintain state information across these loops. See "When Not to Use parfor-Loops". `dspunfold` overcomes these limitations by supporting persistent variables.

`dspunfold` Introduces Latency

If your application does not tolerate latency, use `parfor` instead. `parfor` does not introduce latency. Latency is the number of input frames processed before generating the first output frame.

`parfor` Requires Significant Restructuring in Code

`parfor` requires you to restructure your algorithm to have a loop-like structure that is iteration independent. Due to the semantic limitations of `parfor`, replacing a `for`-

loop with a `parfor`-loop often requires significant code refactoring. `dspunfold` does not require you to restructure your code.

`parfor` Used with `dspunfold`

When you call `dspunfold` on an entry-point MATLAB function that contains `parfor`, `parfor` multi-threading is disabled. `dspunfold` calls `codegen` with the −O option set to `disable:openmp`. With this option set, `parfor` loops are treated as `for`- loops. The multi-threading behavior of the generated MEX file is due entirely to `dspunfold`.

See Also
"Generate Code with Parallel for-Loops (parfor)" | "Algorithm Acceleration Using Parallel for-Loops (parfor)" | "MATLAB Algorithm Acceleration" | `dspunfold` | `parfor`

Workflow for Generating a Multi-Threaded MEX File using `dspunfold`

1. Run the entry-point MATLAB function with the inputs that you want to test. Make sure that the function has no runtime errors. Call `codegen` on the function and make sure that it generates a MEX file successfully.

2. Generate the multi-threaded MEX file using `dspunfold`. Specify a state length using the `-s` option. The state length must be at least the same length as the algorithm in the MATLAB function. By default, `-s` is set to `0`, indicating that the algorithm is stateless.

3. Run the generated analyzer function. Use the `pass` flag to verify that the output results of the multi-threaded MEX file and the single-threaded MEX file match. Also, check if the speedup and latency displayed by the analyzer function are satisfactory.

4. If the output results do not match, increase the state length and generate the multi-threaded MEX file again. Alternatively, use the automatic state length detection (specified using `-s auto`) to determine the minimum state length that matches the outputs.

5. If the output results match but the speedup and latency are not satisfactory, increase the repetition factor using `-r` or increase the number of threads using `-t`. In addition, you can adjust the state length. Adjust the `dspunfold` options and generate new multi-threaded MEX files until you are satisfied with the results..

For best practices for generating the multi-threaded MEX file using `dspunfold`, see the 'Tips' section of `dspunfold`.

Workflow Example

Run the Entry Point MATLAB Function

Create the entry-point MATLAB function.

```matlab
function [y,mse] = AdaptiveFilter(x,noise)

persistent rlsf1 ffilt noise_var
if isempty (rlsf1)
    rlsf1 = dsp.RLSFilter(32, 'ForgettingFactor', 0.98);
    ffilt = dsp.FIRFilter('Numerator',fir1(32, .25)); % Unknown System
    noise_var = 1e-4;
end
```

```
d = step(ffilt, x) + noise_var * noise; % desired signal
[y,e] = step(rlsf1, x, d);

mse = 10*log10(sum(e.^2));
end
```

The function models an RLS filter that filters the input signal x, using d as the desired signal. The function returns the filtered output in y and the filter error in e.

Run AdaptiveFilter with the inputs that you want to test. Verify that the function runs without errors.

```
AdaptiveFilter(randn(1000,1), randn(1000,1));
```

Call codegen on AdaptiveFilter and generate a MEX file.

```
codegen AdaptiveFilter -args {randn(1000,1), randn(1000,1)}
```

Generate a Multi-Threaded MEX File Using **dspunfold**

Set the state length to 32 samples and the repetition factor to 1. Provide a state length that is greater than or equal to the algorithm in the MATLAB function. When at least one entry of frameinputs is set to true, state length is considered in samples.

```
dspunfold AdaptiveFilter -args {randn(1000,1), randn(1000,1)} -s 32 -f true

Analyzing input MATLAB function AdaptiveFilter
Creating single-threaded MEX file AdaptiveFilter_st.mexw64
Creating multi-threaded MEX file AdaptiveFilter_mt.mexw64
Creating analyzer file AdaptiveFilter_analyzer
```

Run the Generated Analyzer Function

The analyzer considers the actual values of the input. To increase the analyzer effectiveness, provide at least two different frames along the first dimension of the inputs.

```
AdaptiveFilter_analyzer(randn(1000*4,1),randn(1000*4,1))

Analyzing multi-threaded MEX file AdaptiveFilter_mt.mexw64  ...
Latency = 8 frames
Speedup = 3.5x
Warning: The output results of the multi-threaded MEX file AdaptiveFilter_mt.mexw64 do
single-threaded MEX file AdaptiveFilter_st.mexw64. Check that you provided the correct
```

```
function when you generated the multi-threaded MEX file AdaptiveFilter_mt.mexw64. For k
this problem, see the 'Tips' section in the dspunfold function reference page.
> In coder.internal.warning (line 8)
  In AdaptiveFilter_analyzer

ans =

    Latency: 8
    Speedup: 3.4686
       Pass: 0
```

Increase the State Length

The analyzer did not pass the verification. The warning message displayed indicates that a wrong state length value is provided to the dspunfold function. Increase the state length to 1000 samples and repeat the process from the previous section.

```
dspunfold AdaptiveFilter -args {randn(1000,1),randn(1000,1)} -s 1000 -f true

Analyzing input MATLAB function AdaptiveFilter
Creating single-threaded MEX file AdaptiveFilter_st.mexw64
Creating multi-threaded MEX file AdaptiveFilter_mt.mexw64
Creating analyzer file AdaptiveFilter_analyzer
```

Run the generated analyzer.

```
AdaptiveFilter_analyzer(randn(1000*4,1),randn(1000*4,1))

Analyzing multi-threaded MEX file AdaptiveFilter_mt.mexw64  ...
Latency = 8 frames
Speedup = 1.8x

ans =

    Latency: 8
    Speedup: 1.7778
       Pass: 1
```

The analyzer passed verification. It is recommended that you provide different numerics to the analyzer function and make sure that the analyzer function passes.

Improve Speedup and Adjust Latency

If you want to increase speedup and your system can afford a larger latency, increase the repetition factor to 2.

```
dspunfold AdaptiveFilter -args {randn(1000,1),randn(1000,1)} -s 1000 -r 2 -f true

Analyzing input MATLAB function AdaptiveFilter
Creating single-threaded MEX file AdaptiveFilter_st.mexw64
Creating multi-threaded MEX file AdaptiveFilter_mt.mexw64
Creating analyzer file AdaptiveFilter_analyzer
```

Run the analyzer.

```
 AdaptiveFilter_analyzer(randn(1000*4,1), randn(1000*4,1))

Analyzing multi-threaded MEX file AdaptiveFilter_mt.mexw64  ...
Latency = 16 frames
Speedup = 2.4x

ans =

    Latency: 16
    Speedup: 2.3674
       Pass: 1
```

Repeat the process until you achieve satisfactory speedup and latency.

Use Automatic State Length Detection

Choose a state length that is greater than or equal to the state length of your algorithm. If it is not easy to determine the state length for your algorithm analytically, use the automatic state length detection tool. Invoke automatic state length detection by setting -s to auto. The tool detects the minimum state length with which the analyzer passes the verification.

```
dspunfold AdaptiveFilter -args {randn(1000,1),randn(1000,1)} -s auto -f true

Analyzing input MATLAB function AdaptiveFilter
Creating single-threaded MEX file AdaptiveFilter_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 1000 ... Sufficient
Checking 500 ... Insufficient
Checking 750 ... Insufficient
Checking 875 ... Sufficient
Checking 812 ... Insufficient
Checking 843 ... Sufficient
Checking 827 ... Insufficient
Checking 835 ... Insufficient
```

```
Checking 839 ... Sufficient
Checking 837 ... Sufficient
Checking 836 ... Sufficient
Minimal state length is 836
Creating multi-threaded MEX file AdaptiveFilter_mt.mexw64
Creating analyzer file AdaptiveFilter_analyzer
```

Minimal state length is **836** samples.

Run the generated analyzer.

```
AdaptiveFilter_analyzer(randn(1000*4,1), randn(1000*4,1))
```

```
Analyzing multi-threaded MEX file AdaptiveFilter_mt.mexw64  ...
Latency = 8 frames
Speedup = 1.9x

ans =

    Latency: 8
    Speedup: 1.9137
       Pass: 1
```

The analyzer passed the verification.

See Also

"Limitations of dspunfold" on page 10-27 | "Why Does the Analyzer Choose the Wrong State Length?" on page 10-31 | "Why Does the Analyzer Choose a Zero State Length?" on page 10-34 | dspunfold

Limitations of `dspunfold`

General Limitations

- On Windows and Linux, you must use a compiler that supports the Open Multiprocessing (OpenMP) application interface. See `http://www.mathworks.com/support/compilers/current_release/`.

- If the input MATLAB function has runtime errors, the errors are not caught when you run the multi-threaded MEX file. Before you use the `dspunfold` function, call `codegen` on the MATLAB function and make sure that the MEX file is generated successfully.

- If the generated code uses a large amount of memory to store the local variables, around `4 MB` on Windows platform, the generated multi-threaded MEX file can have unexpected behavior. This limit varies with each platform. As a workaround, reduce the size of the input signals or restructure the MATLAB function to use less local memory.

- `dspunfold` does not support:

  - `varargin` and `varargout` inside the MATLAB function
  - variable-size inputs and outputs
  - P-coded entry-point MATLAB functions
  - cell arrays as inputs and outputs

Analyzer Limitations

The following limitations apply to the analyzer function generated by the `dspunfold` function. For more information on the analyzer function, see 'Self-Diagnostic Analyzer' in the 'More About' section of `dspunfold`.

- If multiple frames of the analyzer input are identical, the analyzer might throw false positive pass results. It is recommended that you provide at least two different frames for each input of the analyzer.

- If the algorithm in the entry-point MATLAB function chooses its state length based on the input values, the analyzer might provide different *pass* results for different input values. For an example, see the FIR_Mean function in "Why Does the Analyzer Choose the Wrong State Length?" on page 10-31.

- If the input to the entry-point MATLAB function does affect the output immediately, the analyzer might throw false positive *pass* results. For an example, see the Input_Output function in "Why Does the Analyzer Choose a Zero State Length?" on page 10-34.

- If the output results of the multi-threaded MEX file and single-threaded MEX file match statistically but do not match numerically, the analyzer does not pass. Consider the FilterNoise function that follows, which filters a random noise signal with an FIR filter. The function calls randn from within itself to generate random noise. Hence, the output results of the FilterNoise function match statistically but not match numerically.

```
function Output = FilterNoise(x)

persistent FIRFilter
if isempty(FIRFilter)
    FIRFilter = dsp.FIRFilter('Numerator',fir1(12,0.4));
end
Output = step(FIRFilter, x+randn(1000,1));

end
```

When you run the automatic state length detection tool run on FilterNoise, the tool detects an infinite state length. Because the tool cannot find a numerical match for a finite state length, it chooses an infinite state length.

```
dspunfold FilterNoise -args {randn(1000,1)} -s auto

Analyzing input MATLAB function FilterNoise
Creating single-threaded MEX file FilterNoise_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 1 ... Insufficient
Checking Infinite ... Sufficient
Checking 2 ... Insufficient
Minimal state length is Inf
Creating multi-threaded MEX file FilterNoise_mt.mexw64
Warning: The multi-threading was disabled due to performance considerations. This ha
equal to (Threads-1)*Repetition frames (3 frames in this case).
> In coder.internal.warning (line 8)
```

```
  In unfoldingEngine/BuildParallelSolution (line 25)
  In unfoldingEngine/generate (line 207)
  In dspunfold (line 234)
Creating analyzer file FilterNoise_analyzer
```

The algorithm does not need an infinite state. The state length of the FIR filter, hence the algorithm is 12.

Call dspunfold with state length set to 12.

```
dspunfold FilterNoise -args {randn(1000,1)} -s 12 -f true
```

```
Analyzing input MATLAB function FilterNoise
Creating single-threaded MEX file FilterNoise_st.mexw64
Creating multi-threaded MEX file FilterNoise_mt.mexw64
Creating analyzer file FilterNoise_analyzer
```

Run the analyzer function.

```
FilterNoise_analyzer(randn(1000*4,1))
```

```
Analyzing multi-threaded MEX file FilterNoise_mt.mexw64  ...
Latency = 8 frames
Speedup = 0.5x
Warning: The output results of the multi-threaded MEX file FilterNoise_mt.mexw64 do
single-threaded MEX file FilterNoise_st.mexw64. Check that you provided the correct
when you generated the multi-threaded MEX file FilterNoise_mt.mexw64. For best pract
see the 'Tips' section in the dspunfold function reference page.
> In coder.internal.warning (line 8)
  In FilterNoise_analyzer
ans =
    Latency: 8
    Speedup: 0.4970
       Pass: 0
```

The analyzer looks for a numerical match and fails the verification, even though the generated multi-threaded MEX file is valid.

Speedup Limitations

- If the entry-point MATLAB function contains code with low complexity, MATLAB overhead or multi-threaded MEX overhead overshadow any performance gains. In such cases, do not use dspunfold.

- If the number of operations in the input MATLAB function is small compared to the size of the input or output data, the multi-threaded MEX file does not provide any speedup gain. Sometimes, it can result in a speedup loss, even if the repetition value is increased. In such cases, do not use `dspunfold`.

See Also

"Workflow for Generating a Multi-Threaded MEX File using dspunfold" on page 10-22 | "Why Does the Analyzer Choose the Wrong State Length?" on page 10-31 | "Why Does the Analyzer Choose a Zero State Length?" on page 10-34 | `dspunfold`

Why Does the Analyzer Choose the Wrong State Length?

In this section...
"Reason for Verification Failure" on page 10-32
"Recommendation" on page 10-33

If the state length of the algorithm depends on the inputs to the algorithm, make sure that you use inputs that choose the same state length when generating the MEX file and running the analyzer. Otherwise, the analyzer fails the verification.

The algorithm in the function `FIR_Mean` has no states when `mean(input) > 0`, and has states otherwise.

```
function [ Output ] = FIR_Mean( input )

persistent Filter
if isempty(Filter)
    Filter = dsp.FIRFilter('Numerator', fir1(12,0.4));
end


 if (mean(input) > 0)
     % stateless
     Output = mean(input);
 else
     % this path contains states
    yFilt = step(Filter,input);
   Output = mean(yFilt);
 end
 end
```

When you invoke the automatic state length detection on this function, the analyzer detects a state length of 14 samples.

```
dspunfold FIR_Mean -args {randn(10,1)} -s auto -f true

Analyzing input MATLAB function FIR_Mean
Creating single-threaded MEX file FIR_Mean_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 10 ... Insufficient
```

```
Checking Infinite ... Sufficient
Checking 20 ... Sufficient
Checking 15 ... Sufficient
Checking 12 ... Insufficient
Checking 13 ... Insufficient
Checking 14 ... Sufficient
Minimal state length is 14
Creating multi-threaded MEX file FIR_Mean_mt.mexw64
Creating analyzer file FIR_Mean_analyzer
```

Run the analyzer function. Use an input with four different frames. Check if the output results match.

```
FIR_Mean_analyzer(randn(10*4,1))

Analyzing multi-threaded MEX file FIR_Mean_mt.mexw64  ...
Latency = 8 frames
Speedup = 0.5x
Warning: The output results of the multi-threaded MEX file FIR_Mean_mt.mexw64 do not ma
single-threaded MEX file FIR_Mean_st.mexw64. Check that you provided the correct state
you generated the multi-threaded MEX file FIR_Mean_mt.mexw64. For best practices and po
'Tips' section in the dspunfold function reference page.
> In coder.internal.warning (line 8)
  In FIR_Mean_analyzer

ans =

    Latency: 8
    Speedup: 0.5040
       Pass: 0
```

`Pass = 0`, and the function throws a warning message indicating a possible reason for the verification failure.

Reason for Verification Failure

The state length of the algorithm depends on the input. When `mean(input) > 0`, the algorithm is stateless. Otherwise, the algorithm contains states. When generating the MEX file, the input arguments choose the code path with states. When the analyzer is called, the multi-frame input chooses the code path without states. Hence, the state length is different in both the cases leading to the verification failure.

Recommendation

The recommendation is to use inputs which choose the same state length when generating the MEX file and running the analyzer.

For best practices, see the 'Tips' section of dspunfold.

See Also
"Workflow for Generating a Multi-Threaded MEX File using dspunfold" on page 10-22 | "Limitations of dspunfold" on page 10-27 | "Why Does the Analyzer Choose a Zero State Length?" on page 10-34

Why Does the Analyzer Choose a Zero State Length?

When the output of the algorithm does not change for any input given to the algorithm, the analyzer considers the algorithm stateless, even if it contains states. Make sure the inputs to the algorithm have an immediate effect on the output of the algorithm.

The function `Input_Output` uses an FIR filter that contains states.

```
function [output] = Input_Output(input)

persistent Filter
if isempty(Filter)
    Filter = dsp.FIRFilter('Numerator', (1:12));
end

y = step(Filter,input);

output = any(y(:)>0);

end
```

When you call automatic state length detection on this function, the analyzer detects a minimal state length of 0.

```
dspunfold Input_Output -args {randn(10,1)} -s auto -f true

Analyzing input MATLAB function Input_Output
Creating single-threaded MEX file Input_Output_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Sufficient
Minimal state length is 0
Creating multi-threaded MEX file Input_Output_mt.mexw64
Creating analyzer file Input_Output_analyzer
```

The analyzer detects a zero state length because the output of the function is the same irrespective of the value of the input. When the analyzer tests the algorithm with zero state length, the outputs of the multi-threaded MEX and single-threaded MEX match. Therefore, the analyzer considers the algorithm stateless and sets the minimal state length to zero.

Recommendation

To prevent the analyzer from choosing the wrong state length, rewrite your algorithm so that inputs have an immediate effect on the output. Also, choose inputs which stress the code path with maximal state length.

For best practices, see the 'Tips' section of `dspunfold`.

See Also

"Workflow for Generating a Multi-Threaded MEX File using dspunfold" on page 10-22 | "Limitations of dspunfold" on page 10-27 | "Why Does the Analyzer Choose the Wrong State Length?" on page 10-31

Define New System Objects

System Objects Methods for Defining New Objects

The following Impl methods comprise the System objects API for defining new System objects. For more information see "Define New System Objects""Define System Objects".

- getDiscreteStateImpl
- getDiscreteStateSpecificationImpl
- getHeaderImpl
- getIconImpl
- getInputNamesImpl
- getNumInputsImpl
- getNumOutputsImpl
- getOutputDataTypeImpl
- getOutputNamesImpl
- getOutputSizeImpl
- isInputSizeLockedImpl
- getPropertyGroupsImpl
- getSimulateUsingImpl
- isDoneImpl
- infoImpl
- isInactivePropertyImpl
- isInputDirectFeedthroughImpl
- isOutputComplexImpl
- isOutputFixedSizeImpl
- loadObjectImpl
- outputImpl
- processTunedPropertiesImpl
- propagatedInputComplexity
- propagatedInputDataType
- propagatedInputFixedSize
- propagatedInputSize
- releaseImpl

- resetImpl
- saveObjectImpl
- setProperties
- setupImpl
- showSimulateUsingImpl
- stepImpl
- supportsMultipleInstanceImpl
- updateImpl
- validateInputsImpl
- validatePropertiesImpl

Define Basic System Objects

This example shows how to create a basic System object that increments a number by one. The class definition file used in the example contains the minimum elements required to define a System object.

Create System Object

You can create and edit a MAT-file or use the MATLAB Editor to create your System object. This example describes how to use the **New** menu in the MATLAB Editor.

In MATLAB, on the Editor tab, select **New** > **System Object** > **Basic**. A simple System object template opens.

Subclass your object from `matlab.System`. Replace `Untitled` with `AddOne` in the first line of your file.

```
classdef AddOne < matlab.System
```

Save the file and name it `AddOne.m`.

Define Algorithm

The `stepImpl` method contains the algorithm to execute when you call the `step` method on your object. Define this method so that it contains the actions you want the System object to perform.

1   In the basic System object you created, inspect the `stepImpl` method template.

```
methods (Access = protected)
    function y = stepImpl(obj,u)
        % Implement algorithm. Calculate y as a function of input u and
        % discrete states.
        y = u;
    end
end
```

The `stepImpl` method access is always set to `protected` because it is an internal method that users do not directly call or run.

All methods, except static methods, expect the System object handle as the first input argument. The default value, inserted by MATLAB Editor, is `obj`. You can use any name for your System object handle.

By default, the number of inputs and outputs are both 1. Inputs and outputs can be added using **Inputs/Outputs**. If you use variable number of inputs or outputs, insert the appropriate `getNumInputsImpl` or `getNumOutputsImpl` method.

Alternatively, if you create your System object by editing a MAT-file, you can add the `stepImpl` method using **Insert Method** > **Implement algorithm**.

2 Change the computation in the y function to add 1 to the value of u.

```
methods (Access = protected)

    function y = stepImpl(~,u)
      y = u + 1;
    end
```

> **Note:** Instead of passing in the object handle, you can use the tilde (~) to indicate that the object handle is not used in the function. Using the tilde instead of an object handle prevents warnings about unused variables.

3 Remove the additional, unused methods that are included by default in the basic template. Alternatively, you can modify these methods to add more System object actions and properties. You can also make no changes, and the System object still operates as intended.

The class definition file now has all the code necessary for this System object.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value one greater than the input value

  % All methods occur inside a methods declaration.
  % The stepImpl method has protected access
  methods (Access = protected)

    function y = stepImpl(~,u)
      y = u + 1;
    end
  end
end
```

See Also

matlab.System | getNumInputsImpl | getNumOutputsImpl | stepImpl

Related Examples

- "Change Number of Step Inputs or Outputs" on page 11-8

More About

- "System Design and Simulation in MATLAB"

Change Number of Step Inputs or Outputs

This example shows how to specify two inputs and two outputs for the `step` method.

If you specify the inputs and outputs to the `stepImpl` method, you do not need to specify the `getNumInputsImpl` and `getNumOutputsImpl` methods. If you have a variable number of inputs or outputs (using `varargin` or `varargout`), include the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively, in your class definition file.

Note: You should only use `getNumInputsImpl` or `getNumOutputsImpl` methods to change the number of System object inputs or outputs. Do not use any other handle objects within a System object to change the number of inputs or outputs.

You always set the `getNumInputsImpl` and `getNumOutputsImpl` methods access to `protected` because they are internal methods that users do not directly call or run.

Update the Algorithm for Multiple Inputs and Outputs

Update the `stepImpl` method to specify two inputs and two outputs. You do not need to implement associated `getNumInputsImpl` or `getNumOutputsImpl` methods.

```
methods (Access = protected)
   function [y1,y2] = stepImpl(~,x1,x2)
      y1 = x1 + 1
      y2 = x2 + 1;
   end
end
```

Update the Algorithm and Associated Methods

Update the `stepImpl` method to use `varargin` and `varargout`. In this case, you must implement the associated `getNumInputsImpl` and `getNumOutputsImpl` methods to specify two or three inputs and outputs.

```
methods (Access = protected)
   function varargout = stepImpl(obj,varargin)
      varargout{1} = varargin{1}+1;
      varargout{2} = varargin{2}+1;
      if (obj.numInputsOutputs == 3)
         varargout{3} = varargin{3}+1;
```

```
            end
        end

        function validatePropertiesImpl(obj)
            if ~((obj.numInputsOutputs == 2) ||...
                (obj.numInputsOutputs == 3))
                error('Only 2 or 3 input and outputs allowed.');
            end
        end

        function numIn = getNumInputsImpl(obj)
            numIn = 3;
            if (obj.numInputsOutputs == 2)
                numIn = 2;
            end
        end

        function numOut = getNumOutputsImpl(obj)
            numOut = 3;
            if (obj.numInputsOutputs == 2)
                numOut = 2;
            end
        end
    end
end
```

Use this syntax to run the algorithm with two inputs and two outputs.

```
x1 = 3;
x2 = 7;
[y1,y2] = step(AddOne,x1,x2);
```

To change the number of inputs or outputs, you must release the object before rerunning it.

```
release(AddOne)
x1 = 3;
x2 = 7;
x3 = 10
[y1,y2,y3] = step(AddOne,x1,x2,x3);
```

Complete Class Definition File with Multiple Inputs and Outputs

```
 classdef AddOne < matlab.System
% ADDONE Compute output values one greater than the input values
```

```matlab
    % This property is nontunable and cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
      numInputsOutputs = 3;   % Default value
    end

    % All methods occur inside a methods declaration.
    % The stepImpl method has protected access
    methods (Access = protected)
      function varargout = stepImpl(obj,varargin)
        if (obj.numInputsOutputs == 2)
          varargout{1} = varargin{1}+1;
          varargout{2} = varargin{2}+1;
        else
          varargout{1} = varargin{1}+1;
          varargout{2} = varargin{2}+1;
          varargout{3} = varargin{3}+1;
        end
      end

      function validatePropertiesImpl(obj)
        if ~((obj.numInputsOutputs == 2) ||...
            (obj.numInputsOutputs == 3))
          error('Only 2 or 3 input and outputs allowed.');
        end
      end

      function numIn = getNumInputsImpl(obj)
        numIn = 3;
        if (obj.numInputsOutputs == 2)
          numIn = 2;
        end
      end

      function numOut = getNumOutputsImpl(obj)
        numOut = 3;
        if (obj.numInputsOutputs == 2)
          numOut = 2;
        end
      end
    end
```

```
end
```

See Also

getNumInputsImpl | getNumOutputsImpl

Related Examples

- "Validate Property and Input Values" on page 11-14
- "Define Basic System Objects" on page 11-5

More About

- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Specify System Block Input and Output Names

This example shows how to specify the names of the input and output ports of a System object–based block implemented using a `MATLAB System` block.

Define Input and Output Names

This example shows how to use `getInputNamesImpl` and `getOutputNamesImpl` to specify the names of the input port as "source data" and the output port as "count."

If you do not specify the `getInputNamesImpl` and `getOutputNamesImpl` methods, the object uses the `stepImpl` method input and output variable names for the input and output port names, respectively. If the `stepImpl` method uses *varargin* and *varargout* instead of variable names, the port names default to empty strings.

```
methods (Access = protected)
    function inputName = getInputNamesImpl(~)
            inputName = 'source data';
    end

    function outputName = getOutputNamesImpl(~)
            outputName = 'count';
    end
end
```

Complete Class Definition File with Named Inputs and Outputs

```
classdef MyCounter < matlab.System

    % MyCounter Count values above a threshold

     properties
        Threshold = 1
     end
     properties (DiscreteState)
        Count
     end

     methods
        function obj = MyCounter(varargin)
            setProperties (obj,nargin,varargin{:});
        end
     end
```

```
methods (Access = protected)
    function setupImpl(obj)
        obj.Count = O;
    end
    function resetImpl(obj)
        obj.Count = O;
    end
    function y = stepImpl(obj,u)
        if (u > obj.Threshold)
            obj.Count = obj.Count + 1;
        end
        y = obj.Count;
    end
    function inputName = getInputNamesImpl(~)
        inputName = 'source data';
    end
    function outputName = getOutputNamesImpl(~)
        outputName = 'count';
    end
end
end
```

See Also

getInputNamesImpl | getNumInputsImpl | getNumOutputsImpl | getOutputNamesImpl

Related Examples

- "Change Number of Step Inputs or Outputs" on page 11-8

More About

- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Validate Property and Input Values

This example shows how to verify that the user's inputs and property values are valid.

Validate Properties

This example shows how to validate the value of a single property using set.*PropertyName* syntax. In this case, the *PropertyName* is Increment.

```
methods
   % Validate the properties of the object
   function set.Increment(obj,val)
       if val >= 10
          error('The increment value must be less than 10');
       end
       obj.Increment = val;
   end
 end
```

This example shows how to validate the value of two interdependent properties using the validatePropertiesImpl method. In this case, the UseIncrement property value must be true and the WrapValue property value must be less than the Increment property value.

```
methods (Access = protected)
   function validatePropertiesImpl(obj)
       if obj.UseIncrement && obj.WrapValue > obj.Increment
          error('Wrap value must be less than increment value');
       end
   end
 end
```

Validate Inputs

This example shows how to validate that the first input is a numeric value.

```
methods (Access = protected)
   function validateInputsImpl(~,x)
      if ~isnumeric(x)
         error('Input must be numeric');
      end
   end
```

```
end
```

Complete Class Definition File with Property and Input Validation

```matlab
classdef AddOne < matlab.System
% ADDONE Compute an output value by incrementing the input value

  % All properties occur inside a properties declaration.
  % These properties have public access (the default)
  properties (Logical)
    UseIncrement = true
  end

  properties (PositiveInteger)
    Increment = 1
    WrapValue = 10
  end

  methods
    % Validate the properties of the object
    function set.Increment(obj,val)
        if val >= 10
          error('The increment value must be less than 10');
        end
        obj.Increment = val;
    end
  end

  methods (Access = protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue > obj.Increment
          error('Wrap value must be less than increment value');
        end
    end

    % Validate the inputs to the object
    function validateInputsImpl(~,x)
        if ~isnumeric(x)
          error('Input must be numeric');
        end
    end

    function out = stepImpl(obj,in)
      if obj.UseIncrement
        out = in + obj.Increment;
```

```
      else
         out = in + 1;
      end
   end
  end
end
```

Note: See "Change Input Complexity or Dimensions" for more information.

See Also
validateInputsImpl | validatePropertiesImpl

Related Examples
- "Define Basic System Objects" on page 11-5

More About
- "Methods Timing" on page 11-94
- "Property Set Methods"
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Initialize Properties and Setup One-Time Calculations

This example shows how to write code to initialize and set up a System object.

In this example, you allocate file resources by opening the file so the System object can write to that file. You do these initialization tasks one time during setup, rather than every time you call the step method.

Define Public Properties to Initialize

In this example, you define the public `Filename` property and specify the value of that property as the nontunable string, `default.bin`. Users cannot change *nontunable* properties after the `setup` method has been called. Refer to the Methods Timing section for more information.

```
properties (Nontunable)
  Filename = 'default.bin'
end
```

Define Private Properties to Initialize

Users cannot access *private* properties directly, but only through methods of the System object. In this example, you define the `pFileID` property as a private property. You also define this property as *hidden* to indicate it is an internal property that never displays to the user.

```
properties (Hidden,Access = private)
  pFileID;
end
```

Define Setup

You use the `setupImpl` method to perform setup and initialization tasks. You should include code in the `setupImpl` method that you want to execute one time only. The `setupImpl` method is called once during the first call to the `step` method. In this example, you allocate file resources by opening the file for writing binary data.

```
methods
  function setupImpl(obj)
    obj.pFileID = fopen(obj.Filename,'wb');
    if obj.pFileID < O
      error('Opening the file failed');
```

```
        end
    end
end
```

Although not part of setup, you should close files when your code is done using them. You use the `releaseImpl` method to release resources.

Complete Class Definition File with Initialization and Setup

```
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step  method has been called.
    properties (Nontunable)
        Filename = 'default.bin' % the name of the file to create
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods (Access = protected)
        % In setup allocate any resources, which in this case
        % means opening the file.
        function setupImpl(obj)
            obj.pFileID = fopen(obj.Filename,'wb');
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end

        % This System object™ writes the input to the file.
        function stepImpl(obj,data)
            fwrite(obj.pFileID,data);
        end

        % Use release to close the file to prevent the
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end
    end
```

```
end
```

See Also

releaseImpl | setupImpl | stepImpl

Related Examples

- "Release System Object Resources" on page 11-35
- "Define Property Attributes" on page 11-24

More About

- "Methods Timing" on page 11-94

Set Property Values at Construction Time

This example shows how to define a System object constructor and allow it to accept name-value property pairs as input.

Set Properties to Use Name-Value Pair Input

Define the System object constructor, which is a method that has the same name as the class (`MyFile` in this example). Within that method, you use the `setProperties` method to make all public properties available for input when the user constructs the object. `nargin` is a MATLAB function that determines the number of input arguments. `varargin` indicates all of the object's public properties.

```
methods
   function obj = MyFile(varargin)
      setProperties(obj,nargin,varargin{:});
   end
end
```

Complete Class Definition File with Constructor Setup

```
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        Filename ='default.bin' % the name of the file to create
        Access = 'wb' % The file access string (write, binary)
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods
        % You call setProperties in the constructor to let
        % a user specify public properties of object as
        % name-value pairs.
        function obj = MyFile(varargin)
          setProperties(obj,nargin,varargin{:});
        end
```

```
        end

    methods (Access = protected)
        % In setup allocate any resources, which in this case is
        % opening the file.
        function setupImpl(obj)
            obj.pFileID = fopen(obj.Filename,obj.Access);
            if obj.pFileID < O
                error('Opening the file failed');
            end
        end

        % This System object™ writes the input to the file.
        function stepImpl(obj,data)
            fwrite(obj.pFileID,data);
        end

        % Use release to close the file to prevent the
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end
    end
end
```

See Also
nargin | setProperties

Related Examples
- "Define Property Attributes" on page 11-24
- "Release System Object Resources" on page 11-35

Reset Algorithm State

This example shows how to reset an object state.

Reset Counter to Zero

`pCount` is an internal counter property of the System object `obj`. The user calls the `reset` method on the locked object, which calls the `resetImpl` method. In this example , `pCount` resets to 0.

Note: When resetting an object's state, make sure you reset the size, complexity, and data type correctly.

```
methods (Access = protected)
   function resetImpl(obj)
      obj.pCount = 0;
   end
end
```

Complete Class Definition File with State Reset

```
classdef Counter < matlab.System
% Counter System object™ that increments a counter

   properties (Access = private)
      pCount
   end

   methods (Access = protected)
      % In step, increment the counter and return
      % its value as an output
      function c = stepImpl(obj)
          obj.pCount = obj.pCount + 1;
          c = obj.pCount;
      end

      % Reset the counter to zero.
      function resetImpl(obj)
         obj.pCount = 0;
      end
   end
```

```
end
```

See "Methods Timing" on page 11-94 for more information.

See Also

resetImpl

More About

- "Methods Timing" on page 11-94

Define Property Attributes

This example shows how to specify property attributes.

Property attributes, which add details to a property, provide a layer of control to your properties. In addition to the MATLAB property attributes, System objects can use these three additional attributes—`nontunable`, `logical`, and `positiveInteger`. To specify multiple attributes, separate them with commas.

Specify Property as Nontunable

Use the *nontunable* attribute for a property when the algorithm depends on the value being constant once data processing starts. Defining a property as nontunable may improve the efficiency of your algorithm by removing the need to check for or react to values that change. For code generation, defining a property as nontunable allows the memory associated with that property to be optimized. You should define all properties that affect the number of input or output ports as nontunable.

System object users cannot change nontunable properties after the `setup` or `step` method has been called. In this example, you define the `InitialValue` property, and set its value to 0.

```
properties (Nontunable)
   InitialValue = 0;
end
```

Specify Property as Logical

Logical properties have the value, `true` or `false`. System object users can enter `1` or `0` or any value that can be converted to a logical. The value, however, displays as `true` or `false`. You can use sparse logical values, but they must be scalar values. In this example, the `Increment` property indicates whether to increase the counter. By default, `Increment` is tunable property. The following restrictions apply to a property with the `Logical` attribute,

- Cannot also be `Dependent` or `PositiveInteger`
- Default value must be `true` or `false`. You cannot use `1` or `0` as a default value.

```
properties (Logical)
  Increment = true
end
```

Specify Property as Positive Integer

In this example, the private property `MaxValue` is constrained to accept only real,
positive integers. You cannot use sparse values. The following restriction applies to a
property with the `PositiveInteger` attribute,

- Cannot also be `Dependent` or `Logical`

```
properties (PositiveInteger)
   MaxValue
end
```

Specify Property as DiscreteState

If your algorithm uses properties that hold state, you can assign those properties the
`DiscreteState` attribute . Properties with this attribute display their state values
when users call `getDiscreteStateImpl` via the `getDiscreteState` method. The
following restrictions apply to a property with the `DiscreteState` attribute,

- Numeric, logical, or fi value, but not a scaled double fi value
- Does not have any of these attributes: `Nontunable`, `Dependent`, `Abstract`,
  `Constant`, or `Transient`.
- No default value
- Not publicly settable
- `GetAccess = Public` by default
- Value set only using the `setupImpl` method or when the System object is locked
  during `resetImpl` or `stepImpl`

In this example, you define the `Count` property.

```
properties (DiscreteState)
   Count;
end
```

Complete Class Definition File with Property Attributes

```
classdef Counter < matlab.System
% Counter Increment a counter to a maximum value

  % These properties are nontunable. They cannot be changed
  % after the setup or step method has been called.
  properties (Nontunable)
```

```matlab
        % The inital value of the counter
        InitialValue = 0
    end
    properties (Nontunable, PositiveInteger)
        % The maximum value of the counter
        MaxValue = 3
    end

    properties (Logical)
        % Whether to increment the counter
        Increment = true
    end

    properties (DiscreteState)
        % Count state variable
        Count
    end

    methods (Access = protected)
        % In step, increment the counter and return its value
        % as an output

        function c = stepImpl(obj)
            if obj.Increment && (obj.Count < obj.MaxValue)
                obj.Count = obj.Count + 1;
            else
                disp(['Max count, ' num2str(obj.MaxValue) ',reached'])
            end
            c = obj.Count;
        end

        % Setup the Count state variable
        function setupImpl(obj)
            obj.Count = 0;
        end

        % Reset the counter to one.
        function resetImpl(obj)
            obj.Count = obj.InitialValue;
        end
    end
```

```
end
```

More About

- "Class Attributes"
- "Property Attributes"
- "What You Cannot Change While Your System Is Running"
- "Methods Timing" on page 11-94

Hide Inactive Properties

This example shows how to hide the display of a property that is not active for a particular object configuration.

Hide an inactive property

You use the `isInactivePropertyImpl` method to hide a property from displaying. If the `isInactiveProperty` method returns `true` to the property you pass in, then that property does not display.

```matlab
methods (Access = protected)
  function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
      flag = obj.UseRandomInitialValue;
    else
      flag = false;
    end
  end
end
```

Complete Class Definition File with Hidden Inactive Property

```matlab
classdef Counter < matlab.System
  % Counter Increment a counter

  % These properties are nontunable. They cannot be changed
  % after the setup or step method has been called.
  properties (Nontunable)
    % Allow the user to set the initial value
    UseRandomInitialValue = true
    InitialValue = 0
  end

  % The private count variable, which is tunable by default
  properties (Access = private)
    pCount
  end

  methods (Access = protected)
    % In step, increment the counter and return its value
    % as an output
    function c = stepImpl(obj)
      obj.pCount = obj.pCount + 1;
```

```matlab
      c = obj.pCount;
    end

    % Reset the counter to either a random value or the initial
    % value.
    function resetImpl(obj)
      if obj.UseRandomInitialValue
        obj.pCount = rand();
      else
        obj.pCount = obj.InitialValue;
      end
    end

    % This method controls visibility of the object's properties
    function flag = isInactivePropertyImpl(obj,propertyName)
      if strcmp(propertyName,'InitialValue')
        flag = obj.UseRandomInitialValue;
      else
        flag = false;
      end
    end
  end
end
```

See Also

isInactivePropertyImpl

Limit Property Values to Finite String Set

This example shows how to limit a property to accept only a finite set of string values.

Specify a Set of Valid String Values

String sets use two related properties. You first specify the user-visible property name and default string value. Then, you specify the associated hidden property by appending "Set" to the property name. You must use a capital "S" in "Set."

In the "Set" property, you specify the valid string values as a cell array of the `matlab.system.Stringset` class. This example uses `Color` and `ColorSet` as the associated properties.

```
properties
   Color = 'blue'
end

properties (Hidden,Transient)
   ColorSet = matlab.system.StringSet({'red','blue','green'});
end
```

Complete Class Definition File with String Set

```
classdef Whiteboard < matlab.System
% Whiteboard Draw lines on a figure window
%
% This System object™ illustrates the use of StringSets

   properties
     Color = 'blue'
   end

   properties (Hidden,Transient)
     % Let them choose a color
     ColorSet = matlab.system.StringSet({'red','blue','green'});
   end

   methods (Access = protected)
     function stepImpl(obj)
         h = Whiteboard.getWhiteboard();
         plot(h, ...
           randn([2,1]),randn([2,1]), ...
           'Color',obj.Color(1));
```

```
        end
        function releaseImpl(obj)
            cla(Whiteboard.getWhiteboard());
            hold on
        end
    end

    methods (Static)
      function a = getWhiteboard()
          h = findobj('tag','whiteboard');
          if isempty(h)
            h = figure('tag','whiteboard');
            hold on
          end
          a = gca;
      end
    end
end
```

String Set System Object Example

```
%%
% Each call to step draws lines on a whiteboard

%% Construct the System object
hGreenInk = Whiteboard;
hBlueInk  = Whiteboard;

% Change the color
% Note: Press tab after typing the first single quote to
% display all enumerated values.
hGreenInk.Color = 'green';
hBlueInk.Color  = 'blue';

% Take a few steps
for i=1:3
  hGreenInk.step();
  hBlueInk.step();
end

%% Clear the whiteboard
hBlueInk.release();

%% Display System object used in this example
```

```
type('Whiteboard.m');
```

See Also

matlab.system.StringSet

Process Tuned Properties

This example shows how to specify the action to take when a tunable property value changes during simulation.

The `processTunedPropertiesImpl` method is useful for managing actions to prevent duplication. In many cases, changing one of multiple interdependent properties causes an action. With the `processTunedPropertiesImpl` method, you can control when that action is taken so it is not repeated unnecessarily.

Control When a Lookup Table Is Generated

This example of `processTunedPropertiesImpl` causes the `pLookupTable` to be regenerated when either the `NumNotes` or `MiddleC` property changes.

```
methods  (Access = protected)
   function processTunedPropertiesImpl(obj)
      propChange = isChangedProperty(obj,obj.NumNotes)||...
         isChangedProperty(obj,obj.MiddleC)
      if propChange
         obj.pLookupTable = obj.MiddleC *...
            (1+log(1:obj.NumNotes)/log(12));
      end
   endend
```

Complete Class Definition File with Tuned Property Processing

```
classdef TuningFork < matlab.System
  % TuningFork Illustrate the processing of tuned parameters
  %

  properties
    MiddleC = 440
    NumNotes = 12
  end

  properties (Access = private)
    pLookupTable
  end

  methods (Access = protected)
    function resetImpl(obj)
      obj.MiddleC = 440;
      obj.pLookupTable = obj.MiddleC * ...
```

```matlab
            (1+log(1:obj.NumNotes)/log(12));
    end

    function hz = stepImpl(obj,noteShift)
      % A noteShift value of 1 corresponds to obj.MiddleC
      hz = obj.pLookupTable(noteShift);
    end

    function processTunedPropertiesImpl(obj)
      propChange = isChangedProperty(obj,obj.NumNotes)||...
        isChangedProperty(obj,obj.MiddleC)
      if propChange
         obj.pLookupTable = obj.MiddleC *...
            (1+log(1:obj.NumNotes)/log(12));
      end
    end
  end
end
```

See Also

processTunedPropertiesImpl

Release System Object Resources

This example shows how to release resources allocated and used by the System object. These resources include allocated memory, files used for reading or writing, etc.

Release Memory by Clearing the Object

This method allows you to clear the axes on the Whiteboard figure window while keeping the figure open.

```
methods
   function releaseImpl(obj)
      cla(Whiteboard.getWhiteboard());
      hold on
   end
end
```

Complete Class Definition File with Released Resources

```
classdef Whiteboard < matlab.System
% Whiteboard Draw lines on a figure window
%
% This System object™ shows the use of StringSets
%
   properties
     Color = 'blue'
   end

   properties (Hidden)
     % Let user choose a color
     ColorSet = matlab.system.StringSet({'red','blue','green'});
   end

   methods (Access = protected)
     function stepImpl(obj)
         h = Whiteboard.getWhiteboard();
         plot(h, ...
           randn([2,1]), randn([2,1]), ...
           'Color',obj.Color(1));
     end

     function releaseImpl(obj)
         cla(Whiteboard.getWhiteboard());
         hold on
```

```
        end
    end

    methods (Static)
      function a = getWhiteboard()
          h = findobj('tag','whiteboard');
          if isempty(h)
            h = figure('tag','whiteboard');
            hold on
          end
          a = gca;
      end
    end
end
```

See Also
releaseImpl

Related Examples
· "Initialize Properties and Setup One-Time Calculations" on page 11-17

Define Composite System Objects

This example shows how to define System objects that include other System objects.

This example defines a bandpass filter System object from separate highpass and lowpass filter System objects.

Store System Objects in Properties

To define a System object from other System objects, store those other objects in your class definition file as properties. In this example, the highpass and lowpass filters are the separate System objects defined in their own class-definition files.

```
properties (Access = private)
   % Properties that hold filter System objects
   pLowpass
   pHighpass
end
```

Complete Class Definition File of Bandpass Filter Composite System Object

```
classdef BandpassFIRFilter < matlab.System
% Implements a bandpass filter using a cascade of eighth-order lowpass
% and eighth-order highpass FIR filters.

    properties (Access = private)
        % Properties that hold filter System objects
        pLowpass
        pHighpass
    end

    methods (Access = protected)
        function setupImpl(obj)
            % Setup composite object from constituent objects
            obj.pLowpass = LowpassFIRFilter;
            obj.pHighpass = HighpassFIRFilter;
        end

        function yHigh = stepImpl(obj,u)
            yLow = step(obj.pLowpass,u);
            yHigh = step(obj.pHighpass,yLow);
        end

        function resetImpl(obj)
```

```
                    reset(obj.pLowpass);
                    reset(obj.pHighpass);
                end
        end
end
```

Class Definition File for Lowpass FIR Component of Bandpass Filter

```
classdef LowpassFIRFilter < matlab.System
% Implements eighth-order lowpass FIR filter with 0.6pi cutoff

    properties (Nontunable)
        % Filter coefficients
        Numerator = [0.006,-0.0133,-0.05,0.26,0.6,0.26,-0.05,-0.0133,0.006];
    end

    properties (DiscreteState)
        State
    end

    methods (Access = protected)
        function setupImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end
        function y = stepImpl(obj,u)
            [y,obj.State] = filter(obj.Numerator,1,u,obj.State);
        end
        function resetImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end
    end
end
```

Class Definition File for Highpass FIR Component of Bandpass Filter

```
classdef HighpassFIRFilter < matlab.System
% Implements eighth-order highpass FIR filter with 0.4pi cutoff

    properties (Nontunable)
        % Filter coefficients
        Numerator = [0.006,0.0133,-0.05,-0.26,0.6,-0.26,-0.05,0.0133,0.006];
    end

    properties (DiscreteState)
        State
```

```
    end

    methods (Access = protected)
        function setupImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end

        function y = stepImpl(obj,u)
            [y,obj.State] = filter(obj.Numerator,1,u,obj.State);
        end

        function resetImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end
    end
end
```

See Also

nargin

Define Finite Source Objects

This example shows how to define a System object that performs a specific number of steps or specific number of reads from a file.

Use the FiniteSource Class and Specify End of the Source

1   Subclass from finite source class.

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
```

2   Specify the end of the source with the `isDoneImpl` method. In this example, the source has two iterations.

```
methods (Access = protected)
   function bDone = isDoneImpl(obj)
      bDone = obj.NumSteps==2
   end
```

Complete Class Definition File with Finite Source

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
  % RunTwice System object that runs exactly two times
  %
  properties (Access = private)
    NumSteps
  end

  methods (Access = protected)
    function resetImpl(obj)
      obj.NumSteps = 0;
    end

    function y = stepImpl(obj)
      if ~obj.isDone()
        obj.NumSteps = obj.NumSteps + 1;
        y = obj.NumSteps;
      else
        y = 0;
      end
    end

    function bDone = isDoneImpl(obj)
```

```
        bDone = obj.NumSteps==2;
      end
   end
end
```

See Also

matlab.system.mixin.FiniteSource

More About

- "What Are Mixin Classes?" on page 11-98
- "Subclass Multiple Classes"
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Save System Object

This example shows how to save a System object.

Save System Object and Child Object

Define a `saveObjectImpl` method to specify that more than just public properties should be saved when the user saves a System object. Within this method, use the default `saveObjectImpl@matlab.System` to save public properties to the struct, s. Use the `saveObject` method to save child objects. Save protected and dependent properties, and finally, if the object is locked, save the object's state.

```
methods (Access = protected)
  function s = saveObjectImpl(obj)
    s = saveObjectImpl@matlab.System(obj);
    s.child = matlab.System.saveObject(obj.child);
    s.protectedprop = obj.protectedprop;
    s.pdependentprop = obj.pdependentprop;
    if isLocked(obj)
      s.state = obj.state;
    end
  end
end
```

Complete Class Definition Files with Save and Load

The `Counter` class definition file sets up an object with a count property. This counter is used in the `MySaveLoader` class definition file to count the number of child objects.

```
classdef Counter < matlab.System
  properties(DiscreteState)
    Count
  end
  methods (Access=protected)
    function setupImpl(obj, ~)
      obj.Count = 0;
    end
    function y = stepImpl(obj, u)
      if u > 0
        obj.Count = obj.Count + 1;
      end
      y = obj.Count;
    end
```

```
      end
    end

    classdef MySaveLoader < matlab.System

      properties (Access = private)
        child
        pdependentprop = 1
      end

      properties (Access = protected)
        protectedprop = rand;
      end

      properties (DiscreteState = true)
        state
      end

      properties (Dependent)
        dependentprop
      end

      methods
        function obj = MySaveLoader(varargin)
          obj@matlab.System();
          setProperties(obj,nargin,varargin{:});
        end

        function set.dependentprop(obj, value)
          obj.pdependentprop = min(value, 5);
        end

        function value = get.dependentprop(obj)
          value = obj.pdependentprop;
        end
      end

      methods (Access = protected)
        function setupImpl(obj)
          obj.state = 42;
          obj.child = Counter;
        end
        function out = stepImpl(obj,in)
          obj.state = in + obj.state + obj.protectedprop + obj.pdependentprop;
          out = step(obj.child, obj.state);
```

```matlab
        end
      end

      % Serialization
      methods (Access = protected)
        function s = saveObjectImpl(obj)
          % Call the base class method
          s = saveObjectImpl@matlab.System(obj);

          % Save the child System objects
          s.child = matlab.System.saveObject(obj.child);

          % Save the protected & private properties
          s.protectedprop = obj.protectedprop;
          s.pdependentprop = obj.pdependentprop;

          % Save the state only if object locked
          if isLocked(obj)
            s.state = obj.state;
          end
        end

        function loadObjectImpl(obj,s,wasLocked)
          % Load child System objects
          obj.child = matlab.System.loadObject(s.child);

          % Load protected and private properties
          obj.protectedprop = s.protectedprop;
          obj.pdependentprop = s.pdependentprop;

          % Load the state only if object locked
          if wasLocked
            obj.state = s.state;
          end

          % Call base class method to load public properties
          loadObjectImpl@matlab.System(obj,s,wasLocked);
        end
      end
    end
```

See Also
loadObjectImpl | saveObjectImpl

Related Examples

- "Load System Object" on page 11-46

Load System Object

This example shows how to load and save a System object.

Load System Object and Child Object

Define a `loadObjectImpl` method to load a previously saved System object. Within this method, use the `matlab.System.loadObject` to load the child System object, load protected and private properties, load the state if the object is locked, and use `loadObjectImpl` from the base class to load public properties.

```
methods (Access = protected)
  function loadObjectImpl(obj,s,wasLocked)
    obj.child = matlab.System.loadObject(s.child);

    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    if wasLocked
      obj.state = s.state;
    end

    loadObjectImpl@matlab.System(obj,s,wasLocked);
  end
end
```

Complete Class Definition Files with Save and Load

The Counter class definition file sets up an object with a count property. This counter is used in the MySaveLoader class definition file to count the number of child objects.

```
classdef Counter < matlab.System
  properties(DiscreteState)
    Count
  end
  methods (Access=protected)
    function setupImpl(obj, ~)
      obj.Count = 0;
    end
    function y = stepImpl(obj, u)
      if u > 0
        obj.Count = obj.Count + 1;
      end
      y = obj.Count;
```

```matlab
    end
  end
end

classdef MySaveLoader < matlab.System

  properties (Access = private)
    child
    pdependentprop = 1
  end

  properties (Access = protected)
    protectedprop = rand;
  end

  properties (DiscreteState = true)
    state
  end

  properties (Dependent)
    dependentprop
  end

  methods
    function obj = MySaveLoader(varargin)
      obj@matlab.System();
      setProperties(obj,nargin,varargin{:});
    end

    function set.dependentprop(obj, value)
      obj.pdependentprop = min(value, 5);
    end

    function value = get.dependentprop(obj)
      value = obj.pdependentprop;
    end
  end

  methods (Access = protected)
    function setupImpl(obj)
      obj.state = 42;
      obj.child = Counter;
    end
    function out = stepImpl(obj,in)
      obj.state = in + obj.state + obj.protectedprop + obj.pdependentprop;
```

```matlab
      out = step(obj.child, obj.state);
    end
  end

  % Serialization
  methods (Access = protected)
    function s = saveObjectImpl(obj)
      % Call the base class method
      s = saveObjectImpl@matlab.System(obj);

      % Save the child System objects
      s.child = matlab.System.saveObject(obj.child);

      % Save the protected & private properties
      s.protectedprop = obj.protectedprop;
      s.pdependentprop = obj.pdependentprop;

      % Save the state only if object locked
      if isLocked(obj)
        s.state = obj.state;
      end
    end

    function loadObjectImpl(obj,s,wasLocked)
      % Load child System objects
      obj.child = matlab.System.loadObject(s.child);

      % Load protected and private properties
      obj.protectedprop = s.protectedprop;
      obj.pdependentprop = s.pdependentprop;

      % Load the state only if object locked
      if wasLocked
        obj.state = s.state;
      end

      % Call base class method to load public properties
      loadObjectImpl@matlab.System(obj,s,wasLocked);
    end
  end
end
```

```
end
```

See Also

loadObjectImpl | saveObjectImpl

Related Examples

- "Save System Object" on page 11-42

Define System Object Information

This example shows how to define information to display for a System object.

Define System Object Info

You can define your own `info` method to display specific information for your System object. The default `infoImpl` method returns an empty struct. This `infoImpl` method returns detailed information when the `info` method is called using `info(x,'details')` or only count information if it is called using `info(x)`.

```
methods (Access = protected)
   function s = infoImpl(obj,varargin)
      if nargin>1 && strcmp('details',varargin(1))
         s = struct('Name','Counter',...
             'Properties', struct('CurrentCount', ...
              obj.pCount,'Threshold',obj.Threshold));
      else
         s = struct('Count',obj.pCount);
      end
   end
end
```

Complete Class Definition File with InfoImpl

```
classdef Counter < matlab.System
   % Counter Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = 0;
      end

      function resetImpl(obj)
         obj.Count = 0;
      end
```

```matlab
function y = stepImpl(obj,u)
   if (u > obj.Threshold)
      obj.Count = obj.Count + 1;
   end
   y = obj.Count;
end

function s = infoImpl(obj,varargin)
   if nargin>1 && strcmp('details',varargin(1))
      s = struct('Name','Counter',...
      'Properties', struct('CurrentCount', ...
      obj.pCount,'Threshold',obj.Threshold));
   else
      s = struct('Count',obj.pCount);
   end
end
   end
end
```

See Also
infoImpl

Define MATLAB System Block Icon

This example shows how to define the block icon of a System object–based block implemented using a `MATLAB System` block.

Use the CustomIcon Class and Define the Icon

1 Subclass from custom icon class.

```
classdef MyCounter < matlab.System & ...
   matlab.system.mixin.CustomIcon
```

2 Use `getIconImpl` to specify the block icon as `New Counter` with a line break (\n) between the two words.

```
methods (Access = protected)
   function icon = getIconImpl(~)
      icon = sprintf('New\nCounter');
   end
end
```

Complete Class Definition File with Defined Icon

```
classdef MyCounter < matlab.System & ...
   matlab.system.mixin.CustomIcon

   % MyCounter Count values above a threshold

   properties
      Threshold = 1
   end
   properties (DiscreteState)
      Count
   end

   methods
      function obj = MyCounter(varargin)
         setProperties(obj,nargin,varargin{:});
      end
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = 0;
      end
```

```
        function resetImpl(obj)
            obj.Count = O;
        end
        function y = stepImpl(obj,u)
            if (u > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
        function icon = getIconImpl(~)
            icon = sprintf('New\nCounter');
        end
    end
end
```

See Also

matlab.system.mixin.CustomIcon | getIconImpl

More About

- "What Are Mixin Classes?" on page 11-98
- "Subclass Multiple Classes"
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Add Header to MATLAB System Block

This example shows how to add a header panel to a System object–based block implemented using a `MATLAB System` block.

Define Header Title and Text

This example shows how to use `getHeaderImpl` to specify a panel title and text for the `MyCounter` System object.

If you do not specify the `getHeaderImpl`, the block does not display any title or text for the panel.

You always set the `getHeaderImpl` method access to `protected` because it is an internal method that end users do not directly call or run.

```matlab
methods (Static, Access = protected)
   function header = getHeaderImpl
      header = matlab.system.display.Header('MyCounter',...
        'Title','My Enhanced Counter');
   end
end
```

Complete Class Definition File with Defined Header

```matlab
classdef MyCounter < matlab.System

  % MyCounter Count values

   properties
      Threshold = 1
   end
   properties (DiscreteState)
      Count
   end

   methods (Static, Access = protected)
     function header = getHeaderImpl
       header = matlab.system.display.Header('MyCounter',...
         'Title','My Enhanced Counter',...
         'Text', 'This counter is an enhanced version.');
     end
   end
```

```
    methods (Access = protected)
        function setupImpl(obj,u)
            obj.Count = O;
        end
        function y = stepImpl(obj,u)
            if (u > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
        function resetImpl(obj)
            obj.Count = O;
        end
    end
end
```

See Also

matlab.system.display.Header | getHeaderImpl

Add Data Types Tab to MATLAB System Block

This example shows how to add a Data Types tab to the MATLAB System block dialog box. This tab includes fixed-point data type settings.

Display Data Types Tab

This example shows how to use matlab.system.showFiSettingsImpl to display the Data Types tab in the MATLAB System block dialog.

```
methods (Static, Access = protected)
    function showTab = showFiSettingsImpl
        showTab = true;
    end
end
```

Complete Class Definition File with Data Types Tab

Use showFiSettingsImpl to display the Data Types tab for a System object that adds an offset to a fixed-point input.

```
classdef FiTabAddOffset < matlab.System
% FiTabAddOffset Add an offset to input

    properties
        Offset = 1;
    end

    methods
        function obj = FiTabAddOffset(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end

    methods (Access = protected)
        function y = stepImpl(~,u)
            y = u + obj.Offset;
        end
    end

    methods(Static, Access=protected)
        function header = getHeaderImpl
            header = matlab.system.display.Header('Title',...
                'Add Offset','Text','Add an offset to the input');
```

```
        end

        function isVisible = showFiSettingsImpl
            isVisible = true;
        end
    end
end
```
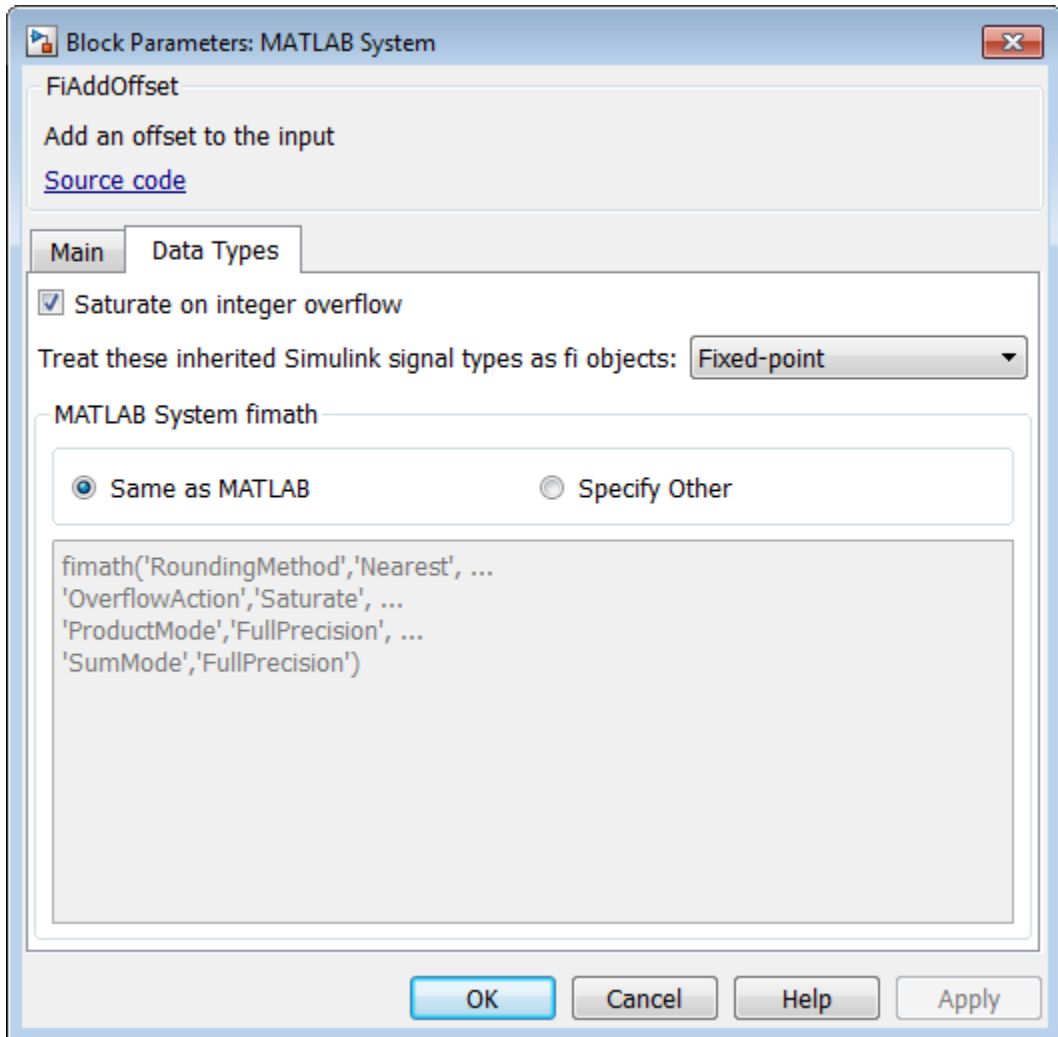
Add Property Groups to System Object and MATLAB System Block

This example shows how to define property sections and section groups for System object display. The sections and section groups display as panels and tabs, respectively, in the MATLAB System block dialog.

Define Section of Properties

This example shows how to use `matlab.system.display.Section` and `getPropertyGroupsImpl` to define two property group sections by specifying their titles and property lists.

If you do not specify a property in `getPropertyGroupsImpl`, the block does not display that property.

```
methods (Static, Access = protected)
   function groups = getPropertyGroupsImpl
      valueGroup = matlab.system.display.Section(...
            'Title','Value parameters',...
            'PropertyList',{'StartValue','EndValue'});

      thresholdGroup = matlab.system.display.Section(...
            'Title','Threshold parameters',...
            'PropertyList',{'Threshold','UseThreshold'});
      groups = [valueGroup,thresholdGroup];
   end
end
```

Define Group of Sections

This example shows how to use `matlab.system.display.SectionGroup`, `matlab.system.display.Section`, and `getPropertyGroupsImpl` to define two tabs, each containing specific properties.

```
methods (Static, Access = protected)
   function groups = getPropertyGroupsImpl
      upperGroup = matlab.system.display.Section(...
            'Title', 'Upper threshold', ...
            'PropertyList',{'UpperThreshold'});
      lowerGroup = matlab.system.display.Section(...
            'Title','Lower threshold', ...
            'PropertyList',{'UseLowerThreshold','LowerThreshold'});

      thresholdGroup = matlab.system.display.SectionGroup(...
```

```matlab
            'Title', 'Parameters', ...
            'Sections',  [upperGroup,lowerGroup]);

     valuesGroup = matlab.system.display.SectionGroup(...
            'Title', 'Initial conditions', ...
            'PropertyList',  {'StartValue'});

     groups = [thresholdGroup, valuesGroup];
   end
end
```

Complete Class Definition File with Property Group and Separate Tab

```matlab
classdef EnhancedCounter < matlab.System
   % EnhancedCounter   Count values considering thresholds

   properties
      UpperThreshold = 1;
      LowerThreshold = 0;
   end

   properties (Nontunable)
      StartValue = 0;
   end

   properties(Logical,Nontunable)
      % Count values less than lower threshold
      UseLowerThreshold = true;
   end

   properties (DiscreteState)
      Count;
   end

   methods (Static, Access = protected)
      function groups = getPropertyGroupsImpl
         upperGroup = matlab.system.display.Section(...
               'Title', 'Upper threshold', ...
               'PropertyList',{'UpperThreshold'});
         lowerGroup = matlab.system.display.Section(...
               'Title','Lower threshold', ...
               'PropertyList',{'UseLowerThreshold','LowerThreshold'});

         thresholdGroup = matlab.system.display.SectionGroup(...
               'Title', 'Parameters', ...
```

```matlab
                'Sections',  [upperGroup,lowerGroup]);

            valuesGroup = matlab.system.display.SectionGroup(...
                'Title', 'Initial conditions', ...
                'PropertyList',  {'StartValue'});

            groups = [thresholdGroup, valuesGroup];
        end
    end

    methods (Access = protected)
        function setupImpl(obj)
            obj.Count = obj.StartValue;
        end

        function y = stepImpl(obj,u)
            if obj.UseLowerThreshold
                if (u > obj.UpperThreshold) || ...
                        (u <  obj.LowerThreshold)
                    obj.Count = obj.Count + 1;
                end
            else
                if (u > obj.UpperThreshold)
                    obj.Count = obj.Count + 1;
                end
            end
            y = obj.Count;
        end
        function resetImpl(obj)
            obj.Count = obj.StartValue;
        end


        function flag = isInactivePropertyImpl(obj, prop)
            flag = false;
            switch prop
                case 'LowerThreshold'
                    flag = ~obj.UseLowerThreshold;
            end
        end
    end
end
```
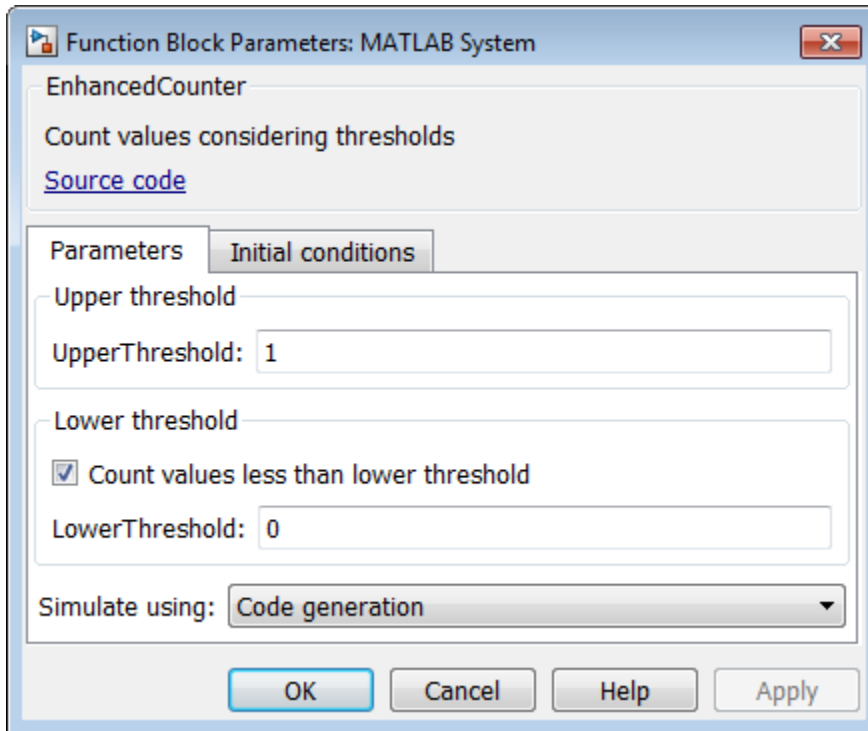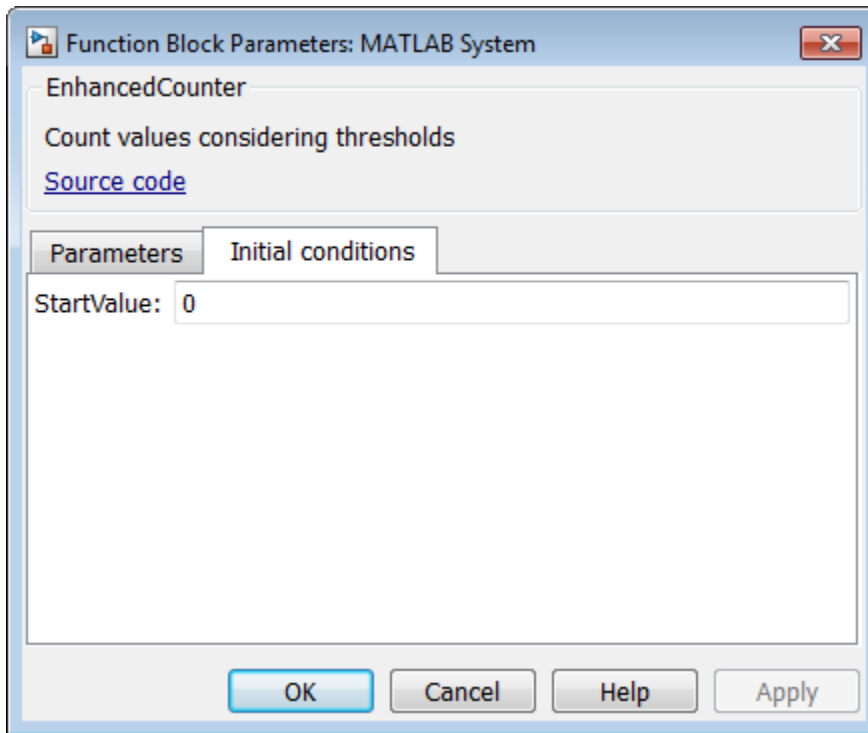
end

See Also

matlab.system.display.Section | matlab.system.display.SectionGroup |
getPropertyGroupsImpl

More About

- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Control Simulation Type in MATLAB System Block

This example shows how to specify a simulation type and whether the **Simulate using** parameter appears on the Simulink `MATLAB System` block dialog box. The simulation options are `'Code generation'` and `'Interpreted mode'`.

If you do not include the `getSimulateUsingImpl` method in your class definition file, the System object allows both simulation modes and defaults to `'Code generation'`. If you do not include the `showSimulateUsingImpl` method, the **Simulate using** parameter appears on the block dialog box.

You must set the `getSimulateUsingImpl` and `showSimulateUsingImpl` methods to `static` and the access for these methods to `protected`.

Use `getSimulateUsingImpl` to specify that only interpreted execution is allowed for the System object.

```
methods(Static,Access = protected)
   function simMode = getSimulateUsingImpl
       simMode = 'Interpreted execution';
     end
end
```

View the method in the complete class definition file.

```
classdef PlotRamp < matlab.System
  % Display a button to launch a plot figure.

  properties (Nontunable)
    RampLimit = 10;
  end

  methods(Static, Access=protected)
    function group = getPropertyGroupsImpl
      group = matlab.system.display.Section(mfilename('class'));
      group.Actions = matlab.system.display.Action(@(~,obj)...
            visualize(obj),'Label','Visualize');
    end

    function simMode = getSimulateUsingImpl
      simMode = 'Interpreted execution';
    end
  end
```
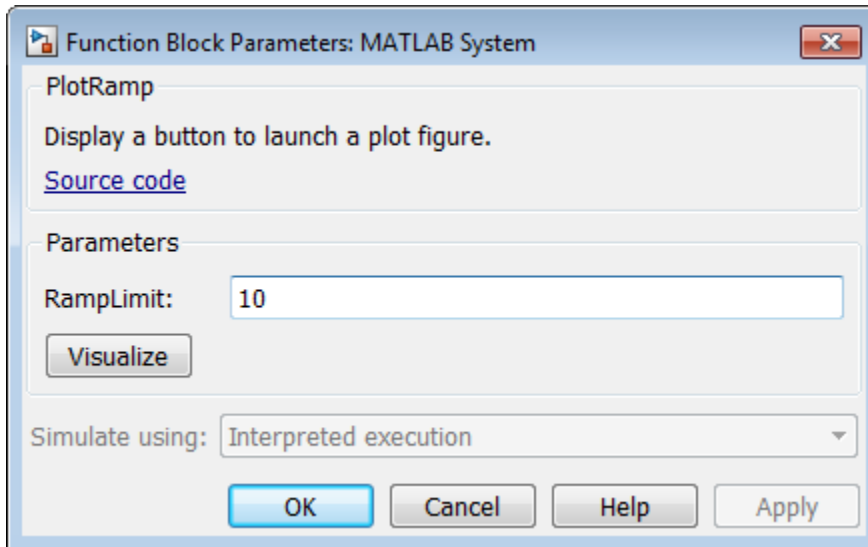
```
methods
  function obj = ActionDemo(varargin)
    setProperties(obj,nargin,varargin{:});
  end

  function visualize(obj)
    figure;
    d = 1:obj.RampLimit;
    plot(d);
  end
  methods(Static,Access = protected)
  end
end
end
```


See Also
getSimulateUsingImpl | showSimulateUsingImpl

More About
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Add Button to MATLAB System Block

This example shows how to add a button to the `MATLAB System` block dialog box. This button launches a figure that plots a ramp function.

Define Action for Dialog Button

This example shows how to use `matlab.system.display.Action` to define the MATLAB function or code associated with a button in the `MATLAB System` block dialog. The example also shows how to set button options and use an actionData object input to store a figure handle. This part of the code example uses the same figure when the button is clicked multiple times, rather than opening a new figure for each button click.

```
methods(Static,Access = protected)
  function group = getPropertyGroupsImpl
    group = matlab.system.display.Section(mfilename('class'));
    group.Actions = matlab.system.display.Action(@(actionData,obj)...
        visualize(obj,actionData),'Label','Visualize');
  end
end

methods
  function obj = ActionDemo(varargin)
    setProperties(obj,nargin,varargin{:});
  end

  function visualize(obj,actionData)
    f = actionData.UserData;
    if isempty(f) || ~ishandle(f)
      f = figure;
      actionData.UserData = f;
    else
      figure(f); % Make figure current
    end

    d = 1:obj.RampLimit;
    plot(d);
  end
end
```

Complete Class Definition File for Dialog Button

Define a property group and a second tab in the class definition file.

```matlab
classdef PlotRamp < matlab.System
  % Display a button to launch a plot figure.

  properties (Nontunable)
    RampLimit = 10;
  end

  methods(Static,Access = protected)
    function group = getPropertyGroupsImpl
      group = matlab.system.display.Section(mfilename('class'));
      group.Actions = matlab.system.display.Action(@(actionData,obj)...
        visualize(obj,actionData),'Label','Visualize');
    end
  end

  methods
    function obj = ActionDemo(varargin)
      setProperties(obj,nargin,varargin{:});
    end

    function visualize(obj,actionData)
      f = actionData.UserData;
      if isempty(f) || ~ishandle(f)
        f = figure;
        actionData.UserData = f;
      else
        figure(f); % Make figure current
      end

      d = 1:obj.RampLimit;
      plot(d);
    end
  end
end
```
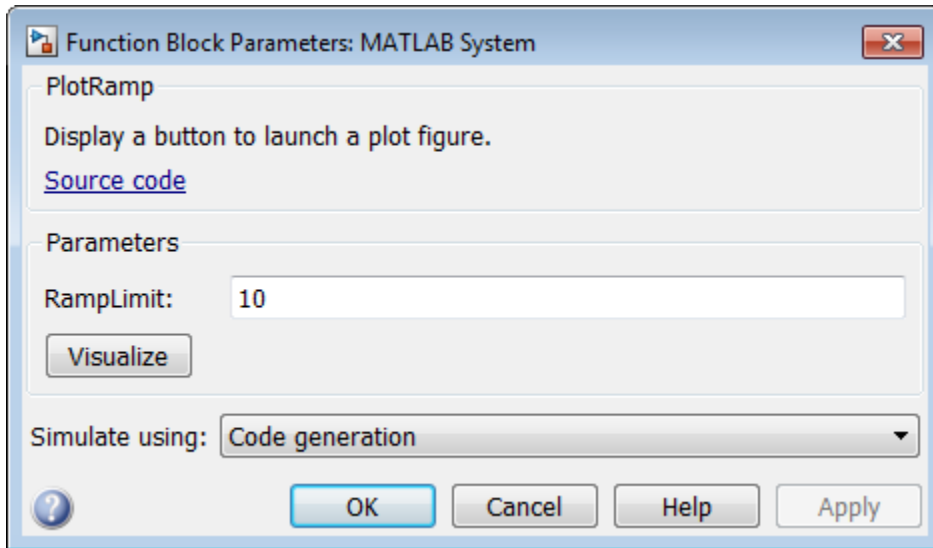
end

See Also

getPropertyGroupsImpl

More About

· "System Object Input Arguments and ~ in Code Examples" on page 11-97

Specify Locked Input Size

This example shows how to specify whether the size of a System object input is locked. The size of a locked input cannot change until the System object is unlocked. Use the `step` method and run the object to lock it. Use `release` to unlock the object.

For information on locking and unlocking, see "What You Cannot Change While Your System Is Running" .

Use the `isInputSizeLockedImpl` method to specify that the input size is locked.

```
methods (Access = protected)
  function flag = isInputSizeLockedImpl(~,~)
    flag = true;
  end
end
```

View the method in the complete class definition file.

```
classdef Counter < matlab.System
  %Counter Count values above a threshold

  properties
    Threshold = 1
  end

  properties (DiscreteState)
    Count
  end

  methods
    function obj = Counter(varargin)
      setProperties(obj,nargin,varargin{:});
    end
  end

  methods (Access=protected)
    function resetImpl(obj)
      obj.Count = O;
    end

    function y = stepImpl(obj, u1)
      if (any(u1 >= obj.Threshold))
        obj.Count = obj.Count + 1;
```

```
      end
      y = obj.Count;
    end

    function flag = isInputSizeLockedImpl(~,~)
      flag = true;
    end
  end
end
```

See Also

isInputSizeLockedImpl

Set Output Size

This example shows how to specify the size of a System object output using the `getOutputSizeImpl` method. Use this method when Simulink cannot infer the output size from the inputs during model compilation.

Note: For variable-size inputs, the propagated input size from `propagatedInputSizeImpl` differs depending on the environment.

- MATLAB — When you first run step on an object, it uses the actual sizes of the inputs.

- Simulink — The maximum of all the input sizes is set before the model runs and does not change during the run.

Subclass from both the `matlab.System` base class and the `Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Use the `getOutputSizeImpl` method to specify the output size.

```
methods (Access = protected)
   function sizeout = getOutputSizeImpl(~)
      sizeout = [1 1];
   end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
```

```matlab
function setupImpl(obj)
   obj.Count = 0;
end

function y = stepImpl(obj,u1,u2)
   % Add to count if u1 is above threshold
   % Reset if u2 is true
   if (u2)
     obj.Count = 0;
   elseif (any(u1 > obj.Threshold))
     obj.Count = obj.Count + 1;
   end
   y = obj.Count;
end

function resetImpl(obj)
   obj.Count = 0;
end

function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
   if strcmp(name,'Count')
      sz = [1 1];
      dt = 'double';
      cp = false;
   else
      error(['Error: Incorrect State Name: ', name.']);
   end
end
function dataout = getOutputDataTypeImpl(~)
   dataout = 'double';
end
function sizeout = getOutputSizeImpl(~)
   sizeout = [1 1];
end
function cplxout = isOutputComplexImpl(~)
   cplxout = false;
end
function fixedout = isOutputFixedSizeImpl(~)
   fixedout = true;
end
function inLocked = isInputSizeLockedImpl(~,idx)
   if idx == 1
     inLocked = false;
   else
```

```
            inLocked = true;
        end
    end
  end
end
```

See Also

matlab.system.mixin.Propagates | getOutputSizeImpl

More About

- "What Are Mixin Classes?" on page 11-98
- "Subclass Multiple Classes"
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Set Output Data Type

This example shows how to specify the data type of a System object output using the `getOutputDataTypeImpl` method. A second example shows how to specify a gain object with bus output. Use this method when Simulink cannot infer the data type from the inputs during model compilation or when you want bus output. To use bus output, you must define the bus data type in the base work space and you must include the `getOutputDataTypeImpl` method in your class definition file.

For both examples, subclass from both the `matlab.System` base class and the `matlab.system.mixin.Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Use the `getOutputDataTypeImpl` method to specify the output data type as a double.

```
methods (Access = protected)
   function dataout = getOutputDataTypeImpl(~)
      dataout = 'double';
   end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = O;
      end

      function resetImpl(obj)
         obj.Count = O;
      end
```

```matlab
        function y = stepImpl(obj,u1,u2)
            % Add to count if u1 is above threshold
            % Reset if u2 is true
            if (u2)
                obj.Count = 0;
            elseif (u1 > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end

        function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
            if strcmp(name,'Count')
                sz = [1 1];
                dt = 'double';
                cp = false;
            else
                error(['Error: Incorrect State Name: ', name.']);
            end
        end
        function dataout = getOutputDataTypeImpl(~)
            dataout = 'double';
        end
        function sizeout = getOutputSizeImpl(~)
            sizeout = [1 1];
        end
        function cplxout = isOutputComplexImpl(~)
            cplxout = false;
        end
        function fixedout = isOutputFixedSizeImpl(~)
            fixedout = true;
        end
    end
end
```

Use the `getOutputDataTypeImpl` method to specify the output data type as a bus. Specify the bus name in a property.

```matlab
properties(Nontunable)
    OutputBusName = 'bus_name';
end

methods (Access = protected)
    function out = getOutputDataTypeImpl(obj)
```

```
        out = obj.OutputBusName;
    end
end
```

View the method in the complete class definition file. This class definition file also includes code to implement a custom icon for this object in the MATLAB System block

```
classdef busGain < matlab.System & matlab.system.mixin.Propagates
% busGain Apply a gain of two to bus input.

    properties
        GainK = 2;
    end

    properties(Nontunable)
        OutputBusName = 'bus_name';
    end

    methods (Access=protected)
        function out = stepImpl(obj,in)
            out.a = obj.GainK * in.a;
            out.b = obj.GainK * in.b;
        end

        function out = getOutputSizeImpl(obj)
            out = propagatedInputSize(obj, 1);
        end

        function out = isOutputComplexImpl(obj)
            out = propagatedInputComplexity(obj, 1);
        end

        function out = getOutputDataTypeImpl(obj)
            out = obj.OutputBusName;
        end

        function out = isOutputFixedSizeImpl(obj)
            out = propagatedInputFixedSize(obj,1);
        end
    end
end
```

See Also

matlab.system.mixin.Propagates | getOutputDataTypeImpl

More About

Set Output Complexity

This example shows how to specify whether a System object output is complex or real using the `isOutputComplexImpl` method. Use this method when Simulink cannot infer the output complexity from the inputs during model compilation.

Subclass from both the `matlab.System` base class and the `Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Use the `isOutputComplexImpl` method to specify that the output is real.

```
methods (Access = protected)
   function cplxout = isOutputComplexImpl(~)
      cplxout = false;
   end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = 0;
      end

      function resetImpl(obj)
         obj.Count = 0;
      end

      function y = stepImpl(obj,u1,u2)
         % Add to count if u1 is above threshold
         % Reset if u2 is true
```

```matlab
            if (u2)
                obj.Count = 0;
            elseif (u1 > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end

        function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
            if strcmp(name,'Count')
                sz = [1 1];
                dt = 'double';
                cp = false;
            else
                error(['Error: Incorrect State Name: ', name.']);
            end
        end
        function dataout = getOutputDataTypeImpl(~)
            dataout = 'double';
        end
        function sizeout = getOutputSizeImpl(~)
            sizeout = [1 1];
        end
        function cplxout = isOutputComplexImpl(~)
            cplxout = false;
        end
        function fixedout = isOutputFixedSizeImpl(~)
            fixedout = true;
        end
    end
end
```

See Also

matlab.system.mixin.Propagates | isOutputComplexImpl

More About

- "What Are Mixin Classes?" on page 11-98
- "Subclass Multiple Classes"
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Specify Whether Output Is Fixed- or Variable-Size

This example shows how to specify that System object output is fixed- or variable-size.

This example shows how to specify that a System object output is fixed-size. Use the `isOutputFixedSizeImpl` method when Simulink cannot infer the output type from the inputs during model compilation.

Subclass from both the `matlab.System` base class and the `matlab.system.mixin.Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Use the `isOutputFixedSizeImpl` method to specify that the output is fixed size.

```
methods (Access = protected)
   function fixedout = isOutputFixedSizeImpl(~)
      fixedout = true;
   end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = O;
      end

      function resetImpl(obj)
         obj.Count = O;
      end

      function y = stepImpl(obj,u1,u2)
```

```matlab
            % Add to count if u1 is above threshold
            % Reset if u2 is true
            if (u2)
                obj.Count = O;
            elseif (u1 > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end

        function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
            if strcmp(name,'Count')
                sz = [1 1];
                dt = 'double';
                cp = false;
            else
                error(['Error: Incorrect State Name: ', name.']);
            end
        end
        function dataout = getOutputDataTypeImpl(~)
            dataout = 'double';
        end
        function sizeout = getOutputSizeImpl(~)
            sizeout = [1 1];
        end
        function cplxout = isOutputComplexImpl(~)
            cplxout = false;
        end
        function fixedout = isOutputFixedSizeImpl(~)
            fixedout = true;
        end
    end
end
```

This example shows how to specify that a System object output is variable-size. Use the `isOutputFixedSizeImpl` method when Simulink cannot infer the output type from the inputs during model compilation.

Subclass from both the `matlab.System` base class and the `matlab.system.mixin.Propagates` mixin class.

```matlab
 classdef CounterReset < matlab.System & ...
      matlab.system.mixin.Propagates
```

Use the `isOutputFixedSizeImpl` method to .

```
methods (Access = protected)
   function xxx = isOutputFixedSizeImpl(~)
      xxx = true;
   end
end
```

View the method in the complete class definition file.

```
classdef SimpleVarSize < matlab.System
    % untitled Add summary here
    %
    % This template includes the minimum set of functions required
    % to define a System object with discrete state.

    properties
        % Public, tunable properties.
    end

    properties (DiscreteState)
    end

    properties (Access = private)
        % Pre-computed constants.
    end

    methods (Access = protected)
        function y = stepImpl(obj,u)
            % Implement algorithm. Calculate y as a function of
            % input u and discrete states.
            y = u;
        end
    end
end


% % We start with a simple System object, SimpleVarSize Variable sized
% vectors in System objects simply means that the system object can support
% different sized vectors at its input for example

obj = SimpleVarSize

% obj =
%  System: SimpleVarSize

%% First call to step gives the same output because the object only copies
```

```
% the input to the output.
y = obj.step(1:5)
% y =
%     1    2    3    4    5

%% and then call with a vector of a different size
y = obj.step((1:3)')
% y =
%     1
%     2
%     3


%% By default System objects accept vectors of different sizes if the
% object's algorithm allows.  If a System object author chooses to disallow
% variable size vectors, he/she can implement the isInputSizeLockedImpl
% method.  isInputSizeLockedImpl accepts the port index and returns true
% if the input size is locked (variable sized vectors are disallowed) and
% false if the input size is not locked (variable sized vectors are
% allowed).
obj2 = SimpleLockedSize

%% Executing step the first time sets the input size for the System object
y = obj2.step(1:5)
%y =
%     1    2    3    4    5

%% Executing step again with a different size throws an error
y = obj2.step((1:5)')
% Error using SimpleLockedSize
% Changing the size on input 1 is not allowed without first calling
% the release() method.

%% System objects can accept structures
% We can create structures as inputs to System objects
s.field1 = 1:5
s.field2 = (1:3)'
% s =
%     field1: [1 2 3 4 5]
%     field2: [3x1 double]

%% step accepts structures as inputs
sout = obj2.step(s)
% sout =
```

```matlab
%      field1: [1 2 3 4 5]
%      field2: [3x1 double]

%% Since all structures are considered the same by System objects, they accept
% changes in structure field sizes and new fields regardless of the
% return of isInputSizeLocked

s2.field1 = s.field2
s2.field3 = s.field1
% s2 =
%      field1: [3x1 double]
%      field3: [1 2 3 4 5]s2.field3 = s.field1
sout2 = obj2.step(s2)
% sout2 =
%      field1: [3x1 double]
%      field3: [1 2 3 4 5]

%% System objects can accept cell arrays
% We can create cell arrays as inputs to System objects
c = {'cars','trucks'}
% c =
%     'cars'     'trucks'

%% step will not accept cells after accepting structures as they are a
% different data type.
cout = obj2.step(c)
% Error using SimpleLockedSize
% Changing the size or datatype on input 1 is not allowed without
% first calling the release() method.

% Calling release to unlock the System object allows it to accept input of
% a new data type.
obj2.release
cout = obj2.step(c)
% cout =
%     'cars'     'trucks'

%% Since cell arrays have size just like other arrays, changes in size
% is detected by System objects.
c2 = c'
% c2 =
%      'cars'
%      'trucks'
cout2 = obj2.step(c2)
```

```
% Error using SimpleLockedSize
% Changing the size on input 1 is not allowed without first calling
% the release() method.

% Note that obj2 is configured to lock its input size.

%% Calling release on the System object allows accept different size cells
obj2.release
cout2 = obj2.step(c2)
% cout2 =
%     'cars'
%     'trucks'

%% An object which does not lock its input size, such as SimpleVarSize,
% can accept cell arrays with different sizes.

obj3 = SimpleVarSize
cout3 = obj3.step(c)
% cout3 =
%     'cars'    'trucks'
cout4 = obj3.step(c2)
% cout4 =
%     'cars'
%     'trucks'
```

See Also

matlab.system.mixin.Propagates | isOutputFixedSizeImpl

More About

- "What Are Mixin Classes?" on page 11-98
- "Subclass Multiple Classes"
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Specify Discrete State Output Specification

This example shows how to specify the size, data type, and complexity of a discrete state property using the `getDiscreteStateSpecificationImpl` method. Use this method when your System object has a property with the `DiscreteState` attribute and Simulink cannot infer the output specifications during model compilation.

Subclass from both the `matlab.System` base class and from the `Propagates` mixin class.

```
classdef CounterReset < matlab.System & ...
    matlab.system.mixin.Propagates
```

Use the `getDiscreteStateSpecificationImpl` method to specify the size and data type. Also specify the complexity of a discrete state property, which is used in the counter reset example.

```
methods (Access = protected)
   function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
      sz = [1 1];
      dt = 'double';
      cp = false;
   end
end
```

View the method in the complete class definition file.

```
classdef CounterReset < matlab.System & matlab.system.mixin.Propagates
   % CounterReset Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = 0;
      end

      function resetImpl(obj)
```

```
            obj.Count = 0;
        end

        function y = stepImpl(obj,u1,u2)
            % Add to count if u1 is above threshold
            % Reset if u2 is true
            if (u2)
                obj.Count = 0;
            elseif (u1 > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end

        function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
            sz = [1 1];
            dt = 'double';
            cp = false;
        end
        function dataout = getOutputDataTypeImpl(~)
            dataout = 'double';
        end
        function sizeout = getOutputSizeImpl(~)
            sizeout = [1 1];
        end
        function cplxout = isOutputComplexImpl(~)
            cplxout = false;
        end
        function fixedout = isOutputFixedSizeImpl(~)
            fixedout = true;
        end
    end
end
```

See Also

matlab.system.mixin.Propagates | getDiscreteStateSpecificationImpl

More About

- "What Are Mixin Classes?" on page 11-98
- "Subclass Multiple Classes"
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Set Model Reference Discrete Sample Time Inheritance

This example shows how to disallow model reference discrete sample time inheritance for a System object. The System object defined in this example has one input, so by default, it allows sample time inheritance. To override the default and disallow inheritance, the class definition file for this example includes the allowModelReferenceDiscreteSampleTimeInheritanceImpl method, with its output set to false.

```
methods (Access = protected)
   function flag = ...
       allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
     flag = false;
   end
end
```

View the method in the complete class definition file.

```
classdef MyCounter < matlab.System

   % MyCounter Count values

    properties
       Threshold = 1;
    end

    properties (DiscreteState)
       Count
    end

    methods (Static, Access = protected)
      function header = getHeaderImpl
        header = matlab.system.display.Header('MyCounter',...
          'Title','My Enhanced Counter',...
          'Text', 'This counter is an enhanced version.');
      end
    end

    methods (Access = protected)
      function flag = ...
          allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
         flag = false
      end
      function setupImpl(obj,u)
```

```
            obj.Count = O;
        end
        function y = stepImpl(obj,u)
            if (u > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
        function resetImpl(obj)
            obj.Count = O;
        end
    end
end
```

See Also

matlab.System | allowModelReferenceDiscreteSampleTimeInheritanceImpl

Use Update and Output for Nondirect Feedthrough

This example shows how to implement nondirect feedthrough for a System object using the `updateImpl`, `outputImpl` and `isInputDirectFeedthroughImpl` methods. In nondirect feedthrough, the object's outputs depend only on the internal states and properties of the object, rather than the input at that instant in time. You use these methods to separate the output calculation from the state updates of a System object. This enables you to use that object in a feedback loop and prevent algebraic loops.

Subclass from the Nondirect Mixin Class

To use the `updateImpl`, `outputImpl`, and `isInputDirectFeedthroughImpl` methods, you must subclass from both the `matlab.System` base class and the `Nondirect` mixin class.

```
classdef IntegerDelaySysObj < matlab.System & ...
   matlab.system.mixin.Nondirect
```

Implement Updates to the Object

Implement an `updateImpl` method to update the object with previous inputs.

```
methods (Access = protected)
   function updateImpl(obj,u)
      obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
   end
end
```

Implement Outputs from Object

Implement an `outputImpl` method to output the previous, not the current input.

```
methods (Access = protected)
   function [y] = outputImpl(obj,~)
      y = obj.PreviousInput(end);
   end
end
```

Implement Whether Input Is Direct Feedthrough

Implement an `isInputDirectFeedthroughImpl` method to indicate that the input is nondirect feedthrough.

```
methods (Access = protected)
```

```
        function flag = isInputDirectFeedthroughImpl(~,~)
            flag = false;
        end
    end
end
```

Complete Class Definition File with Update and Output

```
classdef intDelaySysObj < matlab.System &...
        matlab.system.mixin.Nondirect &...
        matlab.system.mixin.CustomIcon
    % intDelaySysObj Delay input by specified number of samples.

    properties
        InitialOutput = 0;
    end
    properties (Nontunable)
        NumDelays = 1;
    end
    properties (DiscreteState)
        PreviousInput;
    end

    methods (Access = protected)
        function validatePropertiesImpl(obj)
            if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
                error('Number of delays must be positive non-zero scalar value.');
            end
            if (numel(obj.InitialOutput)>1)
                error('Initial Output must be scalar value.');
            end
        end

        function setupImpl(obj)
            obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
        end

        function resetImpl(obj)
            obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
        end

        function [y] = outputImpl(obj,~)
            y = obj.PreviousInput(end);
        end
        function updateImpl(obj, u)
            obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
```

```
         end
         function flag = isInputDirectFeedthroughImpl(~,~)
             flag = false;
         end
     end
end
```

See Also

matlab.system.mixin.Nondirect | isInputDirectFeedthroughImpl | outputImpl |
updateImpl

More About

- "What Are Mixin Classes?" on page 11-98
- "Subclass Multiple Classes"
- "System Object Input Arguments and ~ in Code Examples" on page 11-97

Enable For Each Subsystem Support

This example shows how to enable using a System object in a Simulink For Each subsystem. Include the `supportsMultipleInstanceImpl` method in your class definition file. This method applies only when the System object is used in Simulink via the `MATLAB System` block.

Use the `supportsMultipleInstanceImpl` method and have it return `true` to indicate that the System object supports multiple calls in a Simulink For Each subsystem.

```matlab
methods (Access = protected)
    function flag = supportsMultipleInstanceImpl(obj)
        flag = true;
    end
end
```

View the method in the complete class definition file.

```matlab
classdef RandSeed < matlab.System
% RANDSEED Random noise with seed for use in For Each subsystem

    properties (DiscreteState)
        count;
    end

    properties (Nontunable)
        seed = 20;
    end

    properties (Nontunable,Logical)
        useSeed = false;
    end

    methods (Access = protected)
        function y = stepImpl(obj,u1)
            % Initial use after reset/setup
            % and use the seed
            if (obj.useSeed && ~obj.count)
                rng(obj.seed);
            end
            obj.count = obj.count + 1;
            [m,n] = size(u1);
            % Uses default rng seed
            y = rand(m,n) + u1;
```

```
     end

   function setupImpl(obj)
       obj.count = O;
   end
   function resetImpl(obj)
       obj.count = O;
   end

    function flag = supportsMultipleInstanceImpl(obj)
      flag = obj.useSeed;
    end
  end
end
```

See Also

matlab.System | supportsMultipleInstanceImpl

Methods Timing

In this section...
"Setup Method Call Sequence" on page 11-94
"Step Method Call Sequence" on page 11-95
"Reset Method Call Sequence" on page 11-95
"Release Method Call Sequence" on page 11-96

The call sequence diagrams show the order in which actions are performed when you run the specified method. The background color or each action indicates the method type.

- White background — Sealed method
- Green background — User-implemented method
- White and green background — Sealed method that calls a user-implemented method

Setup Method Call Sequence

This hierarchy shows the actions performed when you call the `setup` method.
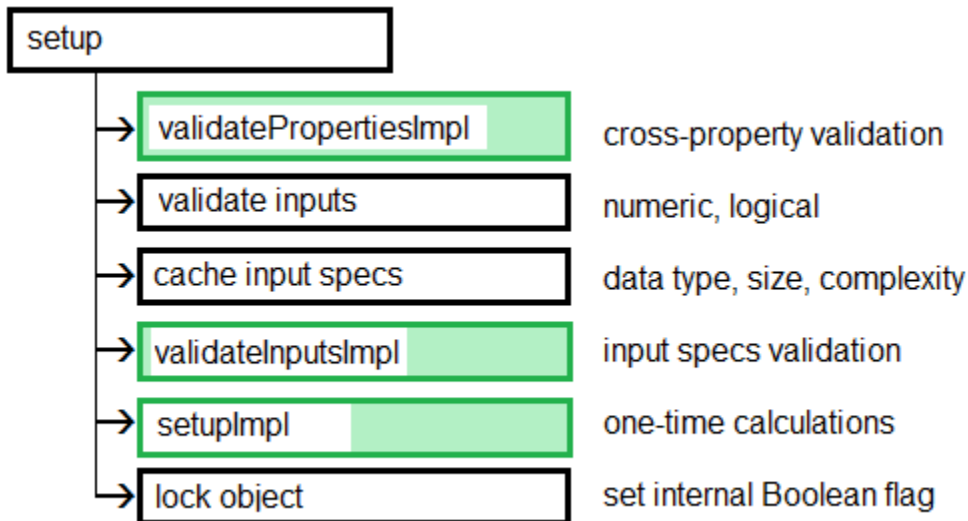
setup	
→ validatePropertiesImpl	cross-property validation
→ validate inputs	numeric, logical
→ cache input specs	data type, size, complexity
→ validateInputsImpl	input specs validation
→ setupImpl	one-time calculations
→ lock object	set internal Boolean flag

Step Method Call Sequence

This hierarchy shows the actions performed when you call the `step` method.
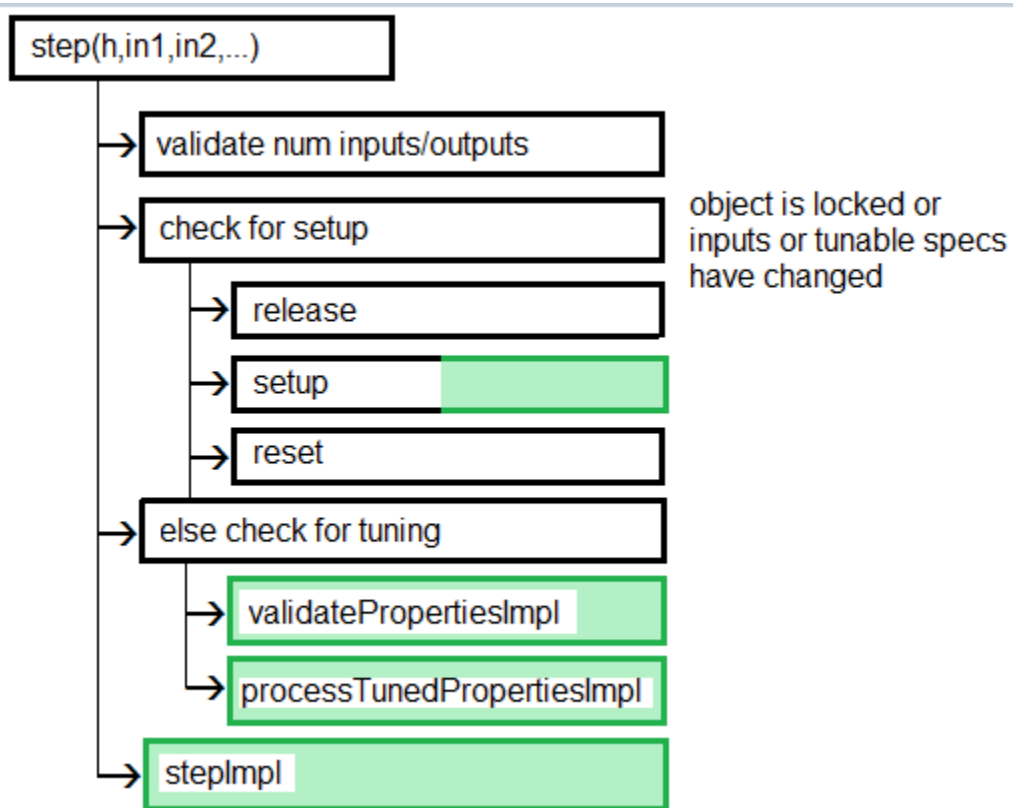
Reset Method Call Sequence

This hierarchy shows the actions performed when you call the `reset` method.
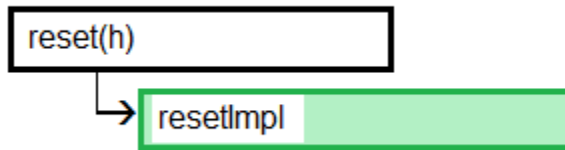
Release Method Call Sequence

This hierarchy shows the actions performed when you call the `release` method.
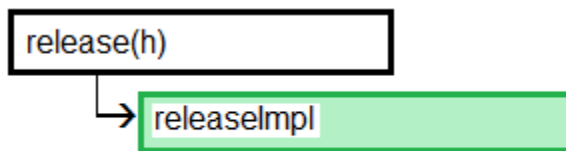
See Also
releaseImpl | resetImpl | setupImpl | stepImpl

Related Examples

- "Release System Object Resources" on page 11-35
- "Reset Algorithm State" on page 11-22
- "Set Property Values at Construction Time" on page 11-20
- "Define Basic System Objects" on page 11-5

More About

- "What Are System Object Methods?"
- "The Step Method"
- "Common Methods"

System Object Input Arguments and ~ in Code Examples

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle. In many examples, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables.

What Are Mixin Classes?

Mixin classes are partial classes that you can combine in various combinations to form desired behaviors using multiple inheritance. System objects are composed of a base class, `matlab.System` and may include one or more mixin classes. You specify the base class and mixin classes on the first line of your class definition file.

The following mixin classes are available for use with System objects.

- `matlab.system.mixin.CustomIcon` — Defines a block icon for System objects in the MATLAB System block
- `matlab.system.mixin.FiniteSource` — Adds the `isDone` method to System objects that are sources
- `matlab.system.mixin.Nondirect` — Allows the System object, when used in the MATLAB System block, to support nondirect feedthrough by making the runtime callback functions, `output` and `update` available
- `matlab.system.mixin.Propagates` — Enables System objects to operate in the MATLAB System block using the interpreted execution

Best Practices for Defining System Objects

A System object is a specialized kind of MATLAB object that is optimized for iterative processing. Use System objects when you need to call the `step` method multiple times or process data in a loop. When defining your own System object, use the following suggestions to help your code run efficiently.

- Define all one-time calculations in the `setupImpl` method and cache the results in a private property. Use the `stepImpl` method for repeated calculations.

- If properties are accessed more than once in the `stepImpl` method, cache those properties as local variables inside the method. A typical example of multiple property access is a loop. Iterative calculations using cached local variables run faster than calculations that must access the properties of an object. When the calculations for the method complete, you can save the local cached results back to the properties of that System object. Copy frequently used tunable properties into private properties. This best practice also applies to the `updateImpl` and `outputImpl` methods.

  In this example, `k` is accessed multiple times in each loop iteration, but is saved to the object property only once.

  ```
  function y = stepImpl(obj,x)
    k = obj.MyProp;
    for p=1:100
      y = k * x;
      k = k + 0.1;
    end
    obj.MyProp = k;
  end
  ```

- Property default values are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

- Do not use string comparisons or string-based switch statements in the `stepImpl` method. Instead, create a method handle in `setupImpl`. This handle points to a method in the same class definition file. Use that handle in a loop in `stepImpl`.

  This example shows how to use method handles and cached local variables in a loop to implement an efficient object. In `setupImpl`, choose `myMethod1` or `myMethod2` based on a string comparison and assign the method handle to the `pMethodHandle` property. Because there is a loop in `stepImpl`, assign the `pMethodHandle` property to a local method handle, `myFun`, and then use `myFun` inside the loop.

```
classdef MyClass < matlab.System
  function setupImpl(obj)
    if strcmp(obj.Method, 'Method1')
      obj.pMethodHandle = @myMethod1;
    else
      obj.pMethodHandle = @myMethod2;
    end
  end
  function y = stepImpl(obj,x)
    myFun = obj.pMethodHandle;
      for p=1:1000
        y = myFun(obj,x)
      end
    end
  end
  function y = myMethod1(x)
    y = x+1;
  end
  function y = myMethod2(x)
    y = x-1;
  end
end
```

- If the number of System object inputs does not change, do not implement the `getNumInputsImpl` method. Also do not implement the `getNumInputsImpl` method when you explicitly list the inputs in the `stepImpl` method instead of using `varargin`. The same caveats apply to the `getNumOutputsImpl` and `varargout` outputs.

- For the `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.

- If the variables in a method do not need to retain their values between calls use local scope for those variables in that method.

- For properties that do not change, define them in as `Nontunable` properties. `Tunable` properties have slower access times than `Nontunable` properties

- Use the `protected` or `private` attribute instead of the `public` attribute for a property, whenever possible. Some `public` properties have slower access times than `protected` and `private` properties.

- Avoid using customized `step`, `get`, or `set` methods, whenever possible.

- Avoid using string comparisons within customized `step`, `get`, or `set` methods, whenever possible. Use `setupImpl` for string comparisons instead.
- Specify Boolean values using `true` or `false` instead of `1` or `0`, respectively.
- For best practices for including System objects in code generation, see "System Objects in MATLAB Code Generation".

Insert System Object Code Using MATLAB Editor

In this section...

Define System Objects with Code Insertion

You can define System objects from the MATLAB Editor using code insertion options. When you select these options, the MATLAB Editor adds predefined properties, methods, states, inputs, or outputs to your System object. Use these tools to create and modify System objects faster, and to increase accuracy by reducing typing errors.

To access the System object editing options, create a new System object, or open an existing one.
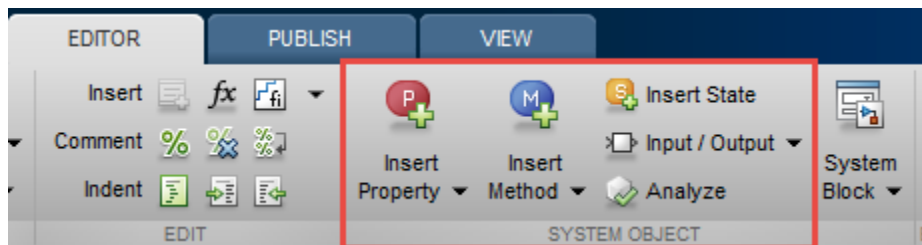
To add predefined code to your System object, select the code from the appropriate menu. For example, when you click **Insert Property** > **Numeric**, the MATLAB Editor adds the following code:
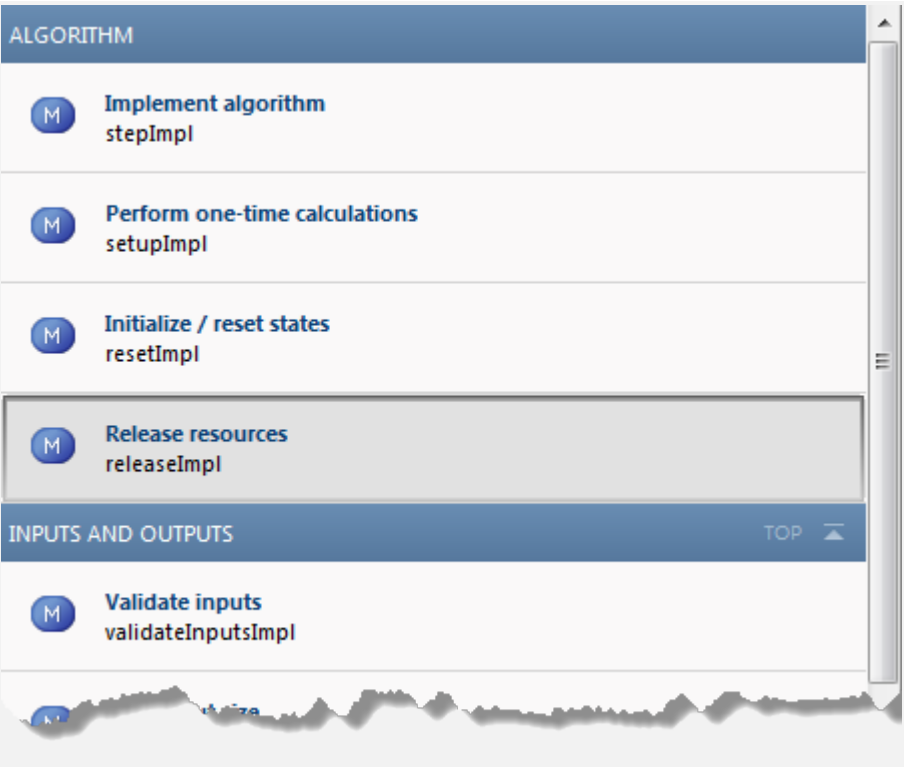
```
properties(Nontunable)
    Property
end
```

The MATLAB Editor inserts the new property with the default name `Property`, which you can rename. If you have an existing properties group with the `Nontunable`

attribute, the MATLAB Editor inserts the new property into that group. If you do not have a property group, the MATLAB Editor creates one with the correct attribute.

Insert Options

Properties	Properties of the System object: Numeric, Logical, String Set, Positive Integer, Tunable Numeric, Private, Protected, and Custom. When you select String Set or Custom Properties, a separate dialog box opens to guide you in creating these properties.
| **Methods** | Methods commonly used in System object definitions. The MATLAB Editor creates only the method structure. You specify the actions of that method.<br><br>The **Insert Method** menu organizes methods by categories, such as **Algorithm**, **Inputs and Outputs**, and **Properties and States**. When you select a method from the menu, the MATLAB Editor inserts the method template in your System object code. In this example, selecting **Insert Method** > **Release resources** inserts the following code:<br><br>```matlab
function releaseImpl(obj)
    % Release resources, such as file handles
end
```<br><br>If an method from the **Insert Method** menu is present in the System object code, that method is shown shaded on the **Insert Method** menu: |

States	Properties containing the `DiscreteState` attribute.

Inputs / Outputs	Inputs, outputs, and related methods, such as **Validate inputs** and **Lock input size**.
	
	When you select an input or output, the MATLAB Editor inserts the specified code in the `stepImpl` method. In this example, selecting **Insert > Input** causes the MATLAB Editor to insert the required input variable `u2`. The MATLAB Editor determines the variable name, but you can change it after it is inserted.
	
	``matlab
	function y = stepImpl(obj,u,u2)
	    % Implement algorithm. Calculate y as a function of input u and
	    % discrete states.
	    y = u;
	end
	``

Create Fahrenheit Temperature String Set

1 Open a new or existing System object.

2 In the MATLAB Editor, select **Insert Property** > **String Set**.

3 In the **String Set** dialog box, under **Name**, replace `Color` with `TemperatureUnit`.

4 Remove the existing `Color` property values with the **-** (minus) button.

5 Add a property value with the **+** (plus) button. Enter `Fahrenheit`.

6 Add another property value with **+**. Enter `Celsius`.

7 Add another property value with **+**. Enter `Kelvin`.

8 Select `Fahrenheit` as the default value by clicking **Default**.

The dialog box now looks as shown:

9 To create this string set and associated properties, with the default value selected, click **Insert**.

Examine the System object definition. The MATLAB Editor has added the following code:

```
properties (Nontunable)
```

```
    TemperatureUnit = 'Fahrenheit';
  end
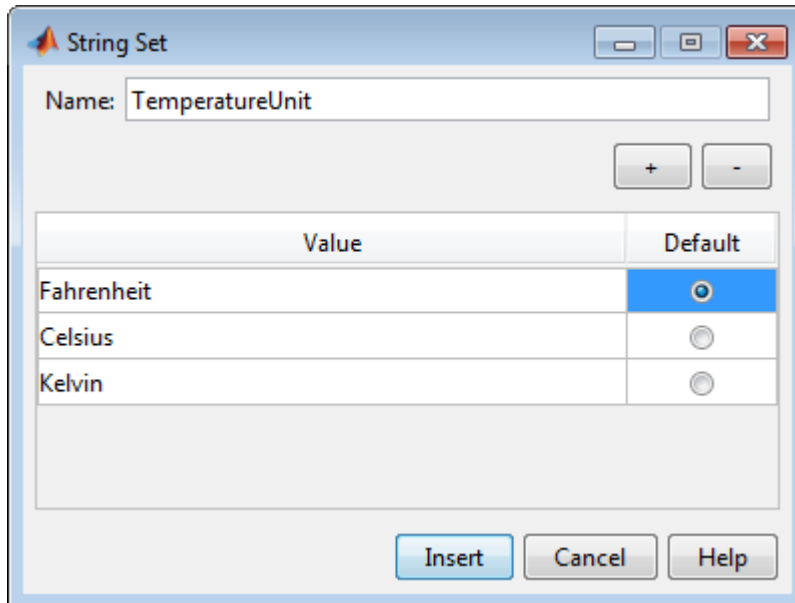

  properties(Constant, Hidden)
    TemperatureUnitSet = matlab.system.StringSet({'Fahrenheit','Celsius','Kelvin'});
  end
```

For more information on the `StringSet` class, see matlab.System.StringSet.

Create Custom Property for Freezing Point

1   Open a new or existing System object.

2   In the MATLAB Editor, select **Insert Property** > **Custom Property**.

3   In the Custom Property dialog box, under **System Object Attributes**, select
    Nontunable. Under **MATLAB Property Attributes**, select **Constant**. Leave
    GetAccess as `public`. **SetAccess** is grayed out because properties of type constant
    can not be set using System object methods.

    The dialog box now looks as shown:

4 To insert the property into the System object code, click **Insert**.

```
properties(Nontunable, Constant)
    Property
end
```

5 Replace `Property` with your property.

```
properties(Nontunable, Constant)
    FreezingPointFahrenheit = 32;
end
```

Define Input Size As Locked

1 Open a new or existing System object.

2 In the MATLAB Editor, select **Insert Method > Lock input size**.

The MATLAB Editor inserts this code into the System object:

```
function flag = isInputSizeLockedImpl(obj,index)
    % Return true if input size is not allowed to change while
    % system is running
    flag = true;
end
```

Related Examples

- "Analyze System Object Code" on page 11-109

Analyze System Object Code

View and Navigate System object Code

View and navigate System object code using the Analyzer.

The Analyzer displays all elements in your System object code.

- Navigate to a specific input, output, property, state, or method by clicking the name of that element.
- Expand or collapse element sections with the arrow buttons.
- Identify access levels for properties and custom methods with the + (public), # (protected), and – (private) symbols.

Example: Go to StepImpl Method Using Analyzer

1 Open an existing System object.
2 Select **Analyze**.
3 Click stepImpl.

The cursor in the MATLAB Editor window jumps to the `stepImpl` method.

```
        d = 1:obj.RampLimit;
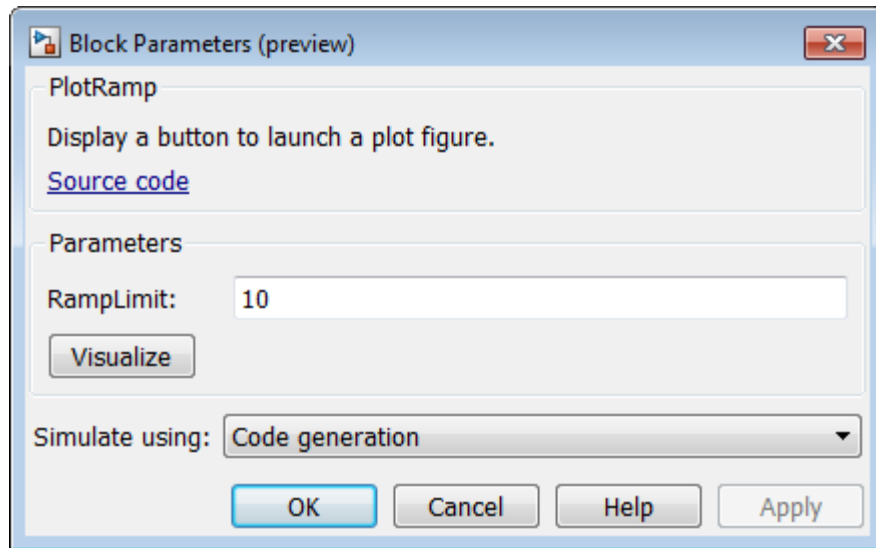        plot(d);
    end
end
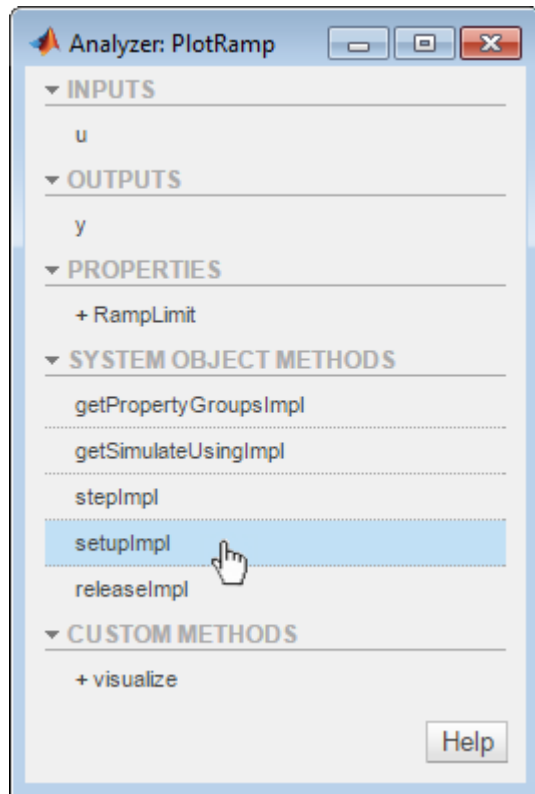
methods(Access = protected, Static)
    function group = getPropertyGroupsImpl
        % Define property section(s) for System block dialog
        group = matlab.system.display.Section(mfilename('class'));
        group.Actions = matlab.system.display.Action(@(~,obj)...
            visualize(obj),'Label','Visualize');
    end

    function simMode = getSimulateUsingImpl
        % Return only allowed simulation mode in System block dialog
        simMode = 'Interpreted execution';
    end
end

methods(Access = protected)
    function y = stepImpl(~,u)
        % Implement algorithm. Calculate y as a function of input u and
        % discrete states.
        y = u;
    end
```

Related Examples

- "Insert System Object Code Using MATLAB Editor" on page 11-102

Define System Object for Use in Simulink

In this section...
"Develop System Object for Use in System Block" on page 11-112
"Define Block Dialog Box for Plot Ramp" on page 11-113

Develop System Object for Use in System Block

You can develop a System object for use in a System block and interactively preview the block dialog box. This feature requires Simulink.

With the **System Block** editing options, the MATLAB Editor inserts predefined code into the System object. This coding technique helps you create and modify your System object faster and increases accuracy by reducing typing errors.

Using these options, you can also:

- View and interact with the block dialog design as you define the System object.
- Add dialog customization methods. If the block dialog box is open when you make changes, the block dialog design preview updates the display on saving the file.
- Add icon methods. However, these elements display only on the MATLAB System Block in Simulink, not in the Preview Dialog Box.

Define Block Dialog Box for Plot Ramp

1   Create a System object and name it `PlotRamp`. This name becomes the block dialog box title. Save the System object.

2   Add a comment that contains the block description.

    % Display a button to launch a plot figure.

    This comment becomes the block parameters dialog box description, under the block title.

3   Select **System Block** > **Preview Block Dialog**. The block dialog box displays as you develop the System object.

4 Add a ramp limit by selecting **Insert Property** > **Numeric**. Then change the property name and set the value to 10.

```
properties (Nontunable)
   RampLimit = 10;
end
```

5 Using the **System Block** menu, insert the getPropertyGrouplsImpl method.

```
methods(Access = protected, Static)
    function group = getPropertyGroupsImpl
        % Define property section(s) for System block dialog
        group = matlab.system.display.Section(mfilename('class'));
    end
end
```

6 Add code to create the group for the visualize action..

```
methods(Access = protected, Static)
    function group = getPropertyGroupsImpl
        % Define property section(s) for System block dialog
        group = matlab.system.display.Section(mfilename('class'));
        group.Actions = matlab.system.display.Action(@(~,obj)...
          visualize(obj),'Label','Visualize');
    end
end
```

7 Add a function that adds code to display the **Visualize** button on the dialog box.

```
methods
  function visualize(obj)
    figure;
```

```
        d = 1:obj.RampLimit;
        plot(d);
    end
  end
```

8 As you add elements to the System block definition, save your file. Observe the effects of your code additions to the System block definition.

The **System Block** menu also displays checks next to the methods you have implemented, which can help you track your development.

The class definition file now has all the code necessary for the PlotRamp System object.

```matlab
classdef PlotRamp < matlab.System
  % Display a button to launch a plot figure.

  properties (Nontunable)
    RampLimit = 10;
  end

  methods(Static, Access=protected)
    function group = getPropertyGroupsImpl
      group = matlab.system.display.Section(mfilename('class'));
      group.Actions = matlab.system.display.Action(@(~,obj)...
            visualize(obj),'Label','Visualize');
    end
  end

  methods
    function visualize(obj)
      figure;
      d = 1:obj.RampLimit;
      plot(d);
    end
  end
end
```
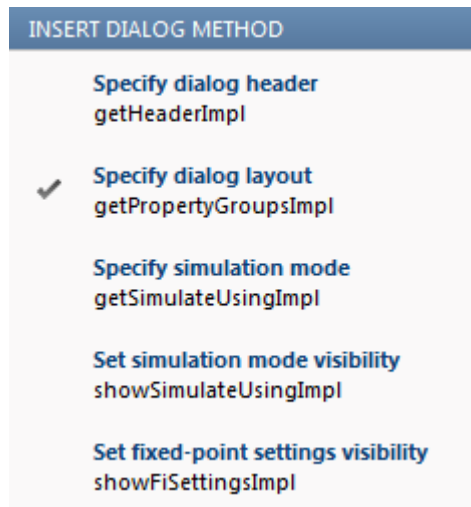
After you complete your System block definition, save it, and then load it into a MATLAB System block in Simulink.

Related Examples

·   "Insert System Object Code Using MATLAB Editor" on page 11-102

HDL Code Generation

HDL Code Generation Support for DSP System Toolbox

In this section...
"Blocks" on page 12-2
"System Objects" on page 12-3

Blocks

You can find DSP System Toolbox blocks that support HDL code generation, in the 'DSP System Toolbox HDL Support' library, in the Simulink library browser. Alternately, you can type dsphdllib in the MATLAB command prompt to open this library. The blocks in dsphdllib have their parameters set for HDL code generation.

Filtering

- Biquad Filter
- CIC Decimation
- CIC Interpolation
- DC Blocker
- Discrete FIR Filter
- FIR Decimation
- FIR Interpolation
- FIR Rate Conversion HDL Optimized
- LMS Filter

Math Functions

- Complex to Magnitude-Angle HDL Optimized

Signal Operations

- Downsample
- NCO
- NCO HDL Optimized
- Repeat
- Upsample

Signal Management

- `Convert 1-D to 2-D`
- `Data Type Conversion`
- `Frame Conversion`
- `Multiport Selector`
- `Selector`
- `Variable Selector`

Sinks

These blocks can be used for simulation visibility in models that generate HDL code, but are not included in the hardware implementation.

- `Display`
- `Matrix Viewer`
- `To Workspace`
- `Spectrum Analyzer`
- `Time Scope`
- `Vector Scope`
- `Waterfall`

Statistics

- `Maximum`
- `Minimum`

Transforms

- `FFT HDL Optimized`
- `IFFT HDL Optimized`

System Objects

HDL Coder™ supports the following DSP System Toolbox `System objects` for HDL code generation:

Filtering

- dsp.BiquadFilter
- dsp.DCBlocker
- dsp.FIRFilter
- dsp.HDLFIRRateConverter

Math Functions

- dsp.HDLComplexToMagnitudeAngle

Signal Operations

- dsp.Delay
- dsp.HDLNCO

Statistics

- dsp.Maximum
- dsp.Minimum

Transforms

- dsp.HDLFFT
- dsp.HDLIFFT

Find Blocks and System Objects Supporting HDL Code Generation

Blocks

You can find libraries of blocks supported for HDL code generation in the Simulink library browser. Find Simulink blocks that support HDL code generation, in the 'HDL Coder' library. You can also type `hdlsllib` at the MATLAB command prompt to open this library.

Create a library of HDL-supported blocks from all products you have installed, by typing `hdllib` at the MATLAB command line. This command requires an HDL Coder license. For more information on this command, see `hdllib` in the HDL Coder documentation.

Refer to the "Supported Blocks" pages in HDL Coder documentation for block implementations, properties, and restrictions for HDL code generation.

System Objects

To find `System objects` supported for HDL code generation, see Predefined System Objects in the HDL Coder documentation.

Links to Category Pages

Signal Management Library

You can find the relevant blocks in the following pages:

- "Buffers, Switches, and Counters"
- "Signal Attributes and Indexing"
- "Signal Operations"

Sinks Library

You can find the relevant blocks in the following pages:

- "Signal Import and Export"
- "Scopes and Data Logging"

Math Functions Library

You can find the relevant blocks in the following pages:

- "Array and Matrix Mathematics"
- "Linear Algebra"

Filtering Library

You can find the relevant blocks in the following pages:

- "Filter Design"
- "Single-Rate Filters"
- "Multirate and Multistage Filters"
- "Adaptive Filters"

Designing Lowpass FIR Filters

Lowpass FIR Filter Design

This example shows how to design a lowpass FIR filter using `fdesign`. An ideal lowpass filter requires an infinite impulse response. Truncating or windowing the impulse response results in the so-called window method of FIR filter design.

A Lowpass FIR Filter Design Using Various Windows

FIR filters are widely used due to the powerful design algorithms that exist for them, their inherent stability when implemented in non-recursive form, the ease with which one can attain linear phase, their simple extensibility to multirate cases, and the ample hardware support that exists for them among other reasons. This example showcases functionality in the DSP System Toolbox™ for the design of low pass FIR filters with a variety of characteristics. Many of the concepts presented here can be extended to other responses such as highpass, bandpass, etc.

Consider a simple design of a lowpass filter with a cutoff frequency of 0.4*pi radians per sample:

```
Fc = 0.4;
N = 100;
Hf = fdesign.lowpass('N,Fc',N,Fc);
```

We can design this lowpass filter using the window method. For example, we can use a Hamming window or a Dolph-Chebyshev window:

```
Hd1 = design(Hf,'window','window',@hamming,'systemobject',true);
Hd2 = design(Hf,'window','window',{@chebwin,50}, ...
             'systemobject',true);
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt,'Hamming window design', ...
       'Dolph-Chebyshev window design')
```
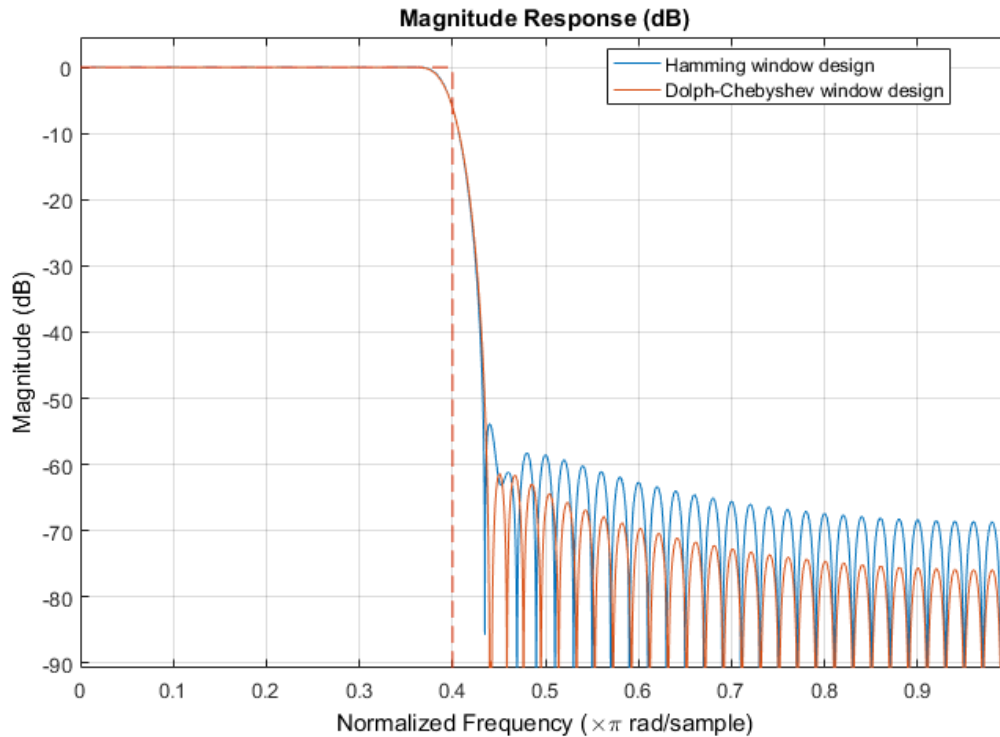
The choice of filter was arbitrary. Since ideally the order should be infinite, in general, a larger order results in a better approximation to ideal at the expense of a more costly implementation. For instance, with a Dolph-Chebyshev window, we can decrease the transition region by increasing the filter order:

```
Hf.FilterOrder = 200;
Hd3 = design(Hf,'window','window',{@chebwin,50},...
            'systemobject',true);
hfvt2 = fvtool(Hd2,Hd3,'Color','White');
legend(hfvt2,'Dolph-Chebyshev window design. Order = 100',...
    'Dolph-Chebyshev window design. Order = 200')
```
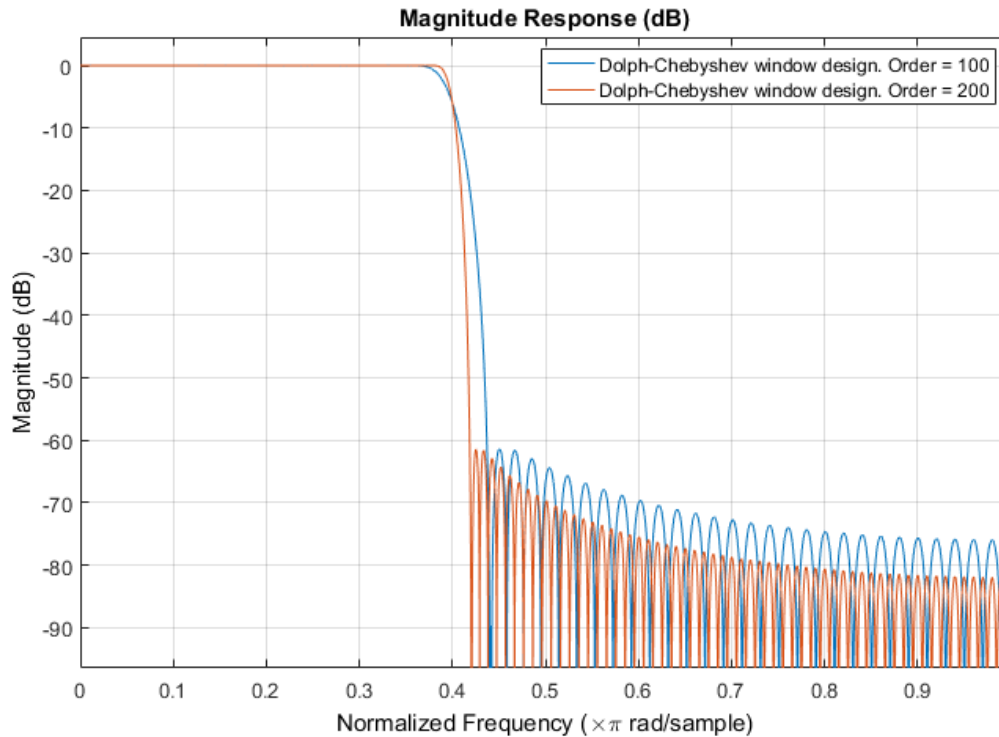
Minimum Order Lowpass Filter Design

In order to determine a suitable filter order, it is necessary to specify the amount of passband ripple and stopband attenuation that will be tolerated. It is also necessary to specify the width of the transition region around the ideal cutoff frequency. The latter is done by setting the passband edge frequency and the stopband edge frequency. The difference between the two determines the transition width.

```
Fp = 0.38;
Fst = 0.42;
Ap = 0.06;
Ast = 60;
setspecs(Hf,'Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
```

We can still use the window method, along with a Kaiser window, to design the low pass filter.

```
Hd4 = design(Hf,'kaiserwin','systemobject',true);
measure(Hd4)


ans =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.38
3-dB Point       : 0.39539
6-dB Point       : 0.4
Stopband Edge    : 0.42
Passband Ripple  : 0.016058 dB
Stopband Atten.  : 60.092 dB
Transition Width : 0.04


ans =

Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.38
3-dB Point         : 0.39539
6-dB Point         : 0.4
Stopband Edge      : 0.42
Passband Ripple    : 0.016058 dB
Stopband Atten.    : 60.092 dB
Transition Width   : 0.04
```

One thing to note is that the transition width as specified is centered around the cutoff frequency of 0.4 pi. This will become the point at which the gain of the lowpass filter is half the passband gain (or the point at which the filter reaches 6 dB of attenuation).

Optimal Minimum Order Designs

The Kaiser window design is not an optimal design and as a result the filter order required to meet the specifications using this method is larger than it needs to be. Equiripple designs result in the lowpass filter with the smallest possible order to meet a set of specifications.

```
Hd5 = design(Hf,'equiripple','systemobject',true);
hfvt3 = fvtool(Hd4,Hd5,'Color','White');
legend(hfvt3,'Kaiser window design','Equiripple design')
```
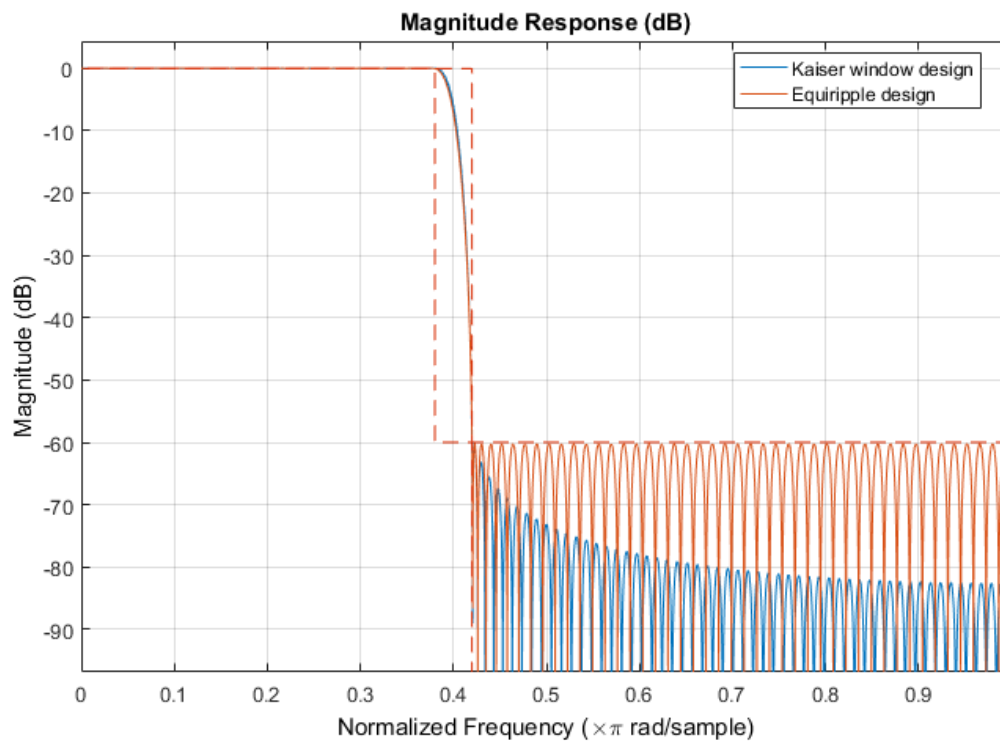
In this case, 146 coefficients are needed by the equiripple design while 183 are needed by the Kaiser window design.

Controlling Design Specifications in Lowpass FIR Design

This example shows how to control the filter order, passband ripple, stopband attenuation, and transition region width of a lowpass FIR filter.

Controlling the Filter Order and Passband Ripples and Stopband Attenuation

When targeting custom hardware, it is common to find cases where the number of coefficients is constrained to a set number. In these cases, minimum order designs are not useful because there is no control over the resulting filter order. As an example, suppose that only 101 coefficients could be used and the passband ripple/stopband attenuation specifications need to be met. We can still use equiripple designs for these specifications. However, we lose control over the transition width which will increase. This is the price to pay for reducing the order while maintaining the passband ripple/stopband attenuation specifications.

Consider a simple design of a lowpass filter with a cutoff frequency of 0.4*pi radians per sample:

```
Ap = 0.06;
Ast = 60;
Fp = 0.38;
Fst = 0.42;
Hf=fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
```

Design an equiripple filter:

```
Hd1 = design(Hf,'equiripple','systemobject',true);
```

Set the number of coefficients to 101, which means setting the order to 100:

```
N = 100;
Fc = 0.4;
setspecs(Hf,'N,Fc,Ap,Ast',N,Fc,Ap,Ast);
```

Design a second equiripple filter with the given constraint:

```
Hd2 = design(Hf,'equiripple','systemobject',true);
```

Measure the filter variables of the second equiripple filter, and compare the graphs of the first and second filters:

```
measure(Hd2)
```

```
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt,'Equiripple design, 146 coeffcients', ...
       'Equiripple design, 101 coefficients')


ans =

Sample Rate       : N/A (normalized frequency)
Passband Edge     : 0.37316
3-dB Point        : 0.39285
6-dB Point        : 0.4
Stopband Edge     : 0.43134
Passband Ripple   : 0.06 dB
Stopband Atten.   : 60 dB
Transition Width  : 0.058177
```
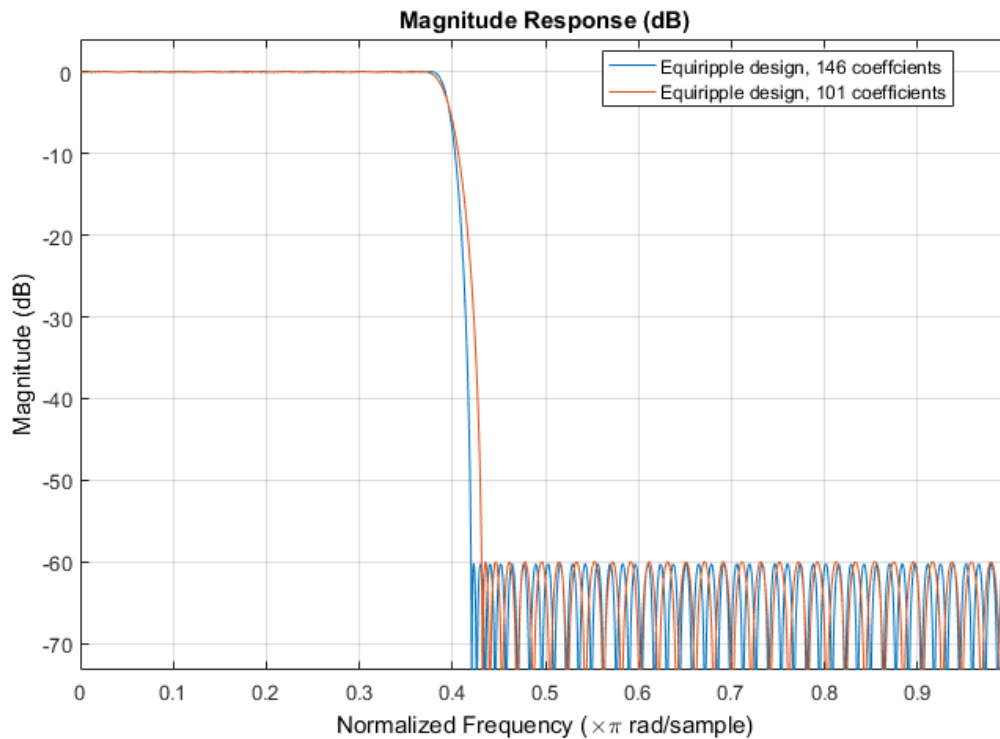
Notice that the transition has increased by almost 50%. This is not surprising given the almost 50% difference between 101 coefficients and 146 coefficients.

Controlling the Transition Region Width

Another option when the number of coefficients is set is to maintain the transition width at the expense of control over the passband ripple/stopband attenuation.

```
setspecs(Hf,'N,Fp,Fst',N,Fp,Fst);
Hd3 = design(Hf,'equiripple','systemobject',true);
measure(Hd3)
hfvt2 = fvtool(Hd1,Hd3,'Color','White');
legend(hfvt2,'Equiripple design, 146 coefficients',...
       'Equiripple design, 101 coefficients')
```

```
ans =

Sample Rate       : N/A (normalized frequency)
Passband Edge     : 0.38
3-dB Point        : 0.39407
6-dB Point        : 0.4
Stopband Edge     : 0.42
Passband Ripple   : 0.1651 dB
Stopband Atten.   : 40.4369 dB
Transition Width  : 0.04
```
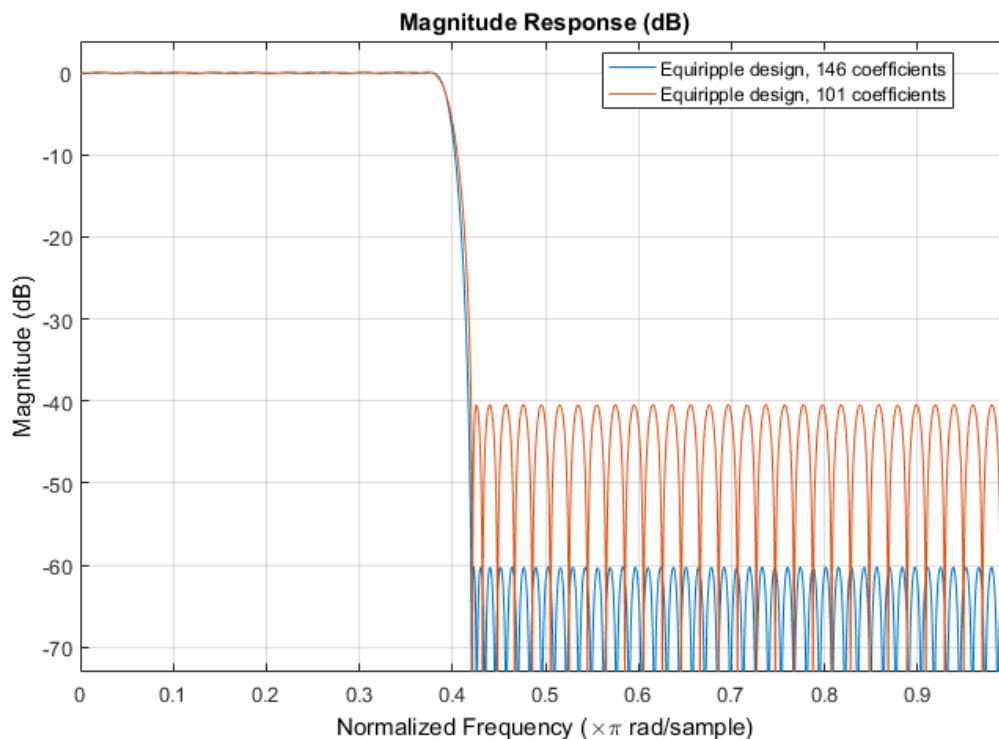
Note that in this case, the differences between using 146 coefficients and using 101 coefficients is reflected in a larger passband ripple and a smaller stopband attenuation.

It is possible to increase the attenuation in the stopband while keeping the same filter order and transition width by the use of weights. Weights are a way of specifying the relative importance of the passband ripple versus the stopband attenuation. By default, passband and stopband are equally weighted (a weight of one is assigned to each). If we increase the stopband weight, we can increase the stopband attenuation at the expense of increasing the stopband ripple as well.

```
Hd4 = design(Hf,'equiripple','Wstop',5,'systemobject',true);
measure(Hd4)
hfvt3 = fvtool(Hd3,Hd4,'Color','White');
legend(hfvt3,'Passband weight = 1, Stopband weight = 1',...
       'Passband weight = 1, Stopband weight = 5')
```

```
ans =

Sample Rate        : N/A (normalized frequency)
Passband Edge      : 0.38
3-dB Point         : 0.39143
6-dB Point         : 0.39722
Stopband Edge      : 0.42
Passband Ripple    : 0.34529 dB
Stopband Atten.    : 48.0068 dB
Transition Width   : 0.04
```
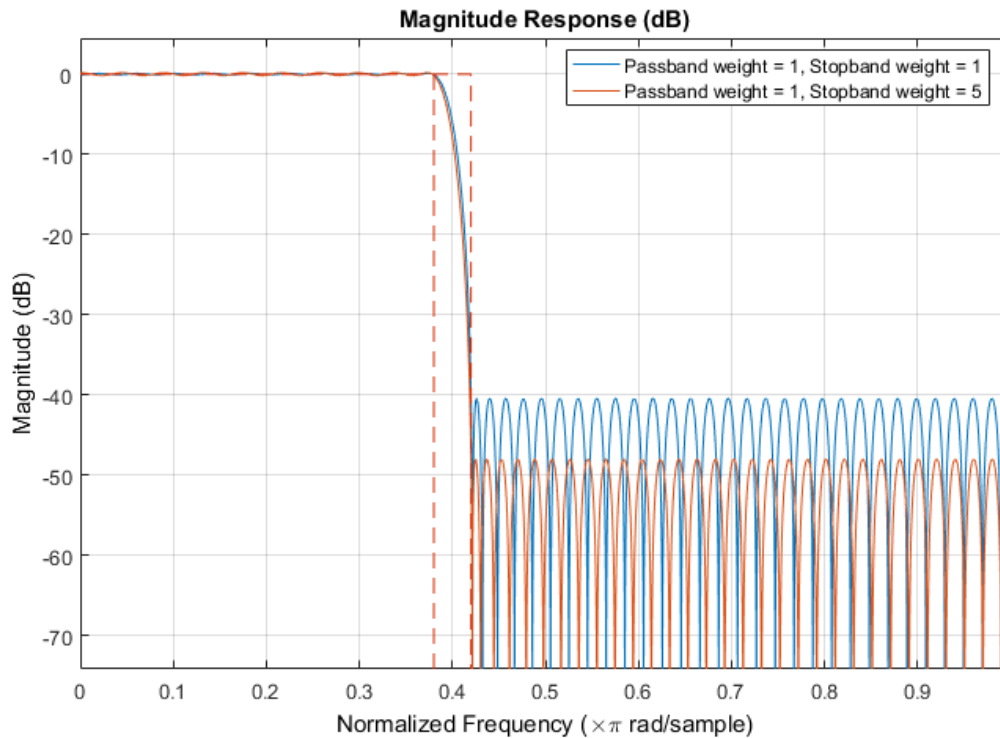

Another possibility is to specify the exact stopband attenuation desired and lose control over the passband ripple. This is a powerful and very desirable specification. One has control over most parameters of interest.

```
setspecs(Hf,'N,Fp,Fst,Ast',N,Fp,Fst,Ast);
Hd5 = design(Hf,'equiripple','systemobject',true);
hfvt4 = fvtool(Hd4,Hd5,'Color','White');
legend(hfvt4,'Equiripple design using weights',...
       'Equiripple design constraining the stopband')
```
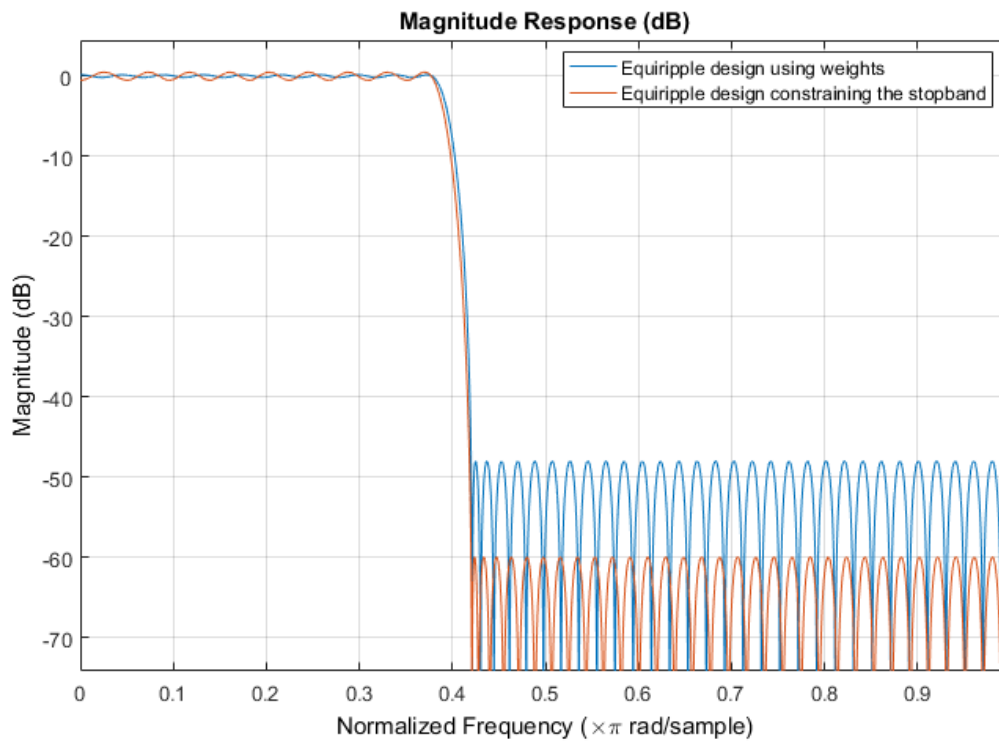
Designing Filters with Non-Equiripple Stopband

This example shows how to design lowpass filters with stopbands that are not equiripple.

Optimal Non-Equiripple Lowpass Filters

To start, set up the filter parameters and use `fdesign` to create a constructor for designing the filter.

```
N = 100;
Fp = 0.38;
Fst = 0.42;
Hf = fdesign.lowpass('N,Fp,Fst',N,Fp,Fst);
```

Equiripple designs achieve optimality by distributing the deviation from the ideal response uniformly. This has the advantage of minimizing the maximum deviation (ripple). However, the overall deviation, measured in terms of its energy tends to be large. This may not always be desirable. When low pass filtering a signal, this implies that remnant energy of the signal in the stopband may be relatively large. When this is a concern, least-squares methods provide optimal designs that minimize the energy in the stopband.

```
Hd1 = design(Hf,'equiripple','systemobject',true);
Hd2 = design(Hf,'firls','systemobject',true);
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt,'Equiripple design','Least-squares design')
```
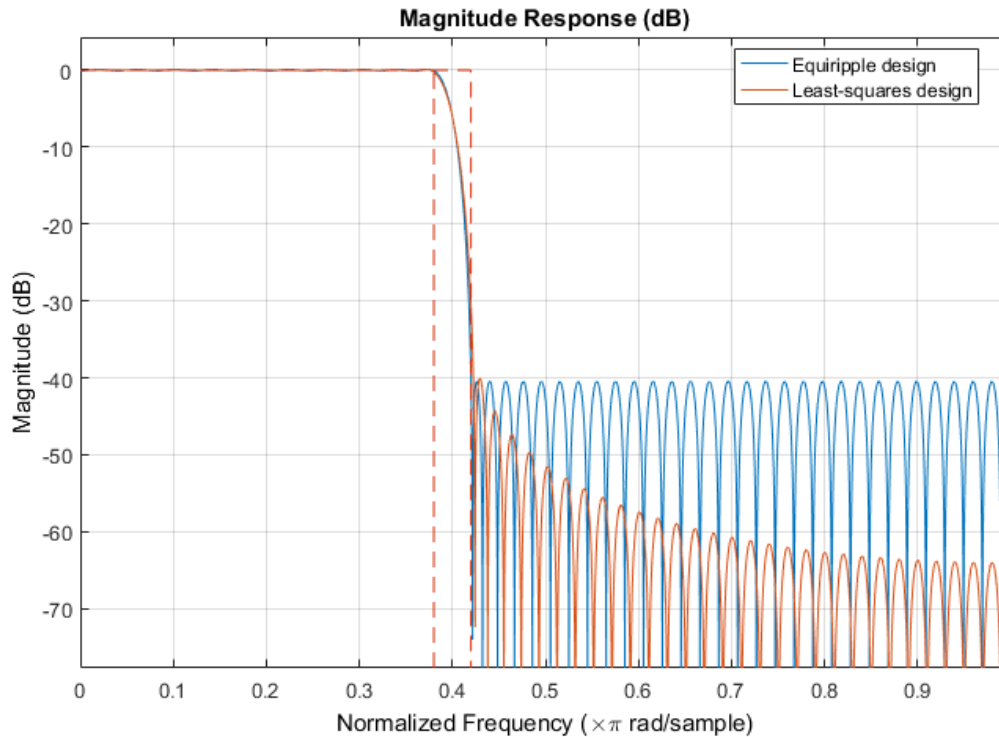
Notice how the attenuation in the stopband increases with frequency for the least-squares designs while it remains constant for the equiripple design. The increased attenuation in the least-squares case minimizes the energy in that band of the signal to be filtered.

Equiripple Designs with Increasing Stopband Attenuation

An often undesirable effect of least-squares designs is that the ripple in the passband region close to the passband edge tends to be large. For low pass filters in general, it is desirable that passband frequencies of a signal to be filtered are affected as little as possible. To this extent, an equiripple passband is generally preferable. If it is still desirable to have an increasing attenuation in the stopband, we can use design options for equiripple designs to achieve this.

```
Hd3 = design(Hf,'equiripple','StopbandShape','1/f',...
             'StopbandDecay',4,'systemobject',true);
```

```
hfvt2 = fvtool(Hd2,Hd3,'Color','White');
legend(hfvt2,'Least-squares design',...
       'Equiripple design with stopband decaying as (1/f)^4')
```


Magnitude Response (dB)
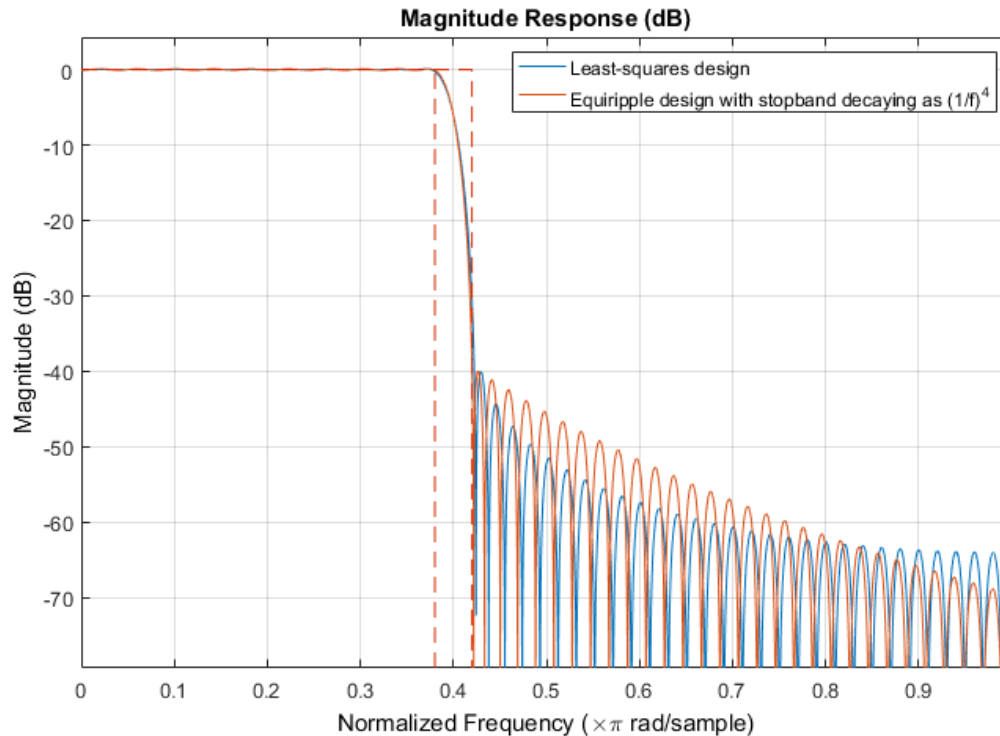
Notice that the stopbands are quite similar. However the equiripple design has a significantly smaller passband ripple,

```
mls = measure(Hd2);
meq = measure(Hd3);
mls.Apass
meq.Apass
```

ans =

    0.3504

```
ans =

    0.1867
```

Filters with a stopband that decays as (1/f)^M will decay at 6M dB per octave. Another way of shaping the stopband is using a linear decay. For example given an approximate attenuation of 38 dB at 0.4*pi, if an attenuation of 70 dB is desired at pi, and a linear decay is to be used, the slope of the line is given by (70-38)/(1-0.4) = 53.333. Such a design can be achieved from:

```
Hd4 = design(Hf,'equiripple','StopbandShape','linear',...
             'StopbandDecay',53.333,'systemobject',true);
hfvt3 = fvtool(Hd3,Hd4,'Color','White');
legend(hfvt3,'Equiripple design with stopband decaying as (1/f)^4',...
    'Equiripple design with stopband decaying linearly and a slope of 53.333')
```
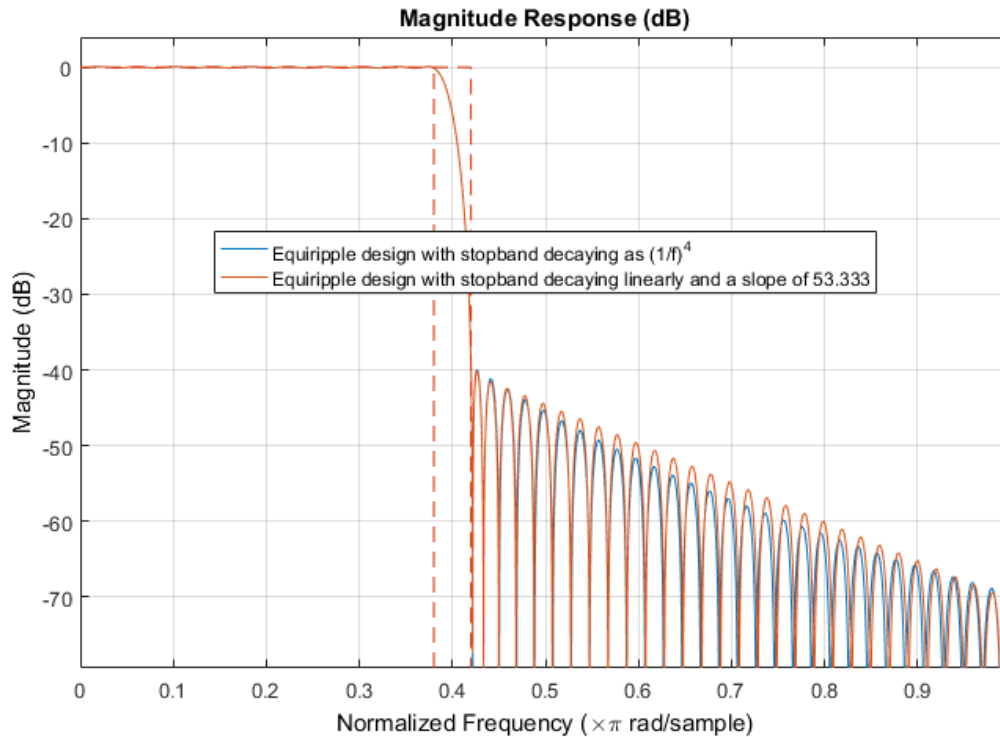
Yet another possibility is to use an arbitrary magnitude specification and select two bands (one for the passband and one for the stopband). Then, by using weights for the second band, it is possible to increase the attenuation throughout the band.

```
N = 100;
B = 2;  % number of bands
F = [0 .38 .42:.02:1];
A = [1 1 zeros(1,length(F)-2)];
W = linspace(1,100,length(F)-2);
Harb = fdesign.arbmag('N,B,F,A',N,B,F(1:2),A(1:2),F(3:end),...
                      A(3:end));
Ha = design(Harb,'equiripple','B2Weights',W,...
           'systemobject',true);
fvtool(Ha,'Color','White')
```
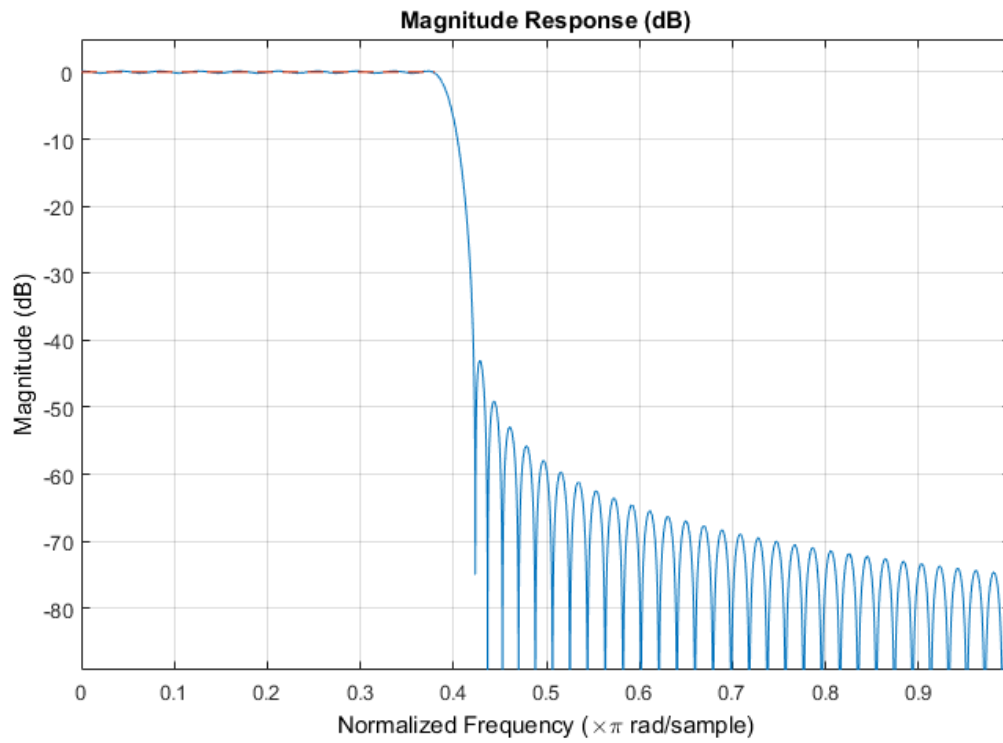
Minimizing Lowpass FIR Filter Length

This example shows how to minimize the number coefficients, by designing minimum-phase or minimum-order filters.

Minimum-Phase Lowpass Filter Design

To start, set up the filter parameters and use `fdesign` to create a constructor for designing the filter.

```
N = 100;
Fp = 0.38;
Fst = 0.42;
Ap = 0.06;
Ast = 60;
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
```

So far, we have only considered linear-phase designs. Linear phase is desirable in many applications. Nevertheless, if linear phase is not a requirement, minimum-phase designs can provide significant improvements over linear phase counterparts. For instance, returning to the minimum order case, a minimum-phase/minimum-order design for the same specifications can be computed with:

```
Hd1 = design(Hf,'equiripple','systemobject',true);
Hd2 = design(Hf,'equiripple','minphase',true,...
            'systemobject',true);
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt,'Linear-phase equiripple design',...
       'Minimum-phase equiripple design')
```
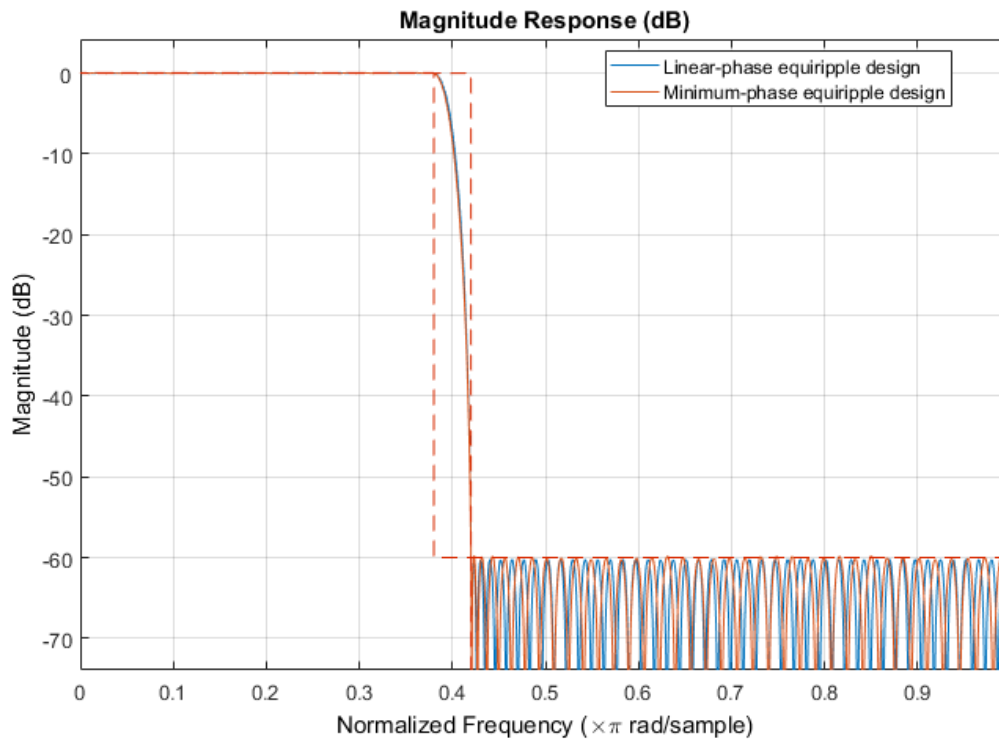
Notice that the number of coefficients has been reduced from 146 to 117. As a second example, consider the design with a stopband decaying in linear fashion. Notice the increased stopband attenuation. The passband ripple is also significantly smaller.

```
setspecs(Hf,'N,Fp,Fst',N,Fp,Fst);
Hd3 = design(Hf,'equiripple','StopbandShape','linear',...
    'StopbandDecay',53.333,'systemobject',true);
setspecs(Hf,'Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
Hd4 = design(Hf,'equiripple','StopbandShape','linear',...
    'StopbandDecay',53.333,'minphase',true,'systemobject',true);
hfvt2 = fvtool(Hd3,Hd4,'Color','White');
legend(hfvt2,'Linear-phase equiripple design with linearly decaying stopband',...
    'Minimum-phase equiripple design with linearly decaying stopband')
```
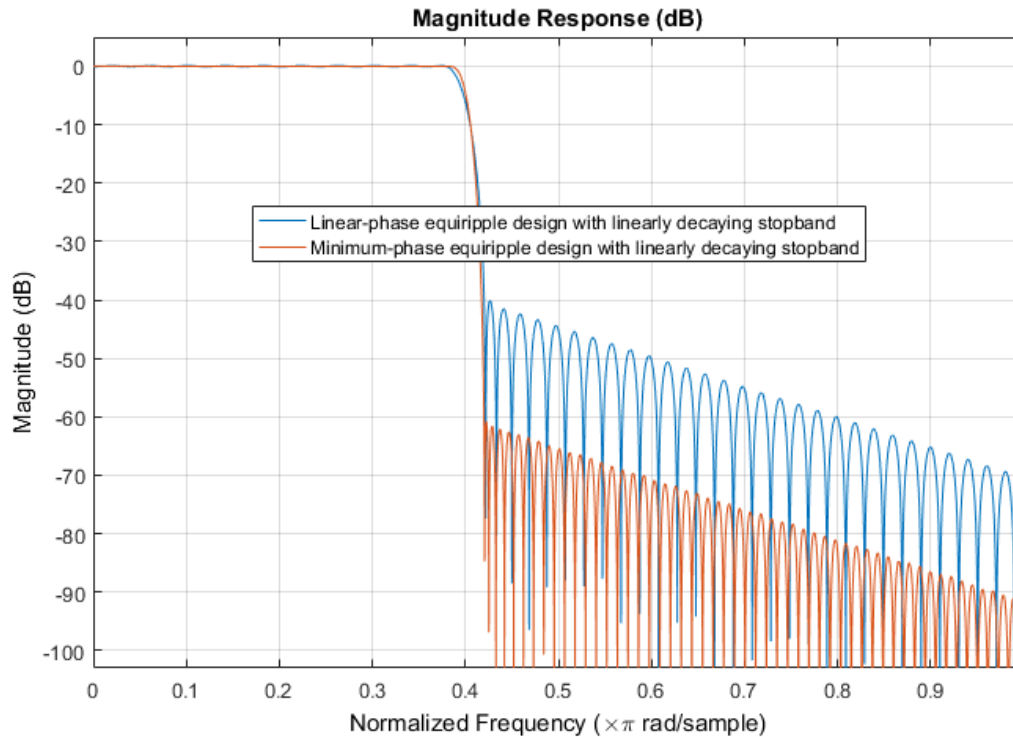
Minimum-Order Lowpass Filter Design Using Multistage Techniques

A different approach to minimizing the number of coefficients that does not involve minimum-phase designs is to use multistage techniques. Here we show an interpolated FIR (IFIR) approach.

```
Hd5 = ifir(Hf);
hfvt3 = fvtool(Hd1,Hd5,'Color','White');
legend(hfvt3,'Linear-phase equirriple design',...
       'Linear-phase IFIR design')
```
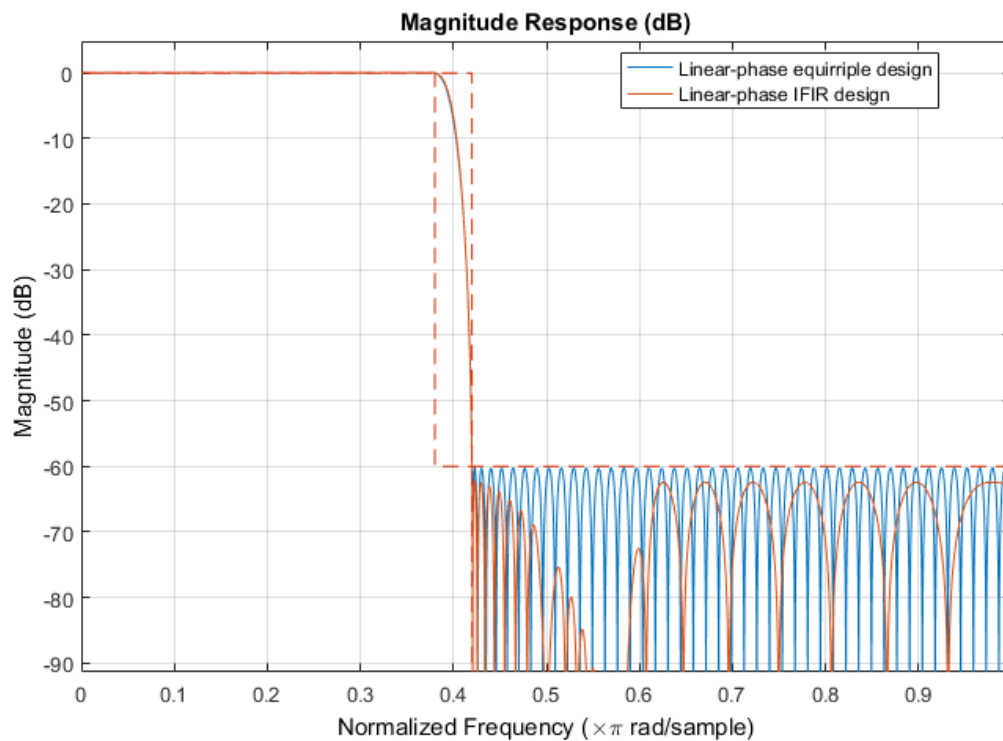
The number of nonzero coefficients required in the IFIR case is 111. Less than both the equiripple linear-phase and minimum-phase designs.

FDATool: A Filter Design and Analysis GUI

Overview

FDATool

The Filter Design and Analysis Tool (FDATool) is a user interface for designing and analyzing filters quickly. FDATool enables you to design digital FIR or IIR filters by setting filter specifications, by importing filters from your MATLAB workspace, or by adding, moving or deleting poles and zeros. FDATool also provides tools for analyzing filters, such as magnitude and phase response and pole-zero plots.

Filter Design Methods

FDATool gives you access to the following Signal Processing Toolbox filter design methods.

Design Method	Function
Butterworth	`butter`
Chebyshev Type I	`cheby1`
Chebyshev Type II	`cheby2`
Elliptic	`ellip`
Maximally Flat	`maxflat`
Equiripple	`firpm`
Least-squares	`firls`
Constrained least-squares	`fircls`
Complex equiripple	`cfirpm`

Design Method	Function
Window	`fir1`

When using the `window` method in FDATool, all Signal Processing Toolbox window functions are available, and you can specify a user-defined window by entering its function name and input parameter.

Advanced Filter Design Methods

The following advanced filter design methods are available if you have DSP System Toolbox software.

Design Method	Function
Constrained equiripple FIR	`firceqrip`
Constrained-band equiripple FIR	`fircband`
Generalized remez FIR	`firgr`
Equiripple halfband FIR	`firhalfband`
Least P-norm optimal FIR	`firlpnorm`
Equiripple Nyquist FIR	`firnyquist`
Interpolated FIR	`ifir`
IIR comb notching or peaking	`iircomb`
Allpass filter (given group delay)	`iirgrpdelay`
Least P-norm optimal IIR	`iirlpnorm`
Constrained least P-norm IIR	`iirlpnormc`
Second-order IIR notch	`iirnotch`
Second-order IIR peaking (resonator)	`iirpeak`

Using the Filter Design and Analysis Tool

There are different ways that you can design filters using the Filter Design and Analysis Tool. For example:

- You can first choose a response type, such as bandpass, and then choose from the available FIR or IIR filter design methods.

- You can specify the filter by its type alone, along with certain frequency- or time-domain specifications such as passband frequencies and stopband frequencies. The filter you design is then computed using the default filter design method and filter order.

Analyzing Filter Responses

Once you have designed your filter, you can display the filter coefficients and detailed filter information, export the coefficients to the MATLAB workspace, and create a C header file containing the coefficients, and analyze different filter responses in FDATool or in a separate Filter Visualization Tool (`fvtool`). The following filter responses are available:

- Magnitude response (`freqz`)
- Phase response (`phasez`)
- Group delay (`grpdelay`)
- Phase delay (`phasedelay`)
- Impulse response (`impz`)
- Step response (`stepz`)
- Pole-zero plots (`zplane`)
- Zero-phase response (`zerophase`)

Filter Design and Analysis Tool Panels

The Filter Design and Analysis Tool has sidebar buttons that display particular panels in the lower half of the tool. The panels are

- Design Filter. See "Choosing a Filter Design Method" on page 15-7 for more information. You use this panel to

  - Design filters from scratch.
  - Modify existing filters designed in FDATool.
  - Analyze filters.

- Import filter. You use this panel to

  - Import previously saved filters or filter coefficients that you have stored in the MATLAB workspace.

- Analyze imported filters.
- Pole/Zero Editor. See "Editing the Filter Using the Pole/Zero Editor" on page 15-17. You use this panel to add, delete, and move poles and zeros in your filter design.

If you also have DSP System Toolbox product installed, additional panels are available:

- Set quantization parameters — Use this panel to quantize double-precision filters that you design in FDATool, quantize double-precision filters that you import into FDATool, and analyze quantized filters.
- Transform filter — Use this panel to change a filter from one response type to another.
- Multirate filter design — Use this panel to create a multirate filter from your existing FIR design, create CIC filters, and linear and hold interpolators.

If you have Simulink installed, this panel is available:

- Realize Model — Use this panel to create a Simulink block containing the filter structure.

Getting Help

At any time, you can right-click or click the **What's this?** button, , to get information on the different parts of the tool. You can also use the **Help** menu to see complete Help information.

Using FDATool

To open FDATool, type

```
fdatool
```
at the MATLAB command prompt.

The Filter Design and Analysis Tool opens with the Design Filter panel displayed.
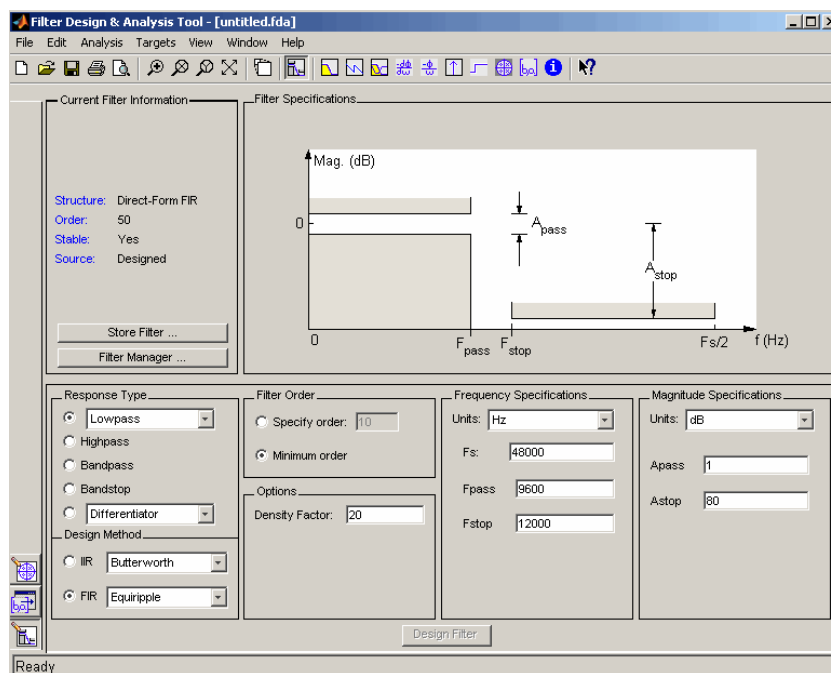
Note that when you open FDATool, **Design Filter** is not enabled. You must make a change to the default filter design in order to enable **Design Filter**. This is true each time you want to change the filter design. Changes to radio button items or drop down menu items such as those under **Response Type** or **Filter Order** enable **Design Filter** immediately. Changes to specifications in text boxes such as **Fs**, **Fpass**, and **Fstop** require you to click outside the text box to enable **Design Filter**.

Choosing a Response Type

You can choose from several response types:

- Lowpass
- Raised cosine
- Highpass
- Bandpass
- Bandstop
- Differentiator
- Multiband
- Hilbert transformer
- Arbitrary magnitude

Additional response types are available if you have DSP System Toolbox software installed.

To design a bandpass filter, select the radio button next to **Bandpass** in the Response Type region of the GUI.
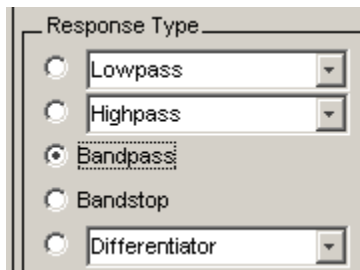
Note: Not all filter design methods are available for all response types. Once you choose your response type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected response type are removed from the Design Method region of the GUI.

Choosing a Filter Design Method

You can use the default filter design method for the response type that you've selected, or you can select a filter design method from the available FIR and IIR methods listed in the GUI.

To select the Remez algorithm to compute FIR filter coefficients, select the **FIR** radio button and choose `Equiripple` from the list of methods.
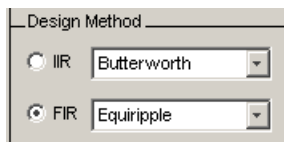
Setting the Filter Design Specifications

- "Viewing Filter Specifications" on page 15-8
- "Filter Order" on page 15-9
- "Options" on page 15-9
- "Bandpass Filter Frequency Specifications" on page 15-10
- "Bandpass Filter Magnitude Specifications" on page 15-11

Viewing Filter Specifications

The filter design specifications that you can set vary according to response type and design method. The display region illustrates filter specifications when you select **Analysis > Filter Specifications** or when you click the **Filter Specifications** toolbar button.

You can also view the filter specifications on the Magnitude plot of a designed filter by selecting **View > Specification Mask**.
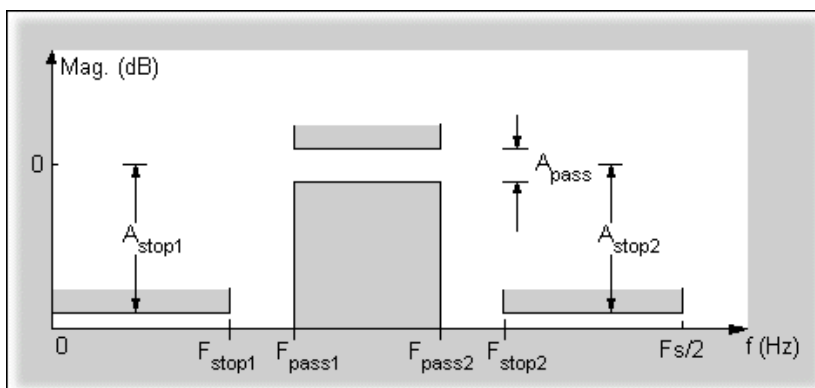
Filter Order

You have two mutually exclusive options for determining the filter order when you design an equiripple filter:

- **Specify order**: You enter the filter order in a text box.
- **Minimum order**: The filter design method determines the minimum order filter.

Select the **Minimum order** radio button for this example.
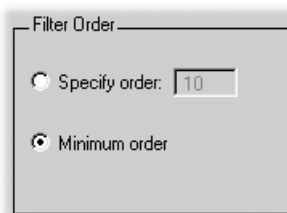
Note that filter order specification options depend on the filter design method you choose. Some filter methods may not have both options available.

Options

The available options depend on the selected filter design method. Only the FIR Equiripple and FIR Window design methods have settable options. For FIR Equiripple, the option is a **Density Factor**. See `firpm` for more information. For FIR Window the options are **Scale Passband**, **Window** selection, and for the following windows, a settable parameter:

Window	Parameter
Chebyshev (`chebwin`)	Sidelobe attenuation
Gaussian (`gausswin`)	Alpha
Kaiser (`kaiser`)	Beta
Taylor (`taylorwin`)	Nbar and Sidelobe level
Tukey (`tukeywin`)	Alpha
User Defined	Function Name, Parameter

You can view the window in the Window Visualization Tool (`wvtool`) by clicking the **View** button.

For this example, set the **Density factor** to 16.
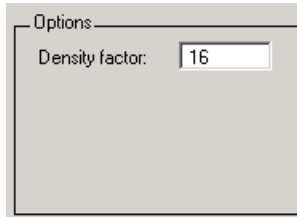
Bandpass Filter Frequency Specifications

For a bandpass filter, you can set

- Units of frequency:

    - Hz
    - kHz
    - MHz
    - Normalized (0 to 1)

- Sampling frequency
- Passband frequencies
- Stopband frequencies

You specify the passband with two frequencies. The first frequency determines the lower edge of the passband, and the second frequency determines the upper edge of the passband.

Similarly, you specify the stopband with two frequencies. The first frequency determines the upper edge of the first stopband, and the second frequency determines the lower edge of the second stopband.

For this example:

- Keep the units in **Hz** (default).
- Set the sampling frequency (**Fs**) to 2000 Hz.
- Set the end of the first stopband (**Fstop1**) to 200 Hz.
- Set the beginning of the passband (**Fpass1**) to 300 Hz.
- Set the end of the passband (**Fpass2**) to 700 Hz.

• Set the beginning of the second stopband (**Fstop2**) to 800 Hz.
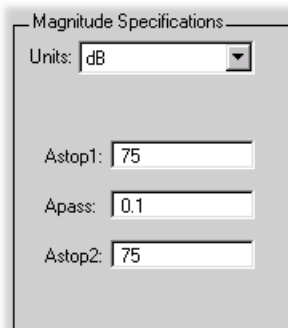
Bandpass Filter Magnitude Specifications

For a bandpass filter, you can specify the following magnitude response characteristics:

• Units for the magnitude response (dB or linear)
• Passband ripple
• Stopband attenuation

For this example:

• Keep **Units** in dB (default).
• Set the passband ripple (**Apass**) to 0.1 dB.
• Set the stopband attenuation for both stopbands (**Astop1**, **Astop2**) to 75 dB.

15-11

Computing the Filter Coefficients

Now that you've specified the filter design, click the **Design Filter** button to compute the filter coefficients.

Notice that the Design Filter button is disabled once you've computed the coefficients for your filter design. This button is enabled again once you make any changes to the filter specifications.

Analyzing the Filter

- "Displaying Filter Responses" on page 15-12
- "Using Data Tips" on page 15-14
- "Drawing Spectral Masks" on page 15-14
- "Changing the Sampling Frequency" on page 15-16
- "Displaying the Response in FVTool" on page 15-16

Displaying Filter Responses

You can view the following filter response characteristics in the display region or in a separate window.

- Magnitude response
- Phase response
- Magnitude and Phase responses
- Group delay response
- Phase delay response
- Impulse response
- Step response
- Pole-zero plot
- Zero-phase response — available from the *y*-axis context menu in a Magnitude or Magnitude and Phase response plot.

If you have DSP System Toolbox product installed, two other analyses are available: magnitude response estimate and round-off noise power. These two analyses are the only ones that use filter internals.

For descriptions of the above responses and their associated toolbar buttons and other FDATool toolbar buttons, see `fvtool`.

You can display two responses in the same plot by selecting **Analysis** > **Overlay Analysis** and selecting an available response. A second *y*-axis is added to the right side of the response plot. (Note that not all responses can be overlaid on each other.)

You can also display the filter coefficients and detailed filter information in this region.

For all the analysis methods, except zero-phase response, you can access them from the **Analysis** menu, the Analysis Parameters dialog box from the context menu, or by using the toolbar buttons. For zero-phase, right-click the *y*-axis of the plot and select **Zero-phase** from the context menu.
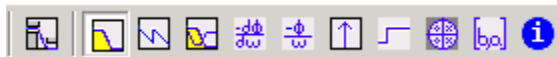
For example, to look at the filter's magnitude response, select the **Magnitude Response** button on the toolbar.
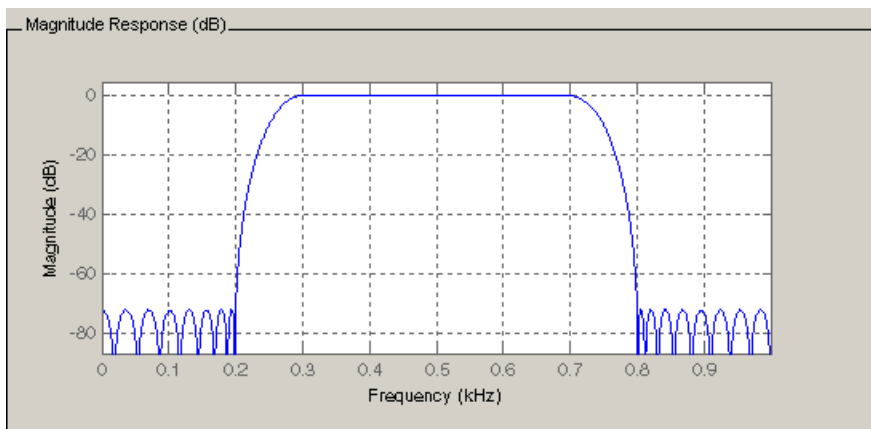
You can also overlay the filter specifications on the Magnitude plot by selecting **View** > **Specification Mask**.

Note: You can use specification masks in FVTool only if FVTool was launched from FDATool.
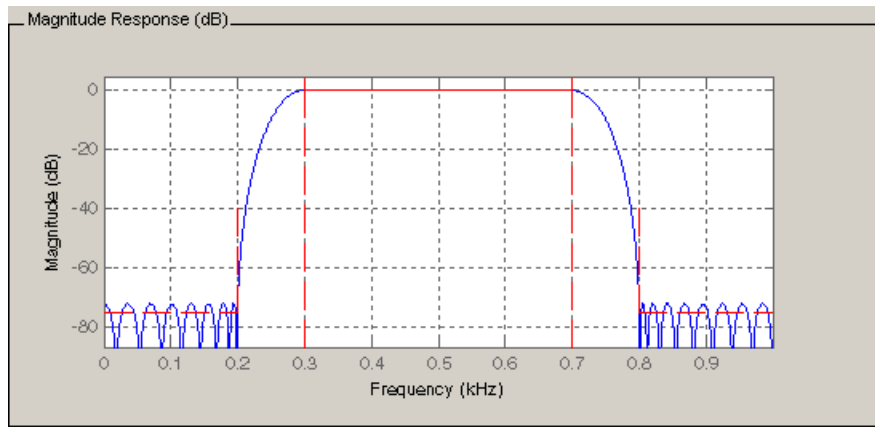
Using Data Tips

You can click the response to add plot data tips that display information about particular points on the response.
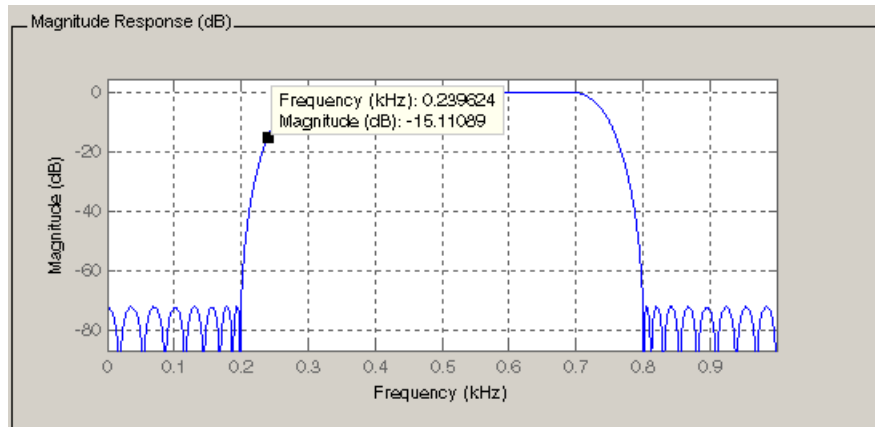
For information on using data tips, see "Display Data Values Interactively" in the MATLAB documentation.

Drawing Spectral Masks

To add spectral masks or rejection area lines to your magnitude plot, click **View** > **User-defined Spectral Mask**.

User-defined Spectral Mask

☑ Enable Mask

☑ Normalized Frequency

Frequency Vector: [0 0.2 0.3 0.7 0.8 1]   0 to 1

Magnitude Units: dB

Magnitude Vector: [-72 -72 0 0 -72 -72]

OK   Cancel   Help   Apply

The mask is defined by a frequency vector and a magnitude vector. These vectors must be the same length.

- Enable Mask — Select to turn on the mask display.
- Normalized Frequency — Select to normalize the frequency between 0 and 1 across the displayed frequency range.
- Frequency Vector — Enter a vector of *x*-axis frequency values.
- Magnitude Units — Select the desired magnitude units. These units should match the units used in the magnitude plot.
- Magnitude Vector — Enter a vector of *y*-axis magnitude values.

The magnitude response below shows a spectral mask.

Magnitude Response (dB)
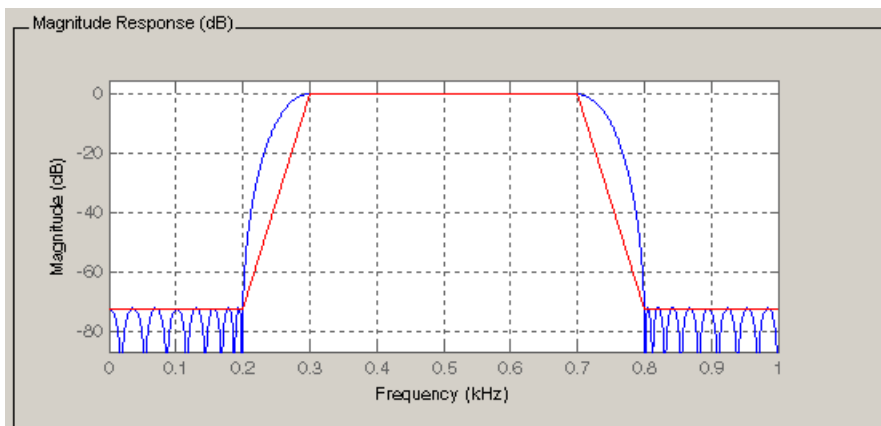
Changing the Sampling Frequency

To change the sampling frequency of your filter, right-click any filter response plot and select **Sampling Frequency** from the context menu.
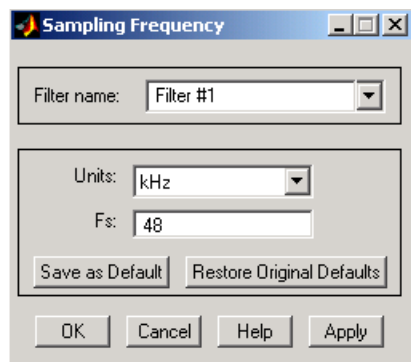
To change the filter name, type the new name in **Filter name**. (In `fvtool`, if you have multiple filters, select the desired filter and then enter the new name.)

To change the sampling frequency, select the desired unit from **Units** and enter the sampling frequency in **Fs**. (For each filter in `fvtool`, you can specify a different sampling frequency or you can apply the sampling frequency to all filters.)

To save the displayed parameters as the default values to use when FDATool or FVTool is opened, click **Save as Default**.

To restore the default values, click **Restore Original Defaults**.

Displaying the Response in FVTool

To display the filter response characteristics in a separate window, select **View** > **Filter Visualization Tool** (available if any analysis, except the filter specifications, is in the

display region) or click the **Full View Analysis** button:

This launches the Filter Visualization Tool (`fvtool`).

Note: If Filter Specifications are shown in the display region, clicking the **Full View Analysis** toolbar button launches a MATLAB figure window instead of FVTool. The

associated menu item is **Print to figure**, which is enabled only if the filter specifications are displayed.

You can use this tool to annotate your design, view other filter characteristics, and print your filter response. You can link FDATool and FVTool so that changes made in FDATool are immediately reflected in FVTool. See `fvtool` for more information.
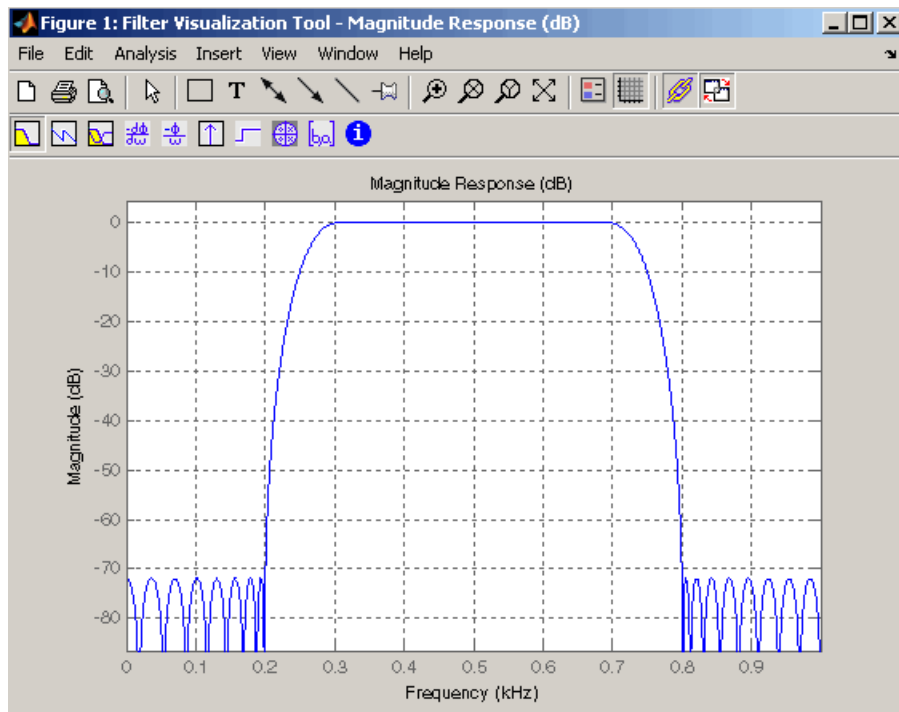
Editing the Filter Using the Pole/Zero Editor

- "Displaying the Pole-Zero Plot" on page 15-17
- "Changing the Pole-Zero Plot" on page 15-18

Displaying the Pole-Zero Plot

You can edit a designed or imported filter's coefficients by moving, deleting, or adding poles and/or zeros using the Pole/Zero Editor panel.

Note: You cannot generate MATLAB code (**File** > **Generate MATLAB code**) if your filter was designed or edited with the Pole/Zero Editor.

You cannot move quantized poles and zeros. You can only move the reference poles and zeros.

Click the **Pole/Zero Editor** button in the sidebar or select **Edit** > **Pole/Zero Editor** to display this panel.
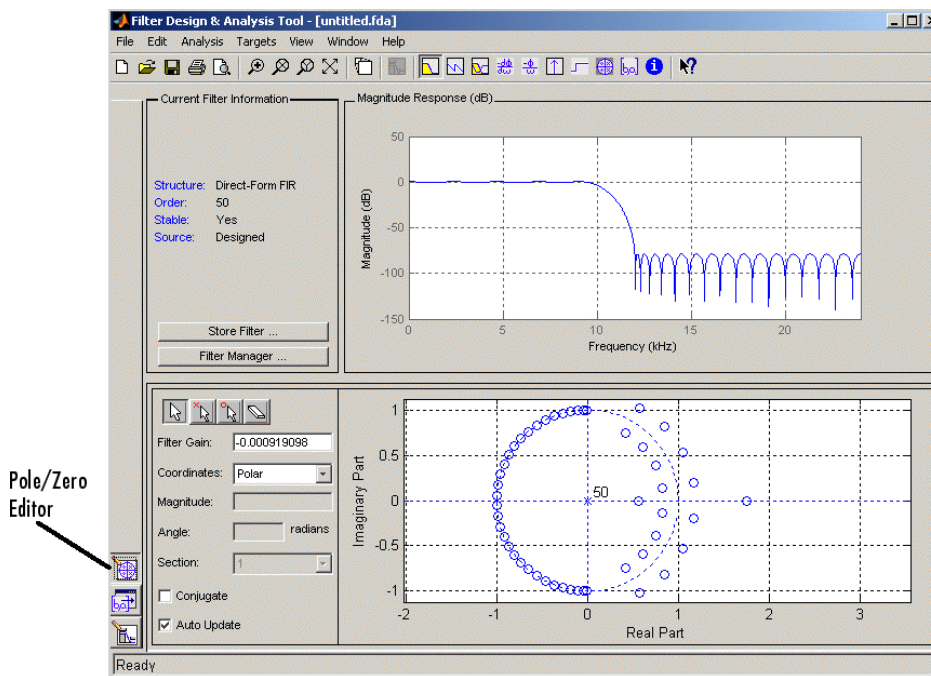
Poles are shown using x symbols and zeros are shown using o symbols.

Changing the Pole-Zero Plot

Plot mode buttons are located to the left of the pole/zero plot. Select one of the buttons to change the mode of the pole/zero plot. The Pole/Zero Editor has these buttons from left to right: move pole, add pole, add zero, and delete pole or zero.
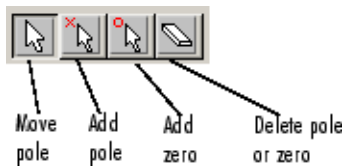
The following plot parameters and controls are located to the left of the pole/zero plot and below the plot mode buttons.

- **Filter gain** — factor to compensate for the filter's pole(s) and zero(s) gains
- **Coordinates** — units (`Polar` or `Rectangular`) of the selected pole or zero
- **Magnitude** — if polar coordinates is selected, magnitude of the selected pole or zero
- **Angle** — if polar coordinates is selected, angle of selected pole(s) or zero(s)
- **Real** — if rectangular coordinates is selected, real component of selected pole(s) or zero(s)
- **Imaginary** — if rectangular coordinates is selected, imaginary component of selected pole or zero
- **Section** — for multisection filters, number of the current section
- **Conjugate** — creates a corresponding conjugate pole or zero or automatically selects the conjugate pole or zero if it already exists.
- **Auto update** — immediately updates the displayed magnitude response when poles or zeros are added, moved, or deleted.

The **Edit > Pole/Zero Editor** has items for selecting multiple poles/zeros, for inverting and mirroring poles/zeros, and for deleting, scaling and rotating poles/zeros.

Select All
Select All Poles
Select All Zeros
Select Inside Unit Circle
Select Outside Unit Circle
Select Below Real Axis
Select Above Real Axis
Select Left Half
Select Right Half

Invert about the Real Axis
Invert about the Imaginary Axis
Invert about the Unit Circle

Mirror about the Real Axis
Mirror about the Imaginary Axis
Mirror about the Unit Circle

Delete current
Scale by a Factor
Rotate Counter-Clockwise

These items are enabled only when one or more poles/zeros are selected.

Moving one of the zeros on the vertical axis produces the following result:

- The selected zero pair is shown in green.
- When you select one of the zeros from a conjugate pair, the Conjugate check box and the conjugate are automatically selected.
- The Magnitude Response plot updates immediately because **Auto update** is active.

Converting the Filter Structure

- "Converting to a New Structure" on page 15-21
- "Converting to Second-Order Sections" on page 15-23

Converting to a New Structure

You can use **Edit** > **Convert Structure** to convert the current filter to a new structure. All filters can be converted to the following representations:

- Direct-form I
- Direct-form II
- Direct-form I transposed

- Direct-form II transposed
- Lattice ARMA

Note: If you have DSP System Toolbox product installed, you will see additional structures in the Convert structure dialog box.

In addition, the following conversions are available for particular classes of filters:

- Minimum phase FIR filters can be converted to Lattice minimum phase
- Maximum phase FIR filters can be converted to Lattice maximum phase
- Allpass filters can be converted to Lattice allpass
- IIR filters can be converted to Lattice ARMA

Note: Converting from one filter structure to another may produce a result with different characteristics than the original. This is due to the computer's finite-precision arithmetic and the variations in the conversion's roundoff computations.

For example:

- Select **Edit** > **Convert Structure** to open the Convert structure dialog box.
- Select `Direct-form I` in the list of filter structures.
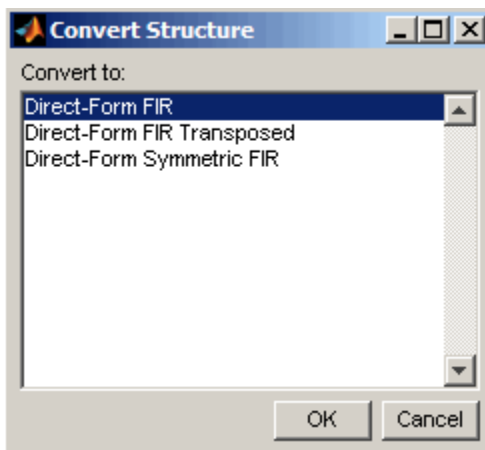
Converting to Second-Order Sections

You can use **Edit** > **Convert to Second-Order Sections** to store the converted filter structure as a collection of second-order sections rather than as a monolithic higher-order structure.

Note: The following options are also used for **Edit** > **Reorder and Scale Scale Second-Order Sections**, which you use to modify an SOS filter structure.

The following **Scale** options are available when converting a direct-form II structure only:

- None (default)
- L-2 (L^2 norm)
- L-infinity (L^∞ norm)

The **Direction** (Up or Down) determines the ordering of the second-order sections. The optimal ordering changes depending on the **Scale** option selected.

For example:

- Select **Edit** > **Convert to Second-Order Sections** to open the Convert to SOS dialog box.
- Select L-infinity from the **Scale** menu for L^∞ norm scaling.
- Leave Up as the **Direction** option.

Note: To convert from second-order sections back to a single section, use **Edit** > **Convert to Single Section**.

Exporting a Filter Design

- "Exporting Coefficients or Objects to the Workspace" on page 15-24
- "Exporting Coefficients to an ASCII File" on page 15-25
- "Exporting Coefficients or Objects to a MAT-File" on page 15-25
- "Exporting to SPTool" on page 15-26

- • "Exporting to a Simulink Model" on page 15-26
- • "Other Ways to Export a Filter" on page 15-29

Exporting Coefficients or Objects to the Workspace

You can save the filter either as filter coefficients variables or as a `dfilt` or `mfilt` filter object variable. (Note that you must have DSP System Toolbox product installed to save as an `mfilt`.) To save the filter to the MATLAB workspace:

1   Select **File** > **Export**. The Export dialog box appears.

2   Select `Workspace` from the **Export To** menu.

3   Select `Coefficients` from the **Export As** menu 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** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

For objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.

5   Click the **Export** button.

Exporting Coefficients to an ASCII File

To save filter coefficients to a text file,

1 Select **File** > **Export**. The Export dialog box appears.

2 Select `Coefficients File (ASCII)` from the **Export To** menu.

3 Click the **Export** button. The Export Filter Coefficients to .FCF File dialog box appears.

4 Choose or enter a filename and click the **Save** button.

The coefficients are saved in the text file that you specified, and the MATLAB Editor opens to display the file. The text file also contains comments with the MATLAB version number, the Signal Processing Toolbox version number, and filter information.

Exporting Coefficients or Objects to a MAT-File

To save filter coefficients or a filter object as variables in a MAT-file:

1 Select **File** > **Export**. The Export dialog box appears.

2 Select `MAT-file` from the **Export To** menu.

3 Select `Coefficients` from the **Export As** menu 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** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

For objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.

5 Click the **Export** button. The Export to a MAT-File dialog box appears.

6 Choose or enter a filename and click the **Save** button.

Exporting to SPTool

You may want to use your designed filter in SPTool to do signal processing and analysis.

1 Select **File** > **Export**. The Export dialog box appears.

2 Select SPTool from the **Export To** menu.

3 Assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.

4 Click the **Export** button.

SPTool opens and the current FDATool filter appears in the Filter area list as the specified variable name followed by (Imported).

Note: If you are using the DSP System Toolbox software and export a quantized filter, only the values of its quantized coefficients are exported. The reference coefficients are not exported. SPTool does not restrict the coefficient values, so if you edit them in SPTool by moving poles or zeros, the filter will no longer be in quantized form.

Exporting to a Simulink Model

If you have the Simulink product installed, you can export a Simulink block of your filter design and insert it into a new or existing Simulink model.

You can export a filter designed using any filter design method available in FDATool.

Note: If you have the DSP System Toolbox and Fixed-Point Designer installed, you can export a CIC filter to a Simulink model.

---

1   After designing your filter, click the **Realize Model** sidebar button or select **File** >
     Export to Simulink Model. The Realize Model panel is displayed.
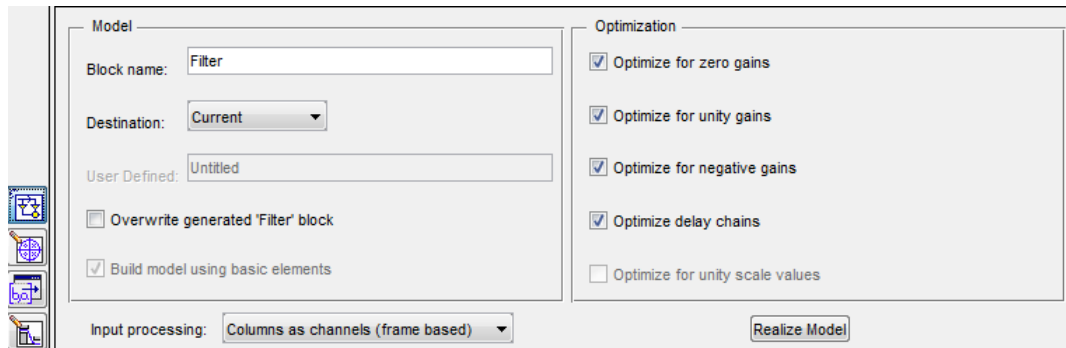
2   Specify the name to use for your block in **Block name**.

3   To insert the block into the current (most recently selected) Simulink model, set the
     Destination to Current. To inset the block into a new model, select New. To insert
     the block into a user-defined subsystem, select User defined.

4   If you want to overwrite a block previously created from this panel, check
     Overwrite generated `Filter' block.

5   If you select the **Build model using basic elements** check box, your filter is
     created as a subsystem block, which uses separate sub-elements. In this mode, the
     following optimization(s) are available:

   •   Optimize for zero gains — Removes zero-valued gain paths from the filter
        structure.

   •   Optimize for unity gains — Substitutes a wire (short circuit) for gains
        equal to 1 in the filter structure.

   •   Optimize for negative gains — Substitutes a wire (short circuit) for gains
        equal to -1 and changes corresponding additions to subtractions in the filter
        structure.

   •   Optimize delay chains — Substitutes delay chains composed of n unit delays
        with a single delay of n.

   •   Optimize for unity scale values — Removes multiplications for scale
        values equal to 1 from the filter structure.

   The following illustration shows the effects of some of the optimizations:
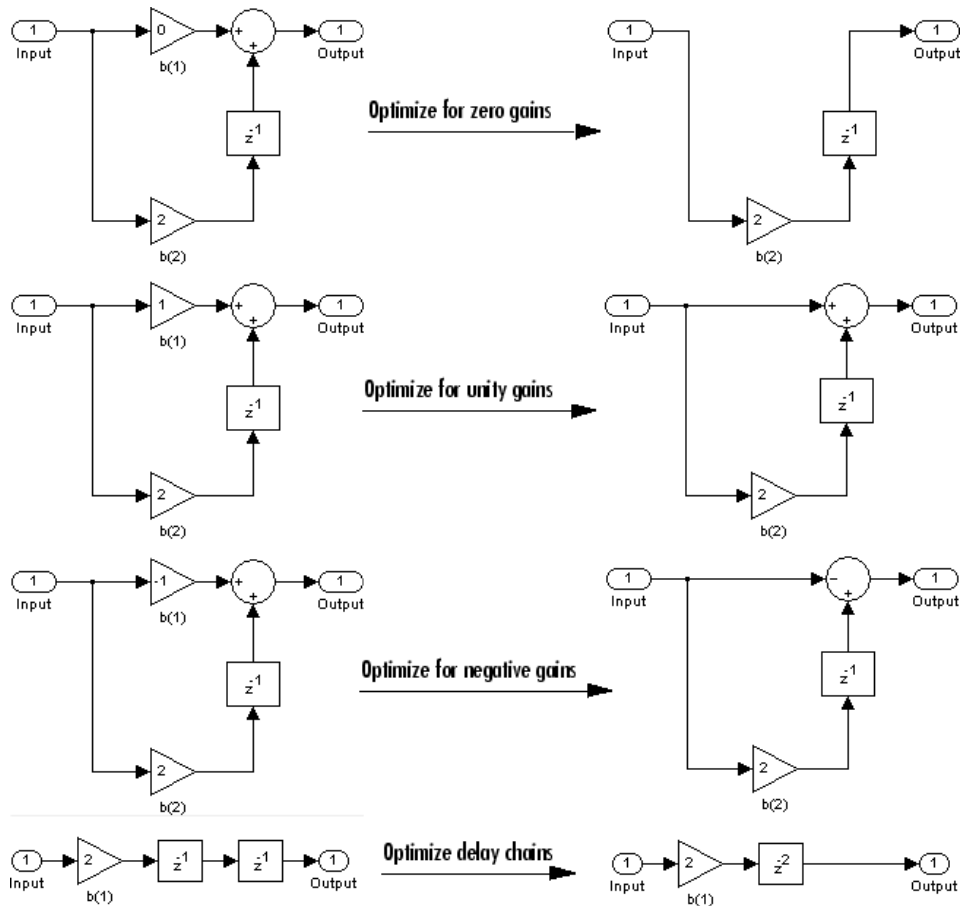
Optimization Effects

Note: The **Build model using basic elements** check box is enabled only when you have a DSP System Toolbox license and your filter can be designed using digital filter blocks from that library. For more information, see the `Filter Realization Wizard` topic in the DSP System Toolbox documentation.

6 Set the **Input processing** parameter to specify whether the generated filter performs sample- or frame-based processing on the input. Depending on the type of filter you design, one or both of the following options may be available:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

7 Click the **Realize Model** button to create the filter block. When the **Build model using basic elements** check box is selected, FDATool implements the filter as a subsystem block using `Sum`, `Gain`, and `Delay` blocks.

If you double-click the Simulink Filter block, the filter structure is displayed.

Other Ways to Export a Filter

You can also send your filter to a C header file or generate MATLAB code to construct your filter from the command line. For detailed instructions, see the following sections:

- "Generating a C Header File" on page 15-29
- "Generating MATLAB Code" on page 15-31

Generating a C Header File

You may want to include filter information in an external C program. To create a C header file with variables that contain filter parameter data, follow this procedure:

1 Select **Targets** > **Generate C Header**. The Generate C Header dialog box appears.
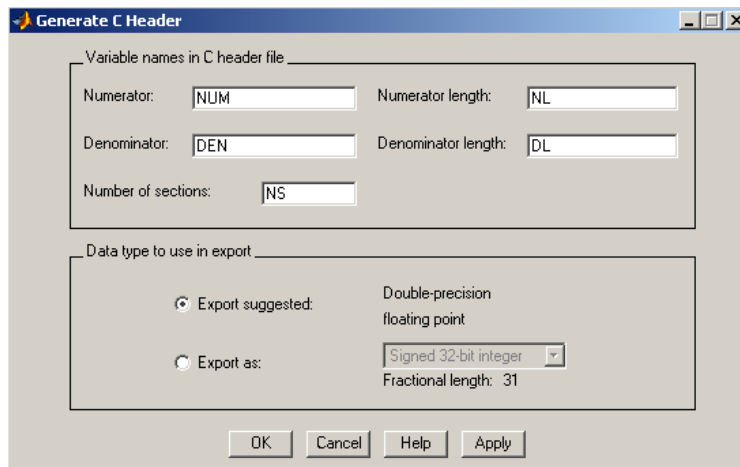
2 Enter the variable names to be used in the C header file. The particular filter structure determines the variables that are created in the file

Filter Structure	Variable Parameter
Direct-form I Direct-form II Direct-form I transposed Direct-form II transposed	Numerator, Numerator length*, Denominator, Denominator length*, and Number of sections (inactive if filter has only one section)
Lattice ARMA	Lattice coeffs, Lattice coeffs length*, Ladder coeffs, Ladder coeffs length*, Number of sections (inactive if filter has only one section)
Lattice MA	Lattice coeffs, Lattice coeffs length*, and Number of sections (inactive if filter has only one section)
Direct-form FIR Direct-form FIR transposed	Numerator, Numerator length*, and Number of sections (inactive if filter has only one section)

*length variables contain the total number of coefficients of that type.

Note: Variable names cannot be C language reserved words, such as "for."

---

3 Select **Export Suggested** to use the suggested data type or select **Export As** and select the desired data type from the pull-down.

Note: If you do not have DSP System Toolbox software installed, selecting any data type other than double-precision floating point results in a filter that does not exactly match the one you designed in the FDATool. This is due to rounding and truncating differences.

---

4 Click **OK** to save the file and close the dialog box or click **Apply** to save the file, but leave the dialog box open for additional C header file definitions.

Generating MATLAB Code

You can generate MATLAB code that constructs the filter you designed in FDATool from the command line. Select **File** > **Generate MATLAB Code** > **Filter Design Function** and specify the filename in the Generate MATLAB code dialog box.

Note: You cannot generate MATLAB code (**File** > **Generate MATLAB Code** > **Filter Design Function**) if your filter was designed or edited with the Pole/Zero Editor.

The following is generated MATLAB code for the default lowpass filter in FDATool.

```
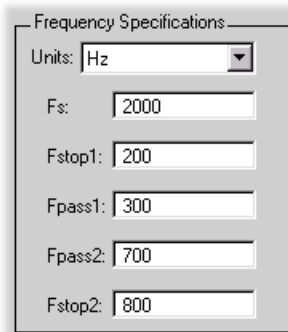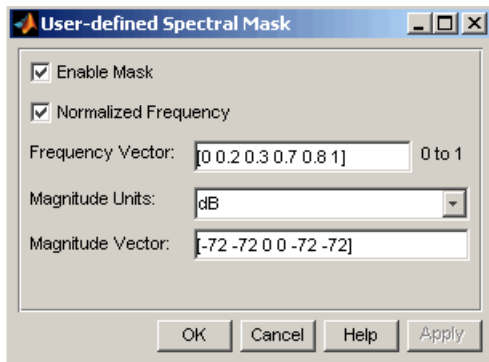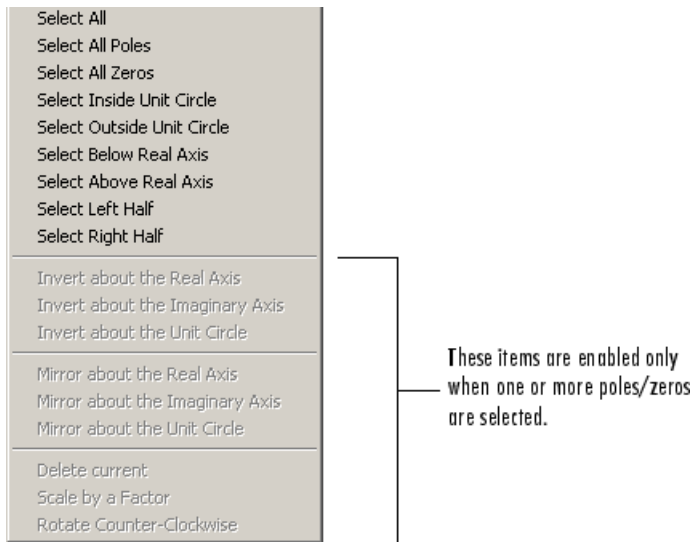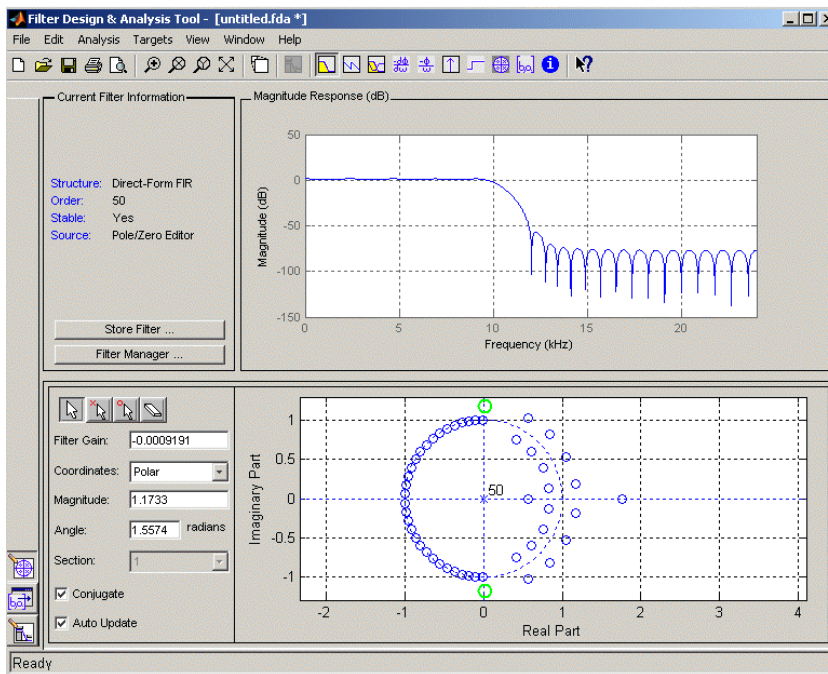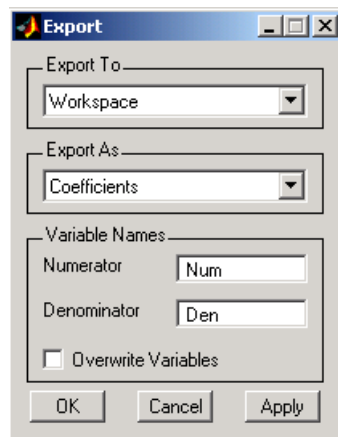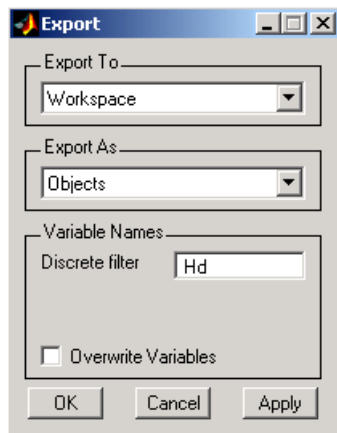function Hd = ExFilter
%EXFILTER Returns a discrete-time filter object.

%
% MATLAB Code
% Generated by MATLAB(R) 7.11 and the Signal Processing Toolbox 6.14.
%
% Generated on: 17-Feb-2010 14:15:37
%

% Equiripple Lowpass filter designed using the FIRPM function.

% All frequency values are in Hz.
Fs = 48000;  % Sampling Frequency

Fpass = 9600;            % Passband Frequency
Fstop = 12000;           % Stopband Frequency
Dpass = 0.057501127785;  % Passband Ripple
Dstop = 0.0001;          % Stopband Attenuation
dens  = 20;              % Density Factor

% Calculate the order from the parameters using FIRPMORD.
[N, Fo, Ao, W] = firpmord([Fpass, Fstop]/(Fs/2), [1 0], [Dpass, Dstop]);

% Calculate the coefficients using the FIRPM function.
```

```
b  = firpm(N, Fo, Ao, W, {dens});
Hd = dfilt.dffir(b);

% [EOF]
```

Managing Filters in the Current Session

You can store filters designed in the current FDATool session for cascading together, exporting to FVTool or for recalling later in the same or future FDATool sessions.

You store and access saved filters with the **Store filter** and **Filter Manager** buttons, respectively, in the Current Filter Information pane.
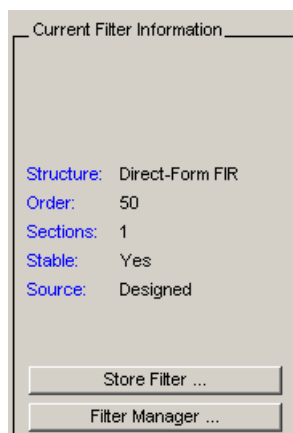
Store Filter — Displays the Store Filter dialog box in which you specify the filter name to use when storing the filter in the Filter Manager. The default name is the type of the filter.
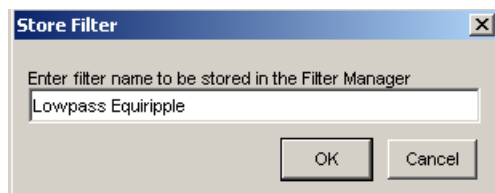
Filter Manager — Opens the Filter Manager.
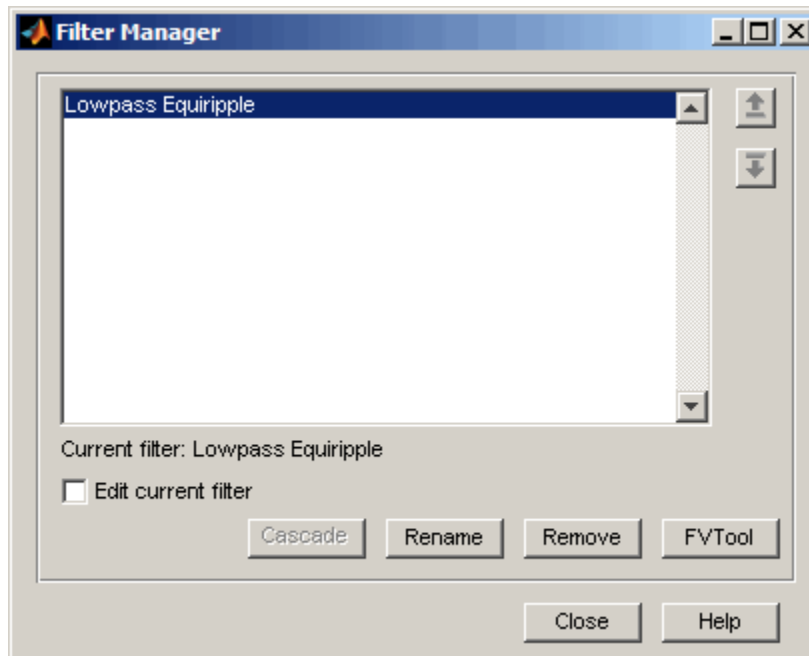
The current filter is listed below the listbox. To change the current filter, highlight the desired filter. If you select **Edit current filter**, FDATool displays the currently selected filter specifications. If you make any changes to the specifications, the stored filter is updated immediately.

To cascade two or more filters, highlight the desired filters and press **Cascade**. A new cascaded filter is added to the Filter Manager.

To change the name of a stored filter, press **Rename**. The Rename filter dialog box is displayed.

To remove a stored filter from the Filter Manager, press **Delete**.

To export one or more filters to FVTool, highlight the filter(s) and press **FVTool**.

Saving and Opening Filter Design Sessions

You can save your filter design session as a MAT-file and return to the same session another time.

Select the **Save session** button ![save] to save your session as a MAT-file. The first time you save a session, a Save Filter Design File browser opens, prompting you for a session name.
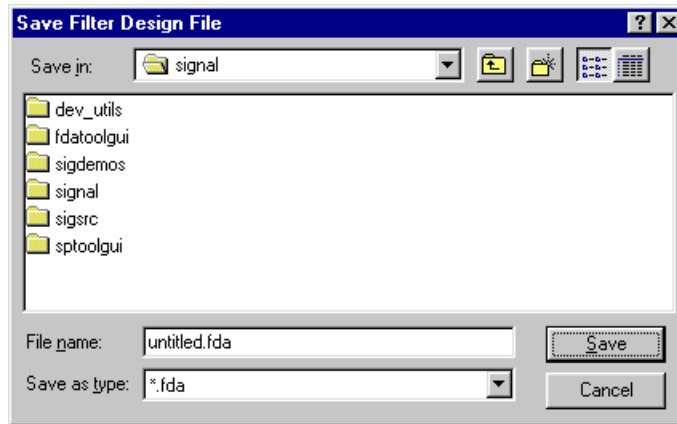
For example, save this design session as `TestFilter.fda` in your current working directory by typing `TestFilter` in the **File name** field.

The `.fda` extension is added automatically to all filter design sessions you save.

Note: You can also use the **File** > **Save session** and **File** > **Save session as** to save a session.

You can load existing sessions into the Filter Design and Analysis Tool by selecting the

Open session button, ![open] or **File** > **Open session** . A Load Filter Design File browser opens that allows you to select from your previously saved filter design sessions.

Importing a Filter Design

In this section...
"Import Filter Panel" on page 15-35
"Filter Structures" on page 15-36

Import Filter Panel

The Import Filter panel allows you to import a filter. You can access this region by clicking the **Import Filter** button in the sidebar.

The imported filter can be in any of the representations listed in the **Filter Structure** pull-down menu. You can import a filter as second-order sections by selecting the check box.

Specify the filter coefficients in **Numerator** and **Denominator**, either by entering them explicitly or by referring to variables in the MATLAB workspace.

Select the frequency units from the following options in the **Units** menu, and for any frequency unit other than Normalized, specify the value or MATLAB workspace variable of the sampling frequency in the **Fs** field.

To import the filter, click the **Import Filter** button. The display region is automatically updated when the new filter has been imported.

You can edit the imported filter using the Pole/Zero Editor panel.

Filter Structures

The available filter structures are:

- Direct Form, which includes direct-form I, direct-form II, direct-form I transposed, direct-form II transposed, and direct-form FIR
- Lattice, which includes lattice allpass, lattice MA min phase, lattice MA max phase, and lattice ARMA
- Discrete–time Filter (`dfilt` object)

The structure that you choose determines the type of coefficients that you need to specify in the text fields to the right.

Direct-form

For direct-form I, direct-form II, direct-form I transposed, and direct-form II transposed, specify the filter by its transfer function representation

$$H(z) = \frac{b(1) + b(2)z^{-1} + b(3)z^{-2} + \ldots b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + a(3)Z^{-3} + \ldots a(n+1)z^{-n}}$$

- The **Numerator** field specifies a variable name or value for the numerator coefficient vector `b`, which contains `m+1` coefficients in descending powers of z.
- The **Denominator** field specifies a variable name or value for the denominator coefficient vector `a`, which contains `n+1` coefficients in descending powers of z. For FIR filters, the **Denominator** is `1`.

Filters in transfer function form can be produced by all of the Signal Processing Toolbox filter design functions (such as `fir1`, `fir2`, `firpm`, `butter`, `yulewalk`). See "Transfer Function" for more information.

Importing as second-order sections

For all direct-form structures, except direct-form FIR, you can import the filter in its second-order section representation:

$$H(z) = G \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

The **Gain** field specifies a variable name or a value for the gain G, and the **SOS Matrix** field specifies a variable name or a value for the L-by-6 SOS matrix

$$SOS = \begin{pmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{22} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{pmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

Filters in second-order section form can be produced by functions such as `tf2sos`, `zp2sos`, `ss2sos`, and `sosfilt`. See "Second-Order Sections (SOS)" for more information.

Lattice

For lattice allpass, lattice minimum and maximum phase, and lattice ARMA filters, specify the filter by its lattice representation:

- For lattice allpass, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where N is the filter order.
- For lattice MA (minimum or maximum phase), the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where N is the filter order.
- For lattice ARMA, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, and the **Ladder coeff** field specifies the ladder coefficients, `v(1)` to `v(N+1)`, where N is the filter order.

Filters in lattice form can be produced by `tf2latc`. See "Lattice Structure" for more information.

Discrete-time Filter (dfilt object)

For Discrete-time filter, specify the name of the `dfilt` object. See `dfilt` for more information.

Multirate Filter (mfilt object)

For Multirate filter, specify the name of the `mfilt` object. See `mfilt` in the DSP System Toolbox product for more information.

Designing a Filter in the Filterbuilder GUI

Filterbuilder Design Process

In this section...

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.

Note: `filterbuilder` requires the Signal Processing Toolbox. The functionality of `filterbuilder` is greatly expanded by the DSP System Toolbox. Many of the features described or displayed below are only available if the DSP System Toolbox is installed. You may verify your installation by typing `ver` at the command prompt.

Design a Filter Using Filterbuilder

The basic workflow in using `filterbuilder` 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 DSP System Toolbox.

Note This step cannot be skipped because it is not automatically completed for you by the 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.

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 are `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 the software. To filter your data, you must explicitly execute this step. In the **Bandpass Design** dialog box, click OK and DSP System Toolbox creates the filter object 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:

```
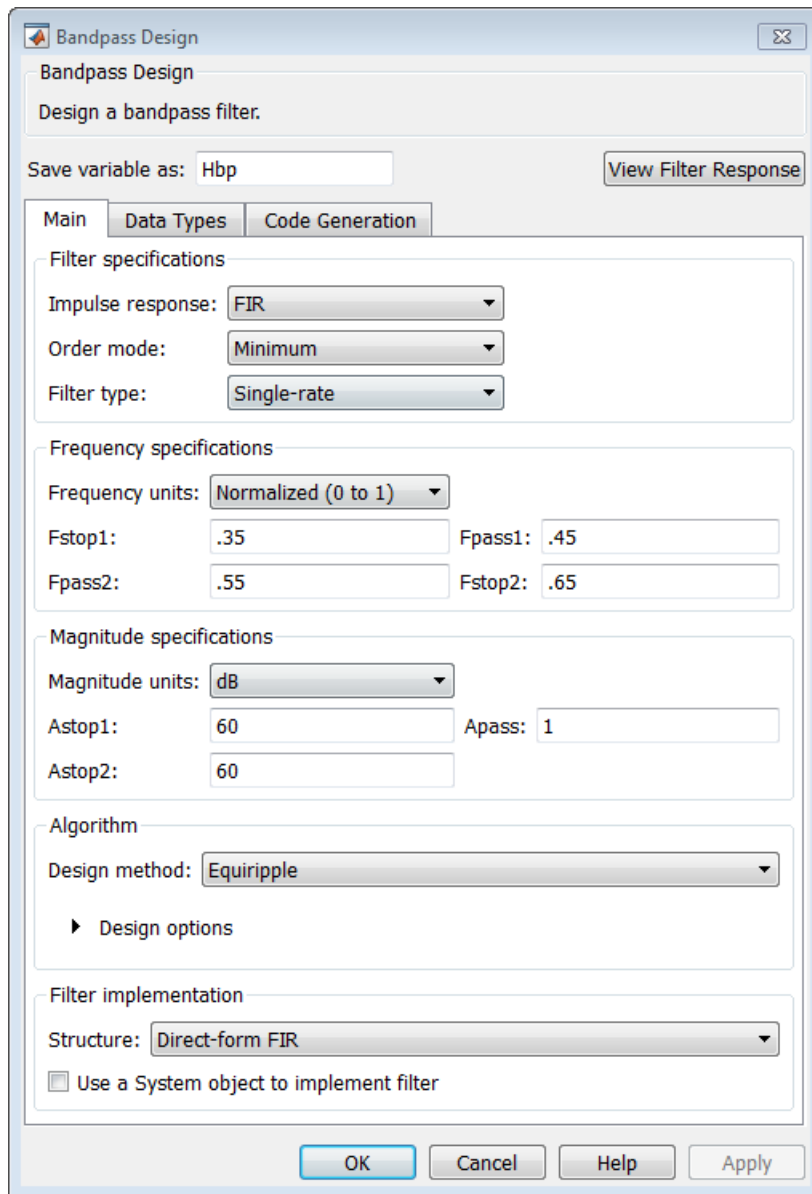>> y = filter (Hbp, x)
```
To understand how the filtering commands work, type:

```
>> 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:

```
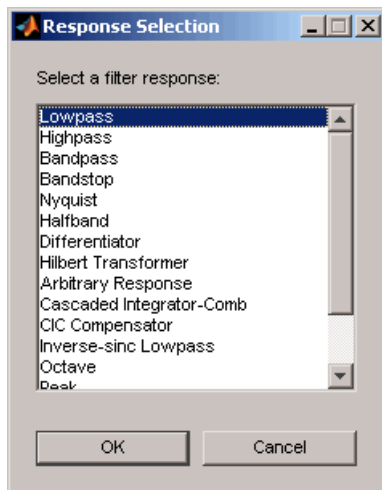filterbuilder
```
The **Response Selection** dialog box appears. In this dialog box, you can select from a list of filter response types. Select `Lowpass` in the list box.

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.
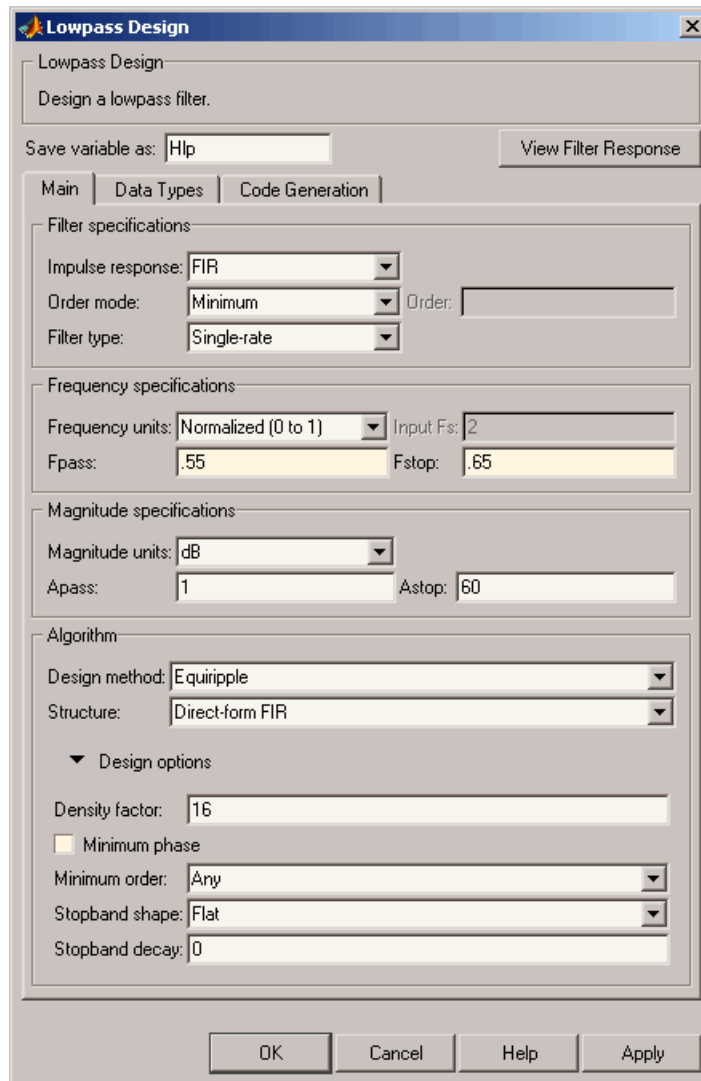
3 Click **Apply**, and the following message appears at the MATLAB prompt:

```
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.
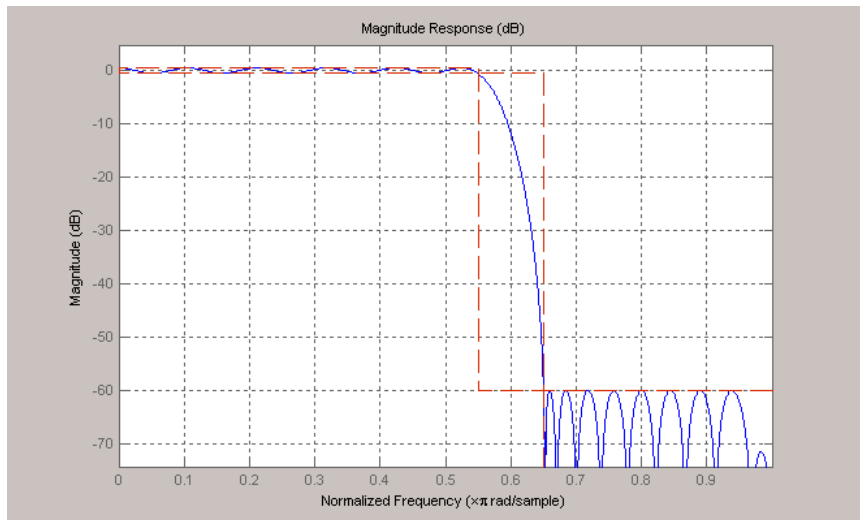
You can change the design and click **Apply**, followed by **View Filter Response**, as many times as needed until your design specifications are met.

Bibliography

References — 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, Springer, 1995.

[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.

References — 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.

References — 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.

References — 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.

References — Fixed-Point Filters

[1] Jackson, L., *Digital Filtering and Signal Processing with MATLAB Exercises*, Third edition, Springer, 1995, pp.373–422.

[2] Dehner, G, "Noise optimized IIR digital filter design: tutorial and some new aspects," *Signal Processing, Vol 83, Issue 8 (August 2003) pp.1565–1582.*